Thomas Hanti

Knowledgebase basiertes Scheduling für hierarchisch asynchrone Multi-Core Scheduler im Systembereich Automotive und Avionik

Wissenschaftliche Schriftenreihe
**Eingebettete, selbstorganisierende Systeme**

Band 17

Prof. Dr. Wolfram Hardt (Herausgeber)

Thomas Hanti

# Knowledgebase basiertes Scheduling für hierarchisch asynchrone Multi-Core Scheduler im Systembereich Automotive und Avionik

TECHNISCHE UNIVERSITÄT
CHEMNITZ

**Universitätsverlag Chemnitz**
**2019**

**Impressum**

**Bibliografische Information der Deutschen Nationalbibliothek**

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über `http://www.dnb.de` abrufbar.

Titelgrafik: Thomas Hanti
Satz/Layout: Thomas Hanti

Technische Universität Chemnitz/Universitätsbibliothek
**Universitätsverlag Chemnitz**
09107 Chemnitz
`https://www.tu-chemnitz.de/ub/univerlag`

readbox unipress
in der readbox publishing GmbH
Am Hawerkamp 31
48155 Münster
`http://unipress.readbox.net`

ISSN 2196-3932 print - ISSN 2196-4815 online

ISBN 978-3-96100-077-7

`http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa2-326409`

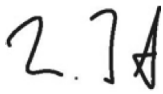
## Vorwort zur Wissenschaftlichen Schriftenreihe „Eingebettete, Selbstorganisierende Systeme“

Dieser Band der wissenschaftlichen Schriftenreihe „Eingebettete, Selbstorganisierende Systeme“ greift ein Thema aus dem Anwendungsbereich automobiler Steuergeräte bzw. der Avionik auf. Der stetig wachsende Funktionsumfang im Bereich der Fahr- und Flugzeugsteuerung stellt die heutige E/E Architektur vor neue Herausforderungen, da Rechenleistung und Kommunikationsbandbreite begrenzt sind. Die Integration neuer Funktionen in das Steuergerät kann jedoch optimiert werden, wenn zusätzliche Kriterien für die Festlegung der Ausführungszeiten sowie das Funktionsscheduling verwendet werden. Herr Hanti befasst sich mit der Erstellung einer Wissensbasis für diese aktuelle Fragestellung. Dazu wird ein neuer Optimierungsansatz entwickelt. Dieser verwendet einen semi-statischen hierarchischen Multi-Core Scheduler.

Herr Hanti beschäftigt sich in seinem Konzept mit der Erstellung der Wissensbasis für den semi-statischen hierarchischen Multi-Core Scheduler und geht dabei insbesondere auf die vorherrschenden Schedulingalgorithmen und die Konfiguration von Steuergeräten in den Domänen Automotive und Avionik ein. Dabei legt Herr Hanti seinen Fokus der Arbeit auf die Eigenschaften von Multi-Core Prozessoren, die bei Erstellung der Wissenbasis mit einbezogen werden können. So kann zur Laufzeit die verfügbare Rechenzeit maximal ausgenutzt werden.

Das Konzept der Wissensbasis wird abschließend mit dem vorherrschenden statischen Ansatz dieser Domänen verglichen. Anhand verschiedener Funktionen, wie z.B. dem Regensensor, der Geschwindigkeitsregelanlage sowie der Einparkhilfe wird der wissensbasierte Optimierungsansatz evaluiert.

Ich freue mich, Herrn Hanti für die Veröffentlichung seiner Arbeit in dieser wissenschaftlichen Schriftenreihe gewonnen zu haben und wünsche allen Lesern viel Freude und Nutzen bei der Lektüre.



**Prof. Dr. Wolfram Hardt**
Professur Technische Informatik

TECHNISCHE UNIVERSITÄT
CHEMNITZ

# Knowledgebase basiertes Scheduling für hierarchisch asynchrone Multi-Core Scheduler im Systembereich Automotive und Avionik

Dissertation
zur Erlangung des akademischen Grades

Dr. -Ing.

M.Eng. B.Eng. Thomas Hanti
geboren am 01. September 1987 in Ingolstadt

Fakultät für Informatik
an der Technischen Universität Chemnitz

| | |
|---|---|
| Gutachter: | Prof. Dr. rer. nat. Wolfram Hardt |
| | Prof. Dr.-Ing. Andreas Frey |
| Tag der Verteidigung: | 12.12.2018 |

# Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Doktorarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herrn Prof. Dr. Andreas Frey von der Technischen Hochschule Ingolstadt und Herrn Prof. Dr. Wolfram Hardt von der Technischen Universität Chemnitz, die meine Doktorarbeit betreut und begutachtet haben. Für die stets hilfreichen Anregungen, die konstruktive Kritik bei der Erstellung dieser Arbeit, das entgegengebrachte Vertrauen und die Rahmenbedingungen möchte ich mich herzlich bedanken.

Ebenfalls möchte ich mich bei meinen Kollegen Michael Ernst und Tobias Meier bedanken, die mir mit viel Geduld, Interesse und Hilfsbereitschaft zur Seite standen. Bedanken möchte ich mich für die zahlreichen interessanten Debatten und Ideen, die maßgeblich dazu beigetragen haben, dass diese Doktorarbeit in dieser Form vorliegt.

Ebenso möchte ich mich bei allen Kollegen und auch ehemaligen Mitarbeitern der technischen Hochschule Ingolstadt und der Firma Airbus für die gute Zusammenarbeit bedanken.

Abschließend möchte ich mich bei meinen Eltern Claudia und Klaus Hanti bedanken, die mich bei der Erstellung dieser Dissertation zu jeder Zeit unterstützt haben und insbesondere für das Gegenlesen und Korrigieren dieser Doktorarbeit.

# Zusammenfassung

In Automobilen Elektrik /Elektronik (E/E) Systemen sowie in der Flugzeugavionik gibt es eine Vielzahl an Funktionen, die den Fahrzeugführer/Piloten bei seinen Aufgaben unterstützen können. Die Anzahl und Verbreitung der Funktionen nahm in den letzten Jahren sehr stark zu und ein Ende dieses Trends ist nicht in Sicht. Neue Technologien, wie komplett autonomes Fahren bei Fahrzeugen sowie eine stetige Erhöhung des Autonomie Levels von Unmanned Aerial Vehicles (UAVs) in der Avionik bringen das heutige E/E Konzept an die Grenzen des Vertretbaren.
Das klassische, statisch konfigurierte E/E Konzept steht somit vor einer neuen Herausforderung, nämlich eine Vielzahl von neuen, zusätzlichen Funktion zu integrieren und dabei die gleiche Funktionalität, Determinismus und Zuverlässigkeit an den Tag zu legen, wie in der Vergangenheit. Mit dem klassischen statischen Konzept ist diese Anforderung nur zu gewährleisten, wenn für jede neue Funktion ein eigenes Steuergerät in das System Automobil/Flugzeug integriert wird. Da dieses Konzept aber hohe Kosten und erweiterten Bauraum nach sich zieht, ist es nicht mehr vertretbar und es wird nach neuen Optimierungsansätzen gesucht. Ein Optimierungsansatz ist der Übergang vom statischen zum semistatischen Scheduling. Dieser Ansatz wird im Hierarchical Asynchronous Multi-Core System (HAMS) mit der statisch generierten Knowledgebase (KB) für Multi-Core Steuergeräte beschrieben. Ziel der Forschungsarbeit ist es, ein Konzept für eine Knowledgebase in HAMS zu entwickeln, welches es erlaubt, das System zur Laufzeit semi-statisch zu rekonfigurieren, ohne dabei den Determinismus von statischen Steuergeräten aus den Domänen Automotive und Avionik zu verletzten und somit mehr Funktionen auf einem Steuergerät zu allokieren als es aktuell möglich ist.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Algorithmusverzeichnis

# 1 Einleitung

In unserem Alltag hat sich der Gebrauch von Fahrzeugen und Flugzeugen etabliert. Diese stetigen Begleiter erleichtern uns den alltäglichen Weg zur Arbeit oder ermöglichen es uns, schnell an einen anderen Ort der Welt zu fliegen. Dabei hat sich das Automobil sowie das Flugzeug seit seiner Erfindung stetig weiterentwickelt.
Automobile der 80er und frühen 90er Jahre sind sehr stark mechanisch geprägt und folglich lag der Schwerpunkt bei der Automobilentwicklung in der Mechanik. So wurden Systeme wie Fensterheber durch Kurbeln realisiert, eine Geschwindigkeitsregelanlage duch mechanisches feststellen des Gaspedals, Kurvenfahrlicht durch mechanisches Schwenken der Scheinwerfer [43] und das Gas-Luftgemisch des Motors mit Hilfe eines Vergasers erzeugt. Mit der Zeit wurden diese mechanischen Systeme durch elektronische Bauelemente ersetzt oder unterstützt, um unter anderem den immer strengeren Anforderungen der Abgasnormen gerecht zu werden [24] oder um den Insassen einen möglichst hohen Sicherheitsschutz [108] oder Komfort zu bieten. So gehört es heute zum Standard, das Gas-Luftgemisch durch eine elektronisch geregelte Einspritzung im Verbrennungsraum zu erzeugen und die Scheiben lassen sich elektronisch öffnen oder schließen. Strengere gesetzliche Vorschriften (Abgasnorm Euro 6) [24] und Innovationen, wie das vollautomatisierte Fahren, treiben die Entwicklung im elektronischen Bereich voran [37] [10].
Auch das Flugzeug unterliegt dem Wandel von rein mechanischen Systemen hin zum elektrifizierten und durch Computer gesteuerten System. Wurden in der Vergangenheit die Steuerflächen noch über Seilzüge, ausgehend von der reinen Muskelkraft der Piloten, gesteuert [16], so bewegen heute elektronische Aktuatoren die Klappen via Fly-By-Wire [66] [16]. Neue elektrische Systeme zur Pilotenführung, zum Beispiel Wegfindung an Flughäfen und halb-automatischer Landeanflug, sowie Infotainment Systeme im Passagierbereich ermöglichten den Weg von Entwicklungen in der Mechanik hin zu elektronischer Avionik [45] [63].

Neue Flugzeugarten wie Unmanned Aerial Vehicles (UAVs) mit Remotesteuerung oder der zukünftigen Fähigkeit, komplett autonom zu agieren, zeigen die Möglichkeiten der Elektrik im Flugzeug auf [70].
Dieser Paradigmenwechsel von mechanischen Systemen hin zu elektrischen Systemen führte zu der notwendigen Entwicklung einer Elektrik /Elektronik (E/E) Architektur [100]. Die E/E Architektur umfasst alle elektronischen Systeme, wie Steuergeräte (ECU), Sensoren und Aktuatoren in einem Fahr-/Flugzeug. Doch die steigende Umwandlung der mechanischen Systeme in elektrische Systeme und deren ständige Weiterentwicklung führt dazu, dass die Anzahl an ECUs in Fahr- und Flugzeugen stetig zunimmt. Die sich daraus ergebende Frage ist, wie passte sich die E/E Architektur an diesen Paradigmenwechsel im Laufe der Zeit an? Ersichtlich aus der steigenden Anzahl an ECUs ist, dass [132] [100]

- die Anzahl der Funktionen schneller steigt, als die Integrationsmöglichkeit
- die Steuergeräte untereinander mit Bussystemen verbunden sind
- die Anzahl der Sensoren und Aktuatoren zunimmt
- bewährte elektrische Hardware zum Einsatz kommt

Da Funktionen zunehmen und die Anzahl an ECUs aus Platzgründen nicht stetig wachsen kann, muss ein Wandel in der aktuellen E/E Architektur stattfinden[10]. Dieser Wandel muss dazu führen, dass weniger ECUs im Gesamtsystem Fahr-/Flugzeug benötigt werden. Dies kann nur geschehen, indem Funktionen intelligent auf weniger ECUs zusammengeführt werden, aber gleichzeitig keine Einschränkungen in den Funktionalitäten entstehen!
Das Ziel dieser Arbeit ist es folglich, einen Weg aufzuzeigen, der es erlaubt Funktionen intelligent zusammenzulegen ohne funktionale Einschränkungen und damit mehr Funktionen auf einem Steuergerät zu allokieren.

## 1.1 Motivation

Werden Funktionen nicht intelligent zusammengelegt, so stehen die Original Equipment Manufacturers (OEMs) der Automobilbranche vor folgenden künftigen Herausforderungen [137]:

1. **Kosten:** Jede ECU im Fahrzeug kostet Geld in der Beschaffung, Lagerung und Montage. Diese Kosten werden an den Käufer direkt weitergegeben, zum einen durch die Erhöhung des Basispreises und zum anderen durch Erwerb von kostspieliger Zusatzausstattung. Ebenso kann sich durch erhöhte Ausgaben in der ECU-Entwicklung der Gewinn des OEMs schmälern.

2. **Bauraum:** Sollen immer mehr ECUs integriert werden, so muss auch mehr Platz für diese verfügbar sein. Müssen ECUs in der Nähe der von ihnen gesteuerten Aktuatoren sein (Brennraumeinspritzung, Abgasnachbehandlung, etc.), so wird auch dort der Platz knapp, welcher unter Umständen schon an andere ECUs vergeben sein kann. Ebenso ist es möglich, Raum aus dem Passagierbereich für die ECUs zu veranschlagen, was den Fahrkomfort schmälern kann.

3. **Energieverbrauch:** Je mehr Steuergeräte in einem Gesamtsystem vorhanden und operativ sind, desto mehr Energie verbrauchen sie. Da im Fahr-/Flugzeug die Energie durch Verbrennen von Kraftstoff erzeugt wird, nimmt der Kraftstoffverbrauch bei Ottomotoren zu und die Reichweite bei E-Fahrzeugen ab, je mehr Verbraucher benötigt werden.

4. **Gewicht:** Jede ECU, die zusätzlich verbaut wird, erhöht das Gewicht des Gesamtsystems. Insbesondere die Kühlung der Steuergeräte beansprucht einen weitläufigen Teil des ECU-Gewichts. Dies wiederum erhöht das Gesamtgewicht des Fahrzeuges und folglich auch den Energieverbrauch.

Auch die OEMs der Avionik Industrie geraten ohne Zusammenlegung der Funktionen in folgende Herausforderungen [48]:

1. **Kosten:** Neben den Anschaffungskosten einer Avionik ECU, kommt noch die Lagerung der ECUs in Servicecentern der Airlines

auf der ganzen Welt hinzu. Sollte nämlich eine ECU ausfallen, so muss möglichst schnell Ersatz beschafft werden, damit das Flugzeug wieder einsatzbereit ist[1] . Denn der Ausfall eines Flugzeugs ist für eine Airline sehr kostspielig.

2. **Gewicht:** Die ECUs in Flugzeugen sind um ein Vielfaches größer und schwerer, da sie in viel widrigeren Umständen funktionieren müssen als ECUs der Automobilbranche. Hierbei spielt der erweiterte Temperaturbereich sowie die Eigenschaft, mit Höhenstrahlung umgehen zu können, in die Gewichtseigenschaft der ECU hinein. Auch die Anforderungen für doppelte und dreifache Redundanz von ECUs erhöhen das Gesamtgewicht der Avionik. Dies ist insofern problematisch, da in der Avionik das Gewicht eine noch höhere Abhängigkeit zum Energieverbrauch darstellt als im Automotive-Bereich.

3. **Energieverbrauch:** Der Energieverbrauch der ECUs ist auch hier abhängig von der Anzahl der ECUs. Je mehr ECUs mitgeführt werden müssen, umso mehr Energie verbrauchen sie und umso mehr Energie wird benötigt zur Durchführung eines Fluges.

Diese Herausforderungen können durch eine intelligente Zusammenlegung von Funktionen reduziert werden. Ein Beispiel aus dem alltäglichen Gebrauch in den Zieldomänen Automotive und Avionik soll die Logik hinter der Zusammenlegung verdeutlichen.

## Komplementäre Funktionen der Zieldomänen

Moderne Autos sind mit vielen Assistenzsystemen und deren Umsetzung in Softwarefunktionen ausgestattet. Entweder sind diese schon in der Basiskonfiguration vorhanden oder müssen als „Zusatzausstattung“ gekauft werden. In der Basiskonfiguration von Mittelklassefahrzeugen ist überwiegend eine Geschwindigkeitsregelanlage (GRA) vorhanden und eine Einparkhilfe kann optional hinzugefügt werden. Wird bei der Fahrzeugbestellung die zusätzliche Option Einparkhilfe gewählt, so werden bei der Montage des Fahrzeugs die entsprechenden Sensoren und

[1] Dies gilt ebenso im Automobilbereich für OEMs und Service Dienstleister

zusätzlichen Steuergeräte in das Fahrzeug verbaut. Bei der Benutzung der beiden Funktionen ist es aber von vornherein ausgeschlossen, dass beide Funktionen zur gleichen Zeit laufen. Sollte die Geschwindigkeitsregelanlage aktiv sein, so ist es unrealistisch, dass das Fahrzeug in den nächsten Sekunden parken wird. Ersichtlich ist das aus der Bedienung der Geschwindigkeitsregelanlage, welche sich aus verkehrsrechtlichen und sicherheitstechnischen Gründen erst ab einer Geschwindigkeit von $> 30\frac{km}{h}$ einschalten lässt (BMW als Beispiel [18]). Konträr verhält sich die Funktion Einparkhilfe. Die Einparkhilfe ist nur unter einer Geschwindigkeit von $< 30\frac{km}{h}$ aktiv und scannt erst dann seine Umgebung nach passenden Parklücken ab (BMW [19]). Wurde eine passende Parklücke gefunden, so muss die Funktion nochmals aktiviert werden, d.h. das Einfahren in die Parklücke muss bestätigt werden und die Funktion schaltet vom Zustand „Scannen“ in „Ausführen“. Aufgrund der Eigenschaften der beiden Funktionen Einparkhilfe und Geschwindigkeitsregelanlage ist es also ausgeschlossen, dass beide Funktionen zusammen im vollen Umfang rechnen. So wäre es sinnvoll, nicht zwei ECUs zu verwenden, sondern nur eine, die sich je nach Geschwindigkeit auf die aktive Funktion dynamisch anpasst bzw. rekonfiguriert.
Ein weiteres aktuelles Beispiel von sich gegenseitig ausschließenden Funktionen ist das Zusammenspiel von Spurhalte- und Abbiegeassistent. Der Spurhalteassistent ist, abhängig vom OEM, u.U. ausschließlich auf einer Autobahn aktivierbar, aufgrund der dort vorherrschenden Fahrbahnmarkierungen [9]. Dies geschieht unter anderem durch Überwachung der Geschwindigkeit und unter Einbindung der aktuellen GPS-Koordinaten, fusioniert mit den Kartendaten des GPS-Systems [56]. Der Abbiegeassistent hingegen wird nur in der Stadt oder auf ländlichen Bundesstraßen benötigt und nicht auf der Autobahn [21]. Hier unterstützt er den Fahrer beim Linksabbiegen in einem Geschwindigkeitsbereich von $> 10\frac{km}{h}$ und nur sofern der Blinker „Links“ gesetzt ist. Aufgrund der unterschiedlichen Geschwindigkeiten sowie Orte, bei der die beiden Assistenzfunktionen aktiv sind, laufen diese Funktionen niemals in vollem Umfang gleichzeitig, d.h. sie schließen sich gegenseitig aus.
Weitere Beispiele aus dem Bereich der Assistenzfunktionen können nicht nur von der Geschwindigkeit abhängen, sondern auch von der Tageszeit, dem Ort und dem Zustand des Fahrzeuges. So gibt es Assistenzfunktionen, die nur in der Nacht aktiv sind, wie zum Beispiel der

Nachtsichtassistent, welcher Fußgänger und Tiere in der Nacht durch eine Infrarotkamera erkennt und die Fernlichtautomatik, welche das Fernlicht teilweise automatisch abblendet bei Gegenverkehr [27]. Eine konträre Funktion am Tag wäre hier die Fußgängerverfolgung mit automatischem Abbremsen, wobei derzeit zur Erkennung von Fußgängern gute Lichtverhältnisse herrschen müssen.
Ebenso ist der Zustand eines Fahrzeuges wichtig, so werden zum Beispiel in der Werkstatt die Assistenzfunktionen nicht benötigt, sondern es müssen umfangreiche Analysen über Fahrzeugzustand und Fehlerbehebungen durchgeführt werden. Assistenzfunktionen werden somit in der Werkstatt nicht benötigt.
Sich gegenseitig ausschließende Funktionen im Automobil gibt es zwar schon heute, aber der Ausblick auf zukünftige Assistenzfunktionen zeigt diese Unzulänglichkeit noch deutlicher auf:
Ein Pionier in Fahrerassistenzfunktionen ist der Automobilhersteller Tesla Motors. Insbesondere sind hier die Autopilotfunktionen der Tesla Modelle „Model S“ und „Model X“ zu nennen [9] [56]. Hat ein Tesla-Kunde diese Option gewählt, so lässt sich die Autopilotenfunktion auf Straßen mit deutlich erkennbarer Fahrbahnmarkierung aktivieren und das Auto regelt autonom die Geschwindigkeit und hält zugleich die Spur. Diese Voraussetzung ist meistens auf Autobahnen/Highways der Fall. Auf Landstraßen oder im Stadtverkehr sind Fahrbahnmarkierungen nicht immer vorhanden und somit ist es dem Autopiloten nicht mehr möglich, das Fahrzeug in der Spur zu halten. Aktuell wird die Autopilotenfunktion limitiert auf die entsprechenden Straßen.
Laut Aussage von Tesla CEO Elon Musk ist es mit einigen Versionen des Autopiloten nicht möglich, am Stadtverkehr teilzunehmen oder sogar abzubiegen [40]. Die hierfür notwendige Technologie müsse erst entwickelt werden und so ist zurzeit beispielsweise eine Fußgängerüberwachung in Tesla Modellen nicht möglich. Aus diesem Beispiel und der Aussage von Elon Musk ist zu erkennen, dass es mehrere Autopilotenfunktionen in einem Auto geben wird und dass diese sehr situationsbezogen sind, d.h. sich gegenseitig ausschließen [27] [8].
Auch im Bereich der Avionik gibt es sich gegenseitig ausschließende Funktionen. Im Flugzeug sind Funktionen sehr stark von der aktiven Flugphase abhängig, d.h. es gibt Funktionen, die nur beim Start, Landen oder während des Reisefluges „Cruise“ aktiv sind. Ein Vertreter dieser

Funktionen ist zum Beispiel die Stellplatzplatzführung auf dem Rollfeld von Flugplätzen. Damit die Piloten auch bei schlechten Sichtbedingungen oder unübersichtlichen Rollwegen ihren zugewiesenen Stellplatz auf dem Rollfeld finden, sind hierfür spezielle Navigationssysteme im Flugzeug eingebaut (Airbus). Diese Funktion ist allein während des Rollvorgangs am Boden aktiv, während des Reisefluges wird diese nicht benötigt. Ebenso ist die Funktion „Brake to Vacate“ nur während des Landevorgangs aktiv [50]. Diese im Airbus A380 optionale Assistenzfunktion berechnet aufgrund der Wetterdaten, Flugzeugtyps, Aufsetzpunkt und Flughafendaten die beste Kombination aus Bremskraft und Umkehrschub für die Landung [59]. Ebenso gibt es neuere, vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickelte Landeassistenzfunktionen, die einen geräuscharmen Landeanflug ermöglichen.
Viel deutlicher wird die Abhängigkeit von Funktionen der Avionik an die aktuelle Flugphase bei UAVs. Hier geht die zukünftige Entwicklung dahin, dass immer mehr Aufgaben, welche zuvor ein Operator am Boden übernommen hat, in die Luft an Bord des Flugsystems transferiert werden. Damit wird die Autonomie des Flugsystems gesteigert und es kann gleichzeitig eine Entlastung des Operators (Pilot) erreicht werden. Ein Beispiel aus der aktuellen Forschung dieser Flugzeuggattung ist der Sagitta Erprobungsträger von der Firma Airbus Defence & Space GmbH. Dieser zeigt die aktuellen Entwicklungsmöglichkeiten bei der UAV Avionik Entwicklung auf. In diesem Erprobungsträger gibt es Funktionen, die nicht nur von der Flugphase, sondern auch von der Mission abhängen. So sind einige Funktionen ausschließlich aktiv, wenn in der Mission zum Beispiel ein Auto verfolgt oder ein Gebiet überwacht wird. Je nach Mission sind Algorithmen zur Erkennung und Verfolgung von Automobilen aktiv oder Algorithmen zur Personenerkennung und Tracking. Diese Funktionen rechnen nur dann, wenn das UAV im Zielgebiet angelangt ist. Auf dem Weg zum Zielgebiet sind Routingfunktionen aktiv, welche den optimalen Weg zum Einsatzgebiet berechnen.
Anhand dieser Beispiele kann man gut erkennen, dass auch in der Avionik immer mehr Funktionen in das Flugzeug integriert werden, die ausschließlich auf spezielle Flug- / Missionsphasen ausgelegt sind.

## Aktuelle Situation der Zieldomänen

Die Branchen der Zieldomänen haben die Vorteile einer Zusammenlegung von Funktionen auf Steuergeräten erkannt und schon erste Schritte eingeleitet. Die Umsetzung der Zusammenlegung in den Zieldomänen Automotive und Avionik ist aber sehr unterschiedlich.
Im Automotive-Bereich hat die Innovation durch Software im Auto die Anzahl der Steuergeräte nach oben getrieben. Jede Funktion besaß ihr eigenes Steuergerät „one function per ECU“ und war über ein Bussystem mit anderen Steuergeräten verbunden [10] [103]. Die Allokation von Funktion auf ein bestimmtes Steuergerät, wurde statisch zur Entwicklung definiert und änderte sich nicht. Doch mit dem stetigen Anstieg und steigender Komplexität der Funktionen und durch die steigende Variantenvielfalt eines Kraftfahrzeugs ist erkennbar, dass nicht jeder neuen Funktion ein eigenes Steuergerät zugewiesen werden kann. Daraufhin wurde eine Zusammenlegung der Funktionen forciert und eine systemweite E/E Architektur geplant [46].
Um Funktionen auf einem Steuergerät im Automotive-Umfeld zusammenzuführen, müssen die einzelnen Funktionen auf deren Laufzeit- und Hardwareanforderungen geprüft werden. Bei der Laufzeit muss sichergestellt werden, dass die Funktion binnen eines gewissen Zeitpunktes oder Zeitraumes die Berechnung beendet hat, sog. „weiche“ oder „softe“ Echtzeit. Ebenfalls muss die ECU die erforderlichen Schnittstellen, zum Beispiel Controller Area Networks (CANs) Bus, zur Verfügung stellen. Stellt sich bei einer Vorabanalyse heraus, dass bei einer Zusammenlegung die Ausführung der Funktionen nicht beeinträchtigt wird, so werden diese Funktionen auf einem Steuergerät allokiert.
In der Fahrerassistenzumgebung kann dieser Ansatz erweitert werden, bis alle Funktionen auf einem Steuergerät untergebracht sind, sog. Hochintegration [73]. Auch diese Allokation ist statisch und die Funktionen laufen nur auf diesem zusammengeführten System. Sollte bei der Vorabanalyse ein Fehler auftreten, zum Beispiel, dass die Leistung des Prozessors nicht ausreicht, so kann ein leistungsstärkerer Prozessor in das Steuergerät integriert werden, um dennoch die Funktionen zu allokieren. Wenn sich in der Vorabanalyse herausstellt, dass eine Funktion lediglich dann aktiv werden soll, wenn sie vom Kunden für das Fahrzeug bestellt wird, kann auch in diesem Fall eine Zusammenlegung der Funktionen

günstiger sein als die separate Implementierung von zwei Steuergeräten [103]. Neueste Prozessor Hardware, Multi-Core Prozessoren, erlauben eine weitaus umfangreichere Zusammenlegung als Single-Core Prozessoren. So können Funktionen auf Cores aufgeteilt werden und so noch mehr Funktionen auf einem Steuergerät allokiert werden. Auch eine Trennung von Funktionen ist in einem Multi-Core System möglich, so dass Funktionen auch dann auf einem Steuergerät allokiert werden können, wenn sich diese normalerweise gegenseitig beeinflussen. [73] [54] [103]
Im Avionikbereich wurden die Vorteile einer Zusammenlegung von Softwarefunktionen schon früher erkannt und aktuell im weitreichenden Projekt der Integrated Modular Avionics (IMA) erforscht. Ein Beispiel aus der Integrated Modular Avionics (IMA) Entwicklung [48] von Boeings 787 Dreamliner hat durch eine Zusammenlegung ca. 2000lb an Gewicht eingespart. Beim A380 soll die Anzahl der Steuergeräte durch Zusammenlegung um die Hälfte reduziert worden sein [48]. So ist es modernen Flugzeugen wie dem Airbus A380, A350 und der Boeing 787 unter anderem erst möglich, so spritsparend zu fliegen, im Vergleich zu deren Vorgängern. Analog zum Automotive-Bereich werden in der Avionik Funktion statisch zur System Designzeit auf Steuergerät allokiert, unabhängig ob es sich um ein Steuergerät mit IMA Funktionalität handelt oder nicht. So sind die Funktionen fest allokiert und alle Funktionen rechnen, ob diese nun benötigt werden oder nicht.

### Weiterentwicklung

Die Erfolge dieses ersten Stadiums der Zusammenlegung sowie der Ausblick auf die stetig steigende Anzahl von Funktionen sowie die Verfügbarkeit neuer Multi-Core Hardware führen dazu, dass über weitere Schritte in der Zusammenlegung von Funktionen auf Steuergeräten nachgedacht werden muss. Wurden im ersten Stadium nur Funktionen auf ihre gegenseitige softwaretechnische Beeinflussung und die Schnittstellen Hardware zu Funktion hin geprüft, ist der nächste logische Schritt, Funktionen auf Situationsabhängigkeit hin zu überprüfen. Dabei soll geprüft werden, ob sich die Funktionen bei ihrer Benutzung so verhalten, dass diese sich gegenseitig ausschließen. Dieser gegenseitige Ausschluss lässt sich realisieren, indem eine Funktion unterschiedliche

„Phasen“ zugewiesen bekommt. Auf diese Phasen kann sich unter anderem die aktuelle Fahrsituation abbilden lassen. Das in der Einleitung ausgeführte Beispiel aus den aktuellen Fahrerassistenzsystemen, wie das Zusammenspiel Einparkhilfe und Geschwindigkeitsregelanlage, beleuchten den Begriff „Phasen“ noch weiter. Bewegt sich das Fahrzeug mit einer aktuellen Geschwindigkeit von $> 30\frac{km}{h}$, so ist die Geschwindigkeitsregelanlage verfügbar. Sollte die Geschwindigkeit $<= 30\frac{km}{h}$ sein, so ist die Einparkhilfe verfügbar. In diesem Beispiel beschreibt die Geschwindigkeit die aktuelle Fahrsituation und regelt die Verfügbarkeit der Funktionen Geschwindigkeitsregelanlage und Einparkhilfe und ist somit gleichzeitig Namensgeber für die Phase „$30\frac{km}{h}$“. Bewegt sich das Fahrzeug nun mit einer Geschwindigkeit von $> 30\frac{km}{h}$, ist es dem Fahrer erlaubt, die Geschwindigkeitsregelanlage zu aktivieren, damit diese die Geschwindigkeit des Fahrzeuges regelt, oder im Standby-Modus zu belassen. Sollte der Fahrer in dieser Situation aber den Schalter der Einparkhilfe betätigen, so darf diese nicht aktiviert werden und muss dem Fahrer eine Verweigerung der Aktivierung anzeigen. Das Beispiel verhält sich konträr zu einer Geschwindigkeit $< 30\frac{km}{h}$. Für den Bediener des Fahrzeuges ist somit die Benutzung und die Verfügbarkeit der Funktionen klar geregelt. Nicht ersichtlich ist für den Fahrer die Realisierung der Phasen und deren Wechsel „Phasenübergang“ im Steuergerät. Für diesen stellt der Phasenübergang somit keinerlei Einschränkungen in der Funktionalität dar.

Im Steuergerät werden für jeden Zustand der Funktionen unterschiedliche Rechenzeiten reserviert. Das bedeutet, sobald die Geschwindigkeitsregelanlage verfügbar ist ($> 30\frac{km}{h}$), wird der Einparkhilfe ein Teil ihrer Rechenzeit entzogen und der Geschwindigkeitsregelanlage zur Regelung der Geschwindigkeit verfügbar gemacht. Die Einparkhilfe ist aber nie ganz ausgeschaltet, d.h. ihr steht minimale Rechenzeit zur Verfügung, so dass im Falle einer Benutzereingabe eine Rückmeldung an den Fahrer gegeben werden kann.

Um eine dynamische Verlagerung der erlaubten Rechenzeiten für Funktionen während des Betriebes durchführen zu können, muss eine Verlagerung der Rechenzeit von Funktionen unterstützt werden (sog. dynamische Rekonfiguration). In den derzeitigen Systemen ist eine dynamische

Rekonfiguration aber nicht vorgesehen, denn die Funktionen werden statisch zur Designzeit fest auf die Steuergeräte allokiert.
Zur Umsetzung einer dynamischen Rekonfiguration in den Bereichen Automotive und Avionik muss der aktuelle statische Ansatz in den Echtzeitbetriebssystemen einem dynamischen Ansatz weichen. Mit diesem dynamischen Ansatz muss es dem System und den Entwicklern ermöglicht werden, das System geordnet und gezielt zu rekonfigurieren. Nur mit diesen Mitteln lassen sich die Vorteile intelligenter Zusammenlegung durch das Ausnützen von Phasen realisieren, ohne Einbußen an Funktionalität oder Verletzung der Randbedingungen der Zieldomänen. Ein System, das einen geordneten Rekonfigurationsansatz in einer Echtzeitumgebung bietet, ist Hierarchical Asynchronous Multi-Core System (HAMS). Das HAMS System in seiner aktuellen Umsetzung [131] basiert auf dem Linux Betriebssystem mit der Real-Time Kernel Erweiterung. Das HAMS System besteht aus dem Hierarchical Asynchronous Multi-Core System (HAMS) Scheduler und der HAMS Knowledgebase für Multi-Core Steuergeräte. Dieser Verbund hat folgende Vorteile:

- Die Daten der HAMS Knowledgebase werden vom HAMS Scheduler ausgelesen, welche Phasen, Rechenzeitanforderungen und Platz fest vorschreiben

- Eine Rekonfiguration zwischen Phasen erfolgt immer nach einem festen Schema unter den Vorgaben der HAMS Knowledgebase

- Der HAMS Scheduler überwacht die Funktionen und deren Ausführung, basierend auf den gelesenen Daten der HAMS Knowledgebase

- Der HAMS Scheduler kann anhand der Daten der HAMS Knowledgebase mittels Rekonfiguration auf Fehler reagieren

Um mit dem HAMS Scheduler geordnet und gezielt rekonfigurieren zu können, muss der HAMS Scheduler die Daten aus der HAMS Knowledgebase(KB) Datei einmalig auslesen. Der Verbund aus HAMS Scheduler und HAMS KB ist das HAMS System. Aus diesem Verbund ist die HAMS Knowledgebase die wichtigste Einheit, um die Herausforderungen aus den Zieldomänen angehen zu können.

## 1.2 Ziele der Knowledgebase

Um den Übergang zwischen statischen und dem HAMS Systemen in den Zieldomänen Automotive und Avionik zu gewährleisten, muss das Zusammenspiel HAMS Scheduler und HAMS Knowledgebase (KB) näher erläutert werden. Das grundlegende Konzept aus Kapitel 1.1 sieht vor, dass der HAMS Scheduler die exekutive Einheit ist und die Operationen (wie in Abschnitt 1.1) ausführt. Die Informationen für eine gezielte und geordnete Rekonfiguration bzw. welche Funktionen sich ändern und wie, bezieht der HAMS Scheduler aus der KB. Somit ist die KB eine auslesbare Datei im HAMS System. Aus ihr kann entnommen werden, anhand der Phasen aller Funktionen (Systemphase), wie das System sich ändern muss und wie der HAMS Scheduler darauf zu reagieren hat. Somit soll eine gezielte und geordnete Rekonfiguration ermöglicht werden. Damit die HAMS Knowledgebase die Aufgabe der Informationslieferung an den HAMS Scheduler erfüllen kann, müssen die folgende Ziele erreicht werden, um in den Zieldomänen das HAMS System erfolgreich einsetzen zu können. In den folgenden Kapiteln wird folglich das Konzept zur Umsetzung der KB erläutert.

### Ziele bei der Allokierung/Scheduling

In einem statischen Echtzeitsystem der Zieldomänen hat jede Funktion ihre gewissen Eigenschaften und zugewiesenen Attribute, die sich nicht ändern. Diese Attribute sind unter anderem die maximale Ausführungszeit und Periode einer Funktion. Das System wird mittels dieser festen und maximalen Eigenschaften und Attribute in der Designphase geplant und optimal ausgelegt.
Im dynamischen HAMS System jedoch ändern sich diese Eigenschaften und Attribute je nach Phase. So gibt es nicht nur eine einzige optimale Auslegung des Steuergeräts, sondern eine Vielzahl, abhängig von der Anzahl der Phasen. Die optimale Systemkonfiguration zum aktuellen Betriebszeitpunkt ist somit phasenabhängig.
Ziel der HAMS Knowledgebase ist es nun, diese Systemkonfigurationen für jede Phase bereitzustellen und zur Laufzeit vom HAMS Scheduler ausgelesen zu werden. Um dies zu erreichen, muss zur Designzeit eine optimale und funktionstüchtige Konfiguration anhand von verschiedens-

ten Berechnungen (Algorithmen) für jede Phase ermittelt werden, in Abhängigkeit vom Schedulingalgorithmus (sog. HAMS Knowledgebase Erstellung). Wird dieses Ziel von der HAMS Knowledgebase erfüllt, ist es mindestens möglich, die gleiche Funktionalität wie im statischen System herzustellen. Die Einberechnung der Phasen ermöglicht es darüber hinaus, noch mehr Funktionen auf einem Steuergerät mitnehmen zu können. Wie viele Funktionen auf einem Steuergerät allokiert werden können, ist aber abhängig von den Funktionen selbst und deren Phasen.

## Ziele bei der Rekonfiguration

In einem statischen Echtzeitsystem wird keine Rekonfiguration durchgeführt, sondern die Eigenschaften und Attribute der Funktionen unverändert beibehalten.
In einem geordneten und gezielten HAMS System muss ebenso der Weg von der aktuellen Konfiguration zur zukünftigen Konfiguration betrachtet werden, um zu überprüfen, ob die Rekonfiguration durchführbar ist. So ist es ein Ziel der HAMS Knowledgebase, auf Basis des Rekonfigurationschemas des HAMS Schedulers, die Durchführbarkeit einer Rekonfiguration schon zur Designzeit zu evaluieren, hinsichtlich der Einhaltung von zeitlichen Grenzen. Wird dieses Ziel von der HAMS Knowledgebase erfüllt, kann zum einen eine Aussage darüber gemacht werden, wie viele und welche Funktionen auf einem Steuergerät mitgenommen werden und zum anderen kann dessen Rekonfiguration für jede Phase vorab evaluiert werden. So geht das System vom statischen Ansatz zum semi-statischen HAMS System über.

## Ziele bei der Hardware

State-of-the-Art Systeme verwenden einen Multi-Core Prozessor mit zwei oder mehr Kernen (Cores). Der Grund für den Wandel von Single zu Multi-Core Prozessoren ist die Steigerung der Anzahl der Funktionen auf einer ECU. Da in eine Multi-Core (Central Processing Unit) - Prozessor (CPU) mindestens zwei Funktionen gleichzeitig (parallel) ausgeführt werden können ohne sich gegenseitig zu behindern, können folglich mehr Funktionen als auf einem Single-Core Prozessor mitgenommen werden. Im Bereich der Avionik sind Single-Core Prozessoren im

sicherheitskritischen Bereich noch weit verbreitet. Im Missionsbereich hält auch hier der Multi-Core Prozessor Einzug, denn nur dieser kann die Rechenleistung der gehobenen Funktionen von z.B. UAVs bereitstellen. Ziel der HAMS Knowledgebase ist es, Multi-Core Prozessoren zu unterstützen. Die Multi-Core Eigenschaften und Besonderheiten müssen sich in den Algorithmen zu Berechnungen und im Vorgang zur Erstellung der HAMS Knowledgebase widerspiegeln und mit einbezogen werden, sofern sie eine Auswirkung auf das Scheduling und die Allokierung von Funktionen haben.

## Ziele bei der Fehlerfallbehandlung

Systeme, unabhängig davon ob sie statisch oder semi-statisch konfigurierbar sind, werden mit einer bestimmten Wahrscheinlichkeit Fehlerfälle hervorrufen. In den vorherrschenden statischen Systemen ist die Möglichkeit zur Fehlerbehandlung begrenzt. Da diese sich nicht auf die neuen Gegebenheiten, hervorgerufen durch einen Fehler im System, anpassen können. Sollten Fehler auftreten, indem sich Eigenschaften und Attribute von Funktionen nicht gemäß den statischen Gegebenheiten verhalten, ist das System gefährdet. Dies kann sich unter anderem darin bemerkbar machen, dass Funktionen zu wenig Ausführungszeit bekommen und keine Ergebnisse liefern können.
Ziel der HAMS Knowledgebase ist es, dass aus ihr vom HAMS Scheduler Informationen über einen korrekten Systemablauf entnommen werden können. Der HAMS Scheduler kann daraufhin, ebenfalls mit Vorgaben zur Beseitigung des Fehlerzustandes in der HAMS Knowledgebase, korrekt auf diesen Fehler reagieren und die Situation auflösen.

## Einschränkungen durch die Zieldomäne

Die Anwendung des HAMS Schedulers in Verbindung mit der HAMS Knowledgebase lässt sich nicht nur auf die Zieldomänen Automotive und Avionik beschränken, sondern kann auch in anderen Bereichen mit Echtzeitsystemen verwendet werden. Allerdings wären die im Nachhinein vorgestellte Knowledgebase Berechnung und Erstellung bei weitem nicht ausreichend, um alle Formen von Echtzeitsystemen abzudecken. Hier sind insbesondere hoch-sicherheitskritische Systeme zu nennen. Die

HAMS Knowledgebase beschränkt sich nur auf die Zieldomänen Avionik und Automotive und deren nicht sicherheitskritischen Bereich. D.h. die Knowledgebase wird nicht in hart echtzeitfähigen und sicherheitskritischen Gebieten der Zieldomänen eingesetzt.
Im Bereich der Avionik beschränkt sich somit das Einsatzgebiet auf die Steuergeräte der Missionsavionik und im Automotive Bereich auf das Einsatzgebiet „Soft Echtzeitrelevant“ und Systeme mit gemischten Funktionen. Unter diesen Einsatzgebieten finden sich auch die vorgestellten Assistenzsysteme wieder.
Außerdem schließt die Erstellung der Knowledgebase folgende Themen aus:

1. Es wird Asynchronität und keine Nebenläufigkeit betrachtet. D.h. es ist nicht möglich, zum Beispiel eine Funktion A zwingend vor, während oder nach einer Funktion B laufen zu lassen. In einem asynchronen Multi-Core System ist die Synchronisation nicht immer gegeben.

2. Es gibt keine Programmabhängigkeiten wie Semaphore oder Mutexe in den Funktionen. Damit Funktionen sich synchronisieren können oder um „Race-Conditions“ zu einer gemeinsamen Ressource auszuschließen, wird in der Multi-Core Programmierung von diesen Mechanismen Gebrauch gemacht.
   Da die Knowledgebase aber in einem asynchronen Echtzeit System verwendet wird, wären solche Mechanismen kontraproduktiv. Eine Ressource muss daher von mehreren Funktion gemeinsam benutzt werden können, zum Beispiel Message Queues oder ist exklusiv für eine Funktion reserviert.

3. Auch eine gemeinsame direkte Benutzung einer Schnittstellen von unterschiedliche Funktionen ist ausgeschlossen, da der Zugriff zu diesen durch Semaphore geschützt werden müsste. Jede Funktion muss daher seine eigene Schnittstelle zum Gesamtsystem haben.

4. Der Aufstartprozess des Steuergerätes wird nicht betrachtet.

## 1.3 Struktur der Arbeit

In diesem Kapitel wurden die Zieldomänen und deren ECUs vorgestellt sowie der sinnvolle Einsatz der HAMS Knowledgebase, um die Herausforderungen aus den Zieldomänen zu überwinden. In Kapitel 2 wird diese Arbeit in den Kontext Echtzeitscheduling eingeordnet. Dabei wird zuerst ein allgemeiner Überblick über erforderliche Themen zum Verständnis des Echtzeitscheduling gegeben sowie eine Wissensgrundlage für das weitere Verständnis dieser Arbeit geschaffen. Die Zieldomänen Avionik und Automotive werden dort anschließend aufgeführt und deren Stand der Technik im Zusammenhang mit Echtzeitscheduling und Multi-Core Echtzeitscheduling erläutert. Die wichtigsten Punkte daraus werden abschließend in Verbindung zu der HAMS Knowledgebase gebracht.
In Kapitel 3 wird der Stand der Forschung genauer beleuchtet. So werden aktuelle Systeme, die sich mit dem Thema Scheduling, semi-statisch, Multi-Core Prozessoren, Multi-Core und Avionik sowie Multi-Core und Automotive beschäftigen, vorgestellt. Die Ziele, Fortschritte und unter Umständen Ergebnisse werden erläutert. Der letzte Abschnitt dieses Kapitels widmet sich dem HAMS System, dessen Aufbau, Organisation und Rekonfiguration.
Ab Kapitel 4 beginnt der Hauptteil dieser Arbeit. Mit dem Wissen aus den vorhergehenden Kapiteln und durch die Analyse von kommerziell verfügbaren Echtzeitsystemen werden Basisanforderungen an die HAMS Knowledgebase definiert. Anhand dieser Analyse wird aufgezeigt, welche Punkte im Echtzeitscheduling und Systemverhalten für die Zieldomänen von Bedeutung sind und ohne deren Erfüllung ein phasenorientiertes System nicht anwendbar sein würde. Daraus werden die Eingangsparameter, die für Berechnungen der HAMS Knowledgebase nötig sind, abgeleitet (Kapitel 5).
Diese werden mit Hilfe von Modellen erklärt, die sich aus den Anforderungen und den Grundlagen des Echtzeitschedulings ergeben. Die Eingangsparameter und Anforderungen bilden die Grundlage für das nächste Kapitel, die HAMS Knowledgebase Berechnung und Erstellung (Kapitel 6). Hier werden zuerst die erforderlichen Berechnungen (Algorithmen) vorgestellt, um einen gültigen phasenorientierten Schedule zu erhalten und in einer Auslastungsberechnungen festgehalten. Bei diesen Auslastungsberechnungen werden die Algorithmen an das pha-

senorientiere Scheduling angepasst, wobei besonders Wert gelegt wird auf die logische Gruppierung von Funktionen im Zusammenhang mit Multi-Core Scheduling. Ebenso werden Rekonfigurationsberechnungen vorgestellt, durch welche vorab überprüft wird, ob ein System zur Laufzeit rekonfiguriert werden kann. Ebenso wird einer Fehlerfallbehandlung ermittelt. Als letzter Abschnitt dieses Kapitels wird die Struktur der HAMS Knowledgebase und deren auslesbare Datei aus den Berechnungen vorgestellt, die vom HAMS Scheduler benötigt wird, um das System zu betreiben, zu rekonfigurieren und zu überwachen.
Im letzten Kapitel des Hauptteils (Kapitel 7) wird die Knowledgebase Berechnung und der Output für den HAMS Scheduler gegen die Anforderungen evaluiert. Ebenso wird ein Vergleich mit heutigen statischen Systemen angestrebt, um die Vorteile und Einschränkungen der HAMS Knowledgebase hervorzuheben. Den Abschluss dieser Arbeit bildet eine Zusammenfassung der Ergebnisse und einen Ausblick auf zukünftige Weiterentwicklungen des HAMS Systems.

## 1.4 Zusammenfassung

In den aktuellen soft echtzeitfähigen Steuergeräten der Bereiche Avionik und Automotive werden unterschiedlichste Funktionen auf einem Steuergerät implementiert. Diese reichen im Automotive-Umfeld von einfachen Assistenzfunktionen, wie der Geschwindigkeitsregelanlage, bis hin zu hoch komplexen Regelfunktion, wie der Einparkhilfe. In Zukunft wird die Anzahl an Funktionen im Fahrzeug weiter steigen, da immer mehr Assistenzsysteme angeboten werden. Auch in der Avionik ist eine steigende Anzahl von Assistenzsystemen zu verzeichnen. Je mehr Assistenzsysteme in ein Fahrzeug/Flugzeug integriert werden müssen, umso mehr Kosten (Beschaffungskosten) wird für diese benötigt. Gleichzeitig steigt auch der Bauraumbedarf für ECUs und der Energieverbrauch. Um dies zu vermeiden, müssen verschiedene Funktionen auf einem Steuergerät implementiert werden (Hochintegration). Der aktuell verwendete statische Implementierungsansatz zur Systemzusammenlegung ist ein erster Schritt und lässt weitere Verbesserungen zu. So ist der nächste logische Schritt bei der Systemzusammenlegung, das statische System in ein semi-statisches, rekonfigurierbares System zu überführen. Ein

System, welches diesen Vorgang für die Domänen Automotive und Avionik unterstützt, ist das HAMS System. Das HAMS System besteht aus einem HAMS Scheduler, der delegiert, welche Funktionen wann auf welchem Core in einem Multi-Core System laufen. Das Wissen wann, wie lange und wo welche Funktionen laufen müssen, entnimmt der HAMS Scheduler aus der HAMS Knowledgebase. Diese zur Systemdesignzeit generierte Wissensbasis wird in das HAMS Steuergerät geladen und erlaubt es somit, das System gezielt zu rekonfigurieren.
In dieser Arbeit wird die Berechnung und Erstellung der HAMS Knowledgebase genauer betrachtet. D.h. es werden Modelle für Funktionen und Hardware anhand von Anforderungen erstellt, welche als Input für Auslastungs- und Verteilungsberechnungen genützt werden. Die Ergebnisse dieser Berechnungen dienen der Erstellung einer für den HAMS Scheduler auslesbaren Datei, welche diesem zur Verfügung gestellt. Den Abschluss dieser Arbeit bilden die Ergebnisse und ein Ausblick.

# 2 Echtzeit Multi-Core Scheduling in Zieldomänen

Damit Steuergeräte ihren Zweck, wie z.B. die Geschwindigkeitsregelanlage, ausführen können, werden auf ihnen Programme/Funktionen implementiert. Diese Programme/Funktionen haben verschiedenste Aufgaben, wie Daten empfangen, verarbeiten und weiterleiten (Eingabe Verarbeitung Ausgabe (EVA) Prinzip). In den Zieldomänen werden Programme/Funktionen „embedded Tasks“ oder nur „Tasks“ genannt, welche zum Systemstart gestartet werden und erst mit dem Herunterfahren des Systems beendet werden. Damit alle Tasks auf einem Steuergerät korrekt ablaufen und ihre Funktionalität korrekt erfüllen können, wird dem Bereich Echtzeitscheduling und Multi-Core Scheduling eine große Bedeutung zugewiesen. Für ein korrektes Echtzeitscheduling müssen zuerst die Tasks analysiert und ihre Laufzeitparameter in Taskmodellen abgebildet werden. Mit Hilfe dieser Taskmodelle wird der anzuwendende Schedulingalgorithmus bestimmt. Zur korrekten Verteilung von Tasks in einem Multi-Core System kommen Verteilungsalgorithmen zum Einsatz. Diese Verteilungsalgorithmen beziehen in ihren Berechnungen auch die Multi-Core Eigenschaften mit ein, insbesondere wie dieses Multi-Core Systeme aufgebaut ist. Diese Eigenschaften und Modelle werden im folgenden Kapitel beschrieben und zum Schluss werden zwei Multi-Core Steuergeräte vorgestellt, die sich aktuell im Einsatz bei den Zieldomänen wiederfinden.

## 2.1 Echtzeitscheduling in der Zieldomäne

Um ein Echtzeitscheduling in den Zieldomänen zu realisieren, müssen zuerst die Tasks und deren Verhalten in einem Taskmodell definiert werden. Dazu zählt, wie oft der Task periodisch aufgerufen wird und

welche Priorität der Task hat. Auf Basis dieses Taskmodells und der dem Steuergerät zugewiesenen Tasks wird dann der passende Schedulingalgorithmus gewählt. Dieser wertet den aktuellen Zustand aller ihm zugewiesenen Tasks aus und entscheidet, wer die Recheneinheit der CPU gewinnt und für wie lange.

### 2.1.1 Taskmodell der Zieldomänen

#### Basis-Taskmodell

Das Basis-Taskmodell unterscheidet sich je nach zeitlichem Verlauf des Tasks. So gibt es periodische, aperiodische und sporadische Tasks [11].

**Periodische Tasks** Diese Tasks müssen jedes Zeitintervall **P** (Periode) erneut rechnen. So muss z.B. ein Spurverlassensassistent alle 250 ms die Straßeninformationen auslesen und bewerten, um bei der Überschreitung einer Spur ohne gesetzten Blinker den Fahrer zu warnen (Abbildung 4.2). Im Avionik-Bereich sind die Zeiträume ebenfalls im Millisekunden-Bereich und können in [1] für die einzelnen Systeme entnommen werden.

**Aperiodische Tasks** Diese Tasks aktivieren unvorhersehbar und haben keinen periodischen Ablauf. Als Beispiel dienen hier Benutzereingaben vom Fahrer, wie das An- und Ausschalten einer Geschwindigkeitsregelanlage. Die Benutzereingabe ist hier gleichzusetzen mit einem Event [11] [143].

**Sporadische Tasks** Auch sporadische Tasks aktivieren unvorhersehbar, aber die Zeit zwischen zwei Aufrufen ist begrenzt. Auch hier sind Events die ausschlaggebende Einheit für einen Aufruf des Tasks [11] [143].

Unabhängig vom zeitlichen Verlauf des Tasks, benötigt dieser Ausführungszeit auf der CPU, um seine Aufgaben erfüllen zu können. Ist es dem Task nicht erlaubt den Prozessor eine bestimmte Zeit zu beanspruchen oder wird er nicht rechtzeitig aufgerufen, kann er nicht ordnungsgemäß seine Aufgabe erfüllen und es kommt zu einem Fehlverhalten des Tasks bzw. Funktionen. Damit dies nicht passiert, ist die Rechenzeit in drei Parameter unterteilt. Der Erste ist die Ausführungszeit Worst Case

Execution Time (WCET), **C**. Die WCET gibt an, wie lange ein Task rechnen muss, um seine Funktion zu erfüllen. In den Zieldomänen sind das die Aufgaben: Daten zu bekommen, zu verarbeiten, Algorithmen anzuwenden und das Ergebnis an die erforderlichen Teilnehmer zu senden. Aufgrund von prozessorspezifischen Eigenschaften, Frequency-Stepping oder Branch Prediction sowie IF-Statements im Code selbst kann es zu Variationen in der Rechenzeit kommen, welche ein Task benötigt [134]. Da ein Task aber immer bis zum Ende rechnen muss und nicht vorher unterbrochen werden darf, muss für die Allokation von Tasks auf Steuergeräten immer die längste Rechenzeit (WCET) zur Allokation genommen werden. Hat der Task seine Berechnungen durchgeführt, suspendiert er sich selbst und setzt seinen nächsten periodischen Aufruf, solange es sich um einen periodischen Task handelt. Der zweite Parameter in Bezug auf Rechenzeit ist somit die Periode, **P**. Die Periode bestimmt, wie oft ein Task innerhalb einer gewissen Zeitspanne aufgerufen werden muss. So muss z.B. ein Air Data Computer alle 50ms den Zustand der Sensoren abfragen umso z.B. die neue Höhe zu berechnen [1]. Wird die Periode verlängert, so kann das u.U. dazu führen, dass der Pilot nicht rechtzeitig die neuesten Informationen auf seinem Display angezeigt bekommt. Im Falle eines sporadischen Tasks wird die Periode durch den „minimaler Abstand zweier Task Aufrufe“ **A** ersetzt. Ein aperiodischer Task hat per Definition weder eine Periode noch einen minimalen Abstand. Der dritte wichtige Parameter ist die Frist (Deadline) eines Tasks, **D**. Innerhalb dieser Deadline muss ein Task zu Ende gerechnet haben. Meistens wird die Deadline gleich der Periode gesetzt, was aber nicht immer der Fall sein muss [99]. Aus diesem Basis-Taskmodell ergeben sich folgende Tupel für einen periodischen Task [11] [143]:

$$T_{\text{perodisch}} = C_{\text{i}}, P_{\text{i}}, D_{\text{i}} \tag{2.1}$$

Für einen sporadischen Task ergibt sich folgender Tupel [11] [143]:

$$T_{\text{sporadisch}} = C_{\text{i}}, A_{\text{i}}, D_{\text{i}} \tag{2.2}$$

#### Erweitertes Taskmodell

Mit der Weiterentwicklung von Hardware, Anwendungen und Scheduling von periodischen, sporadischen und aperiodischen Tasks musste auch das Basis-Taskmodell erweitert werden [11] [143]:

**Rechenzeit Taskmodell**

Zu den Basisrechenzeitparametern eines periodischen Taskmodells können noch weitere Parameter den Task genauer beschreiben.

- Die Bereitzeit des Tasks (ready time) mit Bezug zum Systemstart **r**.
- Eine Antwortzeit **R**, in welcher der Task den Prozessor gewonnen haben muss, um seine Deadline einzuhalten.
- Einen Spielraum **l** (laxity), welcher eine zeitliche Differenz bis zum vollständigen Beenden einer Periode angibt. Dieser Spielraum wird berechnet aus l= D-r-C

**Ein/Ausgabegeräte**

Zur Kommunikation mit anderen Steuergeräten, Benutzereingaben oder mit Sensoren muss der Task mit Ein/Ausgabegeräten verbunden werden. Ein Beispiel dafür wäre der CAN Bus, mit dem Daten von Sensoren abgefragt und Befehle gesendet werden können. Ohne dieses Bussystem wäre es z.B. der Geschwindigkeitsregelanlage nicht möglich, die Fahrzeuggeschwindigkeit von den Raddrehzahlsensoren zu empfangen. Da ein Task u.U. nicht nur ein, sondern mehrere Ein/Ausgabegeräte benötigt, hat ein Task ein Tupel von Ein/Ausgabegeräten in seinem Taskmodell, E/A.

Mit diesen Parametern erweitert sich die Tupel-Darstellung eines periodischen Task wie folgt:

$$T_{\text{perodisch}} = C_{\text{i}}, P_{\text{i}}, D_{\text{i}}, r_{\text{i}}, R_{\text{i}}, E/A_{\text{i}} \tag{2.3}$$

Ein sporadischer Task wird wie folgt in einem Tupel dargestellt.

$$T_{\text{sporadisch}} = C_{\text{i}}, A_{\text{i}}, D_{\text{i}}, r_{\text{i}}, R_{\text{i}}, E/A_{\text{i}} \tag{2.4}$$

### 2.1.2 Schedulingalgorithmen für Echtzeitsysteme

Mit dem Aufstellen eines Taskmodells für einen bestimmten Task kann der notwendige Schedulingalgorithmus bestimmt werden. Hierbei ist zu unterscheiden, ob ein Task periodisch ausgeführt werden muss oder ob eine sporadische Ausführung genügt. Weiterhin ist die Klasse des Echtzeitsystems zu betrachten, welche sich in folgende drei Klassifizierungen unterteilt [11]:

**Harte Echtzeit**

In harten Echtzeitsystemen muss die Deadline eines Tasks unbedingt eingehalten werden. Sollte das nicht möglich sein, so zeigt das System ein Fehlverhalten. Harte Echtzeit wird in sicherheitskritischen Systemen verwendet, wie z.B. dem Airbag Steuergerät oder Flugsteuerung [11].

**Feste Echtzeit**

Im Gegensatz zur harten Echtzeit wird bei festen Echtzeitsystemen dem System kein Fehlverhalten zugeordnet, sobald eine Deadline verletzt ist. Hier wird das Ergebnis der Berechnung verworfen, sobald ein Task eine längere Rechenzeit benötigt als geplant. Grundsätzlich sollte es aber dennoch möglich sein, die Deadlines einzuhalten [11].

**Weiche Echtzeit**

Bei der weichen Echtzeit wird das Ergebnis der Berechnung weder verworfen noch legt das System ein Fehlverhalten bei Verletzung der Deadline vor. Das Einhalten der Deadlines ist somit nicht wichtig für die korrekte Funktionsweise des Systems. Per Definition ist die Einhaltung der Deadlines „bestrebenswert“ [11].

### Cyclisch

Der cyclische Schedulingalgorithmus zeichnet sich dadurch aus, dass die Ausführung der Tasks zeitlich gesteuert ist [143]. So werden Tasks Zeitscheiben (Slots) zugewiesen, in denen sie die CPU gewinnen und ihre Berechnungen durchführen können. Solange der Task in seiner Ausführung ist, wird dieser nicht unterbrochen, bis sein Slot beendet ist. Sobald der Slot aufgebraucht ist, wird der Task von der CPU entfernt und muss auf seinen nächsten Slot warten. Diese Art der Ausführung entspricht der eines harten Echtzeit Systems. Sollte der Task mit seinen Berechnungen früher fertig sein als geplant, so wird dieser Slot nicht für andere Tasks benutzt, sondern die CPU wird in den Warte / IDLE Modus versetzt. Diese Art der Ausführung ist besonders geeignet für periodische und harte Echtzeit anfordernde Tasks.
Um zu ermitteln, ob zu viele Tasks auf einem Steuergerät allokiert werden, gibt es sogenannte „Feasibility Tests“ (Auslastungstest). Im

Falle eines cyclischen Schedulingalgorithmus ist der Feasibility Test folgende Formel 2.5 [143]:

$$t_{\text{cycle}} \leq \sum_{i=0}^{n} C \tag{2.5}$$

In dieser Formel wird im Cyclus eines Tasks (Periode) geprüft, ob die Summe aller Ausführungszeiten in diesem Zyklus kleiner oder gleich der Cycluszeit ist. Dabei muss zuerst eine Sortierung vorgenommen werden, bei der die Tasks, beginnend mit der kleinsten Periode, zuerst zugewiesen werden in sog. Minor Cycle. Der Task mit der größten Periode bestimmt den Major Cycle und wird als letztes zugewiesen. Anhand dieses Feasibility Tests ist zu erkennen, dass das System mit bis zu 100% Prozessorlast ausgelastet werden kann. In einem Multi-Core System müssen die Tasks nicht nur einem Slot, sondern auch einem Core zugewiesen werden. Jeder Core wird im cyclischen Scheduling einzeln betrachtet und kann so theoretisch zu 100% ausgelastet werden.
Ein Vorteil dieses Systems ist, dass im Falle einer Überlast, d.h. ein Task benötigt mehr Rechenzeit als geplant, nur dieser Task von einer Fehlfunktion betroffen ist, alle anderen Tasks bleiben aus Sicht des Schedulers unbetroffen. Ein Nachteil dieses Algorithmus ist, dass sporadische oder aperiodische Tasks nicht unterstützt werden. Sollte das System solche Tasks dennoch unterstützen, so müssen diese in periodische Tasks umgewandelt und ein fester Slot für diese eingeplant werden. Ein weiterer Nachteil ist die Anfälligkeit des Systemtimers im System (Kapitel 2.2). Sollte dieser nicht rechtzeitig auslösen oder falsch eingestellt werden, so hat das komplette System eine Fehlfunktion.

### RMS

Ein weiterer Schedulingalgorithmus ist der auf Prioritäten basierende Rate Monotonic Scheduling (RMS) Algorithmus. Dieser Algorithmus ist für Systeme mit unterbrechbaren, periodischen und sporadischen Tasks vorgesehen, was einem festen Echtzeitsystem entspricht. In einem bestimmten Zeitabstand wird der laufende Task unterbrochen und die Prioritäten aller bereiten Tasks geprüft. Die Auswahl, welcher Task die CPU gewinnt, wird in diesem Algorithmus nicht von der Zeit bestimmt, sondern von Prioritäten, welche einem Task vor Beginn zugewiesen

werden. Höchste Priorität hat der Task mit der geringsten Periode. Ausschlaggebend für die genaue Berechnung der Priorität ist die Auslastung (U) der Formel 2.6. Der Task mit der geringsten Auslastung U bekommt die höchste Priorität [11] [143].

$$U_{\mathrm{i}} = \frac{C_{\mathrm{i}}}{P_{\mathrm{i}}} \tag{2.6}$$

Während der Laufzeit ändert sich die Priorität eines Tasks nicht und wird somit statisch bei der Designzeit zugewiesen. Der RMS Algorithmus ist ein verdrängender (preemptiver) Algorithmus, d.h. in periodischen Zeitabschnitten mit mindestens der Frequenz des höchstprioren Tasks, wird der laufende Task unterbrochen, um zu überprüfen, ob ein höher priorer Task rechenbereit ist. Ist das der Fall, wird immer der höchst priore Task die CPU gewinnen, auch wenn dies bedeutet, dass ein nieder priorer Task seine Deadline nicht einhalten kann. Damit so ein Fall nicht eintritt, gibt es auch für diesen Algorithmus Feasibility Tests. Der Bekannteste ist von Liu und Layland [89] und beschreibt eine sichere Obergrenze in der Gesamtsystemauslastung, abhängig von der Anzahl **n** an Tasks, Periode **P** und WCET **C**, die auf dem System laufen in Formel 2.7. Diese Formel ist ausschließlich gültig sobald $P_i = D_i$ erfüllt ist.

$$U_{\mathrm{i}} = \sum_{i=0}^{n} \frac{C_{\mathrm{i}}}{P_{\mathrm{i}}} \leq n(\sqrt[n]{2} - 1) \tag{2.7}$$

Anhand dieser Formel ist zu erkennen, dass die maximale Auslastung bei einer steigenden Anzahl von Tasks gegen die Grenze $ln2 \approx 0.693$ geht, was einer Auslastung von 69,3% entspricht. Das würde bedeuten, dass ein System nicht zu 100% ausgelastet werden könnte und die restlichen 30,7% im Idle Modus verbringt. Da die von Liu und Layland eingeführte Formel 2.7 eine sichere Grenze darstellt, gibt es dennoch Systeme, die eine höhere Auslastung haben und ebenfalls keine Deadlines verletzt werden [82] [81]. Diese Tatsache und die Anforderung, ein System immer so weit wie möglich auszulasten, gaben den Anstoß für eine Reihe von weiteren Feasibility Tests. Ein Beispiel dafür ist der T-Bound Test [110], berechnet mit der Formel 2.8.

$$U_{\mathrm{i}} = \sum_{i=0}^{m} -1\frac{p_{\mathrm{i+1}}}{P_{\mathrm{i}}} + 2\frac{P_{\max}}{P_{\min}} - m \tag{2.8}$$

Diese Formel gilt für den Fall, dass ein Set von Tasks und deren gesamten Perioden ein Verhältnis $\leq 2$ haben. Für einen anderen Fall gilt der aufwendigere R-Bound Test mit der Formel 2.9 [110].

$$U_{\mathrm{i}} = \sum_{i=0}^{m} -1 \frac{p_{\mathrm{i}+1}}{P_{\mathrm{i}}} + \frac{2}{\frac{P_{\max}}{P_{\min}}} - m \tag{2.9}$$

Auch gibt es Feasibility Tests, die auf Analysen beruhen, wie die „Time Demand Analysis“ [93] und die „Harmonic Analysis“ [82]. Bei Letzterer wird geprüft, ob die Perioden aller Tasks auf dem System harmonisch sind, d.h. ob sie gemeinsame Vielfache voneinander sind. Ist dies der Fall, lässt sich ein anderer Feasibility Test einsetzen [82]. Diese Formeln sind anwendbar bei Single-Core Prozessoren. Bei Multi-Core Prozessoren muss darauf geachtet werden, wie dort das Core Scheduling behandelt wird. Hier wird unterschieden zwischen globalem und partitioniertem Scheduling [117][114][87]. Der Unterschied ist dabei die Runqueue, in welcher alle lauffähigen Tasks gespeichert werden. Bei einer globalen Runqueue gibt es nur eine Runqueue für alle Prozessoren, aus welcher sich die Cores die lauffähigen Tasks herausnehmen können. Bei partitioniertem Scheduling gibt es pro Core eine Runqueue, also so viele Runqueues im System wie es Cores gibt. Dies hat den Vorteil, dass die Single-Core Auslastungstests angewendet werden können. Bei einem globalen Scheduling verändert sich die Auslastungsberechnung von Liu und Layland gemäß der Formel 2.10 [118].

$$U \leq \frac{m^2}{3m - 2} \tag{2.10}$$

Um sporadische oder aperiodische Tasks in einem RMS Scheduling einzuplanen werden sog. Scheduling Server verwendet. Ein Beispiel ist der Polling Server (PS). Für den PS wird im RMS Schedule ein weiterer periodischer Task eingeplant, der als „Platzhalter“ für aperiodische und sporadische Tasks dient. Anhand der vorhandenen Tasks kann die Auslastung und der Abstand zur Feasibility-Grenze berechnet und so ein passender PS eingerichtet werden. Sobald der PS im System geschedult werden kann, können die sporadischen oder aperiodischen Tasks anhand ihrer Ankunftszeit, Deadline etc. der Reihe nach den Prozessor gewinnen.

Der Feasibility Test hierfür ist in Formel 2.11 abgebildet [11].

$$\sum_{i=0}^{n} \frac{C_{\mathrm{i}}}{P_{\mathrm{i}}} + \frac{C_{\mathrm{ps}}}{P_{\mathrm{ps}}} \leq (n+1)(2^{(1/(\mathrm{n}+1))\text{-}1}) \tag{2.11}$$

Je nach Bedarf an sporadischen und aperiodischen Tasks können mehr oder weniger Polling Server eingeplant werden.
Ein Vorteil des RMS Scheduling Algorithmus ist, dass alle Arten von Tasks periodisch, sporadisch und aperiodische unterstützt und auch mit einem Feasibility Test überprüft werden können. So können jedem Task Prioritäten zugeordnet werden. Ein weiterer Vorteil ist die Handhabung einer Überlast. Sollte dieser Fall eintreten, so ist sichergestellt, dass der Task mit der höchsten Priorität immer die CPU gewinnt.

### EDF

Der Earliest Deadline First (EDF) Scheduling Algorithmus ist ein zeitenbasierter und ebenfalls auf Prioritäten ausgelegter Algorithmus für feste und weiche Echtzeitsysteme. Dieser Algorithmus unterbricht periodisch den laufenden Task und untersucht die Deadlines aller bereiten Tasks. Der Task mit der kürzesten Deadline gewinnt den Prozessor und hat somit aktuell die höchste Priorität im System. Auch für diesen Algorithmus gibt es Feasibility Tests von Liu und Layland 2.12 mit der Bedingung $P_i = D_i$ [89][11].

$$U_{\mathrm{n}} = \sum_{i=0}^{N} \frac{C_{\mathrm{i}}}{P_{\mathrm{i}}} \leq 1 \tag{2.12}$$

Die Auslastung eines EDF geschedulten Systems liegt somit maximal bei 100% und übertrifft damit Systeme mit einem RMS Scheduler. In einem Multi-Core System muss weiter unterschieden werden zwischen einem global- und partitionsgeschedulten System.
Bei diesem Scheduling Algorithmus gibt es aber zwei Eigenschaften, welche zu beachten sind. Erstens, die zeitliche Granularität. Die Auflösung der Deadlines hängt direkt ab von der Auflösung der Systemtimer. So können die Deadlines nur ein Vielfaches dieses Timers betragen. Zweitens, kann im Falle einer Überlast nicht vorhergesagt werden, welcher Task den Prozessor gewinnt [143] [105].

Um aperiodische Tasks in EDF zu schedulen, gibt es den Total Bandwidth Server (TBS). Der TBS ist dafür zuständig, dass einem aperiodischen Task eine möglichst kurze Deadline zugewiesen wird, damit dieser sobald wie möglich geschedult werden kann. Dabei muss aber beachtet werden, dass die Systemgesamtauslastung innerhalb der Grenzen von 100% bleibt. Die nicht durch die periodischen Tasks festgelegte Auslastung aus der Formel 2.12 wird dem TBS zugewiesen und darf 100% nicht übersteigen $U_{tbs} = 1 - U_n$. Mit Hilfe der Formel 2.13 wird dem aperiodischen Task während der Laufzeit eine Deadline zugewiesen. Die WCET muss vorher bekannt sein, um diese korrekt berechnen zu können [11].

$$D_i = max(r, d) + \frac{C_i}{U_{tbs}} \tag{2.13}$$

Auch für EDF wurden erweiterte Scheduling Tests entwickelt, für den Fall, dass Tasks mit den Vorrausetzungen $P_i > D_i$ allokiert werden müssen. So hat sich Devi et. al. dieses bei EDF auftretende Problem untersucht. Mit dem „Devis Test“ in [14] hat Devi et. al. einen erweiterten Auslastungstest vorgestellt. Für diesen Test wurde die Formel 2.14 erstellt, in welcher die Tasks in aufsteigender Reihenfolge geordnet werden müssen.

$$\forall(1 \leq k \leq n)|(\sum_{i=1}^{k} \frac{C_i}{P_i} + \frac{1}{D_k} \cdot \sum_{i=1}^{k} (\frac{P_i - min(P_i D_i)}{P_i}) \cdot C_i \leq 1 \tag{2.14}$$

Unter diesem Test gibt es Tasksets, die laut Formel nicht bestehen, aber dennoch schedulbar sind [80]. Burah et al. haben in [68] [67] einen exakten Auslastungstest für EDF aufgestellt, der eindeutig aussagt, ob ein Taskset schedulbar ist oder nicht. Dieser Test ist der sog. „Demand Bound Function“ DBF(I) Test, welcher einen gewissen Zeitraum (Intervall) des Schedules betrachtet. Zum Test (Formel 2.15) werden alle Tasks untersucht, deren Deadlines innerhalb des Intervalls I liegen.

$$\begin{aligned} DBF(I) &= \sum_{i=1}^{n} \lfloor \frac{i - D_i}{P_i} + 1 \rfloor \cdot C_i \\ DBF(I) &\leq 1 \end{aligned} \tag{2.15}$$

Um mit der Formel 2.15 eine genaue Aussage über die Schedulbarkeit eines Tasksets zu bekommen, muss das Intervall passend gewählt werden.

Ggf. müssen mehrere Intervalle getestet werden, um eine ausreichend genaue Aussage über die Schedulbarkeit zu bekommen. Dieses Prinzip der Superposition wurde in [79] [123] vorgestellt und benötigt, abhängig von den Intervallen, eine nicht polynomielle Rechenzeit.

### 2.1.3 Verteilungsalgorithmen für Multi-Core System

In einem Multi-Core System müssen Tasks auf die verschiedenen Cores verteilt werden. Um eine korrekte Verteilung der Tasks zu gewährleisten, wurde diese Aufgabe in Form eines Behälterproblems (Bin Packing) formuliert.

#### Bin Packing Algorithmen

Ein Behälterproblem zeichnet sich dadurch aus, dass es eine begrenzte Anzahl $n$ von Behältern gibt. Diese Behälter haben eine gewisse Kapazität $c$ und ein Gewicht von $j$. Um einen Behälter richtig zu befüllen, muss sichergestellt werden, dass das Gewicht niemals die Kapazität überschreitet, wie in Formel 2.16 [5]:

$$c \leq \sum_{i=1}^{n} j_{\mathrm{i}} \tag{2.16}$$

Um die Behälter z.B. mit Tasks zu befüllen, werden verschiedene Vorgehensweisen / Algorithmen herangezogen. Diese unterscheiden sich in der Zuweisung einer Bin, der Überprüfung eines Behälters und der Aussage, wann ein Behälter als geschlossen markiert wird. Im Folgenden sind die erfolgreichsten Bin Packing Algorithmen aufgeführt, deren Kapazität die gesamte zur Verfügung stehende Rechenzeit ist und die Tasks als Gewicht ihre WCET mit sich bringen [97]:

- First Fit:
  Bei diesem Algorithmus wird jeder Task in den ersten Behälter gepackt, in den er hineinpasst. D.h. Der Algorithmus fängt bei dem ersten Behälter an und prüft das Gewicht des Tasks gegenüber der noch verfügbaren Kapazität. Wenn die Bedingung 2.16 erfüllt ist, wird der Task auf dem Behälter allokiert. Ansonsten wird der nächste Behälter zur Überprüfung herangezogen. Eine Unterart

dieses Algorithmus ist der „First Fit Decreasing“ Algorithmus. Hier wird zuerst das Set an zu allokierende Tasks gemäß ihres Gewichtes absteigend geordnet und dann erst dem Behälter zugewiesen.

- Next Fit:
  Jeder ankommende Task wird in den gerade aktiven Behälter gepackt, solange bis die Bedingung 2.16 nicht mehr erfüllt ist. Sobald das der Fall ist, wird der Behälter geschlossen und der nächste Behälter wird für die Allokation von Tasks geöffnet. Ist der Behälter geschlossen, wird er nicht mehr geöffnet.

- Best Fit:
  Bei diesem Algorithmus wird bei der Allokierung eines Tasks jeder Behälter dahingehend geprüft, ob der Task zugewiesen werden kann oder nicht. Der Task wird dann auf jenen Behälter allokiert, der nach der Allokierung am wenigsten Kapazität übrig hat. Behälter werden nicht geschlossen, sondern immer mit überprüft. So wird sichergestellt, dass möglichst viele Tasks einem Behälter zugewiesen werden.

- Worst Fit:
  Im Gegensatz zum Best Fit steht der Worst Fit Algorithmus. Der Task wird hier dem Behälter zugewiesen, der nach der Allokierung noch am meisten Kapazität zur Verfügung hat. Auch hier werden die Behälter nicht geschlossen.

- Almost Worst Fit:
  Dieser Algorithmus ist gleich zum Worst Fit Algorithmus, mit der Ausnahme, dass hier der Task in den Behälter gepackt wird, der nach der Allokierung noch die zweitmeiste Kapazität zur Verfügung hat.

- Any Fit:
  Der Any Fit Algorithmus benützt zur Zuweisung die Algorithmen First, Next und Worst Fit. Aber die Behälter werden erst dann geschlossen, wenn keine Tasks mehr hinzugefügt werden können. D.h. sollte das noch zur Allokation ausstehende Taskset auf diesen Behälter nicht mehr zugewiesen werden können, dann wird der Behälter geschlossen.

Die Vielfalt dieser Bin Packing Algorithmen ist dadurch begründet, dass je nach Algorithmus die Zeit und die Güte der Zuweisung unterschiedlich sind. So ist die Zuweisung ein NP-schweres Problem und die benötigte Zeit steigt mit der Anzahl der zugewiesenen Tasks.
In Tabelle 2.1 sind die einzelnen Algorithmen und deren Berechnungszeit, soweit bekannt, und deren Güte R für den schlechtesten Fall (Worst Case Performance) gegenübergestellt. Dabei wird die Güte R ermittelt, indem ein Set aus Tasks mit einer bekannten optimalen Verteilung nochmals mit dem Bin Packing Algorithmus aufgeteilt wird. Der Grad der Abweichung von einem vorher bekannten optimalen Ergebnis ist die Güte R [5].

| Algorithmus | Zeit | $R_a$ |
|---|---|---|
| First Fit | $\theta(n \log n)$ | 1.7 |
| Next Fit | $\theta(n \log n)$ | 2 |
| Best Fit | $\theta(n \log n)$ | 1.7 |
| Worst Fit | $\theta(n \log n)$ | 2 |
| Almost Worst Fit | $\theta(n \log n)$ | 1.7 |
| Any Fit | NA | 1.583 |

Tabelle 2.1: Bin Packing Dauer und Güte [5]

### Brute-Force

Mit Bin Packing Algorithmen ist es nicht immer möglich, eine optimale Lösung zu finden (siehe Güte R) oder überhaupt eine Lösung zu bekommen. Daher wird ein Algorithmus benötigt, der sofern es eine Lösung gibt, diese findet, generiert und auch optimal nutzt. Diese optimale Lösung zu finden kann ausschließlich durch einen Suchalgorithmus abgedeckt werden, der alle möglichen Variationen auflistet und auf Machbarkeit überprüft, wie der sog. Brute-Force Algorithmus. Der Brute-Force Algorithmus stellt zuerst jede mögliche Anordnung und Verteilung an Tasks auf die Bins auf. Weiterhin überprüft der Algorithmus, ob die Gewichte der einzelnen Tasks die Gesamtkapazität nicht übersteigen. Ist dies bei einer Lösung der Fall, wird diese verworfen. Solange aus dem Algorithmus eine gültige Lösung resultiert, war der Al-

gorithmus erfolgreich. Wird der Brute-Force Algorithmus nicht bei dem ersten Finden einer Lösung abgebrochen, so können mehrere Lösungen gefunden werden.

## 2.2 Multi-Core Prozessoren der Zieldomäne

Multi-Core Prozessoren sind heute weit verbreitet, da diese der steigenden Nachfrage nach Performance und geringem Bauraum gerecht werden können. Im Vergleich zu Single-Core Prozessoren ist der Aufbau eines Multi-Core Prozessors komplizierter. So muss es neben zusätzlichen Cores auch eine Möglichkeit geben, die Ein-/Ausgabe Geräte zugänglich zu machen. Ebenso müssen Cores untereinander Daten austauschen können. Multi-Core Prozessoren bedürfen so einer neuen Architektur.

### 2.2.1 Architektur der Multi-Core Prozessoren

Die Architektur eines Multi-Core Prozessors ist maßgeblich von den folgenden drei Eigenschaften abhängig.

1. Die arithmetischen Einheiten, die den Multi-Core Hauptteil der Berechnungen durchführen
2. Die Kommunikation auf dem Multi-Core Prozessor und zwischen weiteren Geräten
3. Die Speicher-Architektur auf und außerhalb des Prozessors

Die arithmetischen Einheiten eines Multi-Core Prozessors lassen sich in drei Gewichtsarten der Mikrocontroller und SOCs unterteilen: leicht-, mittel- und schwergewichtig.
Die leichtgewichtigen arithmetischen Einheiten spezialisieren sich auf arithmetische Operationen und werden nur mit prozessorspezifischem Assembler-Code betrieben. So ist es wichtig, dass der Code auf diese Architektur angepasst wird, um das Maximum an Performance aus diesem Core herauszuholen. Ein Beispiel hierfür ist der „Ambric Bric“ aus Abbildung 2.1.
Der Ambric Bric besteht aus je zwei 32 Bit Reduced Instruction Set Computer (RISC) Cores mit lokalen Random-Access Memory (RAM)

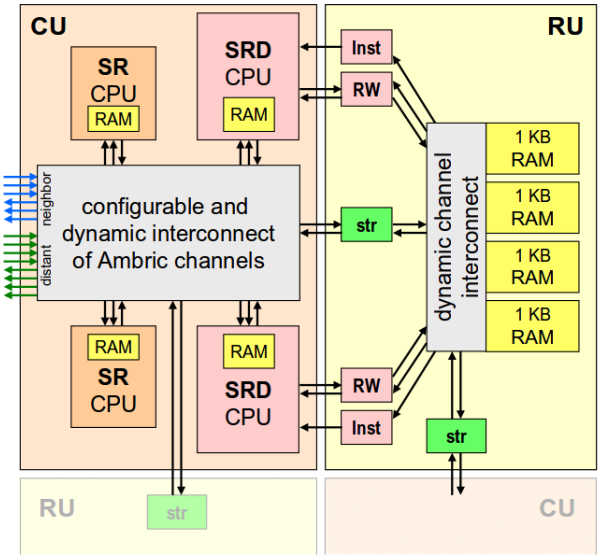



Abbildung 2.1: Ambric Bric [25]

und je zwei 32 Bit RISC Cores mit Digital Signal Processor (DSP) Erweiterung und ebenfalls mit lokalem RAM [25]. Die Verbindungen zwischen den vier Cores und dem Globalen RAM ist durch „Channels“ realisiert. Diese Channels und deren Zuweisung stellen auch die Hauptaufgabe bei der Programmierung dieses Prozessors dar. So werden Tasks einem festen Core zugewiesen und das Routing der Daten (Einstellung der Channels) muss so gewählt werden, dass andere Cores und deren Tasks die Daten übermittelt bekommen. Dieser Prozessor funktioniert ohne Betriebssystem und somit auch ohne Scheduler, da alle Tasks fest zugewiesen werden und die Kommunikation per Design synchronisiert ist [42].
Mittelgewichtige Prozessoren benützen Hochsprachen wie C und sind vielseitiger einsetzbar. Ebenso können Shared Libraries oder Echtzeit Betriebssysteme zum Einsatz kommen. Der Programmierer benötigt aber immer noch ein Verständnis von dem ausführenden Multi-Core

Prozessor und dessen Einschränkungen bzgl. der Allokation von Tasks. Ein Vertreter dieser Klasse ist der Infineon Tri-Core, abgebildet in 2.6. Ein schwergewichtiger Prozessor bedient sich der aktuellen Hochsprachen, wie z.B. C und C++ hat eine mehrstufige Piplene, high speed I/O Anbindungen und auf ihn zugeschnittene Maschinenebefehle zusätzlich zum RISC Befehlssatz. Die Tasks werden nicht mehr zur Designzeit fest einem Core zugewiesen, sondern werden zur Laufzeit verteilt. Diese Prozessoren bedienen sich auch modernen (Real Time Operating System) - Echtzeitbetriebssystem (RTOS), mit deren Möglichkeiten im Scheduling und in der Inter Task Kommunikation [42]. Ein Beispiel für einen schwergewichtigen Prozessor stellt die Intel® Core2™ Microarchitektur dar aus Abbildung 2.2.
In dieser Architektur ist eine 12-stufige Pipeline zu erkennen, die Zentrale Recheneinheit ist in gelb dargestellt (Abbildung 2.2).
Als letztes sind sog. System-on-a-Chips (SOCs) in der Reihe der Multi-Core Prozessoren weit verbreitet. SOCs haben den Vorteil, dass sie neben einer schwergewichtigen Recheneinheit auch noch interne Ein-/Ausgabegeräte mit u.U. PHY verbaut haben. So ist es z.B. üblich, dass auf diesen SOCs eine Grafikeinheit verbaut ist, um ein Graphical User Interface (GUI) wiederzugeben. Auch andere E/A Geräte, wie ein CAN-Bus Zugang und serielle Leitungen sind vorhanden.
So gibt es SOCs, die auf der Intel® Core2™ Microarchitektur basieren und andere wiederum eine ARM Architektur oder eine PowerPC (PPC) Architektur benützen. Da jede Architektur seine Vor- und Nachteile hat, müssen die SOCs auf den jeweiligen Einsatzzweck abgestimmt werden. So wird in mobilen Endgeräten eine ARM Architektur bevorzugt, da sie im Vergleich zu anderen viel Architekturen stromsparender ist [42]. Ein Kandidat für einen SOC ist die Open Multimedia Application Platform (OMAP) Architektur von Texas Instruments (Abbildung 2.3). Hier kann man deutlich erkennen, dass der Prozessor die Verbindungen zum Gesamtsystem wie USB und HDMI mit sich bringt und nicht zusätzliche Geräte verbaut werden müssen.
Eine weitere Besonderheit bei der Architektur von Multi-Core Prozessoren ist der Cache. Dieser ist in mehrere Ebenen (Levels) unterteilt, wie in Abbildung 2.4 zu erkennen ist. Jedes Level hat seine eigenen Besonderheiten und unterscheidet sich hauptsächlich in Schnelligkeit und Vorhersagbarkeit. Der L1 ist am nächsten zum eigentlichen Re-

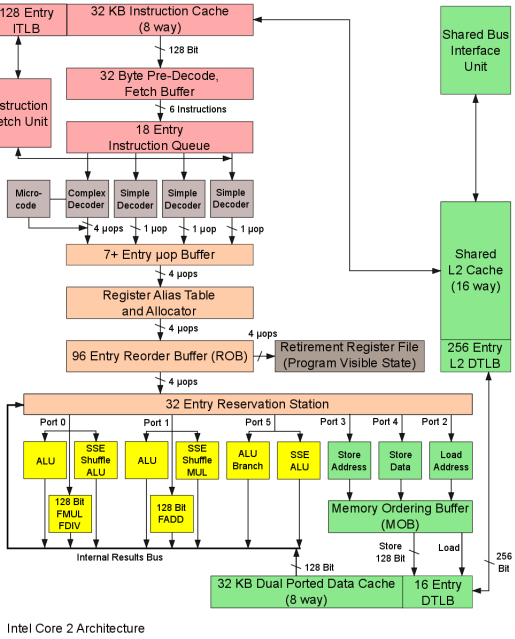



Abbildung 2.2: Intel® Core2™ Microarchitektur [29]

chenkern der CPU und zeichnet sich durch besonders schnelle Zugriffe mit geringer Wartezeit aus. Der L1 Daten Cache Speicher speichert die Daten ab, auf die am häufigsten zugegriffen wird.
Der L2 Cache beinhaltet mehr Speicher, ist langsamer und weiter entfernt von der eigentlichen Recheneinheit. Typischerweise werden L2 und ein möglicher L3 Cache zwischen mehreren Cores zur gemeinsamen Nutzung zur Verfügung gestellt. Der Cache Controller fragt immer erst den L1 Cache ab, ob die erforderlichen Daten zur Verfügung stehen. Ist das nicht der Fall, so wird die nächste Hierarchie, der L2 gefolgt vom L3 Cache, abgefragt.
newpage Stehen die Daten dort zur Verfügung, so wird ein „Cache Hit“ gemeldet, andernfalls ein „Cache Miss“. Sollten spätestens im L3 Cache

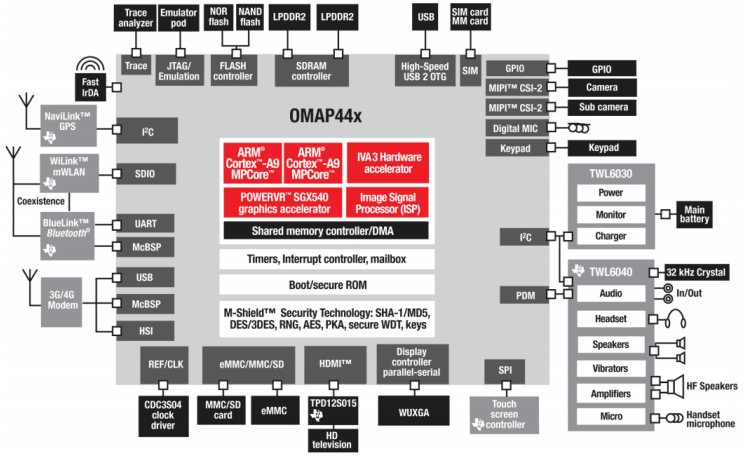



Abbildung 2.3: OMAP Microarchitektur [60]

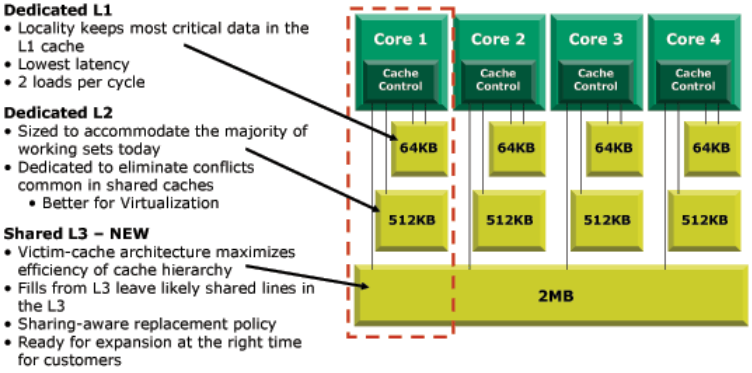



Abbildung 2.4: Cache Architektur eines AMD Barcelona Multi-Core Prozessors [55]

die Daten nicht zur Verfügung stehen, so wird der externe Speicher abgefragt. Die gemeinsame Nutzung von L2 und L3 Speichern ist weiterhin unterteilt in drei verschiedene Unterarten (Abbildung 2.5). So gibt es den Unified Memory Architecture (UMA) Multi-Core Prozessor. Mit diesem Prozessor hat jeder Core einen eigenen L1 Speicher und alle Cores haben Zugriff auf einen gemeinsamen L2/L3 Speicher. Diese Art von Speicherauslegung wird genützt für Symetrische Multi-Core Porzessor (SMP) Betriebssysteme.
Diese ermöglichen es, dass Tasks auf allen Cores ausgeführt werden können. Der Gegensatz dazu ist der Non-Unified Memory Architecture (NUMA) und CC-UMA Multi-Core Prozessor. Hier sind sowohl L1 als auch L2 Caches der Cores untereinander getrennt. Somit ist es nicht möglich, dass Tasks auf unterschiedlichen Cores laufen können.

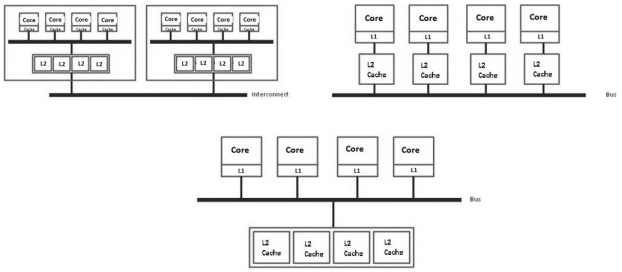



Abbildung 2.5: Links CC-UMA, rechts NUMA, unten UMA

### 2.2.2 Multi-Core und RTOS Aufteilung

Um einen Multi-Core Prozessor zu benutzen, muss erst geklärt werden, wie der Shared Memory Bereich aufgeteilt wird. Hier gibt es die Möglichkeit, sich für ein reines Asymetrisches Multi-Core Processor (AMP) System zu entscheiden. Dazu muss der Shared Memory Bereich so aufgeteilt werden, dass jeder Core nur auf einen bestimmten Bereich Zugriff hat. Falls es keinen Shared Memory Bereich gibt, wie beim Ambric Bric, so kann dort nur ein AMP System installiert werden. Jeder Core bekommt nun sein eigenes Betriebssystem zugewiesen und führt dieses

aus. Im Falle eines Echtzeitsystems wird ein Real Time Operating System (RTOS) ausgeführt, welches nach dem implementierten Scheduling Algorithmus versucht, die gegebenen Deadlines einzuhalten. Auch ist es möglich, und im Falle des Ambric Brics wahrscheinlich, kein RTOS zu implementieren sondern den Task „Bare Metal“ auf einem Core rechnen zu lassen. Das hat den Vorteil, dass bei besonders zeitkritischen Anwendungen die Verwaltungsaufgaben des RTOSes nicht den Task am Rechnen hindern [42].
Eine zweite Möglichkeit ist es ein SMP System mit einem Multi-Core herzustellen. Hier verwaltet lediglich ein Betriebssystem, im Echtzeitbereich ein RTOS, alle Cores und den gesamten Shared Memory Bereich. Die Aufgabe des RTOS Betriebssystems ist es nun, nicht nur die Tasks und deren Deadlines einzuhalten, sondern auch auf die richtigen Cores zu verteilen, die Auslastung der Cores zu balancieren und die Interrupts zu handhaben. Ein SMP System kann weiter unterteilt werden. Zum Beispiel ist es mit dem P4080 [17] möglich, zwei SMP Systeme mit jeweils 4 Cores aufzubauen und voneinander zu trennen. So hat ein dual SMP System, welches durch Hardware getrennt ist, je ein ROTS als Betriebssystem für sich.
Ob AMP System oder SMP System, ist das RTOS Betriebssystem die wichtigste Schnittstelle zwischen Task und Core. Das RTOS bestimmt, welcher Task wann und wie lange laufen darf und über welche Ressourcen er verfügen kann. Gerade in diesem Bereich gibt es viele unterschiedliche Firmen, welche RTOSe für Multi-Core Systeme anbieten. Unter anderem die Firma WindRiver mit ihrem Betriebssystem VxWorks, ein RTOS, welches in der Luft- und Raumfahrt für viele unterschiedliche Bereiche eingesetzt wird.
Ebenso gibt es OpenSource RTOSe, wie z.B. FreeRTOS oder das Linux, erweitert mit dem Preemption Patch. Im Bereich der Automotive Industrie sind Betriebssysteme mit dem AUTOSAR Standard im Einsatz.

### 2.2.3 Multi-Core Steuergeräte der Zieldomänen

Nachfolgend werden die Multi-Core Prozessoren aufgezeigt, welche in den Zieldomänen offiziell und nachweislich eingesetzt werden:

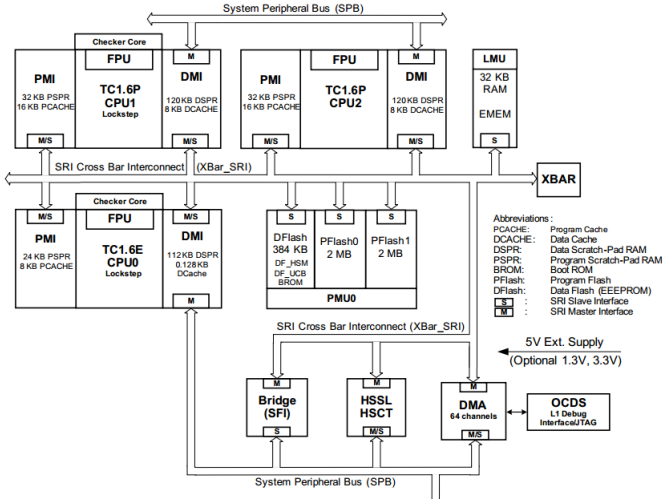



Abbildung 2.6: Infineon Tri-Core TC27X [28]

### Infineon Tri-Core Anwendungen

Der Infineon Tri-Core ist der bekannteste Multi-Core Chip der Automotive Branche. Eingesetzt wird er im Bereich der Motorsteuergeräte und Abgasregelung. Warum der Infineon Tri-Core so häufig eingesetzt wird, liegt in dessen Systemarchitektur. Wie aus Abbildung 2.6 ersichtlich, besitzt der Tri-Core drei voneinander unabhängige Cores mit eigenen Cache und Data Memory Interface (DMI). Jeder Core kann somit als ein AMP System betrieben werden. Da es aber keinen gemeinsamen Speicher gibt, ist der Betrieb als SMP System nur bedingt möglich. Da es keine Coherency Einheit gibt, müssen Daten zwischen den Cores explizit über Core-Interne Nachrichten ausgetauscht werden. Zwei der drei Cores beherbergen einen sog „Checker Core“, der um zwei Takte verzögert die gleichen Berechnungen macht wie der Master Core. So können die Ergebnisse der beiden Cores verglichen und so eine Redundanz erzeugt werden. Besonders auffallend ist hier die Trennung von einem Inter Core Netzwerk (SPI), auf dem die Cores untereinander und

mit dem Programmspeicher Daten austauschen, und dem Peripherie Netzwerk, auf dem die I/O Schnittstellen Daten zum Core senden und ggf. Daten bekommen. So lassen sich Verzögerungen bei der Taskausführung durch I/O Geräte vermeiden. Auch eine Firewall beschützt die Cores vor unbefugten I/O Zugriffen [28].

### Freescale P4080

Der P4080 Multi-Core Prozessor von Freescale ist bekannt durch seinen Einsatz in der Luft- und Raumfahrtindustrie. Spezielle Materialien, die diesen Chip gegenüber Strahlung im Weltraum härten, machen diesen interessant für den Einsatz in Satelliten oder Trägerraketen. Auch in der Luftfahrtindustrie wird dieser Chip eingesetzt[1]. Die Besonderheiten an diesem P4080 Multi-Core SOC sind die Möglichkeiten zur Konfiguration in der Speicherhierarchie sowie die Anzahl und Varianten von I/O Geräten. Wie in Abbildung 2.7 zu erkennen, besitzt jeder Core seinen eigenen L1 und L2 und es gibt zwei gemeinsame L3 Caches mit Anbindung an dem RAM- Speicher. Intern ist der Prozessor so aufgebaut, dass vier Kerne unabhängig voneinander betrieben werden können, also ihre eigene Clock besitzen. So ist es mit dem P4080 möglich, jeden Core einzeln als AMP Systeme zu benutzen oder ein gemischtes System AMP und SMP zu betreiben. Jeder Core kann unabhängig voneinander gestartet oder zurückgesetzt werden, was für unabhängige AMP System von Vorteil ist. Durch die Implementierung von zwei Taktgebern können auch zwei SMP Systeme mit je vier Kernen aufgebaut werden, die z.B. redundant laufen.
Ein weiterer Vorteil für den Einsatz in der Luft- und Raumfahrt ist die „CoreNet Coherency Fabric“ des P4080. Damit lassen sich die Shared Resources zwischen den Cores synchronisieren mit einer schnellen Punkt-zu-Punkt Verbindung, was Latenzen bei der Task Ausführung minimieren soll [17].

[1] Einsatzgebiete unterliegen der Geheimhaltung

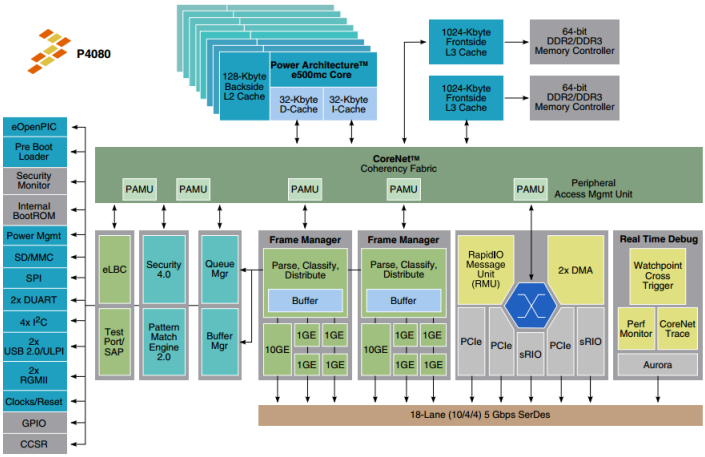



Abbildung 2.7: Freescale P4080 Überblick [17]

## 2.3 Zusammenfassung

In diesem Kapitel wurden alle Grundlagen vorgestellt, welche für ein Echtzeitscheduling in Multi-Core Systemen benötigt werden. Hier wurde insbesondere auf die aktuellen Taskmodelle eingegangen für periodische (Formel 2.1) sowie aperiodische und sporadische Tasks (Formel 2.2). Aufbauend auf diesen Modellen wurden die meistbenutzten Scheduling Algorithmen in Echtzeitsystemen vorgestellt: Cyclisch, RMS und EDF. Für jeden Scheduling Algorithmus wurden Auslastungstests eingeführt, welche eine Vorabprüfung zulassen, ob die auf dem System allokierten Tasks ihre Deadlines einhalten können. Hier sind insbesondere die Formeln für RMS (2.10) und EDF (2.12) von Liu und Layland zu nennen und die Formeln für das Einplanen von aperiodischen und sporadischen Prozessen unter RMS (2.11) und EDF (2.13). Weiterhin wurden Bin Packing Algorithmen, wie First Fit, Best Fit usw. erklärt und deren Nachteile in Bezug auf Optimalität in Tabelle 2.1 aufgezeigt. Als konträre Lösung zu diesem Verteilungsalgorithmus wurde der Brute Force

Algorithmus vorgestellt, der im Gegensatz dazu eine optimale Lösung findet, dafür aber viel Rechenzeit benötigt. Auch die Multi-Core Eigenschaften, die für ein erfolgreiches Echtzeitscheduling benötigt werden, wurden in diesem Kapitel erklärt. Hierbei ist die Architektur, ob es sich um einen leicht- (Bild 2.1), mittel- oder schwergewichtigen (Bild 2.3) Prozessor handelt, von Bedeutung. Schwergewichtige Prozessoren mit unter Umständen einem SOC System kommen einzig für Multi-Core Scheduling in Betracht.

# 3 Stand der Forschung

Schedulingalgorithmen in embedded Systems sind ein sehr breites und weitläufiges Forschungsgebiet. War es zuerst die Aufgabe, eine ECU und die dort allokierten Tasks sicher und ohne Deadlineverletzung zu schedulen, wird heute zusätzlich darauf hingearbeitet, mehr Tasks mit zu nehmen oder aufwändigere, rechenintensivere Algorithmen zu verwenden. Aus diesem Grund, hat sich auch die Forschung darauf eingestellt, Berechenbarkeit der vorherrschenden Auslastungstests von Liu und Layland noch weiter zu verbessern, um durch eine präzisere Vorhersage mehr oder rechenintensivere Tasks auf einem Steuergerät zu allokieren, wie es in Kapitel 2.1.2 und 2.1.2 aufgezeigt wird. Aktuelle Forschung versucht nun zum einen diese Formel zu erweitern und zum anderen Multi-Core Systeme in die Zieldomänen zu integrieren, um durch diese zwei Methoden das Steuergerät optimal auszuschöpfen. Dies geschieht entweder durch die Entwicklung neuer Scheduling Algorithmen oder Adaptierung von Single-Core Scheduling. Insbesondere spielt hierbei die Taskverteilung auf die Multi-Core Systeme eine Rolle, welche entweder durch Algorithmen berechnet, durch Simulation empirisch ermittelt oder durch wissensbasiertes Scheduling gemanagt wird.

## 3.1 Scheduling – Forschung

### Scheduling Algorithmen

Preemptive Scheduling Algorithmen und deren Auslastungstests stellen einen großen Forschungsbereich dar. Dieser Forschungsbereich ist unterteilt in statisches prioritätenbasiertes Scheduling und dynamisches prioritätenbasiertes Scheduling. Zu Ersterem gehört der RMS Algorithmus aus Kapitel 2.1.2 mit dem Auslastungstest 2.7 und den zur Designzeit festgelegten Prioritäten. Zum Zweiten gehört der EDF Algorithmus, beschrieben in Kapitel 2.1.2 mit dem dazugehörigen Auslastungstest 2.12

und den zur Laufzeit, auf Basis der Deadline berechneten Prioritäten. Beide von Liu entwickelten Algorithmen haben die Anforderung, dass Deadlines und Perioden gleich sind. Da dies in den Zieldomänen aber nicht immer der Fall ist, wird der Deadline Monotonic Scheduling (DMS) Algorithmus angewandt [99]. Der DMS Algorithmus basiert ebenfalls auf fest vorgegebenen Prioritäten und kann periodische und sporadische Tasks ohne zusätzlichen Server verarbeiten [61]. In DMS gelten die Voraussetzungen für die Parameter, dass $C_i \leq D_i \leq P_i$ ist. Für die Vergabe der Priorität ist die Deadline ausschlaggebend, wobei der Task mit der niedrigsten Deadline die höchste Priorität hat [90]. In [61] sind die Auslastungstests für den DMS Algorithmus aufgelistet. In einem System mit aperiodischen Tasks bietet sich das Least Laxity First (LLF) Scheduling an. Dieser dynamische Schedulingalgorithmus plant zur Laufzeit den nächsten Prozess anhand der Formel $D_i - r_i - C_i$, wobei der Parameter $r_i$ die Bereitzeit eines Tasks ist [13]. Die Auslastungstests sind in [41] aufgelistet. Da dieser Algorithmus u.U. sehr viele Context Switches durchführt, wurde der Algorithmus zu Modified Least Laxity First (MLLF) erweitert [130]. In MLLF wird durch „Laxity Inversion“ versucht, die Anzahl der Context Switches niedrig zu halten [130].

Letztere Algorithmen sind für Single-Core oder partitionierten Multi-Core ausgelegt, in welchem jeder Core unabhängig voneinander schedult (fully partitioned) [117]. Soll ein Multi-Core Prozessor global geschedult werden, worin ein einziger Scheduler alle Cores simultan
bedient, so müssen Auslastungstest und ggf. der Scheduling Algorithmus angepasst werden. RMS wurde so zu RM-US von Andersson weiterentwickelt [118]. Hier ist die Auslastungsgrenze mit der Formel $U \leq \frac{m^2}{3m-2}$ dargestellt, worin **m** die Anzahl von homogenen Cores entspricht [118]. Laut dieser Formel beträgt die maximal zulässige Auslastung eines Prozessors mit vier Cores 160% anstelle von 276% für lokales RMS Scheduling, unabhängig von der Anzahl der Tasks. Auch der RM-US Algorithmus wurde erweitert, indem die Voraussetzung für identische Prozessoren aufgehoben wurde. Der daraus entstandene Algorithmus ist die Grundlage für entsprechende „Multi-Core Greedy Algorithmen“ im Scheduling [88] [85]. Hierbei wird die maximale Auslastung niedriger je unterschiedlicher die Prozessoren sind [88]. Der EDF Algorithmus wurde ebenfalls zu Global Earliest Deadline First (GEDF) verändert [136] [83].

Hierbei wurde die maximale Auslastung des Systems auf $U \leq \frac{m}{4-\frac{2}{m}}$ beschränkt, wobei die Auslastung eines Tasks nicht größer sein darf als $U < \frac{1}{4-\frac{2}{m}}$ [136] [91]. Auch der LLF Algorithmus wurde im Bereich Multi-Core zum Earliest Deadline Zero Laxity (EDZL) Algorithmus. In diesem Algorithmus wird unterschieden zwischen Tasks mit einer Laxity $> 0$ und $\leq 0$. Sobald der letzte Fall eintritt, werden Tasks als wichtig betrachtet und bekommen eine höhere Priorität als Tasks mit einer positiven Laxity. Der Auslastungstest gibt eine Grenze von $U \leq \frac{m+1}{2}$ vor und besitzt somit eine höhere Auslastung als GEDF [121].
Da globale Algorithmen ein Auslastungsdefizit gegenüber partitionierten Algorithmen haben, letztere aber nicht parallel schedulen können, wurde das „Task Splitting“ eingeführt [117] [114]. Hier wird ein partitionierter Scheduling Algorithmus, wie z.B. PDMS-HPTS-DS „Partitioned Deadline-Monotonic Scheduling with Highest Priority Task Splitting in Decreasing order of size“ verwendet [72] [114]. Mit dem Task Splitting werden so höhere Auslastungswerte als mit globalem Scheduling erlangt, dafür müssen aber Synchronisierungsmechanismen im Scheduler und im Cache des Multi-Core Prozessors vorhanden sein, welche diesen Vorteil schmälern [104] [101] [72].

## Multi-Core Verteilung

Die geringeren Auslastungszahlen in Multi-Core Prozessoren im globalen Scheduling forcieren den partitionierten Einsatz und so wurde die Taskverteilung zu einem weiteren wichtigen Forschungsaspekt.
Unabhängig von der maximalen Auslastung und der Auslegung von global oder partitioniertem Scheduling kann die Verteilung der Tasks die Performance eines Systems beeinflussen. Hier gibt es zwei Aspekte zu betrachten.

**Lastverteilung** Zum einen entscheidet die Verteilung der Tasks über den Energieverbrauch der CPU. So haben Aydin und Yang publiziert, dass eine möglichst gleichmäßige Lastverteilung der Tasks auf die Anzahl der Cores die energiesparendste Variante ist („Load Balancing“) [109]. Aus dieser Strategie heraus entstanden unterschiedlichste Scheduling Algorithmen, welche das Load Balancing und die unterschiedlichen Energiespartechniken der CPUs mit einbeziehen. Untersucht wurden

die Bin Packing Algorithmen in [109] mit dem Ergebnis, dass der Bin Packing Algorithmus Best Fit Decreasing die besten Ergebnisse in Bezug auf Energieverbrauch liefert. Aber auch Load Unbalancing Strategien wurden untersucht, um mit der Abschaltung von Cores bei zu geringer Auslastung Energie zu sparen [135] [102]. Ebenso ist der Dynamic Voltage and Frequency Scaling (DVFS) Mechanismus der CPU zur Reduzierung der Frequenz und Spannung des Prozessors, um die CPU energieeffizienter zu betreiben, sehr weit erforscht [133] [125] [22].
In [92] hingegen wurde der DVS Algorithmus in einem „RT-DVS Cyclic Conserving“ und „RT-DVS Look Ahead“ Scheduling Algorithmus auf einem embedded System umgesetzt. Im Cyclic Conserving Algorithmus wird davon ausgegangen, dass ein Task nicht seine komplette WCET benötigt. Ist das der Fall, wird die Frequenz der CPU gedrosselt und so die Rechenzeit des Tasks verlängert, bis er die ganze WCET benötigt. Der Auslastungstest hat immer noch einen positiven Ausgang, solange die WCET nicht überschritten wird. Zur Laufzeit wird stetig überprüft, ob der Task aktuell seine maximale WCET [92, Seite 93] aufbraucht. Die Vorgehensweise bei einer unerwarteten Steigerung der Rechenzeit eines Tasks bei gleichzeitiger niedriger Frequenz wurde offen gelassen.

**Online Aufnahmetest** Ein weiterer Punkt, um die Verteilung von Tasks in einem System optimaler zu gestalten, sind Aufnahmetests von Tasks auf Cores, wie der Constant Time Admission (CTA) Algorithmus. Der CTA Algorithmus ist ein online Auslastungstest und berechnet, ob zu einem bestehenden Set von Tasks noch ein weiterer Task hinzukommen kann, ohne dass Deadlines der Task verletzt werden. Dazu berechnet dieser die Auslastung der EDF Tasksets und bestätigt die Schedulbarkeit, wenn die Auslastung unter 100% bleibt. Zur Berechnung der Auslastung wird eine Demand Bound Funktion [67] ausgeführt und dabei die Zeit in b+1 Intervalle geteilt. Dabei steigt die Qualität der Ausgabe über die Auslastung mit der Anzahl der Intervalle. Je mehr Intervalle umso genauer kann überprüft werden, zu welcher Zeit eine Auslastung $> 100\%$ auftritt [119].

## Wissensbasiertes Scheduling ohne KB

Viele der in Kapitel 3.1 vorgestellten Auslastungstests sind in die Klasse der „NP schweren Probleme“ eingeordnet, die nicht in konstanter Zeit gelöst werden können. Da diese Tests in Echtzeitsystemen nicht ohne zeitlichen Mehraufwand anwendbar sind, wurden regelbasierte Optimierungsalgorithmen erforscht. Ziel dieser Regelung ist es, so viele Tasks wie möglich auf einem System unterzubringen sowie gewisse, als kritisch deklarierte Tasks ohne Deadlineverletzung rechnen lassen zu können. Eine Art des wissensbasierten Schedulings ist der „Fuzzy Rule-Based Real-Time Scheduler“ von Lee et al. in [69]. In dieser Arbeit wurde das Scheduling auf Ziele reduziert, d.h. treffen mehrere Tasks gleichzeitig ein und es ist ausschließlich möglich einen von zwei Tasks korrekt zu schedulen, so wird als Ziel für diese Situation z.B. den wichtigsten Task zu schedulen, definiert. Für diese Vorgehensweise muss immer die aktuellste Situation im Scheduler betrachtet werden, d.h. welcher Task wird gerade geschedult und welche Tasks sind rechenbereit. Anhand dieser Information wird ein Vergleich durchgeführt, ob dafür ein bestimmtes Ziel vorhanden ist. Diese Vorgehensweise ist gleichzusetzen mit einem „Wenn-Dann-Vergleich“. Dabei kann im Voraus für jede Situation ein anderes Ziel gewählt werden, um so das beste Ergebnis empirisch zu ermitteln.

## Wissensbasiertes Scheduling mit KB

Einen weiteren Schritt im wissensbasierten Scheduling haben Göttge et al. [74] gemacht, indem sie das Tool Real-Time Scheduling Assistant (RTSA) entwickelten, welches die Entwickler dabei unterstützt, komplexe Echtzeitsysteme zu evaluieren. In diesem Tool wird dem Entwickler die Möglichkeit gegeben, vom Konzept über die Analyse bis hin zur Entwicklung von Scheduling Algorithmen für Betriebssysteme, das System im Bereich des Schedulings selbst zu verwalten. Bevor eine offline Analyse getätigt werden kann, muss das RTAS Tool mit den Parametern über das zu untersuchende System initialisiert werden. Hierbei werden sowohl die Task Parameter eingegeben wie die WCET, Deadline und Periode, ein System Model und einem Scheduling Algorithmus. Mit diesem Input wird eine Schedulinganalyse durchgeführt. Aus dieser Analyse des Tools wird für den Entwickler „Wissen“ generiert, um so

Verbesserungen erarbeiten und umsetzen zu können im Bereich des Schedulings. Diese Verbesserungen werden in sog. Regeln (rules) umgewandelt, die auf einem „Wenn-Dann-Vergleich“ basieren. Diese Regeln sind nicht wie im Fuzzy Rule-Based Real-Time Scheduler mathematisch mit einem Vergleich anzuwenden, sondern es handelt sich hierbei um linguistische komplexe Befehle, die ausschließlich vom RTAS Tool verarbeitet und den Tasks mitgegeben werden. Diese Regeln werden zur Laufzeit vom Task ausgewertet und umgesetzt. Wie diese Regeln und das Ergebnis des RTAS Tools zur Laufzeit korrekt angewandt werden, ist der Veröffentlichung nicht zu entnehmen.

## Timing Analyse

Ein weiterer Forschungszweig, der sich aus dem Übergang von Single-Core zu Multi-Core Prozessoren ergeben hat, ist die Simulation von Scheduling Algorithmen [124] [134]. Dieser Forschungszweig wurde mit Hilfe der sog. „Timing Analyse“ kommerziell erweitert. Hierbei werden die Parameter von Funktionen und deren Laufzeitverhalten empirisch ermittelt und das Zusammenspiel mit anderen Funktionen simuliert [3]. Dieser Timing Analyse haben sich u.A. zwei Firmen angenommen, die Firma Timing Architects und die Firma Symptavison GmbH [53]. Diese beiden Firmen stellen unterschiedlichste Tools und Hardware zur Verfügung für die Unterstützung bei der statischen Allokierung von Tasks. Zum einen wird der bestehende Scheduling Algorithmus mit den allokierten Tasks simuliert und stimuliert umso Unzulänglichkeiten festzustellen (Basisanalyse). Die Timing Parameter für alle auf einem Core zu allokierenden Tasks sowie ein Parameter des Multi-Core Systems und des Betriebssystems dienen als Eingang für die Basisanalyse. Um die bestehende Allokation zu verbessern, wird ein weiteres Tool (Inspektor) benötigt, um mit variierenden Parametern und Voraussetzungen ein besseres Ergebnis zu bekommen [3]. Um zu diesen Ergebnissen zu gelangen, werden die Funktionen und deren Timing durch Hardware oder Software Tracing ermittelt. Bei einer erfolgreichen Basisanalyse kann der Schedule optimiert werden. Hier werden z.B. Tasks, die miteinander kommunizieren müssen, auf dem selben Core gelegt, um eine Intra-Core Kommunikation zu vermeiden. Ebenso ist es möglich, die Parameter

der Tasks zu modifizieren und Offsets für Tasks hinzuzufügen, um einen besseren zeitlichen Verlauf zu generieren [122].

## Wissensbasiertes Design/Testen

Ein weiterer Anwendungsbereich ist das wissensbasierte Design bzw. Testen wie es an der TU-Chemnitz durchgeführt wird. In den Arbeiten [112] und [111] wird vorgestellt, aus welchen Softwarebestandsteilen eine AUTOSAR ECU aufgebaut ist, nämlich der Basis Software Layer, die Run-Time Environment (RTE) und die Hauptfunktion. Diese Komponenten werden aus wettbewerbstechnischen Gründen von unterschiedlichen Herstellern geliefert und sind somit anfällig für nicht optimale Konfigurationen [112]. Somit untersuchen die Forscher aus [111] eine neue Methode um diese Konfigurationen zu überprüfen und ggf. die Qualität zu verbessern. Dies geschieht in einem ersten Schritt mit einem statischen Test-Tool. In diesem Tool wird der Source Code der Hautpfunktion und die Konfigurationsdaten (ARXML) für die RTE und des Basiss Software Layers vor der Kompilierung überprüft [112]. Zeitgleich zur Überprüfung wird in einem zweiten Schritt eine „AUTOSAR Knowledge Base“ vom statischen Test-Tool aufgebaut. In dieser AUTOSAR Knowledge Base werden Informationen eingetragen über den Aufbau der AUTOSAR ECU in Bezug auf die Basis-Software, die verschiedenen Funktionen der Hauptfunktionen und Wissen aus den Projekt-Konfigurationsdatein. Mit einem weiteren Tool zur statischen Analyse lassen sich aus den unterschiedlichen Konfigurationen Inkonsistenzen herausfiltern und grafisch anzeigen. Alle, auch nicht inkonsistente Informationen können grafisch aufbereitet werden um das Wissen aus der AUTOSAR Knowledge Base zu überprüfen bzw. zu verbessern. Neben dem statischen Tests erlaubt das Tool auch, dass zur Laufzeit Daten über die Ausführung der Funktionen auf der ECU gesammelt und zur AUTOSAR Knowledge Base hinzugefügt werden, umso das Steuergerät noch weiter zu optimieren bzw das Design zu verbessern.

## 3.2 Ausgewählte Systeme

Neben dem wissenschaftlichen Aspekt Multi-Core Systeme optimaler zu nutzen, haben sowohl Industrie als auch Universitäten an einer Umsetzung geforscht. So gibt es unterschiedliche Systeme, die sich mit dem Einsatz von Multi-Core in den Zieldomänen beschäftigen. Andere Systeme wiederum untersuchen die Zusammenlegung von Funktionen auf ein Steuergerät. Die Ergebnisse dieser Systeme wurden am Ende der Projektlaufzeit festgehalten.

### IMA, DIANA und SCARLETT

In der Avionik Domäne gibt es drei verschiedene Projekte, deren Ergebnisse aufeinander aufbauen und sich heute schon im Einsatz befinden. Eines der bekanntesten Systeme, bzw. deren Ergebnis, ist die Integrated Modular Avionics (IMA) Architektur. Die Entwicklung dafür begann in den 80er Jahren mit unterschiedlichsten Systemen, u.A. NEVADA, PAMELA, ANAIS und VICTORIA [7]. Die Ergebnisse und Entwicklungen erbrachten die IMA $1^{gen}$. Mit IMA $1^{gen}$ war es möglich, die anhaltende Zunahme an Funktionen und Steuergeräten im Flugzeug zu stoppen und im A380 auf dem Niveau des Vorgängers A340-600 zu halten (Abbildung 3.1). IMA $1^{gen}$ hat dies geschafft, indem eine Möglichkeit gefunden wurde, verschiedenste Funktionen, auch mit unterschiedlichen Anforderungen an Kritikalität, auf einem Steuergerät zu vereinen. Die Schwierigkeit die dabei entsteht ist, dass Funktionen sich u.U. gegenseitig beeinflussen können. Dies kann z.B. dadurch entstehen, dass eine Funktion kurzzeitig eine längere WCET benötigt als vorgesehen, was zu Deadline-Fehlern führen kann. Ebenso können Speicherbereiche der Funktionen von anderen Funktionen überschrieben werden, wenn die Programmierung nicht korrekt durchgeführt wurde. Diese Fehler werden in IMA $1^{gen}$ durch einen sog. Level 1 Hypervisor unterbunden. So soll verhindert werden, dass z.B. unkritische Funktionen in den Speicherbereich von kritischen Funktionen schreiben. Auch zeitlich gesehen trennt der Level 1 Hypervisor, die verschiedenen Gast-Betriebssysteme durch einen cyclischen Schedule und zugewiesenen Slots. Level 1 Hypervisor, wie der WindRiver Hypervisor oder der Hypervisor Kernel Based Virtual Machine (KVM), kommen für diesen Einsatz in Frage [7].

In IMA $2^{gen}$ soll eine Rekonfiguration vorgesehen werden. Diese Rekonfiguration wird am Boden, beim Aufstarten des Systems durchgeführt. So können Funktionen von defekten ECUs durch eine Rekonfiguration von anderen Steuergeräten aufgenommen werden, ohne dass dieses ersetzt werden muss. Dadurch kann das Flugzeug zum einen schneller für den nächsten Flug bereit gemacht werden, da es nicht auf Ersatzteile angewiesen ist, zum anderen müssen nicht an jedem Flughafen Ersatzgeräte für jeden Flugzeugtyp und jede ECU vorhanden sein. Damit ist es den Airlines möglich, Geld zu sparen [7] [76]. Wie diese Rekonfiguration in der IMA $2^{gen}$ Architektur umgesetzt wird, damit haben sich die Systeme DIANA und SCARLETT beschäftigt.

**DIANA** Das **D**istributed, equipment **I**ndependent environment for **A**dvanced avio**N**ics **A**pplications (DIANA) System hat sich damit befasst, IMA und die ARINC 653 Spezifikation vereinbar zu machen. Daraus entstand die Plattform Architecture for Independent Distributed Avionics (AIDA), die einerseits eine neue offline Toolkette hervorbrachte als auch Erweiterungen in den Gast-Betriebssystemen und dem Hypervisor, die zur Laufzeit benützt werden konnten [44].
Ziel war es, Kosten bei der Entwicklung von Avionik zu reduzieren. Mit dem offline Tool wurden Funktionen, kritische und unkritische, sowie die Architektur modelliert. Durch diese Modellierung ist es möglich, schon vorab festzustellen, wo Probleme bei der Allokation von Funktionen auf Steuergeräten auftreten und wie diese behoben werden können. Die Erweiterung für die Gast-Betriebssysteme und den Hypervisor besteht aus Bibliotheken und Diensten, welche die Administration und Konfiguration vereinfachen sollen. Zwei Bibliotheken die eingeführt wurden sind: Eine Kommunikationsarchitektur Library, auf der sich eine Funktion anmelden kann, um Daten zu senden und zu empfangen. Diese Library ist nützlich, um bei einer Rekonfiguration wieder benötigte Daten zu erhalten, auch wenn die Funktion sich nicht mehr auf dem gleichen Steuergerät befindet. Auch wurden Vorschläge zur Rekonfiguration unterbreitet. So wurde ein Konzept und eine entsprechende Middleware zur Multi-Static Rekonfiguration vorgeschlagen, für die Rekonfiguration vor und während des Fluges [75]. Das DIANA System hat sich aufgrund von Zertifizierungsgründen auf eine Rekonfiguration vor dem Flug beschränkt. Die Rekonfiguration hatte das Ziel, wie IMA $2^{gen}$ ein defektes

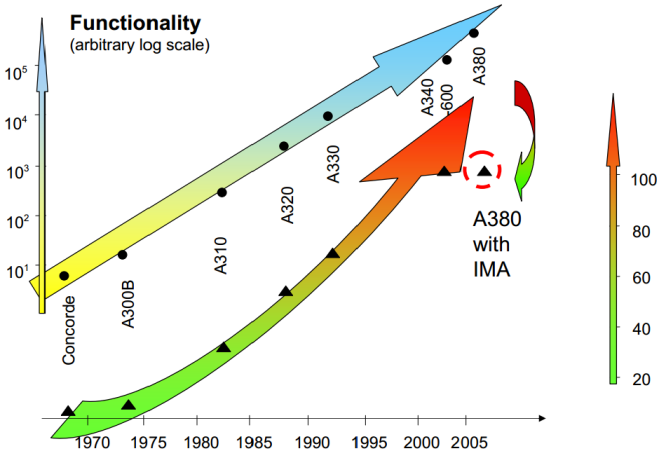



Abbildung 3.1: IMA als Reduzierung der Hardware in A380 [6]

Steuergerät so zu ersetzen, dass die Minimum Equipment List (MEL) erfüllt ist und es dem Flugzeug somit erlaubt war zu starten. Wird nun im Rahmen von Post- oder Pre- Fluguntersuchungen ein defektes Steuergerät gefunden, so wird dem Wartungsteam dieses gemeldet. Anstatt das defekte Gerät zu tauschen, wird es deaktiviert. Um die MEL zu erfüllen, wird mit dem AIDA offline Tool eine Konfiguration gefunden, die es durch Softwaretechnische Rekonfiguration erlaubt, ein „Go“ für den Start zu bekommen. Mit Hilfe der Middleware ist es möglich, die neue Konfiguration aus dem AIDA Tool zu laden. Die geladene Konfiguration ändert sich im Steuergerät nicht mehr ohne menschliche Interaktion. Dies bedeutet aber auch, dass ein Wartungsmitarbeiter an mehreren Steuergeräten diese Einstellungen vornehmen muss. Um die menschliche Interaktion zu minimieren wurde das SCARLETT System aufgesetzt [44] [62].

**SCARLETT** Wie auch DIANA beschränkt sich **SCA**lable and **R**econfigurab**L**e **E**lectronics Pla**T**forms and **T**ools (SCARLETT) auf eine Rekonfiguration am Boden. Aber im Unterschied zu DIANA werden in diesem System die Partitionen der Steuergeräte automatisch rekonfiguriert. Um dies zu ermöglichen, werden neben den Partitionen zusätzliche „Spare“ Partitionen in jedem Steuergerät bereit gehalten, die später für eine Rekonfiguration benützt werden können. Ob im Rahmen eines Wartungschecks rekonfiguriert werden muss, wird vor dem Flug evaluiert, indem von jedem Steuergerät eine Zustandsinformation abgefragt wird. Sollten diese Checks negativ sein, wird eine Rekonfiguration angestoßen. Damit dies automatisch abläuft, gibt es auf jedem Steuergerät drei unterschiedliche Funktionen, die Lade-, Überwachungs- und Powerfunktion. Erstere lädt neue Partitionen auf das Steuergerät, eine weitere überwacht die Ausführung des Steuergeräts und generiert die Zustandsinformation und die letzte Funktion sorgt dafür, dass die Stromzufuhr zu diesem Steuergerät im Fehlerfall unterbrochen wird. Zusätzlich gibt es einen zentralen Konfigurationssupervisor, der die automatische Rekonfiguration steuert. Sollte ein Steuergerät einen Fehler haben, so wird dies durch die Powerfunktion ausgeschalten. Dem Supervisor steht eine Liste von Konfigurationen zu Verfügung, die dieser nun prüft, ob sie auf die aktuelle Situation angewendet werden können. Diese Konfigurationen werden zur Designzeit erstellt und dem Supervisor unveränderlich mitgegeben. Es finden online keine weiteren Berechnungen statt. Sobald eine passende Konfiguration gefunden wurde, wird auf externe Anweisung eine Spare Partition hochgefahren und mit den erforderlichen Daten geladen. Ist dies abgeschlossen, so wird eine Zustandsinformation gesendet, ob die Rekonfiguration erfolgreich war oder nicht. Zu beachten ist hierbei, dass die Partitionen, die auf einem funktionsfähigen Steuergerät allokiert sind, sich auch in einer Rekonfiguration nicht ändern. Es werden lediglich die Spare Partitionen für eine Rekonfiguration eingesetzt [7] [76].
Das SCARLETT System besteht noch aus vielen weiteren technischen Neuerungen, wie eine Übermittlung von Daten über das 28V Bordnetz einen Level 1 Hypervisor von SYSGO mit dem Betriebssystem PikeOS. Diese haben aber keine weitere Bedeutung für den Verlauf dieser Arbeit [47].

## DREAMS

Im System der **D**istributed **RE**al-time **A**rchitecture for **M**ixed Criticality **S**ystems (DREAMS) wird eine Systemarchitektur vorgestellt, die es erlaubt, Multi-Core Systeme in einem Verbund mit anderen zu betreiben. Ein Multi-Core System ist dabei so aufgebaut, dass ein Level 1 Hypervisor (u.A. KVM) für jeden Core ein Gast Betriebssystem zur Verfügung stellt. Dabei ist nicht genauer spezifiziert, ob ein Gast Betriebssystem auch mehrere Cores benutzen kann [106]. Das Gast Betriebssystem an sich beherbergt einen Zustandsmonitor und einen lokalen Scheduler. Der Hypervisor hat ebenfalls einen Zustandsmonitor und einen Scheduler. Letzterer kann sowohl cyclisch als auch prioritätenbasiert schedulen. Im System DREAMS werden vor allem die Schnittstellen beschrieben, die Tasks benützen können, um mit ihrer Umwelt zu kommunizieren. So werden Unterschiede zwischen einer Kommunikation innerhalb des Multi-Core Prozessors und der Kommunikation mit Teilnehmern außerhalb des Multi-Core Prozessors aufgezeigt [94] [113]. Da zwischen den Partitionen aufgrund der Trennung durch den Hypervisor keine direkte Kommunikation möglich ist, übernimmt der Hypervisor nun die Rolle eines Switches. In virtuellen Links lassen sich so von einer Partition zur anderen Kommunikationskanäle aufbauen. Der Hypervisor verteilt empfangene Nachrichten an alle Cores, die sich zum Empfang dieser Nachrichten angemeldet haben. Ebenfalls funktioniert dies mit Off-Prozessor-Netzwerken. Hier werden über externe Switches Nachrichten an die verschiedenen Multi-Core Steuergeräte weitergeleitet. Spezielle Libraries für die DREAMS Switches und Steuergeräte erlauben es dabei, verschiedene Prioritätsstufen für die Nachrichten zu verwenden, um wichtige Nachrichten schneller weiterleiten zu können [94].
Neben der Kommunikation ist die zeitliche Synchronisation in allen vorhandenen Steuergeräten wichtig. Damit die DREAMS Architektur funktioniert, müssen alle verbundenen ECUs die gleiche globale Zeitbasis haben. Ein „Compression master“ gibt dabei die globale Zeit vor und die Clients haben sich durch Korrektur ihrer internen Drift und am Start durch eine Offset Korrektur auf diesen einzustellen. Dieser Vorgang ist transparent und hat für den Task auf dem Gast Betriebssystem keinerlei Auswirkungen. Ebenso müssen Systeme mit unterschiedlichen Prozessortaktraten sich auf die gleiche Zeitbasis aufsynchronisieren können

und die Synchronisation halten, auch wenn deren Taktrate schneller ist als die vom Master.
In der DREAMS Architektur ist auch eine Rekonfiguration für den Ausfall oder Fehler eines Cores vorgesehen. Hier ist die Vorgehensweise ähnlich der des SCARLETT Systems. Der Zustand einer Partition wird periodisch überwacht und an eine Globale Management Einheit geschickt. Sollte eine Partition ausfallen, so wird unterschieden, ob der Multi-Core Prozessor sich lokal selbst rekonfigurieren kann, um eine minimale Funktion zu gewährleisten, oder ob global agiert werden muss. In ersteren Fall werden die Tasks von einem Gast Betriebssystem auf ein anderes zu einem funktionierenden Core migriert, wobei die Tasks das Multi-Core System nicht verlassen. Damit dies nach einem gewissen Schema abläuft und die wichtigen Tasks weiterlaufen, übernimmt ein lokaler, auf dem Hypervisor befindlicher Ressourcen Manager diesen Vorgang. Wie das System bei der Rekonfiguration vorgeht und welche Tasks als wichtig angesehen werden, ist nicht bekannt. Ist es nicht möglich lokal zu agieren, wird ein globaler Ressourcen Manager eingeschaltet, der die Rekonfiguration übernimmt. Auch hier ist nicht bekannt, wie die Rekonfiguration abläuft und wie der globale Ressourcenmanager zu einer neuen Taskverteilung gelangt.
Da das Projekt noch bis September 2017 läuft, werden erst zu einem späteren Zeitpunkt konkrete Ergebnisse im Bereich der Rekonfiguration vorliegen [138].

### ARAMiS

Im System **A**utomotive, **R**ailway and **A**vionics **M**ulti**i**core **S**ystems (ARAMiS) wird der Einsatz von Multi-Core Systemen in den Bereichen Automotive, Avionik und Bahn untersucht [32]. Dabei steht die Vernetzung von Embedded Systemen im Vordergrund. Zuvor hatten sich schon mehrere Systeme mit der Einführung von Multi-Core Systemen in dieser Domäne auseinandergesetzt. Darunter MULTIpartes, welches mit Hilfe von Virtualisierung die Kostenreduktion von Multi-Core Systemen untersucht hat. Ebenso hatte das System RECOMP, welches die Virtualisierung von Multi-Core Steuergeräten als Forschungsziel betrachtet hatte, eine zusätzliche Kommunikationsschicht, um Tasks zwischen virtualisierten Cores kommunizieren zu lassen (siehe DRE-

AMS). Aus dem System CERTAINTY ging als Erkenntnis hervor, dass die Virtualisierungen auf Multi-Core Steuergeräten die Performance von Funktionen und die Kosteneffizienz einschränken. Daher wurde eine neue Architektur vorgeschlagen, welche nicht auf Virtualisierung basiert und dennoch in den Zieldomänen eingesetzt werden kann. Ein Ergebnis dieses Systems wird von dem Projektpartner Kalray in Form eines Prozessors vorgestellt. Dieser Chip besitzt 288 Cores und ist ein sog. Many-Core Prozessor. Die Cores sind voneinander unabhängig und werden separat programmiert. Wie dieser Prozessor eingesetzt wurde und eventuelle weitere Ergebnisse, sind nicht bekannt. Ebenso wurde kein Versuchsträger aufgebaut [26].
Das ARAMiS System hat mehrere Versuchsträger als Resultat. Ein aus dem ARAMiS System hervorgegangenes Tool ist das CADMOS Analysewerkzeug. Hier wurde untersucht, ob die Funktionen, die auf einer einzelnen ECU allokiert waren, auf einer Multi-Core ECU, dem Infineon Tri-Core, portiert werden können. Das Tool sollte dabei unterstützen, eine korrekte statische Allokierung von Tasks im System zu finden unter verschiedensten Eingangsparametern. Für diese Eingangsparameter wurden Modelle entwickelt in Bezug auf Hardware und Software, um so die optimale Verteilung von Tasks auf Cores zu finden. Insbesondere wurde hier das Timing berücksichtigt, insbesondere welche Tasks wann geschedult werden und ob dies mit den Modellen vereinbar ist. Das Resultat zeigt, dass es durchführbar ist und auch so mit Serienhardware umgesetzt werden kann [116].
Ein Demonstrator, der im System ARAMiS entwickelt wurde, war der Virtualized Car Telematics - Simulator von BMW. Aufbauend auf einer Core i7 Plattform wurden mit Hilfe des WindRiver Hypervisors die Cores unterteilt. So wurde für die graphische Darstellung für den Fahrer zwei Cores des i7 Prozessors für das Gast Betriebssystem Ubuntu benützt (Abbildung 3.2). Andere Funktionen auf diesem Prozessor wurden unter dem Gast Betriebssystem Android und einem Linux Server untergebracht. Diese Funktionen untereinander wurden zeitlich und räumlich getrennt[129].

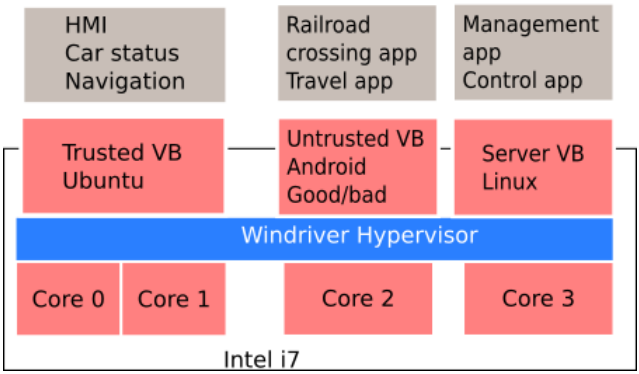



Abbildung 3.2: Hypervisor im Virtualized Car Telematics - Simulator

## Hypervisor oder Container

Wie schon das System CERTAINTY herausgestellt hat, ist die Benutzung von Hypervisoren fragwürdig. Schon relativ früh wurde der Hypervisor in Frage gestellt und ein Ausblick auf zukünftige Technologien gegeben. So sollten laut R. Johnson und M. Loes [98] die Verwendung von Hypervisor eingestellt und nur noch ein Basisbetriebssystem verwendet werden. Tasks, die nun abgetrennt laufen müssen, sollen in einem „Source Wrapper“ eingekapselt werden und nicht mehr in einem eigenen Betriebssystem für sich laufen. So werden die zusätzlichen Betriebssystem Schichten des Gast Betriebssystemes, die bei einer Virtualisierung bereit gestellt werden müssen, nicht mehr benötigt. Dies führt zu einer Performancesteigerung, da nicht zusätzliche administrative Software ausgeführt werden muss[107].
Aktuell wurden und werden diese „Source Wrapper“ entwickelt unter sog. Containern. Insbesondere die LinuxContainers sind hier zu nennen. Deren Ziel ist es, eine Funktion in sich aufzunehmen und eine Umgebung zu bieten, welche so nah wie möglich in einem virtualisierten Gast Betriebssystem zur Verfügung steht. Dabei können mehrere Gast Betriebssysteme emuliert werden, je nach Bedarf der Funktion. In Linux wird hierfür das Paket **L**inu**x** **C**ontainers (LXC) verwendet.

Wie ein Hypervisor können hier die Funktionen auf einen Core begrenzt werden, sowie Speicherzugriffsmechanismen und Netzwerktätigkeiten eingeschränkt werden [38].

## 3.3 Das HAMS System

Eine Forschungsarbeit, welche sich mit deterministischer Rekonfiguration in einem Multi-Core System beschäftigt, ist der Hierarchical Asynchronous Multi-Core System (HAMS) Scheduler. Dieser Scheduler basiert auf einem Linux Kernel und dessen Completely Fair Scheduler (CFS) von Ingo Molnár ab dem Linux-Kernel mit der Version 2.6.23 und dem RT-Preempt Patch. Die Tatsache, dass der HAMS Scheduler ein deterministisches Rekonfigurationsschema in einer Multi-Core Umgebung beherrscht und zudem noch mit Laufzeitveränderungen der Tasks umgehen kann, machen ihn interessant für die Knowledgebase und das semi-statische Scheduling [65] [96] [95] [131].

### Allgemeiner Aufbau des HAMS

Der HAMS Scheduler hat zwei Besonderheiten. Zum einen ist er hierarchisch aufgebaut und zum anderen schedult er das System asynchron. Die hierarchischen Schichten sind in Abbildung 3.4 ersichtlich. Die erste Ebene ist die Hardware selbst. In einem Multi-Core System ist die unterste Ebene ein Core. Auf diesem Core läuft der First Level Scheduler (FLS), welcher die erste Software Ebene des HAMS Schedulers darstellt. In einem Multi-Core System mit beispielsweise vier Cores sind auf dem gesamten Prozessor vier FLS Scheduler zu finden, also einen für jeden Core. Die höchste Software Ebene ist der Second Level Scheduler (SLS). Dieser läuft nur auf einem Core im System. Die Aufgaben von FLS und SLS sind wie folgt [95]:

**FLS**

Der FLS ist der Scheduler des aktuellen Cores und sorgt dafür, dass der nächste Task, ausgehend vom Scheduling Algorithmus, geschedult wird. Neben dieser Aufgabe hat der FLS noch zwei weitere. Der FLS ist dafür verantwortlich, dass der Tick auf diesem Core verwaltet wird. Der Tick sorgt dafür, dass Tasks in

periodischen Abständen unterbrochen werden, damit der FLS untersuchen kann, ob nicht ein höher priorer Task rechenbereit ist. Ist das der Fall, wird der aktuelle Task unterbrochen und der wichtigere Task gewinnt den Core. Eine weitere Aufgabe des FLS ist die Kommunikation mit dem SLS. Der FLS empfängt Befehle vom SLS und führt diese gemäß einem bestimmten Schema aus. Der FLS ist asynchron ausgelegt und jeder FLS bekommt seinen eigenen Tick zugewiesen. Der Tick kann somit auf einem Core 2ms betragen und auf dem anderen 1ms. Je nach Art des Multi-Core Systems können die Ticks dennoch miteinander synchronisiert sein. Dies hängt vom Taktgeber der Cores ab. Besitzen alle Cores einen gemeinsamen Taktgeber, so sind sie über einen gemeinsamen Teiler synchronisiert. Dies ist nicht der Fall, wenn der Multi-Core Prozessor getrennte Taktgeber für die Cores bereitstellt.

**SLS**

Der SLS ist der sog. Dispatcher im System. Der SLS hat den Überblick über alle im System laufenden Tasks, deren Eigenschaften und deren aktuelle Corezuweisung. Dazu kommuniziert der SLS mit allen FLSen auf dem Multi-Core Prozessor System und empfängt deren gesammelten Daten und Änderungswünsche. Sollte der SLS feststellen, dass die aktuelle Systemkonfiguration geändert werden muss, kann dieser durch das globale Wissen einen Zustandsvergleich durchführen und das System rekonfigurieren. Mit Hilfe seines deterministischen und festen Rekonfigurationsschemas aus Abbildung 3.3 kann das System zur Laufzeit den neuen Gegebenheiten angepasst werden.

## Tasks, Phasen und Taskkommunikation

Der HAMS Scheduler ist für weiche Echtzeitsysteme in den Domänen Automotive und Avionik entwickelt worden. Somit ist er für Tasks konzipiert, die auf diesen Systemen laufen. Durch die Verwendung von Linux und dessen Distributionen können schon, wie in ARAMIS, vorhandene und bereits verfügbare Tasks wiederverwendet werden. Zudem ist es, soweit bekannt, nur in HAMS möglich, dass Tasks ihre Laufzeitanforderungen dynamisch zur Laufzeit ändern können. Aktuelle Systeme

aus der Forschung sind statisch aufgebaut und es gibt keine Mechanismen, die mit sich verändernden Tasks umgehen können [131]. Im HAMS Scheduler haben Task Phasen, die einem bestimmten Zustand entsprechen. Um dies zu verdeutlichen kann das Einführungsbeispiel der Tasks „Geschwindigkeitsregelanlage“ und „Einparkhilfe“ herangezogen werden, Abbildung 3.3. Hier verändern sich die Laufzeiten der Tasks, wenn das Fahrzeug in einer bestimmten Phase, z.B. „Einparken“ ist. Da auf diesem System mehrere Tasks mit unterschiedlichen Phasen und Laufzeitabhängigkeiten in einer normalen Konfiguration vorhanden sind, hat der SLS als aktuellen Zustand eine sog. Systemphase definiert. In dieser Systemphase sind alle Phasen aufgelistet und die aktuellen Zustände markiert.

## Ablauf einer Phasenänderung und Taskmigration

Der Ablauf und die Meldungen einer Phasenkonfiguration sind im HAMS Scheduler durch die HAMS API (HAPI) definiert und fest vorgegeben [131] [65]. So meldet, wie in Abbildung 3.3, ein Task eine Phasenänderung an den FLS auf dem aktuellen Core. Sobald dies geschehen ist, geht dieser Task in einen Wartezustand über. D.h. der Task wird nicht mehr vom lokalen FLS geschedult und wartet solange, bis seine Phasenänderung akzeptiert und ausgeführt wurde. Der FLS meldet daraufhin die Phasenänderung an den SLS. Dieser wertet den Befehl auf Validität hin aus und berechnet eine neue Verteilung. Wurde diese Verteilung gefunden, wird vom SLS ein Zustandsvergleich durchgeführt. Dieser sendet anschließend die entsprechenden Befehle an die FLSe der Cores. Bei deren nächsten Aufruf werten diese die Befehle aus und führen diese entweder sofort aus oder warten bis zu einem bestimmten Zeitpunkt. Dies bedeutet, dass sich die FLSe aller Cores kurzzeitig aufsynchronisieren, um simultan eine Phasenänderung durchzuführen. In Abbildung 3.3 bestimmt der FLS mit der niedrigsten Frequenz den Zeitpunkt (andere Vorgehensweisen sind möglich). Sobald alle FLSe diesen Zeitpunkt erreicht haben, wird die Phasenänderung ausgeführt. Dies kann auch bedeuten, dass eine Migration der Tasks, wie in Abbildung 3.3, vorgenommen werden muss.

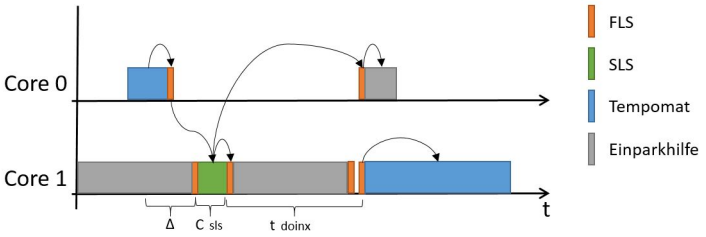



Abbildung 3.3: HAMS Phasenänderung im asynchronen System

## Integration einer Knowledgebase

In den Zieldomänen wird verlangt, dass der Zustand des Systems zu jeder Zeit bekannt ist. In statischen, sich nicht verändernden Systemen ist das der Fall. Da der HAMS Scheduler für sich ein dynamisches System darstellt, das auf Änderungen während der Laufzeit reagieren muss, muss Wissen darüber geschaffen werden, wie er zu Rekonfigurieren hat. Dabei kann auf zwei unterschiedliche Arten vorgegangen werden:
Zum einen, kann der HAMS Scheduler Daten zur Laufzeit sammeln oder sich diese von den Tasks mittels einer Schnittstelle senden lassen. Anhand dieser Daten muss der HAMS Scheduler nun eine lauffähige Konfiguration finden sobald das System startet oder eine Phasenänderung angefragt wird. Dabei muss er sich den vorherrschenden Algorithmen, wie dem Devis Test, dem CTA Algorithmus oder einer anderen Kombination von Bin Packing und Auslastungstests, bedienen. Diese Berechnungen benötigen sehr viel Zeit und haben einen sich ändernden Ausgang.
Die zweite Möglichkeit ist, dem Scheduler eine „Knowledgebase“ aller möglichen Konfigurationen mitzugeben und diese bei jeder Phasenänderung heranzuziehen. Diese Vorgehensweise ist aus der Forschung bekannt und angewandt worden mit dem „Fuzzy Rule-Based Real-Time Scheduler“ von Lee et al. in [69] und dem RTSA Tool von Göttge et al. [74]. Bei diesen Pionieren der Knowledgebaseanwendung im Scheduling wurden

komplexe Berechnungen durch Wenn Dann-Vergleiche oder linguistische Befehle ersetzt (Kapitel 3.1). Somit bedeutet eine Knowledgebase in HAMS, dass jegliche online Berechnungen entfallen und die SLS Rekonfigurationstätigkeit reduziert sich auf einen Zustandsvergleich. Damit sind der HAMS Scheduler und die Knowledgebase eine ideale Kombination für ein semi-statisches und deterministisches Scheduling. Mit Hilfe dieser Kombination kann angestrebt werden, die Ziele der Knowledgebase aus Kapitel 1.2 zu erreichen. Wird die Knowledgebase dem HAMS Scheduler hinzugefügt, ist die Aufteilung in Abbildung 3.4 zu erkennen. Der HAMS Scheduler besteht aus FLS, SLS. Die Knowledgebase ist eine eigene Datei, die vom SLS ausgelesen wird und unveränderlich ist. Das HAMS System ist der HAMS Scheduler und die HAMS Knowledgebase in Kombination [65].

## 3.4 Zusammenfassung

Die Zusammenlegung von Funktionen auf Multi-Core Steuergeräten wurde in der Forschung so wie in der Industrie sehr intensiv untersucht. Verschiedenste Multi-Core Schedulingalgorithmen und Multi-Core Scheduling Varianten wurden aufgestellt, um die Auslastung zu erhöhen. Die erhofften Resultate, Signifikant mehr Tasks ohne Core Grenzen und mit wenig Aufwand unterzubringen, blieben aber aus (Kapitel 3.1). So wurde durch rechenintensive Algorithmen, wissensbasiertes Scheduling, Vorabanalysen und Timing Analysen versucht, Probleme im Scheduling zu finden und diese entweder zu verhindern oder zu vermeiden. Durch Industriebeteiligungen bekamen diese Bemühungen noch eine weitaus größere Bedeutung. Mit IMA 1 $^{\text{gen}}$ und dessen Ausbaustufe IMA 2 $^{\text{gen}}$ bekamen Hypervisoren mehr Bedeutung in der Avionik. Dieser Trend ging auch auf die Automobilelektronik über mit dem Effekt, dass Hypervisoren in den verschiedensten Systemen der Forschung eingesetzt werden. Die Lizenzkosten für einen Hypervisor in der Automobilindustrie und die gleichzeitige Entwicklung von Containern zeigt aber einen Trend weg von Hypervisoren und deren Ressourcenbedarf hin zu ressourcenschonenderen Lösungen auf.
Neben Timing Analysen ist auch die Rekonfiguration von Systemen ein Bereich der Forschung in den Zieldomänen. So wurden auch hier verschie-

denste Systeme entworfen, welche eine Rekonfiguration, gesteuert von einem zentralen Punkt, unter bestimmten Bedingungen (z.B. Wartungsmodus) zulassen. Um aber mehr Software auf einem Steuergerät laden zu können, müssen Steuergeräte zur Laufzeit semi-statisch rekonfiguriert werden. Diese Möglichkeit bietet der HAMS Scheduler. Mit dessen hierarchischem Aufbau und einem deterministischen Migrationsschema in Multi-Core Systemen stellt er die Basis für die Rekonfiguration in einem semi-statischen System dar. Um nun gezielt rekonfigurieren zu können, enthält das HAMS System eine HAMS Knowledgebase. Damit das HAMS System in die Zieldomänen Automotive und Avionik eingesetzt werden kann, müssen bei der Erstellung der Knowledgebase die Anforderungen aus diesen Domänen an ein solches System ermittelt werden.

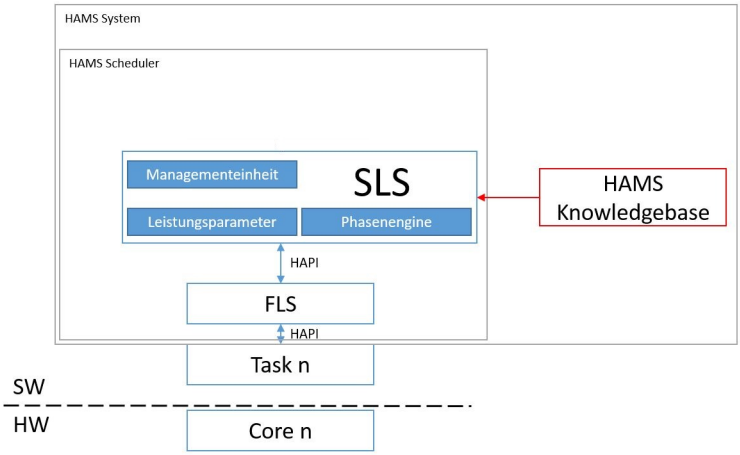



Abbildung 3.4: Das HAMS System im Überblick

# 4 Analyse von Anforderungen aus den Zieldomänen

Um eine Knowledgebase (KB) für ein phasenorientiertes RTOS wie das HAMS System erstellen zu können, muss zuerst eine Analyse stattfinden. Diese Analyse muss klarstellen, welche Anforderungen eine HAMS Knowledgebase für ein phasenorientiertes System in den Bereichen Scheduling/Allokierung, Rekonfiguration und Fehlerfallbehandlung zu erfüllen hat, um in den Zieldomänen eingesetzt werden zu können. Da es schon RTOS Systeme gibt, welche in den Zieldomänen aktiv verwendet werden, müssen diese zuerst untersucht werden. Aus deren Einsatzkonfiguration und dem Umgang mit Tasks im System können so Grundanforderungen für die KB hergeleitet werden. Darüber hinaus muss auch die eingesetzte Hardware betrachtet werden, um Rückschlüsse ziehen zu können, für die Hardwareanforderungen an die KB. Insbesondere sind hier die Multi-Core CPUs der Zieldomänen zu analysieren. Diese Analyse führt zu drei Anforderungskatalogen aus Automobilelektronik, Avionik und Multi-Core Hardware, welche mit den Zielen der KB aus Kapitel 1.2 abgestimmt werden müssen.

## 4.1 Analyse zieldomänenspezifischer RTOS Systeme

### 4.1.1 Anwendungsbereich in der Automobilelektronik

ECUs der Automobilindustrie werden in verschiedenen Domänen gegliedert. Die drei wichtigsten Domänen sind die Powertrain Domäne, u.a. mit der Steuerung des Motors, der Abgasregelung und Schaltung, die Chassis Domäne, in der sich die Fahrerassistenzsysteme und weitere Komfortsysteme befinden und die Body Domäne, mit z.B. der

Steuerung des Innenlichts und der Sitze [115, Kapitel 2]. Für diese Domänen gibt es hauptsächlich zwei Standards die zum Einsatz kommen. Der am meisten genutzte Standard ist **AUT**omotive **O**pen **S**ystems **AR**chitecture (AUTOSAR), das zurzeit in der Version 4.2.2 vorliegt [33]. Ein schon länger existierender Standard in der Automotive Domäne ist **O**ffene **S**ystem und dessen Schnittstellen für die **E**lektronik im **K**raftfahrzeug (OSEK). Diese Standards sind vielfach von Firmen in proprietären RTOS Systemen umgesetzt. Für eine Analyse müssen folgende Punkte untersucht werden:

- Die verwendeten Schedulingalgorithmen
- Ob und wie diese Tasks überwacht werden
- Ob und unter welchen Voraussetzungen eine Rekonfiguration möglich ist
- Ob Multi-Core Systeme eingebunden werden
- Wie eine Fehlerfallbehandlung aussieht

Im Folgenden werden die unterschiedlichen Standards und RTOS Systeme hinsichtlich dieser Punkte für die KB analysiert.

### OSEK

OSEK/ Vehicle Distributed Executive (OSEK/VDX) ist ein Standard für den Automotive Bereich. Das OSEK-OS ist der Vorgänger von AUTOSAR und wurde 1994 von dem OSEK Gremium entworfen. Die Ziele dieses OS sind ähnlich denen des heutigen AUTOSAR, nämlich eine einheitliche Plattform zu schaffen, um Code wiederverwenden zu können. Mit dem Aufkommen von AUTOSAR und dessen Layered Structure sowie dessen Speicherschutzmechanismen, hat OSEK immer mehr an Bedeutung verloren, bis es schließlich ganz eingestellt wurde. Nichtsdestotrotz bilden die Mechanismen von OSEK/VDX und OSEK/Time die Grundlage im Scheduling für den AUTOSAR Standard, die bis heute wirken [115, Kapitel 7].
OSEK/VDX wurde für lose Verbunde von Echtzeitsystemen in der Powertrain- und Chassis Domäne konzipiert. Das Scheduling basiert auf Prioritäten, wobei die höchste Priorität den Prozessor gewinnt. Das

Schedulingverhalten kann preemptiv, nicht preemptiv oder ein Mix aus beiden sein (FIFO-Prinzip) [51]. So können auch bei diesem OS die Auslastungstests RMS und DMS angewandt werden für preemptives Scheduling und die zyklischen Auslastungstests für nicht preemptives Scheduling (OSEK/Time). Eine Besonderheit von OSEK ist die Klassifizierung von Tasks. Hier gibt es zum einen komplexe Tasks, die abhängig von Interrupts, Events oder anderen Ressourcen sind, und erst dann ausgeführt werden können, sobald ein bestimmtes Ereignis eingetreten und abgearbeitet worden ist.
Zum anderen gibt es einfache Tasks, welche von keinen anderen Ressourcen oder Tasks abhängig sind. Ebenso ist die Handhabung von Preemption in diesem System von Bedeutung. So kann die Abarbeitung von „maskable“ Interrupts verschoben werden, bis diese benötigt werden. Alle Verhaltensweisen und Prioritäten werden in OSEK Implementation Language (OIL) Dateien abgelegt, die das Systemverhalten statisch beschreiben. Wie in Abbildung 4.1 zu erkennen ist, werden in OSEK/VDK Tasks mit dem für das Scheduling relevanten Parameter Priorität und der Abhängigkeit von Events versehen. Die Prioritäten werden somit zur Designzeit statisch festgelegt, ebenso wie die Anzahl der Tasks im System. Diese Werte sind zur Laufzeit nicht mehr veränderbar. Damit wird sichergestellt, dass das System sich immer entsprechend den Vorgaben verhält und deterministisch bestimmbar ist. Eine Rekonfiguration ist somit nicht angedacht. Im Falle eines Task Fehlers, wird wie in AUTOSAR nur der Task mit der höchsten Priorität korrekt geschedult bei preemtiven Scheduling [51]. Multi-Core Parameter gibt es in OSEK nicht, da diese Hardware Entwicklung zu neu ist für das beendete OSEK-OS. Der Zusammenschluss von zeitlich sensitiven ECUs benötigt allerdings ein anderes Schedulingverfahren als OSEK/VDK anbietet. Hierfür wurde OSEK um OSEK/Time erweitert, welches auch heute die Grundlage für TIMEX in AUTOSAR bildet. So soll es Tasks, die hart echtzeitfähig sein müssen und zuverlässig einen Output innerhalb eines gewissen Zeitraumes generieren, gelingen, diesen einzuhalten. In OSEK/Time ist es den Tasks nicht erlaubt, auf bestimmte Ereignisse aktiv zu warten. D.h. sollte ein Interrupt oder ein anderer Input benötigt werden und dieser nicht zur Verfügung stehen, so muss der Task dennoch rechnen. Um einen Task immer in einer gewissen Zeitspanne rechnen zu lassen ist ein cyclischer Schedule notwendig.

```
TASK{
    UINT32 PRIORITY;
    UINT32 ACTIVATION;
    ENUM [
        NON,
        FULL
    ] SCHEDULE;
    EVENT_TYPE EVENT[];
    RESOURCE_TYPE RESOURCE[];
    MESSAGE_TYPE MESSAGE[];
    BOOLEAN [
        TRUE {
            APPMODE_TYPE APPMODE[];
        },
        FALSE
    ] AUTOSTART;
};
```

Abbildung 4.1: Konfigurierungsmöglichkeit eines Task für OSEK/VDK in der OIL Datei [35]

So werden in OESK/Time keine Prioritäten angegeben, sondern Zeitabschnitte. Die Planung von Interrupts und Events muss auch cyclisch erfolgen, damit sie sich nicht mit anderen Tasks überschneiden. Sollte das nicht der Fall sein, geht das System in einen Fehlerzustand über. Im Falle von OESK/Time ist dies ein kompletter Neustart des Systems [115, Kapitel 7].

### AUTOSAR

Der AUTOSAR Standard wurde im Jahr 2003 vom AUTOSAR Konsortium ins Leben gerufen. Ziel war es, die Komplexität bei der Entwicklung von ECUs zu verringern, indem die Basis Software (BSW) und die Schnittstellen zu Bussystemen und Tasks standardisiert werden sollten, um so Wiederverwendbarkeit zu gewährleisten. Bei dem Entwurf des AUTOSAR Standards wurden verschiedene Phasen verabschiedet, von der „proof of concept“ Phase mit dem Release 1.0 bis hin zu 2.1 und der aktuellen Version 4.2.2 (seit 2013). Um das Ziel der Wiederverwendbarkeit zu gewährleisten wurde eine „Layered Architecture“ eingeführt, welche die Tasks im Application Layer über ein Runtime Environment

von der BSW trennt. Eine Eigenschaft des AUTOSAR Standards (und auch OSEK) ist, dass die Konfiguration der Schnittstellen offline durchgeführt wird. D.h. es ist zur Laufzeit nicht mehr möglich, zum Beispiel die Busgeschwindigkeit einer CAN Leitung zu verändern. In (Extensible Markup Language) - Erweiterbare Auszeichnungssprache (XML) Dateien wird das AUTOSAR RTOS vorkonfiguriert und bei der Kompilierung werden diese Konfigurationen mit einbezogen und unveränderbar in die auszuführende RTOS Datei gelinkt [36].
Auf AUTOSAR basierende RTOS Systeme werden in allen Domänen eingesetzt. So laufen zeitkritische Tasks der Powertrain Domänen auf diesem System, welche harte Echtzeit erfüllen müssen, sowie nicht zeitkritische Chassis und Body Domänen Tasks. Um zeitkritische Software Funktionen in AUTOSAR integrieren zu können, müssen die erforderlichen Scheduling Algorithmen angeboten werden. Im AUTOSAR Standard wird das Scheduling aus Kompatibilitätsgründen zu OSEK über Prioritäten mit Preemtion realisiert. D.h. der Task mit der höchsten Priorität gewinnt den Prozessor. Die Prioritäten werden statisch zur Designzeit vergeben unter Anwendung der Auslastungstests Rate Monotonic Scheduling (RMS) und Deadline Monotonic Scheduling (DMS) oder anderen, auf statischen Prioritäten basierenden Auslastungstests. Diese Prioritäten werden in Autosar XML (ARXML) Dateien abgelegt und bei der Kompilierung des Systems mit einbezogen [142]. Ebenso unterstützt AUTOSAR ein cyclisches Scheduling mit sog „scheduling tables“. So werden Tasks zu bestimmten Zeitpunkten geladen bzw. Aktionen zu bestimmten Zeitpunkten durchgeführt. Auch diese zeitlichen Einstellungen sind zur Designzeit festgelegt und nicht mehr veränderbar [71].
So sind nach der Kompilierung eines auf AUTOSAR basierenden RTOS Systems für diese ECU die Einstellungen bezüglich der Tasks nicht mehr änderbar. Aus dieser Eigenschaft ergibt sich auch, dass das System zur Laufzeit nicht veränderbar ist und somit auch keine Rekonfiguration stattfinden kann. Das bedeutet auch, dass es in AUTOSAR nicht möglich ist, dynamisch zu schedulen wie bei einem EDF Algorithmus. Auch fehlen die Möglichkeiten zur Überwachung der Taskausführung zur Laufzeit oder sind eingeschränkt verfügbar. In aktuellen Versionen ist die TIMEX Erweiterung verfügbar [126] [34]. Diese Erweiterung der AUTOSAR BSW lässt es zu, dass Funktionen und deren Laufzeit

überwacht werden können. So können Tasks durch Rückmeldung an AUTOSAR genau definiert werden, beispielhaft dargestellt in Abbildung 4.2. Anhand der Rückmeldungen kann überwacht werden, wie lange Tasks insgesamt laufen, welche Teilabschnitte (sog. Runnables in AUTOSAR) sie beinhalten, in welcher Reihenfolge die Teilabschnitte abgearbeitet werden müssen und welche Zeit diese Teilabschnitte benötigen. Mit der Rückmeldung einer Zeitmessung, ob ein Teilabschnitt erfolgreich beendet wurde, kann der korrekte Ablauf von Tasks erfasst werden und somit auch eine Überwachung zur Laufzeit generiert werden. Das sich daraus ergebene Scheduling im Fehlerfall ist wieder auf die Prioritäten zurückzuführen. Sollte der fehlerhafte Task durch die TIMEX Erweiterung an der Ausführung nicht gehindert werden, so wird die Fehlerbehandlung im Sinne des Prioritäten Schedulings abgehandelt. Das bedeutet, der Task mit der höchsten Priorität gewinnt die CPU. Dies stellt jedoch lediglich sicher, dass im Fehlerfall der Task mit der höchsten Priorität sicher ausgeführt wird.

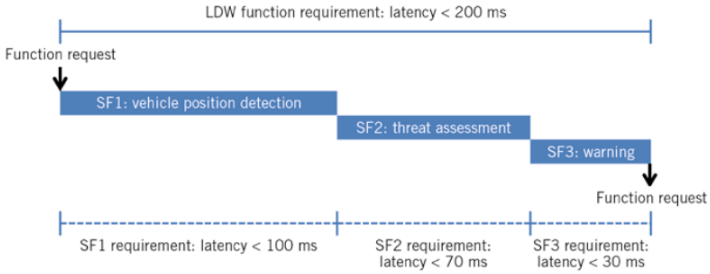



Abbildung 4.2: Teilabschnitte eines Spurverlassenstasks (Lane Departure Warning) und die Teilbereiche in AUTOSAR [126]

Auch unterstützt der AUTOSAR Standard Multi-Core Prozessoren seit der Version 4. Analog zu der Prioritätenzuweisung von Tasks werden im Multi-Core System Tasks statisch einem Core zugewiesen. Somit können die Tasks nicht zwischen Cores frei migrieren.

Das Scheduling ist im Fehlerfall nicht ausreichend und unterliegt Verbesserungen. Auch die Timing Analyse bzw. Überwachung zur Laufzeit von AUTOSAR erfordert eine Erweiterung. So sollten das zeitliche Verhalten von Tasks und noch weitere Laufzeitparameter über den Task enthalten sein. Um dies dennoch im aktuellen AUTOSAR Release zu realisieren, müssen die Analysen vorab durchgeführt werden und sich in den Prioritäten und der Zuweisung von Teilabschnitten in den Tasks widerspiegeln. Zu dieser Vorabanalyse werden die vier Bereiche Software Komponenten (Teilabschnitte und Tasks), Hardware Topologie, BSW Konfiguration und Systemzusammenspiel begutachtet. Im Falle von Software Komponenten müssen unter anderem die Zeiten betrachtet werden, wie WCET Periode und Deadline, mögliche Interrupts und deren zeitliches Verhalten sowie Preemtion von Tasks. Für die Hardware muss insbesondere auf die CPU eingegangen werden. Auch muss die CPU Geschwindigkeit berücksichtigt werden sowie die Hardware Interrupts die benötigt werden. Im Systemzusammenspiel ist vor allem der Kommunikation zwischen Systemen, die je nach Bussystem periodisch oder aperiodisch kommen muss, Beachtung zu schenken.

#### Anforderungskatalog Automobilelektronik

Die Analyse der RTOS Systeme aus der Automotive Domäne gibt konkrete Anforderungen, was eine KB als Basis für ein phasengesteuertes HAMS System unterstützen muss, um im Automotiv Bereich eingesetzt werden zu können. Insbesondere sind hier die RTOS Eigenschaften im Scheduling, bei der Überwachung, bei der Fehlerfallbehandlung und der Rekonfiguration zu betrachten.

#### Schedulingalgorithmen

Das Scheduling in der Automotive Domäne basiert auf Prioritäten. Jeder Task bekommt somit eine eigene statische Priorität zugeschrieben und wird danach im System geschedult. Prioritäten werden mit RMS oder DMS berechnet und in einer Systemdatei abgelegt. Zur Optimierung wird auf Feasibility Algorithmen zurückgegriffen. Das bedeutet, dass Tasks auf ihre zeitlichen Eigenschaften hin überprüft werden und ggf. auf ein anderes passenderes Steuergerät allokiert werden. Da prioritätenbasiertes Scheduling

nicht garantieren kann, dass Tasks immer zur gleichen Periode senden, muss ein anderes Schedulingverfahren eingesetzt werden. So ist es möglich, dass Tasks z.B. periodisch Nachrichten auf den Time Division Multiple Access (TDMA) basierten FlexRay Bus senden können. Tasks sind vom Systemstart an vorhanden und werden erst mit Herunterfahren des Systems beendet. Es kommen zur Laufzeit keine neuen Tasks hinzu.

#### Überwachung

Die Überwachung gewinnt im Automotive Bereich zunehmend an Bedeutung. Mit der TIMEX Erweiterung von AUTOSAR werden Deadlines und Perioden überwacht. Die Überwachung ist sogar so weit fortgeschritten, dass selbst die Ausführung innerhalb des Tasks überwacht wird, so dass zu bestimmten Zeitpunkten Zwischenziele des Tasks erreicht sein müssen. Auch diese Beschreibungen sind statisch und werden dem System in einer Datei mitgegeben.

#### Fehlerfallbehandlung

Doch auch mit einer genauen Beschreibung eines Tasks, wie bei der TIMEX Erweiterung, ist die Fehlerfallbehandlung nicht sehr ausgeprägt. Ein Neustart des Systems wäre bei X-By-Wire fähigen Geräten kontraproduktiv. Auch das sichere Schedulen des höchst prioren Tasks in einem prioritätenbasierten System ist keine erweiterte Fehlerbehandlung, sondern Grundlage von prioritätenbasiertem Scheduling. Ein genaues Konzept fehlt im Automotive Bereich.

#### Rekonfigurationen

Rekonfigurationen gibt es im Automotive Bereich nicht. Alles wird statisch konfiguriert und ist nicht mehr zur Laufzeit veränderbar.

#### Multi-Core Systeme

Multi-Core Systeme sind im AUTOSAR Standard vorhanden und werden aktiv beschrieben. Tasks werden statisch allokiert und jeder Core wird für sich betrachtet.

### 4.1.2 Anwendungsbereich in der Avionik

ECUs der Avionik werden in die Domänen Safety (sicherheitskritisch) und Mission (nicht sicherheitskritisch) gegliedert. In der Safety Domäne befinden sich harte Echtzeitsysteme, die Flugsteuerung und Regelung übernehmen. Hier sind u.a. Fly-By-Wire Systeme angesiedelt, welche die Steuerflächen elektronisch ansteuern [66]. Die Hauptmerkmale dieser Systeme sind korrekte Ergebnisse in einer gewissen Zeit und eine hohe Redundanz.
In der Missions Domäne sind ausschließlich flugunterstützende Systeme angesiedelt, die weder redundant abgesichert sind noch zeitlichen Ansprüchen gerecht werden müssen. Bekannte Betriebssysteme aus der Avionik sind VxWorks und LynxOS, die es je nach Domäne mit verschiedenen Erweiterungen gibt. Eine Analyse dieser zwei Betriebssysteme mündet in den Anforderungskatalog der Avionik. Die zu analysierenden Punkte bleiben gleich wie im Kapitel 4.1.1.

#### VxWorks und LynxOS

Sowohl VxWorks als auch LynxOS sind Echtzeitbetriebssysteme basierend auf Unix. Die Eigenschaft mit POSIX Funktionen umgehen zu können und die sichere Speicherbereichsüberwachung von Tasks haben die Verbreitung in der Avionik gefördert und sind heute Standard. In jedem neuen Flugzeug, z.B. A400M und 787, können je nach OEM eines dieser zwei RTOSe gefunden werden. Auch im Bereich Scheduling ähneln sich diese zwei RTOSe. So findet man die folgenden drei preemtiven Scheduling Vorgehensweisen: (First In -First Out) - der Reihe nach (FIFO), Prioritäten und Round Robin (RR).
Im FIFO Scheduling wird der Task, welcher zuerst sein Bedürfnis zu rechnen am Scheduler angemeldet hat, den Prozessor gewinnen. Dieses Scheduling ist nicht echtzeitfähig und wird daher nicht aktiv eingesetzt. Prioritäten sind auch in der Avionik ein beliebtes Mittel, ein echtzeitfähiges System aufzubauen. Die Prioritäten können wieder mittels RMS oder DMS ermittelt werden. Im Fehlerfall gewinnt der Task mit der höchsten Priorität die CPU.
Round Robin (RR) Scheduling ist ein weiterer beliebter Ansatz, einen echtzeitfähigen Schedule zu erstellen. Im RR Scheduling wird die CPU Rechenzeit für Tasks in bestimmte Zeitabschnitte mit gleicher Größe ein-

geteilt. D.h. jeder Task bekommt den Prozessor für maximal die Länge eines Zeitabschnittes. Wer den Prozessor zuerst bekommt, entscheidet wiederum die Priorität des Tasks, was bedeutet, dass ein Task mit höherer Priorität in seinem Abschnitt zuerst rechnen darf. Gibt es Tasks mit derselben Priorität, so entscheidet das FIFO Prinzip. In VxWorks werden nieder priore Tasks bei ihrer Ausführung unterbrochen, sobald ein höher priorer Task lauffähig ist, auch wenn der Zeitabschnitt des nieder prioren Tasks noch nicht vollendet ist. Die Prioritäten als auch die Größe eines Zeitschlitzes werden vor dem Systemstart mitgegeben und in das ausführbare RTOS kompiliert. Prioritäten sind zur Laufzeit änderbar. Die nötigen Funktionen zur Änderung einer Priorität müssen vom Task selbst aufgerufen werden mit der Angabe der neuen Priorität. Diese RTOSe sind zwar statisch konfiguriert, können aber auf Wunsch dynamisch verändert werden. So ist eine Rekonfiguration in Bezug des Schedulings möglich, wenn auch mit großem Aufwand. In VxWorks ist es möglich, dass Tasks sich selbst andere Prioritäten geben durch Aufrufe der entsprechenden Systemcalls. Dazu muss der Task aber den entsprechenden Code beinhalten.
Beide RTOSe sind Multi-Core fähig und unterstützen sowohl SMP als auch AMP Systeme. In SMP Systemen werden Tasks in VxWorks über die verfügbaren Cores verteilt. Grundlage hierfür ist die Auslastung eines Cores, die zur Laufzeit vom RTOS gesammelt wird. Hat ein Core eine höhere Auslastung als ein anderer, so wird versucht, mit der Migration eines Tasks alle Cores auf die gleiche Auslastung zu bringen. Dabei wird auf Deadlines und andere Task Parameter aktuell keine Rücksicht genommen. Sollte dies nicht gewünscht sein, so muss die Affinität zu einem Core explizit einem Task zugewiesen werden. So wird das RTOS diesen Task nicht mehr verschieben und er ist statisch auf einem Core gebunden. Sollte der Task dennoch migriert werden, so muss der Task selbst dies dem RTOS melden durch Setzen einer neuen Core Affinität. Beim nächsten Scheduleraufruf wird der Task dann migriert. So ist eine Rekonfiguration in Bezug auf Core Affinität und Task-to-Core mapping möglich, aber weder wird dieser Vorgang überwacht noch ist die Implementation einfach gestaltet.
Die Überwachungsmechanismen der Tasks durch das RTOS VxWorks begrenzen sich auf eine Speicherbereichsüberwachung. Die Laufzeiten der Tasks, die Deadlines oder andere Parameter werden nicht überwacht

[140] [39]. Auch hier werden die Tasks zum Systemstart erzeugt und werden erst beim Herunterfahren des Systems beendet.

### VxWorks 653

Um diese Lücke in der zeitlichen Überwachung der Tasks zu füllen, wurden die RTOSe VxWorks 653 und LynxOS 178 entwickelt. Diese RTOSe sind konform mit der ARINC 653 Spezifikation und stellen einen Level 1 Hypervisor dar. In diesen Spezifikationen wird gefordert, dass Tasks zeitlich und räumlich voneinander getrennt sein müssen, um eine wechselseitige Beeinflussung von Tasks zu vermeiden. Dies wird durch sogenannte Partitionen im Scheduler eingestellt. Um eine Partition zu schedulen wird die CPU-Zeit in Zeitabschnitte eingeteilt und jeder Partition ein bestimmter Zeitabschnitt zugewiesen. Einer Partition können ein oder mehrere Tasks zugewiesen werden, so dass auch in einer Partition der Prozess mit der höchsten Priorität die CPU gewinnt. Am Ende des Zeitabschnittes sollten alle Tasks erfolgreich gerechnet haben und auf ihren nächsten Zeitschlitz warten. Ist das nicht der Fall, so beginnt ein Error Handling der Partition. Der fehlerhafte Task wird analysiert und beendet. Ggf. wird die ganze Partition gelöscht, d.h. vom Scheduling ausgenommen.
In ARINC 653 konformen Systemen steht den Tasks eine gesonderte Application Programming Interface (API) zur Verfügung, um das Scheduling aufzubauen. Zuerst werden die einzelnen Zeitabschnitte definiert und den Tasks bzw. Partitionen zugewiesen. Dies geschieht offline, wobei hier die Regeln für das cyclische Scheduling angewandt werden. Sollte ein Zeitabschnitt aus zwei oder mehreren Tasks bestehen, so muss hier zusätzlich die Regel für ein RMS Scheduling in den Partitionen übertragen werden. Anhand dieser Ergebnisse können die Tasks zum Systemstart initialisiert werden [141].

### Anforderungskatalog Avionik

Auch in der Avionik Domäne lassen sich die verschiedenen RTOSe auf ihre Eigenschaften hin analysieren, um wichtige Erkenntnisse für eine KB in diesem Einsatzgebiet zu erlangen. Hier wird ebenfalls der Bereich Scheduling, Überwachung, Fehlerfallbehandlung und Rekonfiguration betrachtet.

#### Schedulingalgorithmen

Der Bereich Scheduling weist in der Avionik Domäne eine größere Vielfalt auf. Das prioritätenbasierte Scheduling ist auch hier von Bedeutung und wird verwendet. Der Einsatz gleicht dem des Automotive Bereichs. Im cyclischen Scheduling kommt aber ein erweitertes Verfahren zum Einsatz. So werden hier nicht nur einzelne Tasks cyclisch geschedult, sondern auch Gruppen von Tasks. Innerhalb dieser Gruppen entscheidet dann wieder die Priorität darüber, welcher Task den Core gewinnt. Diese Gruppen können durch einen Hypervisor zusammengefasst werden und sind für das Basis OS nur als einzelner Task sichtbar. Für diese zwei Schedulingvarianten sind die Feasibility Tests von Bedeutung. Analog zum Automotive Bereich werden auch hier die Tasks und deren Zeiten auf Durchführbarkeit berechnet und ggf. verändert. Ebenso werden dynamische Schedulingvarianten verwendet, wie das RR-Scheduling und das FIFO Prinzip.

#### Überwachung

Das Überwachungsprinzip der Tasks ist im Avionik Bereich deutlicher ausgeprägt als in anderen Bereichen. Neben der Speicherbereichsüberwachung, welche bei Fehlverhalten eines Tasks „löscht“, werden auch Zeiten genau betrachtet. Bei einen cyclischem Schedule wird der Task beendet, sobald dieser seine vorgegebene Zeit überschritten hat.

#### Fehlerfallbehandlung

Die Fehlerbehandlung ist aber beschränkt auf wenige Methoden. So ist die erste Methode, den Task zu löschen. Dies wird vor allem bei einem cyclischen Schedule angewandt, wenn eine zeitliche Verletzung der Partitionen erkannt wird. Ebenso kann der Fehler

ignoriert werden, so dass der Task immer noch geschedult wird, obwohl er bereits einen Fehler begangen hat. So hat man die Möglichkeit, den Task erst einmal weiter berechnen zu lassen und erst später zu reagieren. Im Falle von prioritätenbasiertem Scheduling wird auch hier der höchste priore Task immer mit Sicherheit geschedult.

**Rekonfigurationen**
Eine Rekonfiguration ist in der Avionik nicht vorhanden, so dass Tasks ihre Priorität oder ihren Zeitschlitz in einem statischen Scheduling beibehalten.

**Multi-Core Systeme**
Multi-Core Systeme werden im Avionik Bereich unterstützt. Tasks sind sowohl dynamisch als auch statisch einplanbar nach den entsprechenden Schedulingalgorithmen.

### 4.1.3 Multi-Core Steuergeräte der Zieldomänen

Neben dem RTOS, das auf der ECU läuft, ist die Hardware von gleich großer Bedeutung. Multi-Core Prozessoren weisen eine große Veränderung gegenüber Single-Core Prozessoren auf. Nicht nur, dass hier mehrere Cores auf einem Prozessor verfügbar sind, sondern auch die Funktionalitäten und I/O Möglichkeiten sind erweitert worden. So ist die Komplexität der Multi-Core Prozessoren und deren Handhabung im Vergleich zu Single-Core Prozessoren angestiegen. Diese Komplexitätssteigerung hat auch auf die Allokation einen Einfluss. So müssen mehr Parameter berücksichtigt werden für diesen Vorgang, um eine korrekte Aussage treffen zu können. Um dennoch eine korrekte Aussage über die Taskallokation zu erhalten, muss zuerst das Potential heutiger Multi-Core Prozessoren analysiert werden. Dazu stehen die folgenden Bereiche unter Betrachtung:

- Core
- Speicher
- I/O Geräte
- Firewall / Sicherheitseigenschaften

Wie diese Bereiche in den verschiedenen ECUs in den Zieldomänen vorhanden sind und welche Eigenschaften diese haben zeigt folgende Analyse auf:

### Avionik Multi-Core ECUs

In der Missionsavionik kommen verschiedenste Prozessoren zum Einsatz. Die am häufigsten eingesetzten Prozessoren sind aber die QorlQ Reihe von Freescale/NXP mit dem Vertreter P4080. Ebenso sind Desktop Prozessoren von Intel® wie der Core™ i Serie und der embedded Prozessor Intel® Atom™ anzutreffen.

**Intel** Intel Prozessoren zeichnen sich dadurch aus, dass Tasks ungehindert zwischen den Cores migrieren können. Jeder Core gleicht dem anderen und somit haben auch alle die gleichen Eigenschaften (homogen). Die Stromsparfunktionen der Intel Prozessoren sind sehr stark ausgeprägt. So verändert sich die Taktung der Cores, um dadurch eine geringere Leistungsaufnahme zu erreichen. Für einen Core i5-2520 mit 2,5Ghz sind als Taktrateneinstellung 0.8Ghz, 1.0Ghz 1.5Ghz, 2.0Ghz und 2.5Ghz verfügbar. Der Übergang der Taktfrequenz ist fließend und wird vom OS geregelt durch Setzen der entsprechenden Register. Bei Intel Prozessoren haben alle Cores die gleiche Takteinstellung. Intel Prozessoren haben auch einen gemeinsamen Speicherbereich. So gibt es bei neueren Versionen der Core i Serie privaten L1 und L2 Caches aber einen gemeinsamen L3 Cache sowie RAM, der von allen Cores benützt wird. So sind auch alle Task-Speicherbereiche für jeden Core verfügbar. Die Core i Serie und die Intel Atom Prozessoren sind zuallererst für den Desktop Einsatz gedacht. Aus diesem Grund besitzen die Cores weder I/O Geräte noch eine Firewall und auch keine Eigenschaften, welche die Task Ausführung und das Gesamtsystem absichern. I/O Geräte werden mit dem Data Memory Interface (DMI) Bus via der Northbridge mit dem Prozessor verbunden [30].

**Freescale P4080** Der P4080 besitzt acht identische Cores mit je 1.5 Ghz. Auch hier lässt sich die Taktrate der Cores ändern. Das Besondere an diesem Prozessor ist, dass bestimmte Cores unabhängig voneinander

betrieben werden können. So können Cores unterschiedliche Takteinstellungen haben. Ebenso ist es beim P4080 möglich, dass Cores aus- und eingeschaltet werden können während des laufenden Betriebs. Durch ein Redundanzkonzept ist der P4080 in zwei taktunabhängige Quad-Core Prozessoren unterteilt. Die Speicher sind nicht voneinander getrennt. Neben privaten L0 und L1 Caches ist ein gemeinsamer L2 Cache und RAM vorhanden. So ist es möglich, Tasks zwischen Cores zu migrieren. Ein wichtiges Merkmal ist der Crossbar Switch, der alle Cores und Peripheriegeräte miteinander verbindet. Dieser Crossbar Switch ist sehr schnell dimensioniert, so dass dieser keine negativen Auswirkungen auf die Multi-Core Rechenleistung darstellt. Ebenso sind an diesem Crossbar Switch die Peripheriegeräte angeschlossen. Damit haben die Cores direkten Zugriff auf sämtliche I/O Anwendungen. Eine Firewall für I/O Geräte oder ein weiteres Redundanzkonzept ist beim P4080 nicht vorhanden [17].

### Automotive Multi-Core ECUs

In der Automotive Domäne gibt es ebenfalls zwei verbreitete Multi-Core Prozessoren.

**Infineon Tri-Core** Einer dieser Multi-Core Prozessoren ist der Infineon Tri-Core Prozessor. Dieser zeichnet sich besonders durch seine zahlreichen Sicherheits- und Redundanzeigenschaften aus. Zuallererst besitzt der Tri-Core Prozessor drei Cores. Diese Cores sind aber nicht identisch, d.h. der Prozessor ist nicht homogen. Ein Core besitzt eine höhere Taktrate als andere. Je nach Tri-Core Version gibt es auch Varianten, die drei homogene Cores besitzen. Jeder Core hat seinen eigenen Data Memory Interface (DMI) und Program Memory Interface (PMI) Controller. Mit dem DMI hat jeder Core einen eigenen Scratchpad Speicher. Neben dem Scratchpad besitzt ein Core auch noch einen weiteren Data Cache der im DMI ist. Im PMI ist der DMI Aufbau gespiegelt, aber hier befindet sich der Programmcode getrennt von den Daten. Der Speicherbereich kann unterteilt werden in einen nicht-sicherheitsrelevanten und einen sicherheitsrelevanten Bereich. So können sicherheitsrelevante Tasks ausschließlich in die ihnen zugewiesenen Sicherheitsbereiche schreiben, andere wiederum nur in nicht-sicherheitsrelevante Bereiche. Die I/O

Geräte des Tri-Core SoCs sind nur teilweise direkt mit den Cores verbunden. I/O Geräte, die schnelle Datenverbindung unterstützen, werden direkt mit den Cores über einen Bus verbunden, z.B. Ethernet. Alle anderen I/O Geräte müssen über den Direct Memory Access (DMA) eine Verbindung zu dem Core aufbauen. Dieser DMA weist dann das I/O Gerät durch seine Einstellung einem Core zu. So kann der DMA als eine Firewall dienen und bestimmte I/O Geräte per Core blocken. Dies erschwert aber auch das Migrieren von Tasks zu anderen Cores. So muss nicht nur neben einer manuellen Spiegelung der Daten, notwendig durch nicht vorhandene gemeinsame Speicherbereiche, auch der DMA rekonfiguriert werden, um Cores den Zugang zu I/O Geräten zu ermöglichen.
Der Tri-Core besitzt sog. Checker Cores. D.h. ein Core besitzt einen zweiten Core, der sowohl die gleichen Aufgaben erfüllt, als auch Zugang zu den gleichen Daten hat. Dieser Checker Core rechnet ein paar Takte verzögert und kommt im Normalfall auf die gleichen Ergebnisse. So ist im Tri-Core ein Redundanzkonzept eingebaut, welches die korrekte Code Ausführung und Berechnung gleich mitprüft [28].

**R-Car H2** Ein weiterer Prozessor, welcher im Automotive Bereich eingesetzt wird, ist der Renesas R-Car H2. Der R-Car H2 hat insgesamt neun Cores. Vier Cores davon sind Cortex-A-15 Kerne. Vier weitere Cores davon sind Cortex-A-7 Kerne. Diese beiden Cores unterscheiden sich in der Rechenleistung. Bringen A-15 Kerne eine höhere Leistung bei ebenso höherem Strombedarf, so haben A-7 Cores einen geringeren Strombedarf bei ebenso geringerer Rechenleistung. Trotzdem sind die Tasks zwischen beiden Cores austauschbar, da sie auf einer ARM Architektur aufbauen. Ein weiterer Core ist ein SH-4A Kern mit SuperH Architektur und Reduced Instruction Set Computer (RISC) Befehlssatz. Letzterer ist inkompatibel zu den ARM Kernen, so dass Tasks, die für diesen Core entwickelt wurden, nur auf diesem laufen können. Dafür zeichnet er sich durch eine gute Floating Point Unterstützung aus. Jeder A-7 und A-15 Core hat seinen eigenen L0 und L1 Cache. Alle vier A-15 und alle vier A-7 Kerne besitzen einen eigenen gemeinsamen L2 Cache. Die A-15 und A-7 Kerne sind weitestgehend getrennt voneinander, sie haben nur einen Zugriff auf den gemeinsamen RAM. So ist dieser Prozessor ein Anwärter auf ein mögliches gemischtes AMP-SMP System

(Kapitel 2.2). Wobei es hier zwei AMP Systeme geben kann mit je vier Cores, welche wiederum ein SMP System bilden. Sollte es möglich sein, die ARM Cores zu verbinden, so bildet man ein big.LITTLE Multi-Core System (Kapitel 2.2). In diesem System wird versucht, rechenintensive Tasks auf die A-15 Cores zu legen und dort laufen zu lassen. Alle anderen Tasks bleiben auf den A-7 Kernen. Sollte das System nicht ausgelastet sein bzw. die A-15 Cores nicht benutzt werden, so werden diese ausgeschaltet, um Strom zu sparen.
Im Falle von I/O Geräten gibt es keine Einschränkungen in der Core Verfügbarkeit, wie durch eine Firewall. Ebenso gibt es auch kein Hardware Redundanzkonzept auf diesem Multi-Core Prozessor [49] [84].

### Anforderungskatalog Hardware

Multi-Core Prozessoren und deren Architektur haben einen Einfluss darauf, wie die RTOSe mit ihren Eigenschaften in Scheduling, Überwachung, Fehlerfallbehandlung und Rekonfiguration umgehen können. Die Beeinflussungen sind im Detail im Folgenden aufgelistet:

**I/O Geräte**
: Die Scheduling Vorgehensweise ist rein softwaretechnisch. Die Multi-Core Prozessor Hardware hat keinen Einfluss darauf, welche Scheduling Vorgehensweise verwendet wird. Allerdings hat die Hardware einen großen Einfluss darauf, ob ein Task zum Schedulen auf dem aktuellen Core zur Verfügung steht oder nicht. D.h. die Rekonfiguration ist je nach Hardware und deren Einstellung eingeschränkt.

**Firewall**
: Der kann der Zugriff kann für bestimmte I/O Geräte durch eine Firewall für einen bestimmten Core blockiert sein oder der Core hat überhaupt keine Verbindung zu der I/O Komponente. Dies stellt eine Behinderung in der Rekonfiguration dar, so dass bestimmte Tasks für eingeschränkte Cores ausgeschlossen sind.

**Speicher**
: Eine weitere Einschränkung ist der Zugriff auf gemeinsame Speicherbereiche. Sollten Cores nicht Zugriff auf einen gemeinsamen

Speicher besitzen, wie einen Shared Cache oder RAM, so ist die Rekonfiguration nur unter Umständen oder gar nicht möglich. Hat der Prozessor dennoch einen gemeinsamen Speicher aber eine Hardware Einstellung teilt die Speicherbereiche auf oder das RTOS unterbindet das Schreiben/Lesen von Cores auf softwaretechnische Weise, so ist die Rekonfiguration ebenfalls eingeschränkt.

**Core**

Ebenso ist die Rekonfiguration bei big.LITTLE Prozessoren problematisch. So müssen zur effizienten Ausnutzung des Prozessors rechenintensive Tasks von weniger rechenintensiven Tasks getrennt werden oder eine andere Konfiguration eingesetzt werden, um den Core optimal ausnutzen zu können.
Der Spezialfall Redundanz spielt bei Multi-Core Prozessoren eine Rolle. Sollte ein Task redundant ablaufen, so kann je nach Prozessor auf einen Checker Core zugegriffen werden (Tri-Core oder der Task muss zweimal im System laufen und das am besten voneinander getrennt (P4080). Bei letzterer Variante muss der Systemdesigner selbst die Redundanz im System softwareseitig einführen.

**Sicherheitseigenschaften**

Die zeitliche Überwachung der Tasks ist Aufgabe für das RTOS. Das RTOS hat zu bestimmen, wann ein Task seine Deadline überschritten hat und wann nicht. Je nach Ausstattung des Multi-Core Prozessors können die Ergebnisse (Checker Core) oder die Benutzung der I/O Geräte zur Überwachung herangezogen werden. Das RTOS hat aber immer die Entscheidung zu treffen, was mit einem fehlerhaften Task passieren soll.

## 4.2 Anforderungen an die Knowledgebase

Der HAMS Scheduler mit seinem strukturiertem Rekonfigurationsablauf bietet einem RTOS eine Vielzahl an Möglichkeiten zur Rekonfiguration. Wie die vorherige Analyse zeigt, sind Scheduling- und Rekonfigurationsvarianten aktueller RTOSe auf statische Varianten beschränkt. Der Grund dafür ist die einfache Umsetzung und Vorhersehbarkeit von

statischen Systemen. Zur Beherrschung eines semi-statischen, phasenabhängigen HAMS Systems wird eine statische KB dem HAMS Scheduler zur Verfügung gestellt. Damit HAMS und die KB in den Zieldomänen als RTOS eingesetzt werden können, muss das KB Konzept gewisse Eigenschaften mit sich bringen und folgende Anforderungen in Scheduling, Rekonfiguration, weiche Echtzeit/Laufzeitberechnungen und Überwachung erfüllen, um erfolgreich eingesetzt werden zu können.

## Task Allokierung / Scheduling

Um Tasks gemäß der weichen Echtzeit schedulen zu können, muss die KB die gängigen Scheduling Algorithmen aus den Zieldomänen unterstützen. Der erste Schedulingalgorithmus ist ein prioritätenbasierter Algorithmus. Mit Prioritäten lassen sich die am häufigsten angewandten Auslastungstests von RMS und DMS im Prinzip übernehmen und so lässt sich eine konkrete Aussage über das Einhalten der Task Deadlines treffen. Die bereits bestehende und aktive Verwendung eines prioritätenbasierten Schedulings in VxWorks, OSEK usw. zeigt die Bedeutung und Reife dieser Vorgehensweise in den Zieldomänen auf. So können sowohl weiche periodische Tasks geschedult werden, als auch aperiodische oder sporadische Tasks unter Einbindung der korrekten Feasibility Tests, wie aus den Anforderungen der Zieldomänen zu entnehmen ist.
Ein weiterer und wichtiger Scheduling Algorithmus ist der cyclische Schedule. Mit diesem Schedule ist es möglich, abhängig von einer gemeinsamen Zeitbasis, den für diese Zeit zugeordneten Task zu schedulen. Dieser Algorithmus ist besonders geeignet für periodische und zeitsensitive Tasks. Sollten zudem noch sporadische und aperiodische Elemente hinzukommen, so muss ein kooperatives Scheduling hinzugefügt werden. Tasks, die mit einem cyclischen Bussystem kommunizieren müssen, z.B. Flex-Ray, können so ihren Slot einhalten und ggf. die zeitlichen Einstellungen im Multi-Core System anpassen. Somit muss auch dieser Algorithmus in der KB berücksichtigt werden, um den Anforderungen aus den Zieldomänen gerecht zu werden.
Eine weitere Scheduling Vorgehensweise ist ein prioritätenbasiertes, dynamisches Scheduling zur Laufzeit. Hier werden mit Hilfe des EDF Algorithmus die aktuellen Prioritäten der Tasks berechnet. Vergleichbar mit dem FIFO Scheduling von LynxOS und anderen RTOS Systemen

lässt dieser Algorithmus ein rein dynamisches Scheduling zu.
Die Anforderungen an die Allokation von Tasks auf Steuergeräten sind aus den Anforderungskatalogen der Zieldomänen und deren verwendeten Systeme nicht eindeutig ersichtlich. Allerdings sind die Verfahren aus Kapitel 3.1 mit der Taskverteilungsanalyse ein Anhaltspunkt dafür, dass eine Allokation so viele Tasks wie möglich auf ein Steuergerät allokieren muss, ohne dabei Deadlines zu verletzen. Die dabei übliche offline Prüfung der Prioritäten und der Auslastung muss auch bei der KB zum Einsatz kommen. Denn nur durch intensives Rechnen und Vergleichen der Berechnungen kann es ermöglicht werden, dass eine Taskverteilung gefunden wird.

### KB Konzept Allokierung / Scheduling

Um die Schedulingalgorithmen und deren Auslastungstests sowie die Algorithmen zur Allokation in einem rekonfigurierbaren HAMS System einsetzten zu können, müssen diese erweitert werden. So unterstützen die aktuellen Algorithmen und Tests keine Phasen der Tasks, sondern unterstützen nur die statischen maximalen Ausführungszeiten. So ändern sich die Ergebnisse eines Feasibility Tests mit den Werten der Tasks über deren Laufzeit. Daraus kann sich ergeben, dass ein System u.U. nicht mehr schedulbar sein kann, wenn ein gewisser Phasenzustand erreicht ist. Damit dies nicht geschieht, müssen die Schedulingalgorithmen und deren Auslastungstests erweitert werden, um für phasengesteuerte HAMS Systeme eingesetzt werden zu können.
Zusammenfassend werden folgende Anforderungen aus dem Bereich Scheduling /Allokation an die HAMS Knowledgebase gestellt:

- Handhabung des phasenorientierten Schedulings
- Unterstützung von periodischen und aperiodischen Tasks
- Verwendung von bekannten Feasibility Tests
- Unterstützung von prioritätenbasiertem und cyclischem Scheduling
- Allokation am sicheren Maximum der Feasibility Tests
- Ersetzten von Bin Packing Algorithmen durch Brute Force

## Rekonfiguration

In einem HAMS System ist die Rekonfiguration an Phasen gebunden und kann zu jedem Phasenwechsel entstehen. So kann ein Phasenwechsel bei unterschiedlichsten Events, welche sich in den Zieldomänen schnell ändern können, auftreten. Um die Phasenwechsel zeitlich korrekt durchführen zu können, müssen bei der Erstellung der KB alle möglichen Phasenübergänge betrachtet und evaluiert werden. Mit den Erkenntnissen aus den Anforderungen bezüglich der Rekonfiguration in den Zieldomänen bedeutet dies zum einen, dass zur Designzeit alle möglichen Phasen und deren Konfigurationen bekannt und evaluiert sein müssen, zum anderen muss der Phasenwechsel zeitlich korrekt ablaufen.

### KB Konzept Rekonfiguration

Um dies zu bewerkstelligen, müssen zwei zusätzliche Anforderungen bei der Task zur ECU Allokation bei der KB Erstellung erfüllt werden. Zuerst muss eine Phasenübergangsauflistung aller möglichen Phasenübergänge der Task aufgestellt werden, um sicherzustellen, dass kein möglicher Übergang bei der späteren Betrachtung vernachlässigt wird. Als zweites muss ein sog. Rekonfigurationstest durchgeführt werden. Dabei wird geprüft ob bei der Rekonfiguration möglicherweise zeitliche Rekonfigurationsvorgaben durch das HAMS Migrationsschema überschritten werden und so undefinierte Zustände auftreten können. Dieser Rekonfigurationstest ist abhängig vom HAMS Rekonfigurationsschema sowie von dem verwendeten Schedulingalgorithmus bzw. Auslastungstest. So muss es für jeden Auslastungstest einen eigenen Rekonfigurationstest geben. Zusammenfassend werden folgende Anforderungen aus dem Bereich Rekonfiguration an die HAMS Knowledgebase gestellt:

- Aufstellen aller möglichen Phasenübergänge zur Überprüfung
- Überprüfung der Phasenübergänge auf zeitliche Dauer im HAMS Schema
- Rekonfigurationstest muss prioritätenbasiertes und cyclisches Scheduling unterstützen

## Hardware

Die Multi-Core ECUs, welche in heutigen Systemen eingesetzt werden, sind sehr unterschiedlich und in Kapitel 4.1.3 aufgelistet.

### KB Konzept Hardware

Bei der HAMS KB Berechnung ist es wichtig, dass die unterschiedlichen Eigenschaften dieser CPUs widergespiegelt werden, wie Taktrate und I/O Schnittstellen. Dies bedeutet aber auch, dass Multi-Core ECUs, die ein HAMS System nicht unterstützen, weil z.B. die Cores komplett voneinander getrennt sind, nicht in die KB Berechnungen mit einbezogen werden und somit nicht Teil eines HAMS Systems sein können. Ebenso müssen die Auswirkungen auf die Änderungen der Core Taktrate auch rückwirkend auf die Task WCET übertragen werden. So ist festzustellen, um welchen Prozentsatz ein Task seine WCET verlängert oder verkürzt bei der jeweiligen Taktrate. Auch die Zeit, welche es benötigt, einen Core auf einen neue Taktrate einzustellen. Die Anforderungen aus dem Bereich Hardware sind wie folgt:

- Einbezug von Migrationseinschränkungen zwischen Cores
- Einbezug von I/O Verfügbarkeit pro Core
- Einbezug von CPU Taktraten und die damit verbundenen Taskänderungen in der WCET
- Berücksichtigung der CPU Taktratenübergangszeit

## Fehlerfallbehandlung / Überwachung

Die Überwachung der Tasks zur Laufzeit ist in einem RTOS und insbesondere im HAMS System (Phasenwechsel) wichtig, kommt aber in den heutigen Systemen nur rudimentär zum Einsatz. So wird sich in den meisten Systemen darauf verlassen, dass durch den RMS Algorithmus der wichtigste Task immer geschedult wird. In cyclischen Algorithmen ist die Systemzeit ausschlaggebend, ab welche ein Task den Core an den nächsten Task abgeben muss. Wie aus dem AUTOSAR Standard

und der TIMEX Erweiterung ersichtlich ist, wird sich hier nicht ausschließlich auf die Schedulingalgorithmen verlassen, sondern zusätzlich die Taskausführungszeiten und Deadlines überwacht. Daraus ergibt sich die Anforderung, dass auch in einer KB die Möglichkeit gegeben sein muss, die Taskausführungszeiten und Deadlines zu überwachen und bei einer Abweichung ggf. darauf reagiert werden kann.
Sollte ein Task aus unbestimmten Gründen eine längere Ausführungszeit benötigen als sonst und so seine Deadline überschreiten, muss der HAMS Scheduler aus dem Inhalt der KB schnell und ohne Neuberechnungen diesen Fehlerfall erkennen und ggf. lösen. Aber auch die Hardware kann Ausfällen unterliegen. So können bestimmte Transceiver Chips ausfallen und somit die Kommunikation der ECU mit der Umwelt zum Erliegen kommen. Für diesen Fall müssen Backupstrategien vorhanden sein welche einer KB Verteilung entspricht, die eintritt, sobald ein Fehlerfall gemeldet oder erkannt wird. So muss das HAMS System darauf mit der Lösung aus der KB reagieren und einem fest definierten Plan im Fehlerfall folgen. Folgende Anforderungen sind im Bereich Fehlerfallbehandlung / Überwachung zu erfüllen:

- Parameter zur Erkennung von Überschreitungen in Taskausführungszeit und Deadline
- Parameter für die Tasküberwachung anhand von Rückmeldungen in der KB
- Bereitstellung einer Backupstrategie in Form einer einzigen gültigen validen Verteilung

## 4.3 Zusammenfassung

Für die Zieldomänen gibt es eine Vielzahl von RTOS Systemen die zum Einsatz kommen, wie auf dem AUTOSAR Standard basierende RTOSe und VxWorks. Durch eine Analyse aus Einsatz- und Konfigurationsmöglichkeiten dieser RTOSe, lassen sich Anforderungen für die HAMS Knowledgebase ableiten, um in den Zieldomänen und der damit verbundenen weichen Echtzeit verwendet werden zu können. Hierfür sind insbesondere die Bereiche Scheduling/Allokation, Rekonfiguration,

Hardware und Fehlerfallbehandlung zu analysieren die einen Anforderungskatalog aus der Automobilelektronik, Avionik und Multi-Core Hardware darstellen.
Im Bereich Scheduling verwenden diese RTOSe ein auf statische Prioritäten basiertes Scheduling. Die Prioritäten werden durch die Auslastungstests RMS oder DMS bestimmt. Dies geschieht alles zur Designzeit und ändert sich im laufenden Betrieb nicht mehr.
Rekonfiguration ist in den heutigen Systemen somit nicht vorhanden. Neueste Multi-Core Hardware wird zwar unterstützt, aber ein ausgewogenes Konzept dazu fehlt. Das korrekte Systemverhalten wird in heutigen RTOS Systemen teilweise überwacht. Während im AUTOSAR Standard Eigenschaften von anderen RTOS Systemen übernommen wurden, um neben der Speicherbereichsüberwachung eines Tasks auch dessen Ausführungszeit zu überwachen, ist dies z.B. in VxWorks nicht vorhanden. Da die Überwachung nicht oder nur rudimentär vorhanden ist, wird auch keine Rekonfiguration im Fehlerfall benötigt. Auch die Rekonfiguration zur Laufzeit ist auch nur teilweise definiert.
Basierend auf dieser Anforderungsanalyse ist es von Bedeutung, dass folgende Grundeigenschaften durch den Aufbau und Inhalt der HAMS KB übernommen werden müssen, um in den Zieldomänen eingesetzt werden zu können. So muss im Bereich Scheduling ein prioritätenbasierter, ein cyclischer und ein dynamischer Algorithmus vorhanden sein. Die Rekonfiguration muss semi-statisch umgesetzt werden auf Multi-Core Hardware, so dass definierte Ausgangszustände vorhanden sind. Fehlerfallbehandlung/Überwachung von Tasks müssen zur Laufzeit durchgeführt werden und ein Plan zur Auflösung von Fehlern muss vorhanden sein.
Mit diesem Anforderungskatalog wird nun im Folgenden eine HAMS Knowledgebase aufgebaut, welche das Ziel hat, durch das Ausnützen von phasenbezogener Rechenzeit mehr Software auf einem Steuergerät mitzunehmen als die derzeit aktuellen Systeme.

# 5 Knowledgebase Eingangsparameter

Nachdem die Anforderungen und Ziele in Bezug auf Scheduling/Allokation, Dynamik, Hardware, Echtzeit und Überwachung für die KB aufgestellt sind, müssen nun entsprechende Eingangsparameter aus der Analyse von kommerziellen System aufgestellt werden. Anhand dieser Parameter werden die späteren KB Berechnungen sowie die KB Erstellung durchgeführt. Für die aktuell angewandte offline Untersuchung und den dazugehörigen Feasibility Test wurden in anderen Arbeiten Eingangsparameter definiert, um diese durchführen zu können [143] [11]. Da weder diese Arbeiten noch aktuelle kommerziellen Systeme für semi-statische phasenorientierte Systeme ausgelegt sind, müssen diese folglich erweitert oder abgeändert werden. Hierbei handelt es sich um Taskparameter wie z.B. Ausführungszeit, Deadline und Ready Time, logische Taskparameter zur Definition von Abhängigkeiten und Multi-Core Parameter zur Verteilung der Tasks. Für das weitere Vorgehen muss zuerst erläutert werden, was Task Phasen sind sowie das Zusammenspiel von Task Phasen und anderen Tasks in einer ECU und das Gesamtverhalten eines phasenorientierten Systems.

## 5.1 Task Parameter in einem phasenorientierten System

Da es sich bei dem HAMS System um ein phasenorientiertes System handelt, müssen Phasen und deren Einfluss auf das Scheduling für eine ECU zuerst aufgezeigt werden.

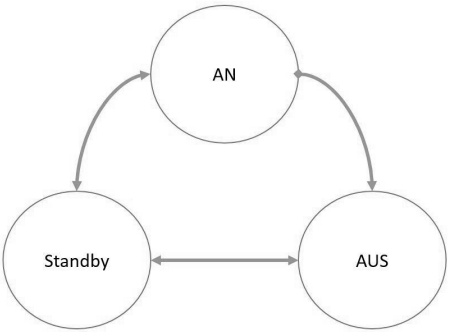



Abbildung 5.1: Zustandsautomat einer GRA aus Sicht des Fahrers

### 5.1.1 Tasks und Phasen

Ein Beispiel aus der Automobilelektronik verdeutlicht den Begriff „Phasen“. Die Geschwindigkeitsregelanlage im Automobil hat in der Regel drei interne Zustände. Sie kann angeschaltet sein und somit die Geschwindigkeit regeln, oder im Standby Modus oder ausgeschaltet sein. Wobei ausgeschaltet im Sinne der Zieldomänen bedeutet, dass der Task dennoch rechnet und Schalterstellungen abfrägt, um wieder regeln zu können. Diese drei Phasen stellen einen internen Zustandsautomaten des Tasks Geschwindigkeitsregelanlage dar (Knoten im Bild 5.1). Die Übergänge der einzelnen Phasen sind eindeutig geregelt (Kanten im Bild 5.1). So kann die Geschwindigkeitsregelanlage nur von „an“ zu „standby“ oder „aus“ gehen und ebenso von der Phase „aus“ in „standby“. Es ist nicht möglich, von „aus“ in den Zustand „an“ zu gehen. In der Phase „standby“ kann die Geschwindigkeitsregelanlage sowohl in „an“ oder „aus“ gehen. Wann eine Phase aktiviert wird, d.h. ein Phasenübergang stattfindet, wird immer vom Task selbst bestimmt. Im Falle der Geschwindigkeitsregelanlage gibt es herstellerübergreifende Vorgaben (ISO 22179:2009)[1]. So darf die Geschwindigkeitsregelanlage unter einer Geschwindigkeit, von z.B. $30\frac{km}{h}$, nicht aktiviert werden. Ebenso beschränken manche Hersteller auch die Maximalgeschwindig-

keit der Geschwindigkeitsregelanlage auf z.B. $180\frac{km}{h}$. So ist ein Event für den Task die aktuelle Geschwindigkeit. Diese Phasen haben eine Auswirkung auf die Ausführungszeit der Tasks bzw. auf die Anzahl der nötigen Berechnungen. Am Beispiel der Geschwindigkeitsregelanlage bedeutet dies, dass dieser eine längere WCET hat im Zustand „an“ als im Zustand „aus“. So ändert sich die Ausführungszeit bzw. WCET der Tasks abhängig von den Phasen.
Nachdem der Begriff Phase geklärt ist, müssen die Bezeichnungen „logisch abhängig“ und „logisch unabhängig“ geklärt werden. Ein Task ist von einem oder mehreren Tasks „logisch abhängig“, sobald mindestens ein gemeinsames Event diese verbindet. Voneinander „logisch unabhängig“ sind Tasks, wenn kein Event diese verbindet. Ein Beispiel mit einem zweiten Task aus der Automobilelektronik, der Einparkhilfe, zeigt die logischen Abhängigkeiten klar auf.
Auch Task der Task Einparkhilfe hat Phasen und Phasenübergänge. Neben den Phasen „an“ und „aus“ für den Task Einparkhilfe ist der Phasenübergang von „aus“ zu „an“ von Bedeutung. Die Einparkhilfe darf nur den Einparkvorgang steuern, sobald die Geschwindigkeit unter $30\frac{km}{h}$ fällt. Somit sind die Tasks Einparkhilfe und Geschwindigkeitsregelanlage logisch abhängige Tasks, da sie das Event Geschwindigkeit mit der Schwelle $30\frac{km}{h}$ verbindet. Anders verhält es sich, wenn die Tasks logisch unabhängig voneinander sind.
Als Beispiel dient hier der Task Regensensor. Der Regensensor im Auto regelt anhand des gemessenen Benetzungsgrades der Scheibe die Geschwindigkeit der Scheibenwischer. Dieser Vorgang ist geschwindigkeitsunabhängig. So muss sowohl bei einer Regelung der Geschwindigkeitsregelanlage als auch bei einem Einparkvorgang der Regensensor funktionieren und die Scheibe reinigen. Da sich der Regensensor mit keinem Event der Geschwindigkeitsregelanlage und der Einparkhilfe verbinden lässt, ist der Task logisch unabhängig.

### 5.1.2 Phasenabhängige Taskparameter

Mit der Änderung einer Phase ändert sich die Ausführungszeit eines Tasks. So benötigt z.B. die GRA mehr Rechenzeit, wenn sie die Geschwindigkeit aktiv regelt, als wenn sie ausgeschaltet ist. Die Parameter

eines periodischen Tasks lassen sich anhand eines Tupels beschreiben[1]

$$FP_m = CP_m, TP_m, DP_m, CMM_m, (NPU_m) \tag{5.1}$$

$CP_m$ Dieser Parameter beschreibt die phasenabhängige WCET des Tasks. Je nachdem in welcher Phase ein Task sich befindet, kann dessen Ausführungszeit (WCET) länger oder kürzer sein. Dieser Parameter darf nie null sein, da dies bedeutet, dass der Task nicht benötigt wird. In den Zieldomänen ist es nicht vorgesehen, dass Tasks sich komplett beenden oder ausschalten. Zur Ermittlung dieses Wertes werden die Ausführungszeiten in den jeweiligen Phasen gemessen und die längste Ausführungszeit als Parameter eingetragen (Worst Case Execution Time).

$TP_m$ ist die Periode der aktuellen Phase. Auch diese kann sich verändern, abhängig davon wie oft der Task benötigt wird. Für diesen Parameter gilt, dass er unter normalen Umständen nicht gleich null sein darf.

$DP_m$ ist die phasenabhängige Deadline eines Tasks. Bei RMS geschedulten Systemen wird die Deadline $DP_m$ gleich der Periode $TP_m$ gesetzt, um die Feasibility Berechnungen durchführen zu können. Da dies nicht immer der Fall ist in realen Systemen, können bei den KB Berechnungen andere Werte für die Deadline eingetragen werden. So kann die Deadline schon vor Periodenende gesetzt werden. Dementsprechend müssen die passenden Scheduling Algorithmen verwendet werden.

$CMM_m$ ist ein Parameter, der angibt, wie oft ein Task während eines Phasenwechsels die Deadline verletzen darf. Das bedeutet, dass der Task während des Phasenwechsels nicht rechnen wird und so auch keine Nachrichten oder Ergebnisse versendet. So lässt sich die erlaubte Rekonfigurationszeit mit diesem Parameter ermitteln.

$NPU_m$ beschreibt die Schnittstellen, welche ein Task benötigt, um mit anderen Geräten zu kommunizieren Needed Peripheral Unit (NPU). Dieser Parameter ist phasenunabhängig.

[1]Index **m** ist eine eindeutige ID des Tasks in der HAMS Knowledgebase

Neben den Parametern für einen periodischen Task gibt es auch noch aperiodische und sporadische Tasks in einem System. Die Tupel für diese Tasks lautet wie folgt.

$$\begin{aligned} FP_m &= \{CP_m, AP_m, DP_m, CEP_m, CMM_m, (NPU_m)\} \\ FP_m &= \{CP_m, DP_m, CEP_m, CMM_m, (NPU_m)\} \end{aligned} \tag{5.2}$$

[$AP_m$] ist die minimale Aufrufzeit zweier Berechnungen für eine aperiodische Funktion. Auch diese ist abhängig von Phasen und kann sich je nach eingetroffenem Event ändern.

[$CEP_m$] ist eine phasenabhängige zusätzliche Ausführungszeit in Prozent für eine Funktion. Da Funktionen ihre Deadline überschreiten können, muss ein weiterer Parameter eingeführt werden, der verhindert, dass unter prioritätenbasiertem Scheduling die CPU von diesem Task nicht dauerhaft blockiert. Sollte die Funktion den Core nicht mehr freigeben und so anderen nieder prioren Funktionen die Rechenzeit „stehlen“, wird durch diesen Parameter der Task zu einer Zwangsbeendigung geführt werden.

Diese Parameter stellen die grundlegenden Eingaben dar, um die Ziele und Anforderungen in Scheduling/Allokation bei der Knowledgebase Erstellung zu erfüllen. Hier sind die zeitlichen Parameter $CP_m$, $TP_m$, $AP_m$, $DP_m$ zu erwähnen, die grundsätzlich für die Auslastungstests der Scheduling Algorithmen benötigt werden. Die Parameter bezüglich der Migration ($CMM_m$,$NPU_m$) werden dazu benötigt, um eine Rekonfiguration einzuplanen, durchzuführen und überprüfen zu können.

### Task Überwachung

Die Überwachung der korrekten Ausführung von Tasks in einem HAMS System ist aufgrund der unterschiedlichen Phasen und der sich daraus ergebenden Veränderung des Systems komplex. So kann der HAMS Scheduler weder zur Laufzeit konkrete Daten über die Ausführung eines Tasks sammeln, noch kann er mit statischen Task Überwachungsparametern versehen werden, wenn sich diese Tasks phasenbezogen ändern. Auch diese Überwachungsparameter müssen somit phasenabhängig gemacht werden. Als Grundlage für die Überwachung dient die Deadline

eines Tasks. Sowohl periodische als auch aperiodische/sporadische Tasks sind mit einer Deadline versehen, die eingehalten werden muss. Wird die Deadline nicht eingehalten, kann je nach Scheduling Algorithmus der Task als fehlerhaft angesehen werden, auch für den Fall, dass Deadline und Periode gleich sind. Ähnlich der Timex Beschreibung, können auch noch weitere phasenabhängige Elemente zur WCET Überwachung eingeführt werden[2].

$$FP_m = \{CP_m, CP_{1.m}, ..., CP_{n.m}, NFO_m\} \tag{5.3}$$

Abhängig von den phasenabhängigen Parametern für einen Task und dessen für diese Phase feste WCET $CP_m$ (CheckPoint) werden relative Punkte für die Ausführungsüberwachung gesetzt. D.h. binnen dieser Zeitspanne muss eine Rückmeldung an den FLS gesendet worden sein. So kann der Parameter $CP_{1.m}$ z.B. auf 15% gesetzt werden, was bedeutet, dass noch vor Ablauf von 15% der WCET eine Nachricht an den FLS des HAMS Schedulers geschickt werden muss.
Der Parameter $NFO_m$ (Needed in Fail Operation) ist ein boolescher Wert, welcher angibt, ob diese Phasen und somit auch dieser Task in einem Fail Operational Modus des HAMS Systems zur Verfügung stehen muss. In der Anzahl aller Phasen dieses Tasks ist dieser Wert nur einmal auf „true“ gesetzt oder immer auf „false“. Diese Parameter zur Überwachung unterstützen das Ziel und die Anforderungen der Fehlerbehandlung und ermöglichen so die korrekte Evaluation des Systems.

### 5.1.3 Phasenunabhängige Taskparameter

Insbesondere die I/O Geräte, welche der Task benötigt, um mit anderen ECUs oder Sensoren zu kommunizieren sind phasenunabhängig und werden immer benötigt. Auch Interrupts können bei gewissen Tasks von Bedeutung sein, um z.B. auf Benutzereingaben zu reagieren. Aber vor allem ist die logische Verknüpfung zwischen Tasks von großer Bedeutung für die KB Berechnung und Systemauslegung.

[2]Der Index **n** steht stellvertretend für die maximale Anzahl

### Logische Verknüpfungsparameter

Um Tasks miteinander logisch zu verknüpfen, muss ein „(Leading Task) - Bestimmender Task (LT)“ definiert werden. Der Leading Task gibt die Phase für seine„ Depending Task - Abhängiger Task (DT)“ vor. Das Tupel 5.4 zeigt auf, wie ein LT und seine DT geordnet werden.

$$\begin{aligned} LT_p &= \{LT_{p.1}, ..., LT_{p.n}\} \\ LT_{p.m} &= \{DT_{p.1}, ..., DT_{p.m}\} \\ DT_{p.1} &= \{DT_{p.1.1}, ..., DT_{p.1.m}\} \\ ULT &= \{(LT_{p.1}; (DT_{p.1.1}, ..., DT_{p.1.m}) \vee ... \\ &\quad \vee (LT_{p.n}(DT_{p.n.1}, ..., DT_{p.n.m}))\} \end{aligned} \tag{5.4}$$

Am ersten Tupel ist zu erkennen, dass ein LT aus mehreren (n) Phasen besteht. Doch muss er mindestens eine Phase besitzen, um überhaupt für das Scheduling in Frage zu kommen. Der Wert $LT_{p.n}$ ist ein Tupel vom Typ $FP_m$ (Formel 5.3) und gibt somit die Zeiten an, die für diese Phase gültig sind. Das zweite Tupel zählt auf, wie viele DTs ein LT hat. Hier kann ein LT eine n-fache Anzahl an abhängigen Tasks haben oder aber auch keinen, womit der zweite Tupel eine leere Menge ist. Beide Tupel sind voneinander unabhängig und können unterschiedliche Größen aufweisen. Im dritten Tupel sind die Phasen des DTs aufgelistet. Ein DT muss immer genauso viele Phasen besitzen wie sein LT. Der Tupel ist geordnet und somit entsprechen die Indices den entsprechenden Phasen des Tupels $LT_p$. Auch in diesem Tupel sind die Werte vom Typ $DT_{p.1.1}$ ein Tupel vom Typ $FP_m$. Ist nun der LT und seine DTs definiert, wird die Beschreibung im Tupel $ULT$ zusammengefasst. Die logischen Abhängigkeiten eines LTs sind vollständig dargestellt und der gegenseitige Ausschluss wird durch ein „oder $\vee$“ vervollständigt. Da in einem System mehrere Tasks aktiv sind, muss noch ein logischer Zusammenhang zwischen diesen hergestellt werden.

$$U_{Sys} = \{U_{LT.1} \wedge ... \wedge U_{LT.n}\} \tag{5.5}$$

Die logischen Abhängigkeiten der ganzen ECU werden im Tupel $U_{Sys}$ dargestellt. Hier werden alle voneinander unabhängigen LTs mit einem logischen „und $\wedge$“ miteinander verknüpft. Dies bedeutet, dass die Tasks gleichzeitig auf dem System laufen und sich gegenseitig zeitlich

beeinflussen können. D.h. in der KB Berechnung muss somit für alle Vorkommnisse ein gültiger Schedule berechnet werden können.
Um diese Vorkommnisse besser einplanen zu können, muss der Zustandsautomat des logisch abhängigen Tasksets ebenfalls betrachtet werden. Hier kann es vorkommen, dass ein Task nicht von jedem Zustand in einen beliebigen anderen Zustand springen kann wie in Kapitel 5.1.1. Sollte dies der Fall sein, reicht eine logische „oder“ Verknüpfung nicht aus, um die späteren KB Berechnungen korrekt durchführen zu können bzw. nicht mögliche Konfigurationen auszuschließen. Diese Möglichkeit muss in der Formel 5.1 abgebildet werden durch eine Reihe MPU für „Mögliche Phasenübergänge“ und führt zu der Formel:

$$FP_m = CP_m, TP_m, DP_m, CMM_m, (NPU_m), (MPU_m) \tag{5.6}$$

#### Peripherie und Interrupt

Neben der Rechenzeit, die ein Task auf der CPU benötigt, bedarf es Peripheriegeräte, um eine korrekte Funktionalität zu gewährleisten. In einem Tupel mit insgesamt vier Reihen:

$$T = \{\{NP\}, \{NM\}, \{NCP\}, \{NI\}\} \tag{5.7}$$

Der Parameter Needed Peripherals -Benötigte Peripherie (NP) gibt an, welche I/O Geräte benötigt werden. Hier befinden sich u. a. CAN, Flex-Ray und bestimmte General-Purpose Input/Output (GPIO) Ports wieder. Da es sich hier um eine grundlegende funktionale Eigenschaft des Tasks handelt, z.B. das Kommunizieren mit Sensoren, sind eingetragene Eigenschaften im NP phasenunabhängig und die Peripheriegeräte müssen immer zur Verfügung stehen. Sollte es in einem System mehrere I/O Geräte des gleichen Typs geben, z.B. CAN, so muss spezifiziert werden, welcher CAN Bus benötigt wird. Da nicht jede CAN Einheit einer ECU mit dem selben CAN Bus verbunden ist, muss hier genau angegeben werden, welche CAN Einheit verwendet wird, z.B. CAN-Chassis. Ein weiterer Parameter (Needed Memory) - Benötigter Speicher (NM) gibt an, wie viel Memory ein Task benötigt. In den Zieldomänen werden die Speicherbereiche beim Aufstarten des Tasks reserviert und erst beim Herunterfahren / Beenden des Tasks wieder freigegeben, um mehr Sicherheit zu gewährleisten. Aus diesem Grund wird der Parameter

als phasenunabhängig behandelt, auch wenn es bedeutet, dass in der aktuellen Phase ein zugewiesener Speicherbereich des Tasks zurzeit nicht beschrieben / benötigt wird. Der Parameter (Needed Core Peripherals) - Benötigte Peripherie des Cores (NCP) gilt explizit für den Core eines Multi-Core Prozessors. Wird hier z.B. eine Floating Point Unit für Gleitkommaoperationen benötigt, muss ein Core diese verfügbar haben, damit der Task Gleitkommaoperationen durchführen kann. Ähnlich verhält es sich mit unterschiedlicher Core Architektur die in BIG.little Systemen zum Einsatz kommt. Der letzte Parameter (Needed Interrupt) - Benötigter Interrupt (NI) gibt an, welche Interrupts benötigt werden. So werden im späteren HAMS System die Funktionen für den Interrupt an die Verteilung der Tasks angepasst.
Um dies auch in der KB Berechnung zu berücksichtigen, werden für Interrupts aperiodische Tasks angelegt. Die Parameter für einen Interrupt werden somit gleich der Formel 5.2 beschrieben und findet sich im OS Interrupt Model wieder. Bei der Erstellung der KB wird die Zeit für die Ausführung des Interrupthandlers mit eingeplant.
Diese vier Parameter grenzen die Rekonfiguration ein. D.h. je mehr Einschränkungen ein Task mit sich bringt, umso weniger ist es möglich, diesen Task auf einen anderen Core zu migrieren. Unter Umständen kann er sogar fest an einen Core gebunden sein. Dies ist der Fall, wenn z.B. der CAN-Chassis aufgrund von Firewalleinschränkungen nur für Core 1 verfügbar ist. So muss jeder Task, der Zugriff auf diesen Bus benötigt, auf Core 1 laufen.

#### Knowledgebase Taskmodel Single-Core

Das Taskmodel für LT und DT der KB ist in Formel 5.8 zusammengefasst.

$$\begin{aligned} ULT = \{(FP_{LT.p.1}; (FT_{DT.p.1.1}, ...FT_{DT.p.1.m})) \vee ... \\ \vee (FT_{LT.p.n}(FT_{DT.p.n.1}, ..., FT_{DT.p.n.m}))\} \\ UDT = \{FP_{DT.p.1.1}, ..., FP_{DT.p.1.m}\} \\ U_{Sys} = \{ULT_1 \wedge ... \wedge ULT_n\} \end{aligned} \tag{5.8}$$

Dieses Taskmodel stellt die Grundlage für eine KB Berechnung und Erstellung dar. Ausschlaggebend sind hier die Task Zeiten und Phasen.

$$
\begin{aligned}
FP_{temp.p.1} &= \{5, 35, 35\} \\
FP_{temp.p.2} &= \{25, 35, 35\} \\
FP_{einp.p.1} &= \{25, 35, 35\} \\
FP_{einp.p.2} &= \{5, 35, 35\} \\
UDT &= \{FP_{einp.p.1}, FP_{einp.p.2}\} \\
ULT_{temueinp} &= \{(FP_{temp.p.1}; (FP_{einp.p.1})) \vee (FP_{temp.p.2}; (FP_{einp.p.2}))\} \\
U_{Sys} &= \{ULT_{temueinp}\}
\end{aligned}
\tag{5.9}
$$

Anhand des Beispiels 5.9 lässt sich das Taskmodel noch anschaulicher erklären. Die zwei voneinander abhängigen Tasks Geschwindigkeitsregelanlage (temp), zugleich der LT, und Einparkhilfe (einp) werden in 5.9 beschrieben. Dabei gibt es für je zwei Phasen „An“ und „Standby“. Anhand ULT ist zu erkennen, dass entweder Phase eins aktiv ist und somit die Einparkhilfe regelt oder Phase zwei, in welcher die Geschwindigkeitsregelanlage angeschaltet ist. Im gesamten System laufen daher auch nur die beiden Tasks Geschwindigkeitsregelanlage und Einparkhilfe und es gibt keine voneinander unabhängigen Tasks.

## 5.2 Multi-Core Parameter

In der Formel 5.8 wurden noch keine speziellen Multi-Core Parameter berücksichtigt und sie ist somit allgemein gültig für Single-Core Prozessoren. Um die Ziele und Anforderungen bei der Hardware umzusetzen, müssen Multi-Core Parameter eingeführt werden. Diese Parameter müssen für die KB Berechnung beschrieben und mit einbezogen werden. Doch zuerst muss geklärt werden, welche Auswirkungen diese Multi-Core Eigenschaften auf die KB haben.

### WCET bei Migration

Damit eine Migration in HAMS zwischen zwei Cores stattfinden kann, muss es sich um ein SMP-System handeln, d.h. der aktuelle Core des

Tasks und der Ziel Core müssen Zugriff auf die gleichen Ressourcen haben. Dies bezieht nicht nur die I/O Geräte mit ein, wie CAN, Flex-Ray etc., sondern auch Speicherbereiche. Nur durch Zugriff auf gemeinsame Speicherbereiche kann vom Task Kontext auf dem Ziel Core zugegriffen werden und die erforderlichen Bereiche kopiert werden. Ist dies nicht der Fall, kann im aktuellen HAMS System keine Migration durchgeführt werden. Die Übermittlung dieses Vorgangs geschieht hier durch spezielle Mechanismen, welche im HAMS Scheduler vorhanden sind [131]. Der Ablauf der Migration ist im HAMS Scheduler fest vorgegeben und somit auch die zeitlichen Bedingungen, für die Übermittlung des Tasks an den Ziel Core (Kapitel 3.3). Wird ein Task migriert, so sind in den privaten Caches des Ziel Cores die Daten noch nicht vorhanden und müssen erst nachgeladen werden. Hier tritt ein sog. „Cache Miss“ ein. Über die Multi-Core Mechanismen zur Speicherverwaltung (write through) werden die erforderlichen Daten aus den gemeinsamen Speicherbereichen nachgeladen und in den privaten Speicher transferiert. Der Cache ist nun „Cache HOT“. Dieser Vorgang führt zu einer Verlängerung der WCET des Tasks nachdem er migriert wurde. Die Verlängerung des Tasks ist abhängig von der Prozessor Architektur und wie viele Speicherbereiche nachgeladen werden müssen. Dies ist ersichtlich in der Abbildung 5.2. Hier sind die Ergebnisse aus der Forschung von D. Hardy et al. [86] [120] aufgelistet und die erforderlichen Zyklen mit einem Prozessor von 1.2GHz erweitert, um so die Verlängerung in Millisekunden anzuzeigen. Wie deutlich zu erkennen ist, findet sich in den Tasks „CNT“ und „Stats“ eine Verlängerung der WCETs von max. 1ms wieder. Im Fall des Testtasks „Matmult“ ist keine signifikante Verlängerung zu erkennen. Um zu vermeiden, dass der Task als fehlerhaft erkannt wird und um die Auslastung nach der Migration nochmals zu überprüfen, muss für diesen Fall eine Verlängerung der WCET im Taskmodel angegeben werden. Dies geschieht mit dem Parameter $CMP$ (WCET after Migration extension Percentage) der eine Verlängerung der WCET in Prozent abhängig der aktuellen Phase angibt. Er gilt für periodische Tasks, wie in Formel 5.10, als auch für sporadische und aperiodische Tasks.

$$FP_m = \{CP_m, TP_m, DP_m, CMP_m\} \tag{5.10}$$

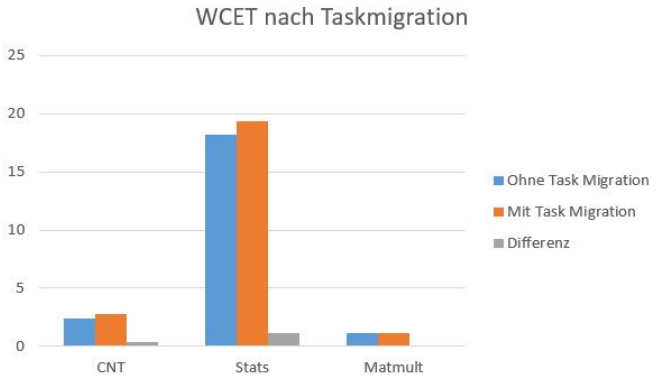



Abbildung 5.2: WCET Verlängerung nach Migration in ms [86]

## WCET bei Core Frequenz

Im Zuge der Energiesparfunktionen von Multi-Core Prozessoren wurden zwei Eigenschaften eingeführt. Die erste Eigenschaft ist die sich verändernde Frequenz eines Cores. Abhängig vom Multi-Core Prozessor können die Cores ihre jeweiligen eigenen Frequenzen unabhängig von den anderen Cores variieren. Dabei sind die Ergebnisse der Variation festgelegt und die Cores können nur bestimmte Stufen an Frequenzen einnehmen. Andere Prozessorarten wiederum reduzieren die Frequenz der Cores insgesamt und alle Cores haben somit die gleiche Frequenz. Die Bestimmung der Frequenz wird über Register im MC-Prozessor festgelegt. In Linux und im HAMS Scheduler ist das Powermanagement dafür verantwortlich und setzt die sog. States selbst in den Registern des Prozessors. Sobald die erforderlichen Register durch das Powermanagement gesetzt sind, wird die Taktrate sofort ohne Verzögerung geändert [17] [28].
Eine weitere Eigenschaft sind BIG.little Prozessoren. Hier gibt es Cluster an Cores, die untereinander verbunden sind (SMP System), aber die Cores unterschiedliche Eigenschaften haben. Dies kann der Fall sein, wenn Cores z.B. schneller in gewissen Berechnungen sind oder

energiesparender als andere in einem BIG.little Prozessor. Diese beiden Eigenschaften, sich verändernde Frequenz und unterschiedliche Core Architekturen, müssen sich im Task Model wiederfinden.

$$FP_m = \{(CP_m), TP_m, DP_m, CMP_m, (CA)\} \tag{5.11}$$

Diese Veränderung ist abgebildet in der Formel 5.11. Hier ist aus dem Parameter $CP_m$ für die WCET eine Reihe erstellt worden. Diese beinhaltet für jede mögliche Frequenz des Cores eine WCET des Tasks. Eine zweite Reihe, die $CA$ (Core Architektur) gibt an, auf welchen Architekturen ein Core laufen darf bzw. gibt die Core Nummern des SMP Systems an.

## Messung der Core Frequenzänderung

Für die Messung der Core Frequenzänderung von Multi-Core Prozessoren wird zuerst beachtet, welche Parameter dabei in die KB Berechnung mit einfließen. In der Formel 5.11 werden die unterschiedlichen Frequenzen, auf die ein Multi-Core Prozessor eingestellt werden kann, berücksichtigt. Diese Frequenzen müssen sich bei den Tasks und deren Phasen wiederfinden, damit für jede Systemphase die Cores mit dem FLS auf die korrekte Frequenz eingestellt werden. Hierbei werden absolute Zeiten für die WCETs der Tasks gemessen, also keine prozentualen Verlängerungen der WCETs abhängig von der Core Frequenz. Ebenso wird eine Zeit, vom Übergang der alten Frequenz zur neuen Frequenz, bei der KB Erstellung nicht berücksichtigt.
Um das Missachten dieser „Core-Frequenz Rekonfigurationszeit“ und der „absoluten Zeiten“ im Rahmen der Zieldomänen zu bestätigen, werden auf der HAMS Evaluationsumgebung entsprechende Messungen durchgeführt. Hierbei wird mit ftrace und Kernelshark zuerst die Laufzeit eines oder mehrerer Tasks unter bestimmten Frequenzen gemessen. Damit dieser Task die erforderlichen Operationen durchführen kann, muss zum einen eine Kreiszahlberechnung nach der Leibnitz Reihe [58] in C umgesetzt werden, sowie in Kooperation die Erkennung von Kanten mit Hilfe des Gauss- und Canny-Algorithmus [2] [57]. Mit der Kreiszahlberechnung nach Leibnitz wird die ALU und die Floatingpoint Unit eines Cores belastet.


Abbildung 5.3: Messung der maximalen Rekonfigurationszeit

Mit dem Kantenerkennungstask wird der Speicherzugriff, insbesondere der Zugriff auf Caches der verschiedenen Ebenen, getestet. Beide Algorithmen werden in den Zieldomänen verwendet. Wobei die Kreiszahlberechnung hier nur stellvertretend ist für einen Task mit überwiegend mathematischen Berechnungen.
In Abbildung 5.3 (Anhang in Abbildung 9.1) ist zu erkennen, wie sich die Ausführungszeit des Tasks „pi_berech-2756“ mit dem Setzen einer höheren CPU Frequenz ändert. Die Ausführungszeit ändert sich im Mittel von 3,1s zu 0.8s. Die CPU Frequenz wurde hierbei von 800MHz auf 2500Mhz erhöht. Diese Zeiten wurden mit dem Core i5 der in Kapitel 7.1 einzuführenden Evaluationsumgebung ermittelt auf Core 0 (kein Hyperthreading) über mehrere Testreihen. Somit wurde die Core Frequenz um ca. 315% erhöht, die Ausführungszeit des Tasks verringert sich um ca. 387%. Das gleiche Ergebnis lässt sich feststellen, wenn die CPU Frequenz von 2500Mhz auf 800MHz reduziert wird (Anhang in Abbildung 9.2). Auch der Kantenerkennungstask verhält sich wie erwartet und hat eine Ausführungszeit von 4s bei 800MHz und 1,2s bei 2500MHz, was einer Steigerung von 333% entspricht (Anhang in Abbildung 9.3). Durch diese Tasks wird deutlich, dass die Änderung der CPU Frequenz unverzüglich abläuft und somit keine Verzögerungen durch unbekannte Multi-Core Hardware Eigenschaften entstehen. Dies ist daran zu erkennen, dass nach einer Frequenzänderung sofort die nominelle Ausführungszeiten anliegen, ohne dass eine einmalige Verlängerung der Ausführungszeit stattfindet. Auch wird deutlich, dass mit der Steigerung der Frequenz die Ausführungszeit der Tasks nicht linear steigt. Somit ist die Aufnahme der Parameter $CP_m$, d.h. der WCET für jede Core Frequenz aus Formel 5.11 berechtigt.

## 5.3 System Model

Die Beschreibung der Tasks ist mit dem Multi-Core Model zwar vollständig, aber noch nicht ausreichend, um die KB für das HAMS System zu berechnen. Ein weiteres Model, das System Model, muss eingeführt werden, damit bei der KB Berechnung die Hardware Eigenschaften mit einbezogen und auf die Taskparameter abgebildet werden zu können.

### Beschreibung der Hardware

Analog zum Tupel des Task Models über Peripherie und Interrupt 5.7 muss die Hardware beschrieben werden. Dies ist im nachstehenden Tupel 5.12 abgebildet.

$$\begin{aligned} C_i &= \{\{AP\}, \{ACP\}, \{AI\}, \{ACF\}, \{CTM\}\} \\ MC &= \{(C_0, ..., C_n), AM\} \end{aligned} \tag{5.12}$$

Die ersten vier Parameter sind eine Liste der verfügbaren Peripheriegeräte (Available Peripherals) - Verfügbare Peripherie (AP), wie CAN Ethernet etc., der verfügbaren Core Eigenschaften, (Available Core Peripherals) - Verfügbare Peripherie des Cores (ACP), wie Gleitkommaeinheit oder Architektur, sowie die verfügbaren Interrupts auf diesem Core (Available Core Interrupt) - Verfügbarer Interrupt des Cores (AI). Unterschiedliche Parameter zum Tupel 5.7 sind die Eigenschaften (Available Core Frequencies) - Verfügbare Frequenzen des Cores (ACF) und Core To Migrate to (CTM). In ACF werden die verfügbaren Core Frequenzen aufgelistet, die ein Core einnehmen kann. So kann daraus abgeleitet werden, wie lange die WCET des Tasks ist, abhängig von der eingestellten Taktrate. Eine weitere wichtige Auflistung ist der Parameter CTM. Er gibt an, auf welchen Core der Task migriert werden kann, ausgehend von diesem Core. So können Cores manuell in Gruppen aufgeteilt werden und Einschränkungen in der Migration vorgenommen werden, wie es z.B. bei Interrupts und den benötigten Funktionen der Fall ist. Das gesamte Tupel des Multi-Core Prozessors MC beinhaltet die Tupel der einzelnen Cores und die maximale Verfügbarkeit des Gesamtspeichers.

## Beschreibung von OS Prozessen/Interrupts

Neben den Tasks auf dem System benötigt ein OS immer eine gewisse Rechenzeit zur Verwaltung des Systems und Abarbeitung von System Calls oder Interrupts. Damit diesen Prozessen eine gewisse Rechenzeit zugeschrieben und so eine korrekte Systemausführung gewährleistet wird, müssen sie bei der Berechnung der KB mit eingeplant werden. Da es sich bei diesen Vorgängen auch um periodische oder aperiodische Vorgänge handelt, werden diese wie Tasks behandelt. So gelten auch hier die Tupel aus 5.1 und 5.2 mit der jeweiligen Anpassung aus den Core Frequenzen und der Einschränkung auf bestimmte Cores. Diese Beschreibung ist aber stark abhängig von dem Betriebssystem, welches verwendet wird. So kann bei manchen Betriebssystemen ersichtlich sein, welcher Teil eines Kernels gerade aktiv ist, wie z.B. in Linux, oder es wird maskiert und nur der Kernel ist aktiv. Sollte letzteres der Fall sein, muss durch empirische Versuche ermittelt werden, wann und in welchem Zusammenhang (bestimmte Tasks) der Kernel periodisch und aperiodisch läuft und so in kleineren Einheiten für die KB Berechnung aufgespalten werden. Im Falle von System Calls, die bei Tasks angewandt werden, werden diese zur WCET des Tasks hinzugezählt und mit Einhaltung der Deadline überprüft.

### Knowledgebase OS Model

Das OS Model für die KB ist in folgenden Tupeln dargestellt 5.13:

$$\begin{aligned} OS &= \{F_{OS.1}, ..., F_{OS.m}\} \\ FOS_m &= \{(C_m), T_m, D_m, CMP_m, (CL)\} \end{aligned} \tag{5.13}$$

Für alle Tasks des Betriebssystems $OS$ wird ein eigenes Tupel aufgestellt. Neben der WCET ($C$), Periode ($T$) und Deadline ($D$) wird auch hier der Parameter für die Verlängerung der WCET nach einer Migration angegeben. Phasen gibt es bei Betriebssystemtasks nicht. Der Parameter $CL$ (Core Lock) gibt an, ob der Task auf einem bestimmten Core vorhanden sein muss und nicht migriert werden darf.

## 5.4 Zusammenfassung

Um eine KB Berechnung und KB Erstellung durchführen zu können, müssen Eingangsparameter für das HAMS System identifiziert, aufgestellt und beschrieben werden. Ausgangspunkt sind dafür die Anforderungen und Ziele in Bezug auf Scheduling/Allokation, Dynamik, Hardware, Echtzeit und Überwachung aus der Analyse bestehender Systeme im Kapitel4.
Für die Durchführung der KB Berechnung und KB Erstellung wurden drei verschiedene Modelle vorgestellt: das Taskmodel, das Systemmodel und das OS-Model. Im Taskmodel werden alle Parameter eines Tasks zusammengefasst und diese nach Phasen geordnet, wie in Formel 5.11 ersichtlich. Dies bedeutet, dass es Taskparameter gibt, wie z.B. die WCET oder Periode, die sich je nach Phase des Tasks ändern. Andere Parameter wiederum wie z.B. die benötigte Periphere, bleiben gleich und ändern sich mit der Phase des Tasks nicht. Ebenso ist beschreiben, wie Tasks in voneinander logisch abhängige und unabhängige Tasks geordnet werden, ersichtlich in Formel 5.8, um ein phasengesteuertes Scheduling zu ermöglichen. So gibt es einen Leading Task, der die aktuelle Phase anhand z.B. der Sensorik vorgibt. Die davon anhängigen Tasks (Depending Tasks) sind von der Bestimmung der Phase des LTs abhängig und stellen sich auf dessen vorgaben ein.
Das Systemmodel beschreibt die Architektur des Multi-Core Prozessors, die eingestellte Konfiguration und dessen verfügbare Peripherie. So können Core Einschränkungen bzgl. Verfügbarkeit und Architektur bei der KB Berechnung mit einbezogen werden, wie sie in Formel 5.12 beschrieben sind. Ebenso müssen aber auch die grundlegenden Parameter mit einbezogen werden wie die Anzahl der Cores und die generell Verfügbare Peripherie. Die Parameter sind weiter heruntergebrochen bis hin zu Periphere die u.U. nur auf einem Core verfügbar ist.
Zuletzt müssen noch die OS Prozesse betrachtet werden, die zur Administration benötigt werden. Diese werden im OS Modell dargestellt, welches sich an das Taskmodell anlehnt. Unterschied dazu ist, dass ein OS Prozess keine Phasen aufweist.

# 6 Knowledgebase Berechnungen und Erstellung

Nach der Parameter Ermittlung der auf dem System zu laufenden Tasks (Kapitel 5) werden die erforderlichen Algorithmen vorgestellt um einen gültigen phasenorientierten Schedule zu erhalten. Dies geschieht unter Berücksichtigung der Ziele aus Kapitel 1.2 und den Anforderungen aus Kapitel 4.2. In dieser Knowledgebase Berechnung gibt es mehrere Algorithmen, welche durchlaufen werden müssen, um zu einem Ergebnis zu kommen. Erstens wir anhand aller Tasks eine Baumstruktur aufgestellt um alle zukünftigen Systemkonfigurationen zu finden. Dies ist die Ausgangsbasis für alle weiteren Algorithmen. Als zweites muss anhand des ausgewählten Schedulingalgorithmus eine valide Taskverteilung anhand der Taskphasen und Multi-Core Abhängigkeit erstellt und auf Machbarkeit überprüft werden. Dies geschieht mittels einer Auslastungsberechnung. Als Drittes wird die Rekonfiguration überprüft mittels einer Rekonfigurationsberechnung. Als letztes muss eine Fail Operational (FO) Konfiguration erstellt werden, welche eingenommen wird, sobald ein Fehler im HAMS System erkannt wird. Der Ausgang dieser Algorithmen muss in eine Knowledgebase Datei umgewandelt werden, welche dem HAMS Scheduler bei der Systemkompilierung mitgegeben wird. Das sequenzielle Durchlaufen aller dieser Schritte für eine gegebenes Taskset bis hin zur Knowledgebase Datei stellt die KB Erstellung dar, die Algorithmen sind die KB Berechnung.

## 6.1 Ziele/Konzept der Knowledgebase Erstellung

Im ersten Teil der Knowledgebase Berechnung, der Taskverteilung, finden sich folgende Konzeptpunkte wieder (Kapitel 4.2):

- Handhabung des phasenorientierten Schedulings im Bereich von prioritätenbasiertem (statisch und dynamisch) und cyclischem Scheduling
- Unterstützung von periodischen, aperiodischen und sporadischen Tasks
- Allokation am sicheren Maximum von bekannten Feasibility Tests
- Ersetzen von Bin Packing Algorithmen durch Brute Force
- Aufstellen aller möglichen Phasenübergänge zur Überprüfung und Auswahl
- Überprüfung der Phasenübergänge auf zeitliche Dauer im HAMS Schema
- Unterstützung des phasenorientierten und cyclischen Schedulings durch Rekonfigurationstests
- Einbeziehen von Migrationseinschränkungen zwischen Cores (I/O Verfügbarkeit, CPU Taktraten und Task Einschränkungen)
- Berücksichtigung der CPU Taktratenübergangszeit und WCET bei Migration

Im zweiten Abschnitt der Knowledgebase Berechnung, der FO Konfiguration, werden nachfolgende Konzeptpunkte umgesetzt:

- Parameter zur Erkennung von Überschreitungen in Taskausführungszeit und Deadline
- Parameter für die Tasküberwachung anhand von Rückmeldungen in der KB
- Bereitstellung einer Backupstrategie in Form einer einzigen gültigen validen Verteilung

## 6.2 Taskverteilung

Zur Erstellung einer Knowledgebase ist in Abbildung 6.1 das Vorgehen zur KB Erstellung schrittweise abgebildet. Nach der Ermittlung der Eingangsparameter muss der Brute Force Bin Packing Algorithmus alle möglichen Taskverteilungen finden (Schritt 1) und diese zum nächsten Validierungsschritt weitergeben. In diesem Validierungsschritt überprüfen Feasibility Tests (Schritt 2) die Auslastung und geben bestandene Taskverteilungen an einen nächsten Schritt weiter. In Schritt 3 wird die Rekonfigurationsdauer geprüft und ein Vergleich auf valide similäre Lösungen durchgeführt. Zusätzlich werden in Schritt 3 alle bestandenen Taskverteilungen aufgelistet und bezüglich Power Performance und Auslastung untereinander verglichen. Ist dies erfolgt, wird im letzten Schritt 4 die KB als XML Datei erstellt. Die allgemeine Vorgehensweise und welche zusätzlichen Prüfungen für die Verwendung der HAMS Knowledgebase in einem HAMS System durchgeführt werden müssen, wird in den nachfolgenden Kapiteln erklärt. Dabei wird die Reihenfolge der Beschreibung der Algorithmen von Abbildung 6.1 geringfügig abweichen. Zum besseren Verständnis des Algorithmus werden die Feasibility Tests zuerst beschrieben.

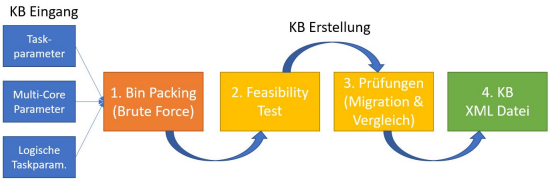



Abbildung 6.1: Vorgang zur KB Erstellung

### 6.2.1 Prioritätenbasiertes Scheduling

Der Schedulingalgorithmus für RMS und EDF ist prioritätenbasiert. Bei RMS ist die Priorität eines Tasks fest vorgegeben und wird offline generiert, wohingegen bei EDF die Priorität eines Tasks zur Laufzeit

(online) berechnet wird. Trotz dieser Unterschiede lassen sich die Feasibility Tests für beide Schedulingvarianten offline berechnen und so eine Aussage über die Einhaltung der Deadlines und Auslastung treffen. Diese Aussage ist von Bedeutung für das Erreichen der Ziele im Bereich Scheduling und Allokation. So lassen sich durch eine Vorausberechnung und Validierung die funktionstüchtigen und optimalen Allokationen im Vergleich untereinander finden.

### Phasenorientierter Feasibility Test

**RMS** Damit der Basis Liu und Layland Feasibility Test mit den Formeln 2.6 und 2.7 für das HAMS System anwendbar ist, müssen zwei Aspekte betrachtet werden. Zum einen verwendet der HAMS Scheduler einen Partitioned Scheduling Ansatz und durch die Ergebnisunabhängigkeit der Tasks untereinander ist eine globale Erweiterung des Feasibility Tests nicht nötig. Somit können die Partitioned Feasibility Tests angewandt werden. Zweitens muss die Phasenabhängigkeit der Tasks mit einbezogen werden. Dies ist im folgendem Algorithmus 1 berücksichtigt.

Der Algorithmus 1 ist in zwei Funktionen aufgeteilt. Die erste Funktion erstellt einen Baum (Teil 1), ausgehend vom Taskset ULT (Formel 5.4), das auf diesem Steuergerät untergebracht werden muss. Die Wurzel (rootnode) des Baumes wird mit allen Leading Tasks und deren Aufstartphasen initialisiert. Dieser Wurzelknoten bekommt in einem weiteren Schritt Blätter hinzugefügt. Das erste Blatt beinhaltet einen ausgewählten Leading Task (LT) und dessen abhängige Tasks (DT). Im nächsten Schritt (Teil 2) werden die Phasen des Leading Tasks betrachtet und ein weiterer Knoten erstellt für jede Phase des Leading Tasks und der entsprechenden abhängigen Task Konfigurationen (Abbildung 6.2). So wird sichergestellt, dass für jede mögliche Phase eines Leading Tasks und dessen abhängigen Tasks ein Blatt auf Ebene 2 im Baum vorhanden ist. Die weitere Funktion (Teil 3) ist ebenfalls unterteilt. Zuerst werden die OS Tasks und deren Auslastung der Variable U (Formel 2.6) hinzugefügt. Im weiteren Verlauf wird die rekursive Funktion „doRekursiveFeasibility“ aufgerufen (Teil 4). In dieser Funktion wird immer ein Blatt aus Level 1 für einen Leading Task herausgenommen. Anhand dessen Anzahl an Phasen (Level 2) wird eine Phase herausgenommen

**Algorithmus 1** Algorithmus für phasenorientiertes prioritätenbasiertes Scheduling - Create Tree

```
1:  function CreateTree(tree)
2:      for i=0; i < size(ULT); i++ do
3:          if ULT[i].isLeadingTask()==true then                ▷ Teil 1
                tree.rootnode.init(ULT[i].startupPhaseLT())
                inod1.add(ULT[i])
4:              for a=0; a < ULT[i].getDepTask.size();
                    a++ do
                        inod1.add(ULT[i].getDepTasks(a))
5:              end for                tree.addinod1List(inod1List)
6:          end if
7:      end for
8:      for i=0; tree.NrInods1() > i; i++ do                    ▷ Teil 2
9:          for a=0; tree.inod1[i].nrPhases() > a; a++ do
                tree.inod1[i].addinod2(tree.inod1[i].
                ULT[a].getPhaseTimes())
                tree.inod1[i].addinod2List(tree.inod1[i].
                getinod1List())
10:         end for
11:     end for
12: end function
```

**Algorithmus 2** Algorithmus für phasenorientiertes prioritätenbasiertes Scheduling - Rekursives feasibility Prüfung

```
1: function doRekursiveFeasibility(U, indexnode1, indexnode2)
   ▷ Teil 4
2:     if indexnode1==tree.nrinod1() then
3:         for i=0; tree.inod1[indexnode1].size() > indexnode2;
               i++ do
                   U+ = tree.inod[indexnode1].inod2[i].C / tree.inod[indexnode1].inod2[i].P
                   CheckAuslastung(U, taskcount)
4:         end for
5:     else
6:         while tree.inod1[indexnode1].size() > indexnode2 do
                   U+ = tree.inod[indexnode1].inod2[indexnode2].C / tree.inod[indexnode1].inod[indexnode2].P
                   doRekursiveFeasibility (U,
                   indexnode1++, indexnode2)
                   indexnode2++
7:         end while
8:     end if
9: end function

10: function FeasibilityTest(tree)
11:     for i=0; listOS.size() > i; i++ do                    ▷ Teil 3
          U+ = listOS[i].C / listOS[i].P
12:     end for
         doRekursiveFeasibility 0,0,0
13: end function
```

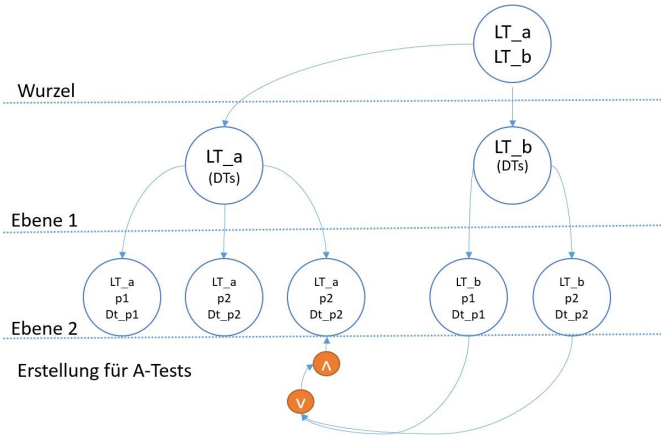



Abbildung 6.2: Die Ebenen des Baumes erstellt durch Algorithmus 1

und der Auslastungswert zu U hinzugefügt. Die rekursive Funktion wird solange aufgerufen, bis das Ende des Baumes, also der letzte Ebene 1 Knoten erreicht ist. Hier wird ebenfalls eine For-Schleife für alle Phasen durchgegangen und nach Hinzufügen jeder Auslastung für diese Phase der Auslastungstest durchgeführt. So ist sichergestellt, dass für jede Systemkonfiguration der Auslastungswert überprüft wird. Dies kann z.B. mit Hilfe des Liu und Layland Tests (Formel 2.7) oder mit DMS geschehen .
Anstelle des Liu und Layland Tests können auch andere Feasibility Tests zur Überprüfung der Auslastung herangezogen werden. Hier sind die bekannten T-Bound (Formel 2.8) und R-Bound (Formel 2.9) Tests ebenso anwendbar wie die Hyperbolic Bound Tests. Diese Tests sind ausschließlich für periodische Tasks anwendbar, aperiodische Tasks oder sporadische werden nicht unterstützt.
Hierfür muss der Polling Server verwendet werden (Formel 2.11). Der Polling Server für sich ist ein periodischer Task mit einer festen Ausführungszeit. In dieser Ausführungszeit ist es möglich, die aperiodischen und ggf. sporadischen Tasks zusammenzubringen. Bei aperiodischen Tasks wird der Parameter $A_i$ dazu genutzt, um diese in aperiodische Tasksets

von gleichen Perioden zusammenzufassen. Die Ausführungszeit, welche ein aperiodischer Taskset benötigt, wird anhand der Feasibility Tests berechnet und so die Laufzeit für den Polling Server angegeben. Diese Tasksets bekommen dann eine Periode und eine Ausführungszeit zugewiesen, die sich aus der Summe aller Ausführungszeiten im aperiodischen-Taskset ergibt. Anhand der RMS Feasibility Tests wird nun überprüft, ob der Task noch auszuführen ist.

**EDF** Ebenso wie bei RMS Feasibility Tests muss auch der EDF Feasibility Test angepasst werden. Auch hier gilt, dass aufgrund des lokalen Schedulings des HAMS Schedulers und der Ergebnisunabhängigkeit der Tasks, keine globale Feasibility Test Erweiterung durchgeführt werden muss. Für den Algorithmus der EDF Feasibility Berechnung können Teil 1, 2 und 3 übernommen werden ohne zusätzliche Änderung. Lediglich in Teil 4 muss der Feasibility Test geändert werden nach Formel 2.12, um so bei EDF die Auslastung nicht über 100% steigen zu lassen.
Für aperiodische und sporadische Tasks wird der Total Bandwidth Server (TBS) angewandt. Bei diesem Verfahren wird die restliche Auslastung, nach der Allokation aller periodischen Tasks, einem TBS Server zugewiesen. Online wird dann die Deadline des aperiodischen Tasks berechnet und diesem zugewiesen. Offline wird überprüft, ob die gesamte Auslastung dieser aperiodischen Tasks nicht den Maximalwert von 100% überschreitet. Sporadische Tasks können nicht überprüft werden, da sie keine feste Periode besitzen. Für diese muss ebenso die Deadline online berechnet werden, wenn die momentane Auslastungssituation auf dem Core dies zulässt.

**Konzept und Ziele** Mit Blick auf das HAMS Knowledgebase Konzept findet sich in Teil 1 und Teil 2 des Algorithmus 1 der erste Schritt des Ziels, alle möglichen Phasenübergänge aufzustellen, wieder. Dafür wird eine Baumstruktur verwendet und die Blätter der untersten Ebene sind die einzelnen Phasen. Teil 3 mit der Allokation von OS Tasks ist allgemein gültig und muss durchgeführt werden für ein funktionierendes System. Teil 4 bezieht sich auf das Scheduling. Somit finden sich hier insbesondere die Ziele, Verwendung von bekannten Feasibility Tests sowie der Einbezug von prioritätenbasiertem (statisch und dynamisch) Scheduling und die Einbindung von sporadischen und aperiodischen

Tasks wieder. Ein weiteres Ziel, das erfüllt wird, ist die Handhabung von phasenorientiertem Scheduling durch Ermittlung einer Baumstruktur aus Algorithmus 1.

### Verteilungsberechnung für phasenorientierte HAMS Systeme

**Basis HAMS Bin Packing Algorithmus** Die Algorithmen 1 von RMS und EDF beziehen sich lediglich auf Auslastungstests von partitioned Systemen. Ein Verteilungsalgorithmus, der Tasks über mehrere Cores hinweg verteilt, ist in diesen Tests nicht vorhanden. Für das Ziel der Unterstützung von Multi-Core Hardware, müssen Verteilungsalgorithmen vorhanden sein. Um dies zu ermöglichen, muss ein Bin Packing Verfahren angewandt werden. In diesem Verfahren stellen die Bins die verfügbaren Cores im Gesamtsystem dar. In die Bins werden nun Tasks allokiert, bis sie einen gewissen Schwellwert erreichen bzw. überschreiten. Der Schwellwert ist in diesem Fall die Schwelle des Feasibility Tests des zugrundeliegenden Schedulingverfahrens. So liegt in einem HAMS System mit EDF Scheduling die Schwelle jeder Bin bei 100%.
Bin Packing Algorithmen könnten angewandt werden, um die Tasks schnell auf die Cores zu verteilen. Diese Algorithmen haben aber den entscheidenden Nachteil, dass sie nicht immer die optimale Lösung finden bzw. auch keine Lösung finden, obwohl es eine Lösung gibt. Um dies auszuschließen, werden bei der Erstellung der KB Bin Packing Algorithmen wie First Fit und Next Fit o.Ä. nicht berücksichtigt. Stattdessen wird ein Brute Force Algorithmus angewandt, der alle möglichen Aufteilungen an Tasksets betrachtet und so die optimale Lösung findet, sofern es diese gibt. Der Nachteil ist der Rechenaufwand, der benötigt wird, um alle Variationen zu erstellen und zu vergleichen. Da die Berechnung aber offline geschieht und nicht zur Laufzeit ist dieser Nachteil vernachlässigbar.
Die Bin Packing Algorithmen für RMS und EDF teilen sich in zwei Abschnitte auf. Zuerst müssen alle möglichen Variation der Tasksets und Phasen generiert und erstellt werden. Desweiteren müssen die Multi-Core Eigenschaften mit einbezogen werden.

**Algorithmus 3** Bin Packing Algorithmus mit Phasen

```
1: function RotiertenRekursive(list_ret, list_in, index1)
2:     if index1==list.size-1 then
         list_in.putIndexToEnd(index1-1)
         list_ret.copy(list_in)
3:     else if index1==list.size-2 then
         list_ret.copy(list_in)
         RotiertenRekursivelist_ret, list_in, index1 + 1
4:     else
5:         while list.size − 2 > index1 do
6:             while nr_iterations < list.size − index1 do
                   RotiertenRekursivelist_ret, list_in, index1 + 1
                   list_in.putIndexToEnd(index1)
                   nr_iterations++
7:             end while
8:         end while
9:     end if
10: end function
```

**HAMS Bin Packing Algorithmus mit Phasen** Der Algorithmus 3 zeigt die Erweiterung des Algorithmus 1 um den Bin Packing Aspekt. Vor der eigentlichen Zuweisung der Tasks auf die Bins werden mit Hilfe eines Baumes aus Abschnitt 6.2.1 die Tasksets in logisch abhängige und unabhängige Blätter geordnet. Ist dies geschehen, wird eine rekursive Funktion „doRekursiveFeasibilityABinCheck“ aufgerufen, Algorithmus 4. In dieser Funktion wird zuerst eine Liste über die gesamten Leading Tasks mit deren abhängigen Tasks erstellt, und zwar auf Ebene der Phasen (Teil 1). So wird die Liste „list“ immer mit einer Phase eines bestimmten Leading Tasks und dessen abhängigen Tasks befüllt. Dies geschieht solange, bis der Algorithmus beim letzten Blatt auf Ebene 1 angekommen ist. Hier wird durch eine For-Schleife immer eine Phase dieses Tasks hinzugefügt (Teil 2). Ist die Liste vollständig, beginnt der eigentliche Bin Packing Algorithmus. Zuerst wird das Taskset rotiert, um alle möglichen Variationen zu überprüfen (Brute Force). Danach werden in Teil 3 die Tasks auf die einzelnen Bins allokiert. Aus der Liste wird ein Task herausgenommen und mit Hilfe des Liu Layland

**Algorithmus 4** Rekursiver Auslastungstests im prioritätenbasierten Bin Packing 1

```
1: function doRekursiveFeasibilityABinCheck(U, indexnode1,
   indexnode2, list2, coreid, Bins)
2:     if indexnode1==tree.nrinod1() then
3:         for i=0; tree.inod1[indexnode1].size() > indexnode2;
               i++ do
               list.add(tree.inod1[indexnode1].getLTaDTTasks())
                                                          ▷ Teil 2
               RotiertenRekursive(list, list2, 0)
4:              for a=0; list.size() > a; a++ do
5:                 for b=0; list[a].size() > b; b++ do
6:                     for t=0; Bins.size() > t; t++ do      ▷ Teil 3
7:                         if LimitierteFunktionen(list[a].get(b),
                           coreid) then
                                doAllocationCheck(list, a, b, t, Bins)
8:                         end if
9:                     end for
10:                end for
11:                if a==list.size() && allocate == true then
                    saveConfigurationToList(bins)
12:                end if
13:             end for
               list.remove(tree.inod1[indexnode1].getLTaDTTasks())
14:        end for
15:    else
        doRekursiveFeasibilityBBinCheck (tree, indexnode1,
        indexnode2, list)
16:    end if
17: end function
```

**Algorithmus 5** Rekursiver Auslastungstests im prioritätenbasierten Bin Packing 2

```
1: function doAllocationCheck(list, a, b, t, Bins)
       U_temp = List[a].get(b).C / List[a].get(b).P
2:     if Bins[t].U + U_temp ≤ T_t(2^(1/T_t) − 1) then
         Bins[t].U+ = U_temp Allokiere Task
         Bins[t].AddTask(List[a].get(b))
         allocate=true
3:     else
           Nächste Bin
4:     end if
5:     if t == MaxBins && !allocate then
         CheckFrequency()                                  ▷ Teil 4
6:       if !IncreaseFrequency(Bins[t], coreid) then
             Fehler Bei Allokierung
7:       end if
8:     else
         Solution.add(Bins)
9:     end if
10: end function

11: function doRekursiveFeasibilityBBinCheck(tree, indexnode1, indexnode2, list)
12:     while tree.inod1[indexnode1].size() > indexnode2 do
                                                           ▷ Teil 1
            list.add(tree.inod1[indexnode1].getLTaDTTasks())
            U_temp = List[a].C / List[a].P
            doRekursiveFeasibilityABinCheck (U + U_temp,
            indexnode1+1, indexnode2, list, coreid, Bins)
            list.remove(tree.inod1[indexnode1].getLTaDTTasks())
            indexnode2++
13:     end while
14: end function
```

Auslastungstests o.ä. überprüft, ob die Gesamtauslastung der Bin noch unter dem Schwellwert liegt. Ist dies der Fall, wird der Task auf diese Bin allokiert. Ansonsten wird solange überprüft, bis alle Bins untersucht worden sind. Konnte der Task nicht allokiert werden, so ist für diese Taskanordnung bzw. für diese Rotation keine gültige Konfiguration gefunden worden (Teil 4). Der Auslastungstest ist bestanden, sobald für jede Systemkonfiguration mindestens eine gültige Verteilung gefunden wurde. Diese Verteilung wird dann in einer Liste gültiger Verteilungen gespeichert.
Wie in den Bin Packing Algorithmen gezeigt wird, kann z.B. eine Sortierung der Tasks nach aufsteigender Auslastung ein besseres Verteilungsergebnis erbringen als eine ungeordnete Taskaufreihung. Damit wurde aufgezeigt, dass die Reihenfolge, nach der die Tasks auf die Bins allokiert werden, sich auf das Ergebnis auswirkt. Dies muss auch im Brute Force Algorithmus der KB berücksichtigt werden. In der Funktion „RotiertenRekursive“ wird das Taskset rotiert und die Tasks abwechselnd von ihrem Index an das Ende der Liste geschoben. Dies wird solange gemacht, bis alle Variationen/Iterationen gefunden wurden. Somit wird erreicht, dass alle Variationen von Taskverteilungen für eine erfolgreiche Allokierung auf die Bins überprüft werden (Brute Force).

**Konzept und Ziele** In Algorithmus 3 und Algorithmus 4 findet sich der zweite und komplett ausgearbeitete Teil der Ziele wieder, alle möglichen Phasenübergänge aufzustellen sowie Bin Packing durch Brute Force zu ersetzen. Brute Force wird in Algorithmus 4 Teil 2 veranschaulicht und durch das Rotieren werden alle möglichen Systemkonfigurationen zur späteren Überprüfung und Auswahl gefunden (Teil 3 Algorithmus 4 und Algorithmus 3 komplett). Ebenso finden sich die Ziele des Schedulings in Teil 4 von Algorithmus 4 wieder. Hier werden die Feasibility Tests ausgeführt und so sichergestellt, dass das Ziel einer Allokation am sicheren Maximum für die aktuelle Verteilung eingehalten wird. Verteilungen werden im Teil 1 und 2 von Algorithmus 4 erstellt (Brute Force) und mit bestandenem Feasibility Test in Teil 4 gespeichert und so das Ziel, Aufstellen aller möglichen Phasenübergänge zur Überprüfung und Auswahl, erreicht.

**Bin Packing Algorithmus mit MC System Vorgaben** In dem vorherigen Algorithmus 3 sind die Eigenschaften eines MC (Multi-Core) Systems nicht berücksichtigt bzw. werden durch den Funktionsaufruf „Limitierte Funktionen“ überprüft (Teil 3 von Algorithmus 4). Geschieht dies nicht, wird davon ausgegangen, dass jeder Core Zugriff auf alle Ressourcen hat. Da dies nicht immer der Fall ist, müssen eingeschränkte Tasks mit dem Algorithmus 6 vorab auf einer passenden Bin/Core allokiert werden.

**Algorithmus 6** Allokation limitierter prioritätenbasierter Funktionen

```
1: function LIMITIERTE FUNKTIONEN(Task, Core)
2:     if NP_ult ∩ AP_t  &&  NCP_ult ∩ ACP_t  &&
       NI_ult ∩ A_t then
           return true
3:     else
4:         if IsTaskAssigendManuallyToThisCore(Task, Core) then
               return true
5:         end if
6:     end if
       return false
7: end function
```

In diesem Algorithmus wird überprüft, ob der zu untersuchende Core Zugriff auf die nötigen Ressourcen hat. So wird zuerst untersucht, ob der Core die nötige Peripherie bereitstellen kann, danach ob der Core die gewünschten Eigenschaften, die der Task benötigt, aufweisen kann. Zuletzt werden die Interrupts und die manuelle Core Allokation untersucht.
Eine weitere Eigenschaft des MC Systems, welche noch nicht näher erklärt wurde, ist die Einstellung der Core Frequenz im System (Teil 4 von Algorithmus 4). Dies muss bei dem Auslastungstest mit eingeführt werden. Je schneller ein Core getaktet ist, umso mehr Rechenschritte kann er in der Sekunde ausführen. Somit hat der Task bei einer anderen Frequenz eine andere WCET (Kapitel 5.2). Da der HAMS Scheduler ausschließlich eine Rekonfiguration der Frequenz zur Phasenänderung zulässt, wird vor dem eigentlichen Auslastungstest die Frequenz festgelegt und dann das Set geprüft. Aus Performancegründen wird mit der nied-

rigsten Frequenz angefangen und solange geprüft, bis alle Frequenzen getestet wurden (Kapitel 3.1).

**Konzept und Ziele** Im Algorithmus 6 findet sich das Ziel wieder, Einschränkungen bei der Allokation und Migration von Tasks zwischen Cores mit einzubeziehen. Dies wird durch die interne Periphere, wie z.B. FPU und den Zugriff auf externe Periphere, wie z.B. URAT und die erforderlichen Interrupts geprüft (Algorithmus 6). Die Frequenz der einzelnen Cores wird ebenso mit einbezogen und überprüft, wie das manuelle Allokieren eines Tasks auf einem Core.

### Rekonfigurationstests

In einem phasengesteuerten HAMS System muss sich das System an Veränderungen, ausgelöst durch Events, anpassen. Diese Anpassung wird im HAMS Scheduler u.A. durch eine Migration von Tasks vollzogen. Um zu überprüfen, ob diese Migration in den gesetzten Rahmenbedingungen aus Formel 5.2 abläuft, werden die im Nachhinein vorgestellten Algorithmen bei der KB Berechnung angewandt.
Somit wid das Ziel, die Durchführbarkeit einer Rekonfiguration schon in der Designzeit zu überprüfen, erfüllt. Ebenso wird die Analyse zeigen, wie Rekonfiguriert wird, m gemäß den Anforderungen, keine unbekannten Zustände einzunehmen.

**Phasenänderung** Eine Phasenänderung findet statt, sobald ein logisch abhängiges Taskset die Notwendigkeit einer Phasenänderung erkennt. Dies wäre der Fall, wenn z.B. die Geschwindigkeit $> 30\frac{km}{h}$ annimmt und die Einparkhilfe zugunsten der Geschwindigkeistregelanlage in den Standby Zustand übergeht. Aus den Algorithmen 6 und 3 ist zu erkennen, dass es nicht nur eine, sondern mehrere valide Lösungen auf dem MC System geben kann. Ausgehend von der Formel 5.4 kann ein LT mehrere Phasen aufweisen, die nicht gleichzeitig aktiv sind. Die davon abhängigen Tasks haben exakt dieselbe Anzahl von Phasen wie der Leading Task. Der Parameter MPU des Leading Tasks gibt an, welche Phasenübergänge möglich sind. In einem logisch abhängigen Taskset ohne Übergangseinschränkungen sind $UE = P * (P - 1)$ (P ist die max. Anzahl an Phasen) möglich. Bei Einschränkungen sind die

nicht möglichen Übergänge abzuziehen. Im kompletten System $USys$ mit allen voneinander unabhängigen Tasksets sind die Übergänge laut Formel 6.1 möglich.

$$U_{tot} = \text{pred}_{i=1}^{n} UE_i \tag{6.1}$$

**Phasenänderungsprüfung - Formel** Bei der Phasenüberprüfung wird getestet, ob die Zeit, die der HAMS Scheduler benötigt, um die Tasks zu migrieren, die erlaubten Rekonfigurationszeiten, Parameter $CMM$ aus Formel 5.2, nicht überschreitet. Um dies zu überprüfen, muss eine Formel aufgestellt werden, die angibt, wie lange eine Migration im HAMS System dauert (Migrationschema) und zudem muss ein Algorithmus gefunden werden, der alle möglichen Phasenübergänge findet und mit dieser Formel überprüft.
Die Phasenübergangsprüfung ist abhängig vom Schedulingalgorithmus und dem zugrunde liegenden HAMS Scheduler (Migrationschema aus Abbildung 3.3). Bei einem prioritätenbasierten Schedulingalgorithmus ist nicht festgelegt, zu welchem Zeitpunkt die Tasks auf dem System laufen. Periodische Tasks laufen in ihrer Periode ausschließlich einmal, aber wann in diesem Zeitraum, ist nicht festgelegt [11]. Der genaue Zeitpunkt, wann der HAMS SLS läuft ist damit unbekannt. Somit ist eine genaue Berechnung der Rekonfigurationszeit nicht möglich.
Dennoch ist es möglich, die bei einer Rekonfiguration benötigt werden kann, mit folgender Vorgehensweise und Einstellungen, zu berechnen:
Um eine minimale Verzögerung bei der Rekonfiguration zu erhalten, läuft der SLS mit der höchsten Frequenz im System bzw. hat die gleiche Periode wie der Task mit der aktuell höchsten Frequenz der Phase. Die Rekonfiguration wird vom HAMS SLS gesteuert. Die Meldezeit vom Task zum SLS $t_{ttosls}$ beträgt maximal die SLS Periode (Abbildung 3.3). Dabei werden Inter-Prozessorkommunikationszeiten (im Mikrosekundenbereich) vernachlässigt, da diese eine Größenordnung kleiner sind als die Tasklaufzeiten (Millisekunden). Der SLS empfängt den Phasenänderungswunsch und ermittelt die Abweichung anhand der KB. Da die Cores nicht synchronisiert im HAMS System laufen, ist daher der zweite Schritt der Rekonfiguration mit einem Maximalwert zu berechnen. Hierbei gibt der Core mit den längsten Zeiten zwischen den Ticks die Maximaldauer an mit der Zeit $t_{doinx}$. Die SLS Nachrichten werden so

verschickt, dass alle Cores, abhängig von einer gemeinsamen Zeitbasis, die neue Konfiguration anwenden. Diese Dauer ist die größte Periode zwischen zwei Ticks.
Mit der Berechnung der Summe aus $t_{ttosls} + t_{doinx}$ kann die maximale Rekonfigurationszeit bestimmt und mit dem Parameter $CMM$ verglichen werden. Die Dauer, einen Task von einem Core auf einen anderen zu verschieben, wird wiederum vernachlässigt (Kapitel 5.2). Ist dies nicht mehr der Fall, kann eine Rekonfiguration in HAMS derzeit nicht stattfinden.
Eine weitere Überprüfung ist die Auslastung nach der Migration. Verändert sich durch die Migration kurzzeitig die Ausführungszeit der migrierten Tasks, u. a. durch aktivierten Cache, so muss die Auslastung neu berechnet werden bzw. auf einen Schwellwert geprüft werden (Abbildung 5.2). Diese Prüfung weicht nicht vom normalen Feasibility Test ab. Hierbei kann entschieden werden, ob eine temporäre Überauslastung (U>100%) toleriert werden kann (weiche Echtzeit).
Ein Beispiel soll dies verdeutlichen: Die Geschwindigkeitsregelanlage ändert ihre Phase von „Standby“ in „Aktiv“ und soll dabei auf einem anderen Core migriert werden. Befindet sich auf diesem Core die Einparkhilfe in „Standby“ kann es u.U. passieren, aufgrund der Prioritäten und der Verzögerung durch das Laden der Caches, dass die Deadlines der Einparkhilfe einmalig verletzt werden. Es ist nun am Systemdesigner zu entscheiden, ob diese Tankverteilung zulässig ist.

**Phasenänderungsprüfung - Algorithmus** Nachdem nun der Rekonfigurationstest mathematisch feststeht, muss noch ein Algorithmus 7 die Migrationen aller voneinander abhängigen und unabhängigen Phasenübergänge finden und prüfen.
Im Algorithmus 7 werden in Teil 1.1 alle validen Lösungen betrachtet, welche in 1 als durchführbar berechnet und in der Liste „listrecon“ gespeichert wurden. Da durch den Brute Force Algorithmus das gleiche Taskset mehrmals betrachtet wurde und u.U. auch mehrere valide Lösungen zu dieser Verteilung gefunden wurden, müssen diese Lösungen zuerst sortiert werden.
Ausgehend vom zu untersuchenden Eintrag wird das Leading Taskset ermittelt, welches sich verändert. Steht das LT fest, wird die komplette Liste auf Tasksets mit abweichenden Phasen des LTs (Teil 1.2) un-

**Algorithmus 7** Rekonfigurationstest für priotätenbasiertes Scheduling Teil 2

```
1: function CheckRekonfiguration(Lists, SList)                    ▷ Teil 2.1
2:     for a=0;a < SLists.size();a++ do
3:         for i=0;i < Lists.size();i++ do
4:             for y=0;y < Lists[i].size();y++ do
5:                 if SList.isValidTransition(Lists[i]) then
                                                                  ▷ Teil 2.2
                       SLS_f=getSLSFreqofMaxFreqTask(SList)
                       listc=getListofTasksToMigrate(SList, Lists[i])
                                                                  ▷ Teil 2.3
6:                     if isMigrationAllowedfornonLTTasks(listc) then
                                                                  ▷ Teil 2.4
                           low_f=getLowestFreqofChangTask(listc)
7:                         if CheckReconfigurationMaxTime
                           (listc, low_c) then
8:                             if CheckWCETSAfterMigration
                               (listc) then                       ▷ Teil 2.5
                                   allowedReconfList.add(SList,
                                   List[i], ListRekon)
9:                             end if
10:                        end if
11:                    end if
12:                end if
13:            end for
                   checkSimilarWriteResult(ListRekon)
14:        end for
15:    end for
16: end function
```

**Algorithmus 8** Rekonfigurationstest für priotätenbasiertes Scheduling Teil 1

```
1: function CheckSimilarSolution( )                          ▷ Teil 1.1
2:     for i=0;i < listrecon.size();i++ do
         LT=listrecon[i].getChanLT();
3:       for y=0;y < listrecon.size();y++ do
4:           if y! = i   &&   listrecon[y].getChanLT() == LT
                 &&   listrecon[y].hasDifPhase(LT) then
                       putSolutionToListsInOrder(Lists, listrecon[y])
                                                             ▷ Teil 1.2
5:           else if y! = i   &&
                 listrecon[y].getChanLT() == LT   &&
                 listrecon[y].hasSamePhase(LT) then
                       putSolutionToLists(SLists, listrecon[y])
                                                             ▷ Teil 1.3
6:           end if
7:       end for
         CheckRekonfiguration (Lists, SList)
8:     end for
9: end function
```

tersucht und auf gleiche Phasen dieses LTs (Teil 1.3) überprüft. Die Übereinstimmungen werden in den entsprechenden Listen festgehalten. Es gibt eine Liste, in der die Phase des LTs immer gleich bleibt (SList) und ein Array an Listen für die anderen Phasen des LTs (Lists). Ist dies geschehen, werden die Listen an die Funktion „CheckRekonfiguration“ übergeben.
In dieser Funktion wird zuerst eine Konfiguration aus der SLS Liste genommen sowie eine Konfiguration für eine andere Phase aus dem Listenarray „List“. Nun wird untersucht, ob dieser Phasenübergang erlaubt ist oder nicht (Teil 2.2). Danach wird die Frequenz des SLS bestimmt, wie in Kapitel 6.2.1, und die Tasks, die zwischen Cores migriert werden. Nach der Ermittlung muss geprüft werden, ob es erlaubt ist, dass Tasks, die nicht ihre Phase ändern, auch migriert werden dürfen. Da die Migration dazu führt, dass Taskzeiten sich verlängern muss überprüft werden, ob es für diesen Task erlaubt ist, auch im u.U. aktiven Zustand einmal seine Deadline zu verpassen (Teil 2.4) gemäß dem Parameter $CMM$. Ist das für alle Tasks erlaubt, wird in Teil 2.5 die niedrigste Frequenz aus der Liste zu migrierender Tasks bestimmt. Diese Frequenz gibt an, wie lange die Rekonfiguration dauert, bis alle FLSen im HAMS Scheduler die neuen Tasks übernehmen. Diese Zeit wird in Teil 2.5 noch mit der maximal zulässigen Gesamtdauer für die Migration überprüft, anhand des niedrigsten $CMM$ Wertes.
Eine weitere Prüfung ist der Check der WCETs nach der Migration. Hier muss nochmals ein Auslastungstest durchgeführt werden mit den verlängerten WCET Werten und dem Parameter $CMP$. Sind alle Prüfungen positiv durchlaufen, gibt es eine Liste erfolgreicher Rekonfigurationen. Da auch hier mehrere erfolgreiche Übergänge und Migrationen durch die Brute Force Methode als Ergebnisse ermittelt, aber nur eine Konfiguration zur Laufzeit eingenommen werden kann, muss das Ergebnis für diesen Phasenübergang verglichen werden gemäß 6.2.1 und ausschließlich ein Ergebnis wird daraufhin in die KB geschrieben.

**Einplanung HAMS SLS, FLS und Tick** Bei den vorherigen Auslastungstests und der Phasenübergangsprüfung wurden zwar OS Tasks mit eingeplant, aber nicht genauer beschrieben. Im HAMS System kann das Verhalten von zwei OS Tasks, nämlich dem SLS und FLS, sehr genau beschrieben bzw. bestimmt werden. Wie in Abschnitt 6.2.1 auf-

geführt, hat die Periode des SLS direkten Einfluss auf die Zeitdauer der Rekonfiguration. Je höher die Periode umso länger wird die Zeit $t_{ttosls}$. Um diese Zeitspanne so gering wie möglich zu halten, sollte der SLS die gleiche Periode wie der höchstfrequentierte Task im System haben. Die Rechenzeit, die der SLS benötigt, muss beim Auslastungstest mit einbezogen werden und zugleich wird dem HAMS SLS die höchste Priorität im System zugewiesen.
Der FLS, im Gegensatz zum SLS, ist ein Teil des OS und wird nach jedem Beenden eines Tasks automatisch aufgerufen. Dieser wählt dann anhand des Schedulingalgorithmuses und der ihm zugewiesenen Tasks den nächsten Task aus. Diese Zeit wird im Auslastungstest nicht berücksichtigt, da sie im Vergleich zu dem auf dem System laufenden Tasks zu gering ist. Befindet sich das System dennoch sehr nahe an der Auslastungsgrenze, sollten diese Zeiten mit einberechnet werden.
Bei der Erstellung der KB wird auch eine anzunehmende Einstellung für einen Systemtick ermittelt. Der Systemtick ist eine periodische Unterbrechung des aktuell laufenden Tasks auf dem jeweiligen Core und führt gleichzeitig zum Aufruf des FLS. Dieser überprüft dann, ob ein höher priorer Task ausgeführt werden muss. Damit die Deadlines im System eingehalten werden, muss der Tick auf die Frequenz des höchstprioren Tasks gesetzt werden, der aktuell auf dem System geschedult werden muss.

**Konzept und Ziele** Durch Algorithmus 7 werden die Ziele Rekonfigurationstest für prioritätenbasiertes Scheduling (Teil 2.2), Migrationseinschränkungen (Teil 2.4), Berücksichtigung von CPU Taktratenübergangszeit (Teil 2.4) sowie Migrationseinschränkungen wie die WCET Verlängerung bei der Migration (Teil 2.5) umgesetzt. Da dieser Algorithmus für mehrerer valide Lösungen diese Überprüfung unternimmt und dabei keine auslässt, wird auch hier das Brute Force Vorgehen unterstützt.

#### Vergleich gleicher valider Lösungen

Da die Auslastungs- und Rekonfigurationstests auf einer Brute Force Methode basieren können mehrere valide Lösungen entstehen (Kapitel 2.1.3). Aus dieser Anzahl von Lösungen muss diejenige ausgewählt

werden, die am „optimalsten“ geeignet ist. So spiegeln sich in diesem Punkt die Ziele und Anforderungen aus Allokation/Scheduling wieder. Zu berücksichtigen ist hier aber, dass die „Optimalität“ sich von System zu System unterscheiden kann und nicht quantitativ evaluiert werden kann. So ist zur Designzeit zu entscheiden, welche der drei folgenden Punkte bei der KB Erstellung besonders beachtet werden müssen, bzw wie der Systemdesigner „optimal“ definiert werden kann:

**Rekonfigurationsmöglichkeit**
: Sollte es mehrere valide Phasenkonfigurationen geben, muss auch der Phasenübergang betrachtet werden, insbesondere die Dauer des Überganges. Eine kürzere Übergangsdauer ist vorzuziehen, damit die Tasks nach der Rekonfiguration so auch schneller wieder verfügbar und schedulbar sind.

**Auslastung**
: Sollten mehrere valide Lösungen vorhanden sein, so ist die Auslastung, auf einem Core zu betrachten. Je höher die Auslastung umso mehr Leistung wird von dem Chip benötigt. So ist es möglich, die Taktfrequenz dieses Cores herunterzusetzen und so die Aufnahme an Energie zu verringern. Sollte das nicht möglich sein, da vorherige Prüfungen dies ausgeschlossen haben, so ist eine geringere Auslastung dennoch im Sinne der geringeren Leistungsaufnahme vorzuziehen. Durch die entstehende freie Zeit wird der „Idle Task“ im System öfter aufgerufen und somit ebenfalls die Leistungsaufnahme verringert.

**MC Power Performance**
: Sollte nach der Überprüfung der gleichen validen Lösungen keine Entscheidung getroffen werden können, so kann die komplette Leistungsaufnahme des MC Systems betrachtet werden. Ein MC System hat immer dann die geringste Leistungsaufnahme, wenn die Cores gleich konfiguriert laufen. D.h. die Auslastung der Cores sollte so gleich wie möglich sein, ebenso wie die Taktrate (sog. Load-Balancing) (siehe Kapitel 3.1).

Bei der KB Erstellung kann der Systemdesigner nun diesen drei Punkten eine Gewichtung zukommen lassen, so dass valide similäre Lösungen,

die nicht optimal sind, verworfen werden, solange bis eine einzige Verteilung pro Phase übrig bleibt. Der Konzeptpunkt „Aufstellen aller möglichen Phasenübergänge zur Überprüfung und Auswahl“ ist mit der Auswahlmöglichkeit in Form der Gewichtung komplett umgesetzt.

### 6.2.2 Zeitbasiertes Scheduling

Neben dem prioritätenbasierten Schedulingalgorithmus gibt es einen zeitlich basierten Schedulingalgorithmus. Dieses sog. cyclische Scheduling wählt anhand der aktuellen Systemzeit aus, welcher Task gerade den Core gewinnt oder auf seinen Rechenzeitpunkt warten muss. Dabei ist die Systemzeit in sog. Slots unterteilt, deren Taskzuweisung sich nicht ändert. Wie Tasks den Slots zugewiesen werden, wird im Folgenden geklärt.

#### Phasenorientierter Auslastungstest

**Cyclisches Scheduling** Im cyclischen Schedule sind die Zeitintervalle in Major und Minor Slots aufgeteilt. Der Major Slot besteht dabei aus einem Vielfachen von Minor Slots. Die Bestimmung ihrer Größen ist von auf dem Core zugewiesenen Taskperioden abhängig. Der Major Slot wird von dem kleinsten gemeinsamen Vielfachen (kgV) der Perioden bestimmt. Dies ist der Task mit der längsten Periode im Taskset bzw. mit der kleinsten Frequenz. Der Minor Slot ist der größte gemeinsame Teiler ggT der Perioden auf diesem Core. Der Idealfall für den Minor Slot wäre die kleinste Periode aus dem Taskset bzw. der Task mit der höchsten Frequenz.
Nachdem die Größe der Major und Minor Slots bestimmt wurde, muss die Auslastung ebenfalls geprüft werden anhand der Formel 2.5. Diese Prüfungen und Berechnungen der Major und Minor Slots wird ausschließlich auf dem lokalen Core angewandt, da durch die Asynchronität im System jeder Core andere Cycluszeiten haben kann. Im Algorithmus 9 ist die Vorgehensweise zur Auslastungsberechnung in einem phasenorientierten System vorgegeben. Dieser Algorithmus 9 baut ebenfalls auf die in Algorithmus 1 vorgestellte Baumstruktur auf. So werden hier die Blätter eines Baumes durchiteriert und so für jede mögliche Phasenkonfiguration die Auslastung überprüft. In Teil 2 wird die cyclische

**Algorithmus 9** Allokation limitierter cyclischer Funktionen

```
1: function CheckAuslastungCyclisch(U, taskcount)        ▷ Teil 1
2:     if U ≤ 1 then
         calculateKgV()
         calculateGgT()
         return true
3:     else
         return false
4:     end if
5: end function                                          ▷ Teil 2

6: function doRekursiveFeasibilityCyclic(U, indexnode1,
       indexnode2)
7:     if indexnode1==tree.nrinod1() then
8:         for i=0; tree.inod1[indexnode1].size() > indexnode2;
               i++ do
               U+ = tree.inod[indexnode1].inod2[i].C / tree.inod[indexnode1].inod2[i].P
               CheckAuslastungCyclisch(U, taskcount)
9:         end for
10:    else
11:        while tree.inod1[indexnode1].size()>indexnode2 do
               U+ = tree.inod[indexnode1].inod2[indexnode2].C / tree.inod[indexnode1].inod2[indexnode2].P
               doRekursiveFeasibility (U, indexnode1++,
               indexnode2)
               indexnode2++
12:        end while
13:    end if
14: end function
```

Auslastung auf ihren Maximalwert 100% überprüft. Deadlines und andere Multi-Core Anforderungen der Tasks werden in diesem Algorithmus nicht überprüft. Danach werden für Major und Minor Cycle die Zeiten berechnet.

**Kooperatives cyclisches Scheduling** Das rein cyclische Scheduling erlaubt per Definition ausschließlich periodische Tasks und keine aperiodischen oder sogar sporadische Tasks. Somit ist es nicht möglich, OS Tasks oder andere nicht periodische Tasks gezielt mit einzuplanen. Dazu muss diesen Tasks ein eigener Slot zugewiesen werden. Dieser Slot kann die Größe von mind. einem Minor Cycle oder mehreren Minor Cycles haben und wird mit EDF geprüft und betrieben für eine maximale Auslastung. Damit Tasks diesem Slot zugewiesen werden können, muss Folgendes neben der EDF-Auslastungsprüfung betrachtet werden:

1. Muss der Task innerhalb dieses Zeitraumes fertig werden bzw. hat der Task am Ende des Slots seine Deadline, muss genügend Rechenzeit für diesen vorhanden sein.

2. Der Abstand zum nächsten freien Slot muss so gewählt sein, dass die Minimum Separation Time eingehalten wird oder die Deadline eingehalten werden kann.

3. Sporadische Tasks werden jedem Slot zugewiesen, der nach Allokation der aperiodischen Tasks noch genügend freie Auslastung zur Verfügung hat

**Konzept und Ziele** Algorithmus 9 mit Teil 1 und Teil 2 ermöglicht es, auch ein cyclisches Scheduling in der HAMS Knowledgebase zu berechnen. Mit der Verwendung der Algorithmen 1 und 9 finden die Ziele eine Betrachtung wie: Verwendung von bekannten Feasibility Tests, Handhabung von phasenorientierten Scheduling, Unterstützung von periodischen aperiodischen und sporadischen Tasks .

#### Verteilungsberechnung für phasenorientierte HAMS Systeme

**Cyclischer Bin Packing Algorithmus für MC Systeme** Der Bin Packing Algorithmus eines cyclischen Schedules in phasengesteuerten

Multi-Core Systemen besteht nicht nur aus der Verteilung der Tasks auf die Bins, sondern auch aus der Verteilung der Tasks auf die Slots. Somit ist der Basis Bin Packing Algorithmus in zwei Teile unterteilt. Im ersten Teil, in der Brute Force Methode, werden die Tasks in rotierender Weise auf die Cores verteilt und deren Auslastung geprüft. Im zweiten Teil werden jedem Minor Cycle zuerst die periodischen und danach aperiodischen Tasks zugewiesen, und zwar solange bis alle Tasks einen festen Slot haben. Das Bin Packing findet unter Einbindung der jeweiligen Phase statt und wird im folgenden Algorithmus 12 dargestellt.
In diesem Algorithmus 12 finden sich zwei Teile bzw. Funktionen wieder. Der erste Teil ist in der rekursiven Funktion „doRekursiveFeasibilityABinCheck“ zu finden. Hier wird der in Algorithmus 1 erstellte Baum durchiteriert, um alle möglichen Systemkonfigurationen zu überprüfen. Hierfür werden die aktuell zu untersuchenden Phasen in einer Liste gespeichert (Teil 1). Wie im Algorithmus 3 werden die in der Liste befindlichen Tasks rotiert, um so einen Brute Force Algorithmus zu gewährleisten (Teil 2). Danach wird für alle Tasks in der Liste eine passende Bin gesucht, d.h. die aktuelle Auslastung der Bins überprüft, ob diese nicht den Schwellwert überschreitet (Teil 3). Sind alle Tasks erfolgreich allokiert, wird der zweite Teil des Algorithmus ausgeführt, die Zuweisung der Tasks auf die entsprechenden Minor Slots.
In der Funktion „AssignTasksToMinorCycle“ des Algorithmus 12 werden zuerst der Minor und Major Cycle für die aktuelle Bin berechnet und gesetzt. Danach wird jeder Task entsprechend dem Verhältnis $\frac{MajorCycletime}{Periodtime}$ zugewiesen. War diese Zuweisung korrekt, wird die Einhaltung der Deadlines und der Perioden des Tasks innerhalb des Major Cycles und zwischen zwei Major Cyclen geprüft. Gibt es ein gültiges Ergebnis für jede Bin im System, wurde eine valide Konfiguration gefunden. Diese wird dann in einer Liste von gültigen Konfigurationen abgespeichert.

**Bin Packing Algorithmus mit MC Systemvorgaben** Auch bei einem zeitlich basierten Schedulingalgorithmus gibt es MC Systemvorgaben, die berücksichtigt werden müssen. Hierfür kann der gleiche Algorithmus wie in Algorithmus 6 angewandt werden. In diesem Algorithmus werden Tasks den jeweiligen Cores zugewiesen, die nur auf bestimmte Peripherie und Interrupts Zugriff haben.

**Algorithmus 10** Rekursiver Auslastungstest im cyclischen Bin Packing

```
1:  function doRekursiveFeasibilityABinCheck(U,
              indexnode1, indexnode2, list2)                    ▷ Teil 1
2:      if indexnode1==tree.nrinod1() then
3:          for i=0; tree.inod1[indexnode1].size() > indexnode2;
              i++ do
              list.add(tree.inod1[indexnode1].getLTaDTTasks())
                                                                ▷ Teil 2
              RotiertenRekursive(list, list2, 0)
4:            for a=0; list.size() > a; a++ do
5:              for b=0; list[a].size() > b; b++ do
6:                for t=0; bins.size() > t; t++ do            ▷ Teil 3
7:                  if LimitierteFunktionen(list[a].get(b)) then
                              doAllocationCheck(list, a, b, t)
8:                  end if
9:                end for
10:               if AssignTasksToMinorCycle(Bins) then
                                                                ▷ Teil 4
                    allocate2==true
11:               end if
12:             end for
13:           end for
              list.remove(tree.inod1[indexnode1].getLTaDTTasks())
14:         end for
15:     else
16:         while tree.inod1[indexnode1].size()>indexnode2 do
                                                                ▷ Teil 1
              list.add(tree.inod1[indexnode1].getLTaDTTasks())
              U_temp = List[a].C / List[a].P
              doRekursiveFeasibilityABinCheck (U + U_temp,
              indexnode1+1, indexnode2)
              list.remove(tree.inod1[indexnode1].getLTaDTTasks())
              indexnode2++
17:         end while
18:     end if
19: end function
```

**Algorithmus 11** Rekursiver Auslastungstest im cyclischen Bin Packing

```
1: function doAllocationCheck(list, a, b, t, Bins)
            U_temp = List[a]..get(b).C / List[a].get(b).P
2:      if bins[t].U + U_temp ≤ then                          ▷ Teil 3.1
            bins[t].U+ = U_temp
            bins[t].AddTask(List[a].get(b))
            allocate1=true
3:      else
            Nächste Bin
4:      end if
5:      if t == MaxBins  &&  allocate1 == false then
            Fehler Bei Allokierung
6:      end if
7: end function
```

D.h. auch im Algorithmus 10 muss dieser Funktionsaufruf stattfinden, bevor die Tasks den jeweiligen Major und Minor Slots zugewiesen werden und die Auslastung überprüft wird. So muss die Zuweisung der limitierten Tasks des Systems vor dem Auslastungstest stattfinden. Bei der Rotation der Tasks werden limitierte Tasks wie alle andere Tasks behandelt, da es in diesem System keine zeit- bzw. ergebnisabhängigen Tasks gibt.

#### Rekonfigurationstests

Auch im cyclischen Schedule muss jeder Phasenübergang geprüft und verifiziert werden (siehe Algorithmus 7). Sobald dies geschehen ist, wird eine gültige Taskverteilung in die KB aufgenommen. Ebenso findet hier eine Unterteilung der Rekonfigurationstests in eine Formel und einen Algorithmus statt.

**Phasenänderungsprüfung - Formel** Im Gegensatz zum prioritätenbasierten Scheduling sind in einem cyclischen Schedule die Zeiten bzw. Slots, in welchem die Tasks laufen, bekannt. So lässt sich nicht nur die Maximaldauer berechnen, sondern eine genaue Zeitspanne für die Dauer der Rekonfiguration. Dadurch kann die prioritätenbasierte Formel

**Algorithmus 12** Algorithmus für phasenorientiertes cyclisches Scheduling

```
1: function AssignTasksToMinorCycle(bins)
2:     for t=0;bins.size() > t;t++ do
         bins[t].calculateMajorCycle(bin[t].getTaskList())
         bins[t].calculateMinorCycle(bin[t].getTaskList())
         bins[t].setMinorandMayorSlots()
3:       for a=0;a < bins[t].nrTasks;a++ do
4:         for b=0; b<bins[t].nrMinSlots; b++ do
5:           if bins[t].minslot(b)==empty then
               bins[t].minslot(b).assignTask(bins[t].getTask(a));
6:             if bins[t].getTask(a).
                   isAssigendFullyInMayorCycle() then
7:               if !checkDeadlines(Taskset[a]) then
                     abort=true;
8:               end if
9:             end if
10:          end if
11:          if bins[t].getTask(a).isMinCycleFully() then
                 b+=bins[t].getTask(a).getMin();
12:          end if
13:        end for
14:      end for
15:    end for
16:    if t == bins.size()  &&  !abort then
         saveConfigurationToList(bins);
17:    end if
18: end function
```

$t_{ttosls} + C_{sls} + t_{doinx}$ auf exakte Zeiten erweitert werden.

$$T = \Delta(SLS_t - LT_t) + C_{sls} + \lfloor (FLS_{bin} - SLS_t) \rfloor \tag{6.2}$$

Das Delta gibt die Differenz an zwischen dem aktuellen Zeitpunkt des LTs und dessen Meldung eine Phasenänderung zu vollziehen und des nächsten Aufrufes des SLS. Auch hier wird der SLS mit derselben Frequenz wie der höchstfrequentierte Task geschedult, um die Reaktionszeiten so gering wie möglich zu halten. Im zweiten Teil wird der Zeitpunkt ausgerechnet vom Senden des Rekonfigurationsbefehls des SLS bis zur Umsetzung in den FLSen der Cores. Auch hier macht sich die Asynchronität des Systems bemerkbar und hat einen Einfluss auf die Gesamtdauer. Da die Minor und Major Slots weder synchronisiert noch auf allen Cores gleich sind, findet auch bei diesem HAMS Rekonfigurationsschema eine Rekonfiguration ab einem bestimmten Zeitabschnitt in der Gesamtsystemzeit statt. Dieser Zeitabschnitt ist die maximale Differenz zwischen dem Ende des SLS und dem letzten FLS Aufruf mit dem größten Abstand zu diesem unter Berücksichtigung aller Cores.
Sollte sich durch eine Migration des Tasks dessen Ausführungszeit verlängern, so kommt im cyclischen Schedule keine temporäre Überlast zusammen, sondern Tasks benötigen mehrere Slots, um alle Daten zu laden und in ihre normale WCET zurückzukehren. Bei cyclischen Tasks kann angegeben werden, wie lange die Zeitspanne dafür ist und ob sie die Grenzen überschreitet mit dem Parameter $CMP$.

**Phasenänderungsprüfung - Algorithmus** Der Algorithmus für eine Phasenübergangsprüfung im cyclischen Schedule ist ähnlich dem prioritätenbasierten Algorithmus. Der Unterschied liegt hier im Teil 2.5 von Algorithmus 7. Im cyclischen Schedule werden nach Formel 6.2 die Phasenübergange geprüft, und zwar nach der exakten Dauer.

### Weitere Cyclische Eigenschaften

Der Vergleich valider similärer Lösungen im cyclischen Schedule entspricht dem Vergleich von prioritätenbasiertem Scheduling. Auch hier werden die Ergebnisse in Bezug auf Auslastung, Rekonfigurationsmöglichkeiten oder MC-Performance geprüft und ggf. ausgewählt. Zusätzlich kann beim cyclischen Schedule der genaue Zeitpunkt, d.h. Minor Slot,

angegeben werden. So kann bei der Auslastung nicht nur die Gesamtsystemauslastung bzw. die Auslastung pro Core überprüft werden, sondern auch die Verteilung auf die einzelnen Minor Slots. Es kann hier auf eine gleichmäßige Verteilung geachtet werden.

**Konzept und Ziele** Im letzten Abschnitt ist die Überprüfung der Phasenübergänge auf zeitliche Dauer im HAMS Schema eindeutig beschrieben. Ebenso wird das Ziel der Berücksichtigung der WCET bei der Migration und der CPU Taktraten unter cyclischem Scheduling ausgeführt.

## 6.3 Scheduling im Fehlerfall

In den Systemen der Zieldomänen können zu jeder Zeit unvorhergesehene Fehlerfälle auftreten, die von den Funktionen nicht abgefangen werden können. Dadurch entsteht ein unerwartetes Laufzeitverhalten der Funktionen und andere Tasks können blockiert werden. Das HAMS System und insbesondere der SLS Scheduler stellen in solchen Fällen eine „Fail-Operational“ Funktionalität des Steuergerätes her, um eine grundlegende Funktionalität zu gewährleisten, bis der Fehler von außerhalb behoben werden kann.
Wie es im Ziel Fehlerbehandlung / Überwachung erörtert ist, werden die Fehler und wie sie zu erkennen sind aufgezeigt.

### 6.3.1 Erkennung von Soft- und Hardware-Fehlern im HAMS System

Um Fehler zu erkennen, muss der SLS mit den nötigen Parametern ausgestattet sein, um eine Online-Analyse der Tasks zur Laufzeit durchführen zu können. Im HAMS System können Soft- oder Hardwarefehler auftreten. Bei Softwarefehlern zeigen Tasks ein unerwartetes Laufzeitverhalten, was bedeutet, dass sie sich selbst komplett beenden, von OS Schutzmechanismen beendet werden oder längere Rechenzeit als ihre WCET benötigen. Bei Hardwarefehlern funktionieren die Tasks korrekt, können aber z.B. mit anderen Steuergeräten oder Sensoren nicht kommunizieren. Es wird somit nur ein Teil ihrer Funktion ausgeführt.

Die nötigen Parameter, die vorhanden sein müssen, sind abhängig von der Art des Fehlerfalls, Soft- oder Hardware und vom verwendeten Scheduling Algorithmus, prioritätenbasiert oder cyclisch.

**Softwarefehler und prioritätenbasierts Scheduling**

In dieser Laufzeitkombination kann es zu einem Komplettausfall eines ganzen Cores kommen. Insbesondere bei RMS, wenn der Task, mit der aktuell auf dem Core höchsten Priorität, den Core nicht mehr frei gibt für andere Tasks (nicht so bei EDF siehe Kapitel 2.1.2). Auch wenn der Task nicht die höchste Priorität besitzt, ist es nötig, dem fehlerhaften Task dennoch Beachtung zu schenken. Somit ist nicht nur der fehlerhafte Task von seinem Fehlverhalten selbst betroffen, sondern auch andere Tasks werden dadurch in Mitleidenschaft gezogen. Aus diesem Grund ist eine schnelle Erkennung des Fehlers nötig. Erschwert wird dies aber durch den RMS und EDF Algorithmus, da Tasks sich selbst managen und nicht in bestimmten Zeitabschnitten rechnen. Deshalb müssen bei periodischen RMS und EDF Tasks die Deadlines und Periodendauer eines Tasks zur Fehlererkennung mit angefügt und bei der KB Datei zum Auslesen mit angegeben werden. Bei aperiodischen und sporadischen Tasks muss dementsprechend die maximale Aufrufzeit mitgegeben werden.

**Softwarefehler und cyclisches Scheduling**

Bei dieser Laufzeitkonfiguration kommt es in einem Softwarefehlerfall zu keiner Beeinträchtigung des Gesamtsystems. Benötigt ein Task mehr Rechenzeit als vorgesehen, wird er am Ende seines zugewiesenen Slots vom FLS Scheduler beendet. Ein Abschalten dieses Tasks ist notwendig, denn sobald ein Task einen undefinierten Zustand erreicht, können u.U. durch Speicherbereichsüberschreitungen andere Tasks beeinträchtigt werden. Die automatische Umschaltung eines Tasks zu einem anderen Task am Ende des Slots macht es schwieriger, diese Art von Fehler zu erkennen. Möglich ist die Rückmeldung der Tasks mit dem Parameter $CP_{X.m}$ aus Kapitel 5.1.2. So kann der FLS mit dem ausgelesenen Parameter der KB erkennen, dass binnen der Ausführungszeit die nötige Rückmeldung ausblieb oder zu spät gekommen ist. Sollten in einem cyclischen Schedule aperiodische und sporadische Tasks

vorkommen, so werden diese wie prioritätenbasierte Tasks in ihrem zugewiesenen Slot auf die Länge ihrer Laufzeit und die Einhaltung ihrer Periode geprüft.

**Hardwarefehler**

Hardwarefehler können vom HAMS System an sich nicht erkannt werden, da dieses keine Hardware Schnittstellen benötigt und diese nicht überprüft. Die Diagnose der Schnittstellen muss entweder vom Task selbst oder durch einen periodischen (Build in Self Test) Selbstdiagnose (BIST) durchgeführt werden, was wiederum ein eigenständiger Task ist. Der HAMS Scheduler bekommt vom BIST oder einen seiner Tasks Rückmeldung über die Funktionalität der Schnittstellen bzw. Hardware. Dies geschieht u.a. durch Einnahme einer bestimmtem Phase oder er sendet mit der HAMS API (HAPI) direkt an den FLS, dass er einen Hardware Fehler entdeckt hat.

So kann eine Phase bedeuten, dass der Task Schnittstellen nicht mehr benutzen kann, aber dennoch versucht, seine normale Funktionalität wiederaufzunehmen. Sollte so ein Signal oder Phase in einem Task vorhanden sein, muss dies in der KB für den HAMS Scheduler eingetragen werden. So können bestimmte Phasen auf einen Hardwarefehlerfall hinweisen oder Signale für diesen Fall definiert werden. Am sinnvollsten ist eine Kombination von beiden. So kann der Task ein Signal an den FLS senden und ebenso eine Phase einnehmen, in welcher der Task noch seine Restfunktionalitäten ausführen kann. Die Signale und müssen in der HAMS Knowledgebase mit eingebracht werden, damit der HAMS Scheduler diese später richtig interpretiert.

### 6.3.2 Fail-Operational Funktion

Um das HAMS System im Fehlerfall dennoch betreiben zu können, wird ein Fail-Opera- tional Status hergestellt, um das System, wie im Ziel „Fehlerüberwachung / Überprüfung“ vorgestellt, mit eingegrenztem Fehler minimal betreiben zu können. Dadurch können Fehler nicht verhindert werden, aber die Auswirkungen auf das Gesamtsystem werden begrenzt. Dieser Status ist definiert und tritt erst dann ein, sobald ein Task Fehlverhalten gemeldet oder erkannt wird. Damit dieser Status

nicht online berechnet werden muss und einen definierten Zustand hat, wird in der KB dieser Status zum Auslesen hinterlegt.
Auch kann für jeden vorkommenden Fehlerfall ein eigener Schedule in der KB berechnet und dem HAMS System mitgegeben werden. Auf diese Weise kann auf jede Art von Fehler optimal reagiert werden. Das HAMS System an sich unterstützt z.Z. aber nur die Einnahme einer einzigen Fail Operational (FO) Konfiguration unabhängig von der Fehlerart und Systembetroffenheit. Damit ist es umso wichtiger, eine stabile und minimale FO Konfiguration zu finden und zur Laufzeit herzustellen. Welche Tasks in einem Fehlerfall zur Verfügung stehen müssen, ist zur Designzeit zu entscheiden. Die entsprechenden Leading und Depending Tasks können dann mit einer Phase und dem Parameter $NFO_m$ gekennzeichnet werden. Im FO Zustand können ggf. Leading Tasks ohne ihre Depending Tasks laufen und umgekehrt. Damit wird die logische Verknüpfung von Tasksets aufgegeben, um eine minimale Operation der ECU gemäß den Zielen zu gewährleisten.
Zur Berechnung dieser Konfiguration, wird der EDF Algorithmus verwendet. Dieser Algorithmus hat im Gegensatz zu RMS und Cyclisch den Vorteil, sich dynamisch an die neuen Gegebenheiten anzupassen (Kapitel 5.1.2). Sollte der fehlerhaft agierende Task in der FO Konfiguration vorhanden sein, wird er durch die Prioritätenberechnung von EDF erkannt und vom HAMS System geblockt, so dass andere Tasks nicht an ihrer Ausführung gehindert werden. Sollten mehrere fehlerhafte Tasks in der FO Konfiguration vorhanden sein, werden auch diese durch die EDF Fehlerbehandlung erkannt und geblockt.
Um eine korrekte Verteilung aller Tasks der FO Konfiguration zu gewährleisten, muss auch für diesen Fall eine Verteilungs- und Auslastungsberechnung durchgeführt werden. Hierbei wird der Algorithmus 3 dahingehend erweitert, dass in der Liste „List“ nur diese Tasks aufgenommen werden, die mit dem erforderlichen $NFO_m$ Parameter gekennzeichnet wurden. Die Auslastung und die Verteilung der Tasks wird dann wie gehabt fortgeführt, ebenso die Rotation, um eine optimale Verteilung zu berechnen. Sollte aber zur Laufzeit ein Core oder mehrere Cores ausfallen, wie z.B. beim P4080, sind die Tasks, die auf die ausgefallenen Cores zugewiesen wurden, nicht mehr verfügbar. Damit der SLS auch im Ausfall eines oder mehrere Cores verfügbar ist, wird jedem Core der SLS zugewiesen.

## 6.4 Datenformat der Knowledgebase

Nach der Berechnung aller möglichen Konfigurationen und der Auswahl einer passenden Taskverteilung für die jeweilige Konfiguration, muss das Ergebnis der KB Berechnung für den HAMS Scheduler auslesbar abgespeichert werden. Zu diesem Zweck dient die (Extensible Markup Language) - Erweiterbare Auszeichnungssprache (XML) Beschreibungssprache zur Darstellung der KB Berechnungsergebnisse.
Wie diese KB Ausgangsdatei aufgebaut werden muss, damit der HAMS Scheduler ohne online Berechnungen schedulen kann, wird im Folgenden erklärt. Eine vollständige KB Datei ist im Anhang zu finden 9.1.

### 6.4.1 XML Aufbau

Die KB Ausgangsdatei ist in drei Abschnitte unterteilt und wird wie folgt vom HAMS SLS gelesen:

#### Systemeigenschaften in der KB

Der erste Abschnitt spiegelt die vorhandenen Systemeigenschaften wieder. D.h. hier werden die KB Eingangsparameter aus den Tupeln $C_i$ und $MC$ aus Formel 5.12 wiedergegeben. So kann der HAMS SLS die aktuellen Systemparameter, wie Core Anzahl und Frequenz mit seinen aktuellen Systemparametern vergleichen und Rückschlüsse dahingehend ziehen, ob die KB Ausgangsdatei für dieses System anwendbar ist. Somit ist das Laden einer nicht unterstützten KB Ausgangsdatei oder mit falschen Eingangsparametern beschriebenen KB Ausgangsdatei unterbunden. Ebenso sind hier die Peripherie Geräte und die entsprechenden Interrupts sowie OS Systemtasks aufgelistet.
Im zweiten Abschnitt und zugleich dem größten Anteil der KB Ausgangsdatei, stehen die einzelnen Systemkonfigurationen für die jeweilige Phase. So kann anhand der eindeutigen Phasenbezeichnung aller laufenden Tasks, die Systemkonfigurationsbeschreibung für die jeweilige Phase gesucht und angewandt werden.
Im letzten Abschnitt befindet sich die Systemkonfiguration für den

Fail-Operational Betrieb. Hier werden die Tasks und Systemprozesse aufgelistet, die auch noch im Fehlerfall geschedult werden sollen.

### Taskset und Phasenerkennung für SLS/FLS in XML

Den Hauptteil der KB Ausgangsdatei stellen die verschiedenen Systemkonfigurationen dar, die es für jede mögliche Phasenkonfiguration geben muss. Damit der HAMS SLS, die Systemkonfiguration richtig einlesen und anwenden kann, müssen folgende Regeln und Attribute in einer Systemkonfiguration umgesetzt sein:
Jede Systemkonfiguration muss mit einer eindeutigen und einmaligen ID versehen sein. Hierbei kann die ID eine fortlaufende Nummer sein oder eine eindeutige und einmalige Zeichenkette als Phasenbezeichnung, die unabhängig vom Core angibt, welche Phasen aller Tasks in dieser Konfiguration aktiv sind. Mit Hilfe dieser Zeichenkette und einem Vergleich der aktiven Phasen ist es dem HAMS Scheduler möglich, schnell die entsprechenden Konfiguration zu finden und umzusetzen. In Anhang 9.1 ist unter dem Attribut „<Beschreibung>“ die Phasenbeschreibung zu finden, wie z.B. “Parkgeschwindigkeit/ Init_Norm/ Scheibe_Nass/ Kworker0_Norm/ Kworker1_Norm/ SLS_Norm/“. Zusätzlich zu dieser eindeutigen Beschreibung ist der Systemkonfiguration („<Configuration>“) im Anhang 9.1 eine eindeutige ID zugewiesen unter dem Attribut „<ID>“.
Nach der Identifizierung der eindeutigen Systemkonfiguration wird diese weiter unterteilt. So müssen je Systemkonfiguration alle Cores aufgelistet sein. Handelt es sich z.B. bei dem System um ein Quad-Core System mit vier Cores, so müssen für jede Systemkonfiguration alle vier Cores aufgelistet sein, mit deren jeweiligen Frequenzeinstellungen und allokierten Tasks, auch wenn ein Core keinen Task beinhaltet. Um die Cores in der KB voneinander zu unterscheiden besitzt eine Systemkonfiguration Elemente mit der Bezeichnung „<Core>“ in Abhängigkeit von der Core Anzahl. Jeder Core hat ein Attribut „<Core_ID>“, welches angelehnt ist an die Core Beschreibung unter Linux.
Jeder Core in der Systemkonfiguration hat eine Systembeschreibung (Abbildung 6.3). So muss für jeden Core ein Schedulingalgorithmus angegeben werden und ob es sich hierbei um ein prioritätenbasiertes Scheduling handelt (RMS oder EDF) oder um cyclisches Scheduling

(Attribut „<Sched_Class>“ des Elements Core). Insbesondere ist hier auch anzugeben, ob es sich um einen kooperativen Schedulingalgorithmus handelt. Werden z.B. asynchrone und sporadische Tasks auf diesem Core gleichzeitig zu einem cyclischen Schedule angegeben, so muss auch dies in diesem Attribut vermerkt sein (Attribut „<Use_Only_RMS>“ des Elements Core).
Als Nächstes müssen die spezifischen Einstellungen für jeden Schedulingalgorithmus im Attribut Core aufgelistet werden. Im Falle von RMS und EDF müssen hier die Ticks eingestellt werden, in denen der FLS regelmäßig prüft, ob ein höherpriorer Task den Core gewinnt. Diese Ticks wurden vorher in der KB berechnet und validiert (Attribut „<RMS_Tick>“ der Abbildung 6.3). Im Falle eines cyclischen Schedulings stehen hier die Minor und Major Cycle für den FLS, die ebenfalls zur Einplanung des Systemticks dienen. Sollte es sich um einen Scheduling Server handeln, stehen hier Start- und Endzeiten des jeweiligen Servers in Abhängigkeit des Major Cycles.
nach den vorgenannten Attributen werden noch zwei weitere Eigenschaften benötigt. Zum einen muss die Core Frequenz für den jeweiligen Core festgehalten werden (Attribut „<Frequency>“) und zum anderen muss eine Angabe darüber gemacht werden, wie viele Tasks auf diesem Core laufen, damit der SLS diese richtig auslesen kann (Attribut „<Tasks_on_Core>“). Nach diesen allgemeinen Core Einstellungen befindet sich eine Liste von Tasks, die auf diesem Core laufen (Abbildung 6.3). Neben dem Namen (Task_Name) und einer eindeutigen ID des Tasks (Task_ID) muss zu erkennen sein, in welcher Phase der Task läuft (Sub_Phase). So muss hier exakt diese Zeichenkette vorhanden sein, die vom Task an den FLS gemeldet wird, um zu erkennen und um zu überprüfen, in welcher Phase sich der Task befinden muss. Ebenso werden die weiteren Parameter angegeben, abhängig vom Schedulingalgorithmus und Feasibility Test.
Bei RMS und EDF geschedulten Tasks werden die Deadline, die WCET und die Periode mit angegeben. Bei EDF dienen die Parameter zur Online-Berechnung der Priorität und ebenso zur Tasküberwachung. Bei RMS werden ebenso Deadline, WCET und Periode angegeben zur Online-Überwachung. Zusätzlich wird für RMS die offline berechnete Priorität hinzugefügt, nach welcher ausgewählt wird, welcher Task den Core gewinnt.


Abbildung 6.3: KB Ausschnitt zur Phasenkonfiguration für RMS (o. links), EDF (o. rechts) und Cyclisch (u. links)

Bei cyclischen Tasks werden neben der WCET und Deadline eine Auflistung von Minor Cycles in Abhängigkeit vom Major Cycle angegeben in welcher der Task laufen darf. Ebenso wird mit Rücksicht auf die Periode des Tasks eine Wiederholungsinstanz angegeben, ebenfalls in Abhängigkeit vom Major Cycle (Cyclic_Times_Start).
Bei einem serverbasierten Schedulingalgorithmus werden die cyclischen Tasks aufgeführt wie ein einem rein cyclischen Schedule. Zusätzlich werden die Server aufgelistet, in denen sich auch die zugewiesenen Minor Cycle befinden in Abhängigkeit vom Major Cycle und die Namen aller Tasks sowie deren Priorität innerhalb dieses Servers.
Da es in einem Betriebssystem auch noch OS Systemtasks gibt, die u.a.

zur Verwaltung des Systems dienen, müssen diese auch mit angegeben werden. Die genaue Angabe dieser Tasks in der KB ist aber beschränkt. So haben Systemtasks keine Phase und keine genaue oder u.U. nur eine WCET. So wird diesem Task in der Auflistung keine Phase zugewiesen und die WCET wird gleich der Deadline gesetzt. Zusätzlich werden die OS Systemtasks noch im ersten Abschnitt der KB Ausgangsdatei (Systemeigenschaften) beschrieben und können so beim Einlesen der KB korreliert werden.

#### Fail-Operational Funktionen in XML

Im letzten Abschnitt der KB Ausgangsdatei finden sich die Parameter für den Fail-Operational Modus wieder. Für den Fail-Operational Modus ist nur eine gültige Systemkonfiguration vorhanden. Für diese Systemkonfiguration wird die Core Frequenz und der Schedulingalgorithmus angegeben, wobei diese für jeden Core gleich sind. Ebenso werden die Tasks mit ihrem jeweiligen Core aufgelistet. Die Beschreibung und die Parameter der Tasks unterscheiden sich nicht wesentlich von der allgemeinen Beschreibung der Tasksets, nur ist die Phasenerkennung mit der Phase beschrieben, die ausschließlich im Fail-Operational Modus zu finden ist.

**Konzept und Ziele** Mit dem Aufbau der Knowledgebase Datei und dem Abschnitt der Fail-Operational Verteilung sind folgende Ziele erfüllt:

- Parameter zur Erkennung von Überschreitungen in der Taskausführungszeit und Deadline
- Parameter für die Tasküberwachung anhand von Rückmeldungen in der KB
- Bereitstellung einer Backupstrategie in Form einer einzigen gültigen validen Verteilung

## 6.5 Zusammenfassung

In diesem Kapitel wurde die Berechnung und Erstellung der Knowledgebase vorgestellt. Als Erstes wurde dazu ein Algorithmus aufgestellt,

durch welchen alle möglichen Konfigurationen an Phasen, die in einem System vorkommen können, ermittelt werden. Dazu wurde eine Baumstruktur verwendet, deren unterste Blätter und das Kreuzprodukt aus diesen, alle Systemkonfigurationen bzgl. Phasen aufweisen. Dieser Algorithmus ist die Ausgangsbasis für alle weiteren Algorithmen und sorgt dafür, dass keine Phase bei den Berechnungen ausgelassen wird.
Die darauffolgenden Berechnungen sind abhängig vom Schedulingalgorithmus und wurden daher getrennt betrachtet. Dies bedeutet, dass prioritätenbasierte Berechnungen von cyclischen Berechnungen für sich betrachtet wurden.
Bei prioritätenbasierten Schedulingalgorithmen wie RMS und EDF und auch bei cyclischem Scheduling sind die Feasibility Tests bekannt und können mit einer Modifikation für phasenorientiertes Scheduling auch für die KB Berechnung angewandt werden. Hier wurde zuerst der Input aus der Baumstruktur genommen und die Tasks und deren Konfiguration für die jeweilige Phase ermittelt. In einem nächsten Schritt werden die Tasks der jeweiligen Phase auf dem Core allokiert. Die vorhandenen Bin Packing Algorithmen wie First Fit und Next Fit wurden durch einen Brute Force Algorithmus ersetzt. Bei der Allokation der Tasks werden die aktuelle Auslastung mit der mit dem ermittelten Maximalwerte der Feasibility Tests überprüft. Hier unterscheiden sich prioritätenbasierte und cyclische Schedulingalgorithmen. Müssen in einem prioritätenbasiertem Schedulingalgorithmus die Auslastungswerte überprüft werden, so muss im cyclischen Schedulingalgorithmus zusätzlich die Verteilung auf die einzelnen Cyclen geschehen. Einschränkung bei der Allokation stellen für beide Schedulingalgorithmen die Multi-Core Konfiguration sowie eine manuelle Allokation von Tasks dar. Neben periodischen Tasks ist es auch möglich aperidoische und sporadische Tasks in Scheduling Servern unterzubringen. Dies erfordert in beiden Schedulingalgorithmus eine extra Prüfung. Zuletzt wird eine FO-Konfiguration erstellt anhand der dafür ermittelten Tasks.
Ist für jede Systemkonfiguration eine gültige Taskverteilung gefunden, wird die Rekonfiguration zwischen den verschiedenen Phasen geprüft. Dafür wurden, abhängig vom Schedulingalgorithmus, unterschiedliche Formeln und Algorithmen entworfen, die einem Vorabtest der Rekonfiguration erlauben und so Schlussfolgerungen über die Machbarkeit bzw. Dauer der Rekonfiguration zulassen. Je nach mitgegebenen Parameter

werden Verteilungen verworfen, weil die Rekonfiguration zu lange dauert.
Wurden alle Algorithmen durchlaufen muss es für jede Systemkonfiguration eine gültige und von Feasibility Tests bestätigte Verteilung geben und keine Rekonfiguration darf länger dauern als der eingegebene Maximalwert. Da der Brute Force Algorithmus nicht nur eine einzige Konfiguration überprüft sondern alle, kann es vorkommen, dass mehrere valide Konfigurationen entstehen. Hier ist es möglich vorab z.B. nur die Konfiguration zu nehmen, die eine möglichst gleichmäßige Verteilung von Tasks aufweist, oder welche ein möglichst schnelle Rekonfiguration hat. Dies ist dem Systemdesigner überlassen. Im letzten Abschnitt wurde die Erstellung der Knowledgebase XML basierte Ausgangsdatei für den HAMS Scheduler und deren verschiedene Parameter erklärt. Es wurde aufgezeigt, welche Parameter in der KB Ausgangsdatei vorhanden sein müssen und welcher Inhalt. Zusätzlich wurde hier darauf eingegangen, wie der HAMS Scheduler die unterschiedlichen Systemkonfigurationen findet.

# 7 Knowledgebase Evaluation

Ziel der Evaluation ist es, die Erfüllung der Ziele und Anforderungen, welche an die HAMS Knowledgebase gestellt werden, aufzuzeigen. Dazu ist das folgende Kapitel in zwei Abschnitte unterteilt. Im ersten Abschnitt werden die Ziele und Anforderungen an die KB aus den vorherigen Kapiteln aufgelistet und in Kategorien zusammengefasst. Danach wird die allgemeine Vorgehensweise erklärt, wie die Ziele und Anforderungen an die KB überprüft werden können. Im zweiten Abschnitt wird die eigentliche Evaluation durchgeführt. Hier wird der Output der KB Berechnungen und die KB Ausgangsdatei mit dem aktuellen Algorithmus für Auslastungs- und Verteilungsberechnung für Multi-Core Systeme verglichen und bezüglich der Ziele und Anforderungen bewertet.

## 7.1 Konzept der Evaluation

### Vorgehensweisen bei Ziele und Anforderungen

#### Allokierung und Scheduling

Zur Evaluation des Schedulingverhaltens des HAMS Schedulers mit der KB, muss sich diese in den Zielen und Anforderungen beweisen. Da der HAMS Scheduler und die KB für die Zieldomänen Automotive und Avionik ausgelegt sind, stammen die Evaluationsbeispiele aus diesem Bereich. Diese Evaluationsbeispiele sind Tasks aus den Zieldomänen. Diese Tasks werden mit realistischen/vorgeschriebenen Ausführungszeiten und weiteren Parametern versehen, um die KB Berechnung durchführen zu können. Das Ergebnis wird auf Funktionstüchtigkeit sowie der Vielzahl an allokierten Tasks überprüft und mit dem aktuellen Algorithmus zur Auslastungs- und Verteilungsberechnung verglichen. Zu beweisen ist die Einsatzfähigkeit in diesen Zieldomänen bzw. die Vorteile darin aufzuzeigen.

Im Bereich Scheduling unterteilt sich diese Validierung der Ziele und Anforderungen in zwei Abschnitte. Zum einen in die Anzahl der allokierten Tasks, die auf ein System gebracht werden können und ein lauffähiges System bilden und zum anderen, mit welchem Verteilungsverfahren (Bin Packing) diese Tasks in diesem Multi-Core System verteilt werden. Hinsichtlich der Allokation wird betrachtet, wie das phasenorientierte Scheduling umgesetzt wird, welche Tasks (periodisch, sporadisch, aperiodisch) untergebracht werden können, ob die erforderlichen Scheduling Algorithmen, prioritätenbasiert, cyclisch und dynamisch unterstützt werden, wie Tasks allokiert werden können und welche Feasibility Tests angewandt werden. Im Bereich Bin Packing wird untersucht, wie nahe das phasenorientierte Bin Packing Verfahren der HAMS Knowledgebase an alternative Bin Packing Verfahren herankommt.
Mit dieser Evaluation soll das Konzept bewiesen werden, dass die HAMS Knowledgebase mehr Tasks auf ein System laden kann. Gleichzeitig soll aber ein sicheres Maximum an allokierten Funktionen nicht überschritten werden.

#### Rekonfiguration

Zur Evaluation der Rekonfiguration des Systems aus den Angaben der HAMS Knowledgebase, wird der Migrationsvorgang während eines Phasenübergangs geprüft. Dazu werden die Tasks aus dem Kapitel „Allokierung / Scheduling“ wiederverwendet und ein Übergang zwischen zwei Phasen mit zusätzlicher Migration beleuchtet. Bei diesem Migrationsvorgang wird zuerst die Formel 6.2 angewandt, um zu testen, ob die Migration in dieser Form durchführbar ist. Die sich aus der Berechnung ergebenden Zeiten oder Zeiträume dienen als weitere Grundlage für die Untersuchung. In einem Evaluierungssystem (Kapitel 7.1) wird die Migration, wie sie in einem HAMS System üblich ist, mit den in der HAMS Knowledgebase beschriebenen Eingangs- und Ausgangskonfiguration entsprechend durchgeführt. Die Zeiten/Zeiträume die dazu im Evaluierungssystem benötigt werden, werden mit dem Ausgang der Formel verglichen und evaluiert. Mit diesem Abschnitt der Evaluation soll ebenso nachgewiesen werden, dass die Zeiten bei der Migration und somit die Formeln anwendbar sind,

in Abhängigkeit des zugrundeliegenden Scheduling Algorithmus. Ebenso wird die Ausgabe der KB Berechnung auf die Anzahl der Konfigurationen hin überprüft, d.h. dass alle für die Operation notwendigen Konfigurationen vorhanden sind.
Somit wird das Konzept aus Kapitel 4.2 bestätigt, das verlangt, schon zur Designzeit die Durchführbarkeit einer Migrationen zu überprüfen und somit das Ziel einer deterministischen Rekonfiguration zu erreichen.

### Hardware

Die HAMS Knowledgebase ist darauf ausgelegt, Multi-Core Hardware zu unterstützen. Hierzu wurden Parameter in die KB zur Berechnung mit einbezogen, welche vor allem bei der Migration von Tasks zum Tragen kommen. Ebenso wurden Parameter über die Core Anzahl, Core Frequenz, Core Peripherie und Einschränkungen der Core Migration in die KB zur Berechnung mit aufgenommen. Diese zwei Eigenschaften und Besonderheiten der Multi-Core Hardware treten nur dann auf, wenn es sich um ein dynamisches System handelt. Somit können keine gezielten Vergleiche mit bestehenden statischen Systemen aus den Zieldomänen zur Evaluation angeführt werden.
Um diese Parameter dennoch zu untersuchen, werden in einem Evaluationssystem Migrationen und Rekonfigurationen durchgeführt und dabei auf die zu beachtenden Parameter eingegangen. Somit soll bestätigt werden, dass diese Migrationsparameter zur Durchführbarkeit der Migration beitragen und die Core Frequenzparameter es ermöglichen, den Prozessor noch besser auszulasten.
Im Bezug auf die Hardware wird auch eine ausgangsoptimierte HAMS Knowledgebase evaluiert. Diese wird aufzeigen, dass Hardwareparameter einen Vorteil gegenüber den aktuellen Tests wie CTA und Bin Packing Algorithmen bringen.
In diesem Kapitel wird nicht untersucht, ob die vorgestellten Algorithmen aus Kapitel 6.2.1 die Tasks anhand der Parameter Core Peripherie und Einschränkungen korrekt allokieren. Dieser Vorgang ist trivial und wird mit „if“ Bedingungen im Algorithmus 6 abgefragt. Das in Kapitel 4.2 ausführlich aufgestellte Konzept der HAMS Knowledgebase, neue Multi-Core Hardware nicht nur zu

unterstützen, sondern auch die Besonderheiten und Eigenschaften mit einzubeziehen, wird mit diesem Abschnitt evaluiert.

**Fehlerfallbehandlung**

Zur Evaluation der Fehlerfallbehandlung der HAMS Knowledgebase können nicht, wie bei den anderen Zielen, quantitative Aussagen gemacht werden. Sollte ein Task sich fehlerhaft verhalten, so sind in den Zieldomänen bestimmte Regeln und Vorgehensweisen zur Auflösung dieser Situation vorgegeben. Dazu zählen das Ausschalten der ganzen oder einem Teil der Funktionalität oder das Aussenden einer Warnung mit gleichzeitigem Ignorieren des Fehlers. Ziel der Fehlerfallbehandlung auf Basis der KB ist es, die Fehler im System zu erkennen und die Auswirkungen einzugrenzen. Zur Evaluierung wird daher simuliert, was passiert, wenn sich ein Task im Sinne des Schedulings fehlerhaft verhalten sollte. Daraus werden Rückschlüsse gezogen, ob die Parameter in der KB zur Erkennung ausreichend sind, und ob der Fehler mit der Herstellung Fail-Operational Konfiguration eingegrenzt werden kann. Das Konzept der HAMS Knowledgebase Tasks zu überwachen, Fehler bei der Systemausführung zu erkennen und eine entsprechende Backupstrategie bereit zu halten wird so mit diesem Abschnitt evaluiert.

## Die Evaluationsumgebung

Um einen qualitativen und quantitativen Vergleich des HAMS Systems mit der KB und den bekannten aktuellen Systemen sowie Forschungssystemen ermitteln zu können, wird eine Umgebung benötigt, die dies erlaubt. Dabei wurden folgende Systeme und Algorithmen verwendet:

**Multi-Core Systeme**

Hauptbestandteil der Evaluation sind zwei Multi-Core Systeme. Das erste System ist das PandaBoard mit dem OMAP 4430 Prozessor der Firma Texas Instruments. Das PandaBoard ist ein Open Source Project und wird von zwei Herstellern, CircuitCo und SVTronics Inc. produziert . Der OMAP 4430 Prozessor des PandaBoards ist Automotive-zertifiziert und für den Einsatz im Bereich Infotainment konzipiert. Der Prozessor besitzt zwei Cores

mit jeweils 1GHz, baut auf einer ARM Architektur auf und ist für ein SMP Betriebssystem vorbereitet. D.h. Tasks können zwischen diesen Cores migrieren mit Hilfe des Zugriffs auf gemeinsame Caches und RAM. Das installierte OS ist das Linux Betriebssystem, basierend auf der Distribution Ubuntu Server Edition 12.04 und dem selbst erstellten Kernel in der Version 3.2.0-48. Der Kernel wurde um den Real-Time Kernel Patch (Preempt RT) in der Version 3.2-rt erweitert. Neben den selbst geschriebenen Testtasks und den um HAMS erweiterten Kernel befinden sich noch Ftrace und Kernel Shark auf dem System, welche unverändert aus den Repositories übernommen wurden bzw. im Kernel aktiviert wurden.

Das zweite Testsystem ist ein Intel Core i5 Dual Core Prozessor mit der Kennung i5-2520M. Dieser Intel® Core Prozessor mit deaktiviertem Hyperthread- ing™ ist in einem Desktop System verbaut. Er erzeugt 2.5GHz bei deaktiviertem Speed-Step™ und besitzt einen gemeinsamen 3Mb Cache über alle zwei Cores, welche mit der Intel® Smart-Cache Technologie ausgestattet sind. D.h. jeder Core bekommt nach Bedarf eine variable Größe des Caches zugeordnet. Auch auf diesem Prozessor läuft das Ubuntu OS in der Version 12.04 in der Desktop-Variante. Der Kernel ist gleich dem des PandaBoards.

#### Evaluationstasks

Um das HAMS System mit der KB zu analysieren werden Tasks benötigt, welche die Funktionen der Zieldomänen umsetzen. Dazu werden bekannte und öffentlich verfügbare Algorithmen und Berechnungen verwendet, die das grundsätzliche Verhalten der Tasks bilden. So kann eine Geschwindigkeitsregelanlage mit Matlab / Simulink®grundsätzlich nachgebildet und dessen Laufzeitverhalten und Ansprüche analysiert werden. Ebenso können aus Veröffentlichungen Parameter über die Tasks herausgefunden und zur KB Evaluation herangezogen werden. Die erforderlichen Angaben sind in [127] [126] [1] und in Tabelle 7.1 aufgeführt.

| Name | Phase On (ms) | Phase Stand-by (ms) | Deadline (ms) | Period (ms) |
|---|---|---|---|---|
| GRA | 25 | 2 | 35 | 35 |
| Einpark-assistent | 20 | 2 | 35 | 35 |
| Abbiege-assistent | 25 | 2 | 35 | 35 |
| Spurhalte-assistent | 25 | 5 | 35 | 35 |
| Fernlicht-assistent | 25 | 5 | 35 | 35 |
| Berganfahr-assistent | 25 | 5 | 35 | 35 |
| Kollisions-vermeidungs-assistent | 25 | 5 | 35 | 35 |
| Regensensor | 25 | 2 | 35 | 35 |

Tabelle 7.1: Task Laufzeiten bei 1 Ghz Taktrate

**Evaluationssystem Hardware mit Algorithmen**

Um das Scheduling der HAMS KB mit aktuellen Systemen vergleichen zu können, kann aus einer Vielzahl von Systemen ausgewählt werden. Lizenz-, Hardwa- re- und analysetechnische Eigenschaften und Bestimmungen schränken diese Auswahl ein. Aus diesen Gründen wird der Linux Scheduler des vanilla Kernels bzw. mit Real-Time Patch zum Vergleich herangezogen. Dieser besitzt die Fähigkeit, mit RMS zu schedulen. Andere Scheduling Algorithmen müssen mit Hilfe eines Scheduling Simulators aufgezeigt werden. Zur Evaluation der Hardware werden zwei Evaluationstasks verwendet. Ein Task belastet überwiegend die Floatingpoint Unit und Arithmetic Logic Unit (ALU), der zweite Task besteht größtenteils aus Speicherzugriffen. Für die Arithmetischen Operationen wird ein Task implementiert, welcher die Kreiszahl Pi berechnet nach

Leibnitz [58]. Für die Speicherzugriffe wird ein Task implementiert, welcher Kanten detektiert mit Hilfe eines Gaussfilters und des Canny Algorithmus [2] [57].

#### Scheduling Simulator

Um die Ergebnisse der KB Berechnung graphisch darstellen zu können, wird ein Scheduling Simulator benötigt. Der gewählte Simulator, der Simulation of Multiprocessor Scheduling with Overheads (SimSO) [12], wurde entwickelt vom französischen Institut Laboratorie d'Analyse et d'Architecture des Systems (LAAS - CNRS) und wird Open-Source zur Verfügung gestellt. SimSo ist ein Scheduling Simulator für Echtzeit und Multi-Core Architekturen, der auch Rechenzeiten für eine Schedulingberechnung in seine Simulation mit aufnimmt. Das Ergebnis wird in einer graphischen Umgebung mit SimSoGui in Form eines Gantt-Charts dargestellt. Benützt werden die Versionen 0.7 von SimSo und 0.8.1 von SimSoGui [12].

## 7.2 Evaluation

### Allokierung und Scheduling

#### Anzahl der Softwarefunktionen

Zum Vergleich der auf dem System allokierbaren Softwarefunktionen einer HAMS Knowledgebase im HAMS System mit heutigen State-of-the-Art Systemen, wird zuerst ein Subset aus Funktionen zur Allokierung benötigt. Für diesen Vergleich werden aus der Tabelle 7.1 die Tasks Geschwindigkeitsregelanlage (GRA), Einparkassistent und Regensensor ausgewählt und auf einem Dual-Core System allokiert. Aus den verwendeten Linux Systemen (Kapitel 7.1) müssen zu den Tasks noch die Kerneltasks „Kworker:0“, „Kworker:1“ [77] sowie „Lin-Init“ (der Init Prozess von Linux) ergänzt werden. Diese Angaben sind ausreichend, um einen Linux Kernel ohne HAMS Scheduler zu betreiben. Zur Erstellung der KB werden noch weitere Parameter benötigt. Das Multi-Core Model muss angegeben werden basierend, auf Formel 5.12. In diesem Fall ist der Tupel wie folgt aufgebaut:

$$
\begin{aligned}
MC &= \{C_0, C_1\} \\
C_0 &= \{\{serial0\}, \{FPU\}, \{serial0\}, \{1000\}, \{1\}\} \\
C_1 &= \{\{serial0\}, \{PRU\}, \{serial0\}, \{1000\}, \{0\}\}
\end{aligned} \tag{7.1}
$$

Die Frequenz ist hier auf 1GHz festgelegt und ändert sich nicht. Eine Migration zwischen den beiden Cores ist erlaubt. Die Zeiten und die Phasen sind aus Tabelle 7.1 ersichtlich und werden für diese Evaluation auf folgende Werte heruntergebrochen und gemäß der Formel 5.1 in Tabelle 7.2 aufgelistet.

| Name | Phasename (ms) | Tupel |
|---|---|---|
| GRA | Fahrgeschwindigkeit | $FP_m$={(25), 35, 35, 1 ,arm, 10%} |
| GRA | Einparkgeschwindigkeit | $FP_m$={(2), 35, 35, 1 ,arm, 2%} |
| Einparkassistent | Fahrgeschwindigkeit | $FP_m$={(2), 35, 35, 1 ,arm, 2%} |
| Einparkassistent | Einparkgeschwindigkeit | $FP_m$={(25), 35, 35, 1 ,arm, 25%} |
| Regensensor | Scheibe Trocken | $FP_m$={(2), 35, 35, 1 ,arm, 2%} |
| Regensensor | Scheibe Nass | $FP_m$={(25), 35, 35, 1 ,arm, 25%} |
| Kworker:0 | - | $FP_m$={(30), 60000, 60000, 1 ,arm} |
| Kworker:1 | - | $FP_m$={(30), 60000, 60000, 1 ,arm} |
| Lin-Init | - | $FP_m$={(100), 60000, 60000, 1 ,arm} |
| HAMS-SLS | - | $FP_m$={(1), 35, 35, 1 ,arm} |

Tabelle 7.2: Tasks und logische Verknüpfungen zur Evaluation

Aus der Tabelle 7.2 geht hervor, dass die Tasks Geschwindigkeitsregelanlage (GRA) und Einparkassistent voneinander logisch abhängig sind und

der Task Regensensor sowie die Kernel Tasks von allen anderen Tasks logisch unabhängig sind. Dies kann anhand der Formel 5.4 beschrieben werden:
$ULT_1 = (GRA_{fahrg}(Einpark_{fahrg}) \vee (GRA_{einparkg}(Einpark_{einparkg}))$
Alle Parameter für die Erstellung der KB sind nun vorhanden und eine Berechnung anhand des RMS Algorithmus kann ausgeführt werden. Die Knowledgebase als Resultat dieser Berechnung ist im Anhang 9.1 angefügt. Ersichtlich ist, dass es für jede Phase eine eigene Konfiguration gibt und sich diese unterscheiden. So migriert von Konfiguration 1 zu Konfiguration 2 der Einparkassistent auf dem Core 1 und der Regensensor auf dem Core 0. Tasks, die an einen Core gebunden sind, wie Kworker:0 und Kworker:1 verbleiben auf ihren fest angebundenen Cores. Zwischen den Konfigurationen 3 und 4 findet keine Migration statt. Hier ändern sich lediglich die Ausführungszeiten der Tasks. Auch diese Vorgehensweise ist legitim in dynamischen Systemen.
Die Gegenüberstellung der verschiedenen Phasen mit dem statisch geschedulten Linux System mit RMS ist in Bild 7.1 dargestellt.

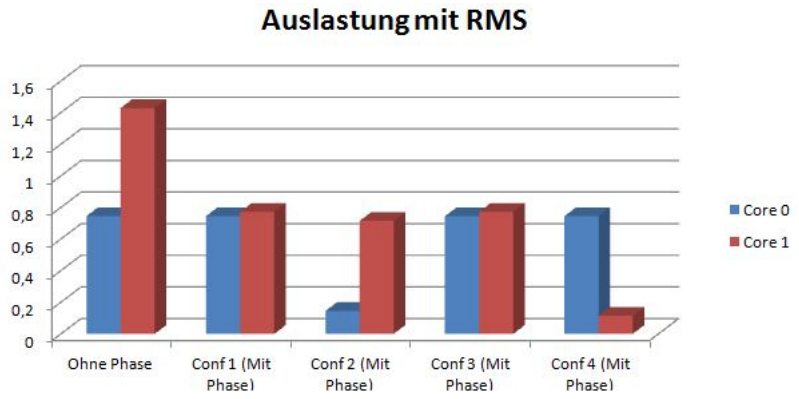



Abbildung 7.1: Vergleich prioritätenbasierter RMS Algorithmus

In Bild 7.1 ist die Auslastungen pro Core in den verschiedenen Phasen für die HAMS KB sowie ohne Phasen zu erkennen. Die verschiedenen Konfigurationen und deren Taskverteilung ist in der HAMS KB Anhang 9.1 zu finden. Um die Auslastungsgrenzen beurteilen zu können, müssen diese ihren maximalen Grenzen nach Formel 2.7 gegenübergestellt werden. Dies ist in Tabelle 7.2 zu sehen. Ein System ohne HAMS

| Konfiguration | Auslastung Core 0 | Auslastungsgrenze Core 0 | Auslastung Core 1 | Auslastungsgrenze Core 1 |
|---|---|---|---|---|
| Ohne Phasen | 0,745 | 0,756 | 1,426 | 0,779 |
| Conf1 | 0,745 | 0,756 | 0,771 | 0,779 |
| Conf2 | 0,145 | 0,743 | 0,714 | 0,828 |
| Conf3 | 0,745 | 0,756 | 0,771 | 0,779 |
| Conf4 | 0,745 | 0,756 | 0,114 | 0,779 |

Tabelle 7.3: Auslastungsvergleich in RMS ohne Phasen und mit Phasen

Scheduler und ohne KB (in Abbildung 7.1 ganz links) kann Phasen und deren Vorteile nicht beim Scheduling mit einbeziehen. Von daher müssen immer die Maximalzeiten der WCETs aller Tasks herangezogen werden. Da die Auslastung nun deutlich die Grenzwerte von Liu und Layland überschreitet, ist das System so nicht ausführbar. Es müsste ein Multi-Core System mit drei Cores benützt werden, um diese Tasks auf einem System zu vereinen.
Wie aus den vorherigen Ergebnissen zu erkennen ist, ist die logische Abhängigkeit und die Möglichkeit der HAMS Knowledgebase diese zu verarbeiten, der Schlüssel zu einer geringeren Gesamtsystemauslastung. Um dies aber zu ermöglichen, müssen logische Abhängigkeiten der Tasks gefunden und in logische Tasksets gegliedert werden. Um aufzuzeigen, wie vielfältig diese Phasen definiert werden können, werden zwei weitere Gegebenheiten erörtert. In der Tabelle 7.2 sind sieben Tasks aufgelistet, die aktuell in Autos implementiert werden und die sich gegenseitig anhand der Fahrzeuggeschwindigkeit logisch ausschließen. Wird eine KB mit diesen Tasksets erstellt, wird diese, wie in Anhang 9.5 aufgelistet, dem HAMS Scheduler mitgegeben. Gegenüber einem üblichen

| Task Name | Aktive Phasen $\frac{km}{h}$ | Standby Phasen $\frac{km}{h}$ |
|---|---|---|
| Geschwindigkeitsregelanlage | 10-60; 60-250 | 0-10 |
| Einparkassistent | 0-10 | 10-60; 60-250 |
| Berganfahrassistent | 0-10 | 10-60; 60-250 |
| Abbiegeassistent | 0-10 | 10-60; 60-250 |
| Kollisionsassistent | 10-60; 60-250 | 0-10 |
| Spurhalteassistent | 60-250 | 0-10; 10-60 |
| Fernlichtassistent | 60-250 | 0-10; 10-60 |

Tabelle 7.4: Erweiterte logische Taskabhängigkeiten

Scheduler mit EDF sind die Auslastungsgrenzen und Vorteile in Bild 7.2 zu erkennen. Ohne ein HAMS System mit Knowledgebase müsste dieses Taskset auf mehrere ECUs aufgeteilt werden. Eine weitere Gege-

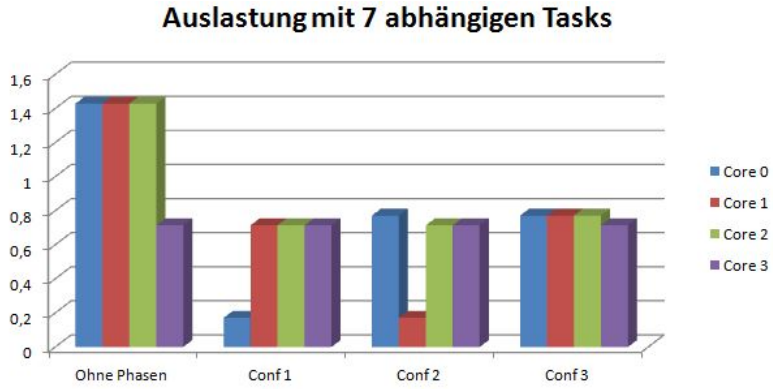



Abbildung 7.2: 7 Tasks mit und ohne Phasen

benheit verdeutlicht, wie die Phase 0-10 $\frac{km}{h}$ noch weiter untergliedert / koordiniert werden kann.
Hier kann das Taskset, bestehend aus den Tasks Einpark-, Berganfahrssistenten, Abbiegeassistent und dem neuen Task LaunchControl in

sich logisch ausschließende Phasen untergliedert werden. In der Phase „Parken“ ist der Einpark- und Berganfahrassistent aktiv, in der Phase „Langsamfahrt“ der Abbiegeassistent sowie die LaunchControl und in der Phase „Schnellstart“ ausschließlich die LaunchControl. So lässt sich anhand des Bildes 7.3 erkennen, dass in diesem Dual-Core System immer vier Tasks gleichzeitig auf einem System laufen können, ohne die Auslastungsgrenzen von RMS zu überschreiten. Ein aktuelles statisch geschedultes Dual-Core System ist nur in der Lage, zwei dieser Tasks sicher zu schedulen.

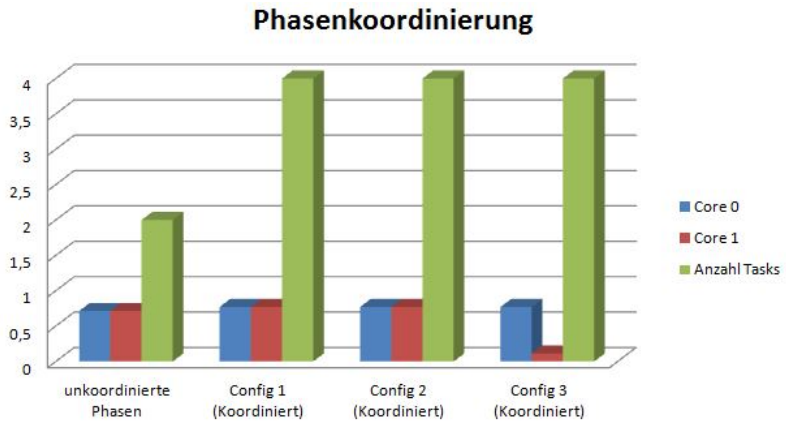



Abbildung 7.3: Koordinierung von Phasen im Bereich 0-10 $\frac{km}{h}$

Durch das Ausnutzen von Phasen und unterschiedlicher Scheduling Algorithmen lassen sich in den Zieldomänen mehr Softwarefunktionen auf einer ECU allokieren, als es mit aktuellen statischen Systemen möglich wäre. Auch kooperative Scheduling Algorithmen können dabei unterstützt werden, wenn es das Zielsystem benötigt. Dabei ist die KB Berechnung nicht nur auf Multi-Core Systeme beschränkt, sondern kann auch aufgrund des „partitioned“ Scheduling Ansatzes auf Single-Core Systeme angewandt werden.
Zur Vervollständigung der Evaluation der HAMS Knowledgebase im Scheduling muss ein cyclischer Schedule auf die Tasks aus Tabelle 7.2

angewandt werden. Wie in Anhang 9.4 ersichtlich, werden nun die Tasks einzelnen Zeitslots zugewiesen (u.A. Zeilen 27 und 28 in Anhang 9.4). Wird z.B. die Konfiguration 1 genauer betrachtet, so ist zu erkennen, dass auf Core 0 ein Major Cycle von 35ms aktiv ist mit einem Minor Cycle von 1 ms. Der SLS läuft im ersten Minor Cycle, der zweite Minor Cycle ist vom Task Geschwindigkeitsregelanlage belegt. Der, ausgehend von der Phase, aktive Task Einparkassistent belegt insgesamt 25 Minor Cycle Slots, um seine Berechnungen durchführen zu können. Die noch verbleibenden 7 Slots des Major Cycles sind für Kooperatives Scheduling vorgesehen. Hier kann ein Polling Server wie aus Kapitel 2.1.2 ausgewählt werden, um als Platzhalter für sporadische oder aperiodische Tasks zu fungieren. Da es in diesem System solche Task nicht gibt, wird der IDLE Task geschedult.
Ebenso werden von der HAMS Knowledgebase Polling Server für sporadische und aperiodische Tasks unterstützt. Anhand der Parameter aus Tabelle 7.2 werden die Kernel Tasks LinInit, Kworker:0 und Kworker:1 als aperiodische Tasks definiert. Diese werden wie in Anhang 9.2 ersichtlich in Polling Server unter RMS zusammengefasst. Dieser Polling Server hat eine WCET Laufzeit, die dieser während seiner Periode aufbrauchen kann. Sollten die Tasks mehr Laufzeit benötigen müssen, sie auf die nächste Periode warten, bis der FLS sie wieder schedult. Eine Priorität gibt es per Definition in diesem Server nicht, es gilt das FIFO Prinzip (Kapitel 2.1.2).

### Vorteile der KB Allokierung

Wie in dem Konzept der HAMS Knowledgebase bei der Task Allokierung definiert (Kapitel 4.2), wird zur KB Erstellung ein Brute Force Algorithmus verwendet, der immer die optimale Verteilung berechnet. Aktuelle, in Kapitel 2.1.3 vorgestellte Verteilungsalgorithmen (Bin Packing Algorithmen) liefern nicht immer die optimalen Ergebnisse, sondern haben eine gewisse Güte (Kapitel 2.1.3), mit der sie an das optimale Ergebnis herankommen.
Um die Unterschiede der beiden Vorgehensweisen weiter zu verdeutlichen, wird die Gegebenheit aus Tabelle 7.2 in der Phase 60-250$\frac{km}{h}$ weiter fortgeführt. Das Ergebnis des Brute Force Algorithmus und somit die Verteilung der Tasks für jede Phase, ist in Anhang 7.2 ersichtlich.

Das Ergebnis ist valide, d.h. alle Tasks können ihre erforderliche Rechenzeit einfordern und keine Deadlines werden verletzt.
Wird nun anstatt des Brute Force Bin Packing Algorithmus ein anderer Algorithmus angewendet, so verändert sich die Taskverteilung. Das Ergebnis ist zum Überblick in Tabelle 7.2 aufgelistet sowie in Bild 7.4:

| Task | Auslastung | Zugewiesener Core |
|---|---|---|
| GRA | 0,714 | Core 0 |
| Kollisionsassistent | 0,714 | Core 1 |
| Spurhalteassistent | 0,714 | Core 2 |
| Fernlichtassistent | 0,714 | Core 3 |
| Einparkassistent | 0,057 | Core 3 |
| Berganfahrassistent | 0,057 | Core 3 |
| Abbiegeassistent | 0,057 | Core 3 |

Tabelle 7.5: Taskverteilung First Fit Deacreasing

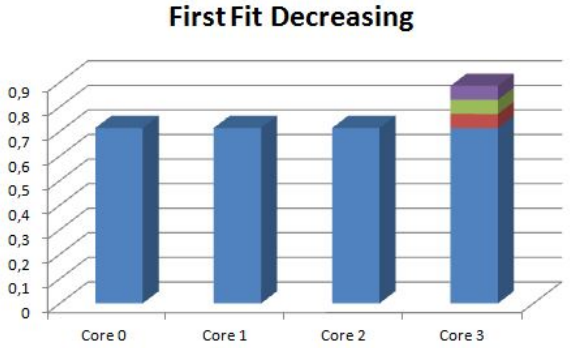



Abbildung 7.4: Anwendung des First Fit Deacreasing Algorithmus

Der Brute Force Algorithmus der KB Erstellung bietet somit Vorteile bei der Allokierung der Tasks auf die verschiedenen Cores gegenüber den klassischen Bin Packing Algorithmen und kann durch das Testen einer jeden Taskverteilung die beste/mögliche Systemkonfiguration für

die weiteren Überprüfungen finden. So ist in Abbildung 7.6 die Baumstruktur für die Tabelle 7.2 dargestellt, die sich nach Algorithmus 1 für prioritätenbasiertes Scheduling ergibt. Kernel eigene Tasks, LinInit, Kworker:0 und Kworker:1, wurden hierfür aus Gründen der der Übersichtlichkeit nicht beachtet. Mit Hilfe der Logischen Operatoren können alle Phasenkonfigurationen ermittelt werden. Der Brute Force Algorithmus mit der Rotation findet als nächstes die optimale Verteilung anhand Algorithmus 4. So ist sichergestellt, dass ,wie im Konzept gefordert, alle Konfigurationen getestet werden.

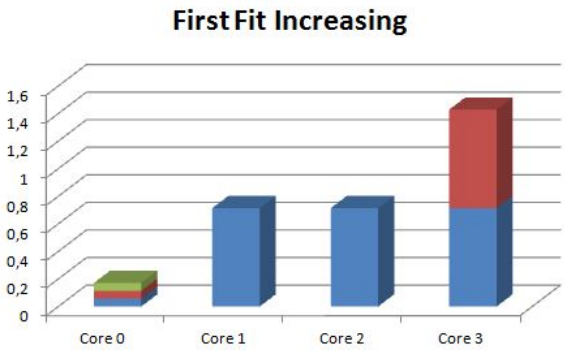



Abbildung 7.5: Anwendung des First Fit Increasing Algorithmus

Zur weiteren Evaluation der Task Allokation wird untersucht, welche Auswirkungen Einschränkungen im System auf den Brute Force Algorithmus haben. Dadurch wird der Brute Force Algorithmus in seiner Vielfalt gestört, z.B. wenn Tasks nur auf bestimmten Cores laufen dürfen oder aufgrund von Peripherieeinschränkungen ausschließlich dort laufen können, wird auch ein Brute Force Algorithmus u.U. keine schedulbare Taskverteilung finden können. Um dies zu verdeutlichen werden die Tasks aus Tabelle 7.1 und 7.2 um fiktive Hardwareeinschränkungen erweitert[1]. Die Tabelle 7.1 wird mit den Parametern NP, NCP und NI aus dem Tupel 5.7 für die Tasks aus Tabelle 7.2 erweitert. Diese

[1]Diese Einschränkungen dienen hier als Evaluationsunterstützung und entsprechen nicht zwangsläufig der Realität

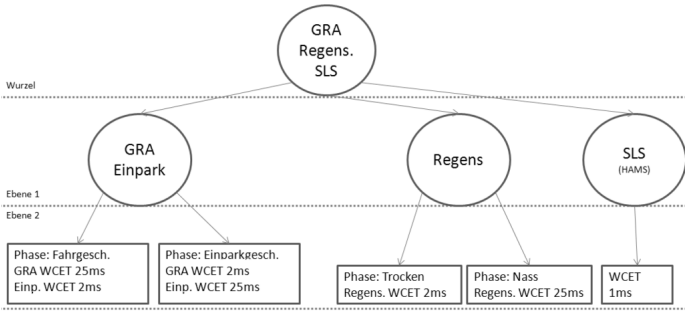



Abbildung 7.6: Baumstruktur der HAMS KB Erstellung

Peripherieeigenschaften müssen sich ebenso im Hardware Modell des Dual-Core Prozessors, basierend auf Formel 5.12 widerspiegeln und sind in Formel 7.2 aufgeführt.

$$C_0 = \{\{CAN_{Motor}\}, \{FPU\}, \{Tast_{Temp}, Tast_{Einp}, Tast_{Regen}\}, \{1Ghz\}, \{1\}$$

$$C_1 = \{\{CAN_{Comfort}, ETH\}, \{FPU\}, \{Tast_{Temp}, Tast_{Einp}, Tast_{Regen}\}, \{1Ghz\}, \{1\} \quad (7.2)$$

$$MC = \{(C_0, C_1), AM\}$$

Wie aus Formel 7.2 zu erkennen ist, sind nicht alle Kommunikationsschnittstellen über jeden Core erreichbar. Dies stellt somit eine Einschränkung bei der Allokierung der Funktionen auf die Cores dar. Mit dem Blick auf Anhang 9.1 und Konfiguration mit der ID 1 ist die Konfiguration nicht einzunehmen. Da die Geschwindigkeitsregelanlage nur auf Core 0 gesetzt werden kann aufgrund der CAN Schnittstelle und der Regensensor mit der Einparkassistent auf Core 1, kann diese Konfiguration nicht eingenommen werden. Sollten, wie gefordert, Einparkassistent und Regensensor auf Core 1 laufen, ist die Auslastung über der Auslastungsgrenze von 100% und das System somit nicht schedulbar.

| Name | Phase On (ms) | Phase Stand-by (ms) | NP | NCP | NI |
|---|---|---|---|---|---|
| GRA | 25 | 2 | CAN_Motor | FPU | Taster_Temp |
| Einpark-assistent | 20 | 2 | CAN_Comfort, ETH | FPU | Taster_Einp |
| Regensensor | 25 | 2 | CAN_Comfort | none | Taster_Reg |

Tabelle 7.6: Tasks erweitert um Hardware Eigenschaften

## Rekonfiguration

Um den Übergang von dynamischen zu statischen Systemen mit der HAMS Knowledgebase und dem HAMS System zu ermöglichen, muss eine deterministische Rekonfiguration vorliegen. Nur so ist gewährleistet, dass während des Rekonfigurationsbetriebes, die Tasks ihre Arbeit fortsetzen können. Da der HAMS Scheduler ausschließlich bei einer Phasenänderung oder dem Übergang in den Fail-Operational Modus das System rekonfiguriert, ist dieser Abschnitt bei der KB Erstellung für alle möglichen Phasenübergänge zu evaluieren. Für diese Evaluation bei der Erstellung der KB wird die Formel 6.2 je nach Schedulingvorgehensweise herangezogen. Die Formeln teilen sich in zwei Abschnitte auf nämlich $\Delta(SLS_t - LT_t)$ und $\lfloor(FLS_{bin} - SLS_t)\rfloor$. Für diese Abschnitte muss nachgewiesen werden, dass diese Annahmen korrekt sind und in welchem Zeitraum sie sich befinden und wie dieser beeinflusst werden kann.
Um den ersten Abschnitt $\Delta(SLS_t - LT_t)$ der Formel zu verdeutlichen, wird das Bild 7.7 benötigt. In diesem Bild befindet sich ein Gantt Diagramm, aufgenommen mit dem trace-cmd Toolkit und visualisiert mit Kernelshark auf dem i5 System. Die ersten drei Balken des Gantt Charts stellen die Cores 0, 1 und 3 dar, auf denen unterschiedliche

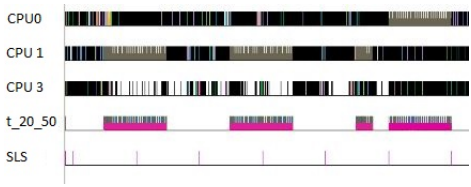



Abbildung 7.7: SLS mit doppelter Frequenz als Task

Tasks aktiv sind. Zu Aufzeichnungszwecken wurde der Hyperthreading Core 3 des i5 Prozessors aktiviert für trace-cmd. Unter anderem laufen die Tasks „`t_20_50`“ und „SLS“, welche in den unteren zwei Gantt Charts dargestellt sind. Der SLS wird mit einer Frequenz von 40Hz a 25ms Periodendauer aufgerufen und der Task „`t_20_50`“ mit einer Frequenz von 20Hz a 50ms Periodendauer. Auch der FLS ist auf dieser Darstellung aktiv, geht aber in den schwarzen Balken der Kernel Tasks unter und ist auch nicht herauszufiltern. Wie aus dem Bild hervorgeht, ist keine zeitliche Abstimmung zwischen SLS und `t_20_50` zu erkennen. So aktiviert sich der SLS zeitlich unabhängig von den anderen Tasks, d.h. weder das Ende noch der Anfang eines Tasks ist mit dem SLS Aufruf gekoppelt, noch ist es erlaubt, dass der SLS während eines Tasks laufen darf. Wird nun, wie am Ende des Bildes zu erkennen ist, ein Phasenänderungswunsch an den SLS gesendet, so ist der erste Abschnitt der Formel in diesem Vorgang abgebildet. Der Task `t_20_50` muss nun auf den nächsten SLS Aufruf warten, welcher in diesem System in den nächsten $\sim 5ms$ erfolgt. Somit ist das Delta in diesem System 5ms. Aus dem Bild lässt sich auch erkennen, was passiert, wenn der SLS Aufruf später erfolgen würde. Der Task müsste somit bis zu einer Maximalzeit von 25ms warten und zwar genau dann in dem Fall, wenn der Task seinen Phasenänderungswunsch sendet, aber der SLS gerade beendet ist. Somit lässt sich auch eine Abhängigkeit der Formel im ersten Abschnitt von der Frequenz des SLS ableiten. Die Migration dauert somit länger je niederfrequenter der SLS ist. So lässt sich die Zeit der Phasenrekonfiguration verkürzen, indem der SLS eine höhere Priorität hat.
Die Verdeutlichung des zweiten Teils der Formel erfordert eine Testreihe. Sobald der SLS den Phasenänderungswunsch evaluiert und seine

Reaktionen berechnet bzw. die Nachrichten an die erforderlichen FLSen verteilt hat, muss auf den nächsten Aufruf des FLS gewartet werden. Der FLS wird durch den Systemtick aufgerufen und nach diesem abgearbeitet. In einem asynchronen System sind die Ticks für die Cores unterschiedlich und können verschiedene Perioden haben. Aus diesem Grund bestimmt der FLS mit der niedrigsten Periode den zweiten Abschnitt der Formel $\lfloor(FLS_bin - SLS_t)\rfloor$. Die Testreihe ermittelt den maximalen Abstand SLS zu FLS. Da diese Testreihen mit einem Linux OS vorgenommen werden, wird erwartet, dass die Maximalzeit nicht länger als 1ms (Tick mit 1000Hz als Standardkonfiguration) ist.

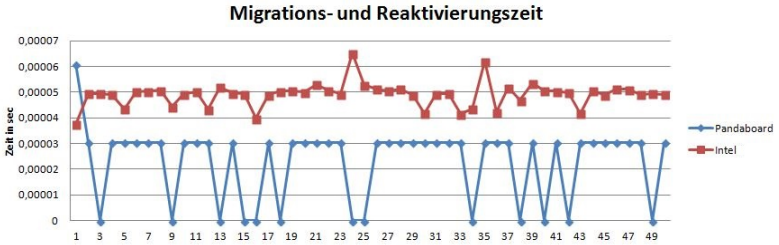



Abbildung 7.8: Messung der Migrations- und Reaktionszeit

Im Bild 7.8 sind die Schwankungen dieser Zeit dargestellt, einmal für das PandaBoard mit OMAP4430 und einmal für den i5 Prozessor. Die Tasks, die auf dem System laufen, sind wieder der `t_20_50` und der SLS Task aus dem vorherigen System. Gemessen wird nun die Zeit vom SLS Ende bis über die Migration von Core 1 auf Core 0 bis zum Wiederaufruf des Tasks (siehe Bild 7.7 Ende). Aus dem Chart 7.8 lässt sich eine Durchschnittszeit von $50\mu s$ erkennen. Da der i5 Prozessor mit einem Standard Linux betrieben wird, ist die Tickfrequenz von 1000Hz(1ms) das zu erwartende Maximum. Der Chart bleibt innerhalb dieser Grenzen. Beim PandaBoard werden die gleichen Tasks verwendet, jedoch ein echtzeitfähiges Linux OS. Man erkennt deutlich, dass die Migrations- und Reaktionszeit in einigen Fällen so schnell abläuft, dass diese nicht messbar ist. Ansonsten läuft diese mit durchschnittlich ca. $30\mu s$ ab. Dennoch bleibt auch diese Zeit im Rahmen des Ticks. Um die

Formel nun auf unser System anzuwenden, wird für das Delta ein Maximum von 25ms und für den Tick ein Maximum von 1ms (Summe 26ms) erwartet. Die Ausführungszeit des SLS wird hierbei vernachlässigt, da diese zu klein ist.
Wie aus Bild 7.9 zu erkennen ist, bleibt die maximale Zeit mit 7ms beim PandaBoard und beim i5 Board mit $500\mu s$ im Limit von 26ms. Somit ist die deterministische Rekonfiguration im HAMS Scheduler durchführbar und für die Formel zur KB Erstellung anwendbar.

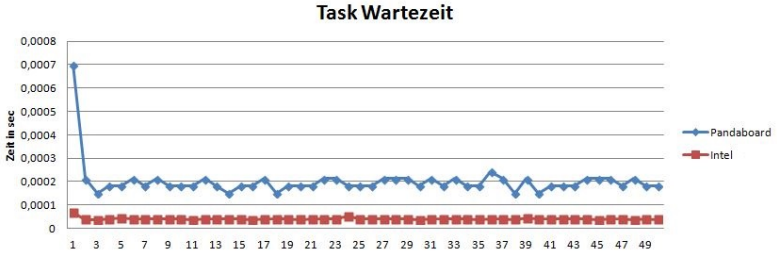



Abbildung 7.9: Messung der maximalen Rekonfigurationszeit

Ausreißer die auf den Bildern 7.8 und 7.9 zu erkennen sind, werden durch 1st Level Hardware Interrupts oder Systeminternen Tasks des Linux Systems erzeugt, die periodisch oder aperiodisch auftreten. Aus diesem Grund wurden die Messungen mehrmals durchgeführt, um diese Einflüsse zu filtern. Der Aufstartvorgang unter ftrace und die dementsprechenden Ausreißer wurden nicht betrachtet.

## Hardware

Für die Evaluation des Konzeptes der HAMS KB im Bereich der Hardware wurden die Eigenschaften eines Multi-Core Prozessors herausgearbeitet und deren Auswirkungen auf die HAMS KB Berechnungen untersucht. Die Konzepterstellung der Hardware Eigenschaften erfolgte in Kapitel 5.2 mit Hilfe von Hardwarebeschreibungen und vorliegenden Forschungsergebnissen und mündete in die Formel 5.11. Hier wurden die zwei Eigenschaften der Core Frequenzänderung und der WCET Verlän-

gerung eines Tasks nach der Migration beschrieben. Einschränkungen in der Hardware und die Auswirkungen auf die HAMS Knowledgebase wurden im vorherigen Kapitel 7.2 beschrieben und evaluiert.
Wie in Kapitel 7.1 vorgestellt, findet die Hardwareevaluation aufgrund der Trivialität nicht in Betrachtung auf die Parameter Core Anzahl, Core Peripherie und Einschränkungen statt.
Wie in Kapitel 7.1 aufgestellt, wird das ausgangsoptimierte Scheduling aus Kapitel 6.2.1 verglichen mit heutigen Lösungen. Ebenso werden Frequenzänderung und Rekonfiguration anhand der Parameter überprüft.

**Zielgerichtete Knowledgebase** Um ein ausgangsoptimiertes Scheduling in der KB Ausgangsdatei zu erzeugen werden, wie in Kapitel 6.2.1 postuliert, nur diese Konfigurationen gewählt, welche die KB in Auslastung, Rekonfiguration oder Power Performance optimiert. Anhand der Optimierung der Power Performance lässt sich diese Eigenschaft der KB am deutlichsten evaluieren. Dazu dienen die Evaluationstasks aus Tabelle 7.2 mit der aktiven Phase $0 - 10\frac{km}{h}$ und der Auslastung von 0,714 für aktive Tasks und 0,051 für nicht aktive Tasks (Tabelle 7.2) sowie das EDF Scheduling als Schedulingalgorithmus. Zur Verfügung steht das für die Phase $10 - 60\frac{km}{h}$ benötigte hypothetische Quad-Core System. In der KB gibt es im Bereich Power Optimierung zwei Möglichkeiten die ausgewählt werden können. Wie in Abbildung 7.10 links zu sehen ist, können die Tasks gemäß ihrer Auslastung möglichst homogen auf die Cores verteilt werden (Load Balancing). Die Stromaufnahme wird durch diese Maßnahme verringert gemäß Kapitel 3.1. Ebenfalls kann ein Core wie in in Abbildung 7.10 rechts dargestellt (Load Unbalancing) konfiguriert werden. Auch durch diese Maßnahme wird die Stromaufnahme verringert. Welche Maßnahme am besten geeignet ist, ist Multi-Core Prozessor abhängig und muss in den entsprechenden Datenblättern ermittelt werden. Basierend auf der Berechnungsformel in [109], wurde die Power Performance eines jeden Cores in Abbildung 7.10 dargestellt. Zu erkennen ist, dass ein gleichmäßig ausgelasteter Core eine bessere Performance hat als als wenn ein Core komplett ausgeschaltet wird.
Werden diese Möglichkeiten der KB Berechnung mit den heutigen Aufnahmetests wie CTA und Devis (aus Kapitel 3.1) verglichen, so werden die Ergebnisse von einer Power Performance optimierten Lösung abweichen. Im CTA Algorithmus spielt die Ankunftszeit der Tasks eine Rolle.

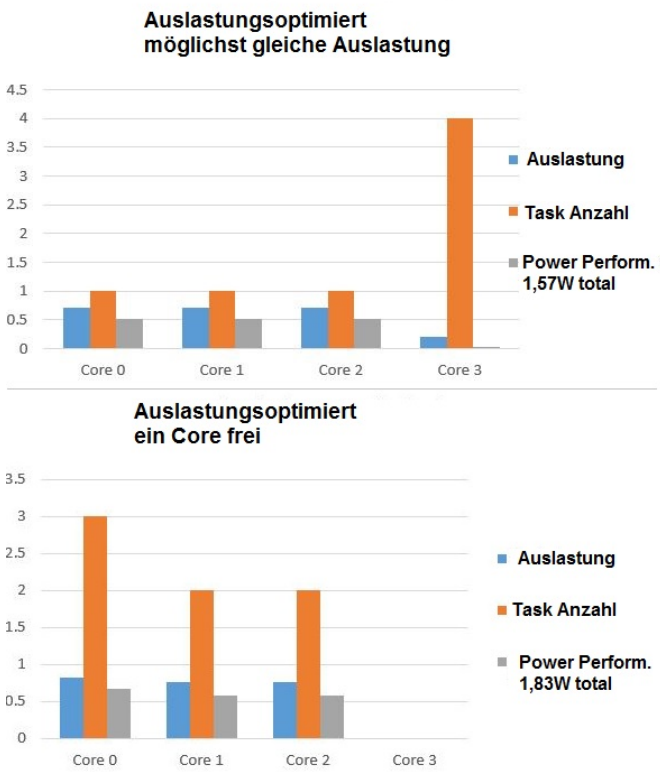



Abbildung 7.10: Möglichkeiten der Auslastungsoptimierung

Diese ist aber zufällig und somit kann keine Performance Optimierung stattfinden. Bei der Verteilung auf die Prozessoren unter EDF, kann es hier u.U. vorkommen, dass die Lösungen der KB gleichen. Dies ist aber auch abhängig von der Ankunftszeit der Tasks und somit nicht immer gegeben. Sollte diese Verteilung in CTA nicht dem Zufall überlassen, werden muss vorab die Ankunftszeit der Tasks unveränderbar festgelegt werden. Die gleiche Aussage ist auch für den Devis Test gültig. Wird aber hier ein First Fit Increasing oder Decreasing Algorithmus verwen-

det wird es nicht möglich sein, einen Core ohne eine Taskallokation zu bekommen.

**Core Frequenzänderung** Eine weitere Möglichkeit der Power Performance Optimierung in der HAMS KB und im HAMS System ist die Änderung der Core Frequenz zur Laufzeit. Wie aus Kapitel 5.2 bekannt, tritt die Änderung der Frequenz bei bestimmten Multi Core Systemen unverzüglich in Kraft und verringert die Energieaufnahme des Cores. Diese Eigenschaft kann benützt werden, um die linke Taskverteilung aus Abbildung 7.10 noch weiter zu optimieren. Wie in Core 3 zu erkennen ist, ist die Auslastung bei 0,204 und gibt so Potential frei für eine Auslastungserhöhung durch Frequenzsenkung. Mit der Annahme, dass bei einer Frequenzverringerung um das 4-fache sich die Rechenzeit der Tasks auch um das 3,8-fache erhöht, ergibt sich eine neue Auslastung von 0.7752. Wie in 7.11 zu erkennen ist, ist eine homogenere Verteilung der Auslastung durch die Senkung der Frequenz entstanden. So wird nicht nur durch das Senken der Frequenz, sondern auch durch ein besseres Load Balancing die Power Performance des Multi-Core Prozessors optimiert. Dies ist in der Power Performance Berechnung aus Abbildung 7.11 zu erkennen. Diese ist niedriger als die im Vergleich dazu stehenden Berechnungen aus Abbildung 7.10. Für Core 3 wurde hier die Stromaufnahme um den Faktor 4 verringert.

**WCET und Rekonfiguration** Um die Verlängerung der WCET durch eine Rekonfiguration zu evaluieren, wird zuerst betrachtet, welche Hardware Eigenschaften diesbezüglich bei der KB Erstellung mit einbezogen wurden. Dabei wurde der Parameter für die WCET Verlängerung $CMP$ in der Formel 5.11, unabhängig der Phase, mit einbezogen. Um zu evaluieren, wie dieser Parameter sich in einem realen HAMS System auswirkt, werden Messungen mit Kernelshark und ftrace anhand der zwei Tasks aus der Frequenzmessung vorgenommen. In der Versuchsreihe werden die Tasks von einem Core auf einen anderen migriert, während diese im rechnenden Zustand sind und den Core besetzen. Es wird nun untersucht, wie sich die Ausführungszeit bei der Migration verändert. Die Frequenz bleibt dabei auf allen Cores gleich. Die Ausführungszeit wird dabei in den Bereich von Millisekunden (Kreiszahlberechnung ca. 39ms und Kantendetektion ca. 128ms) eingestellt, um in die für die

Zieldomänen relevanten Zeitbereiche zu messen. Dies geschieht durch Einstellung der Iterationen. Zur Evaluation werden die Tasks auf dem Core i-5 System zwischen Core 0 und 1 (beide ohne Hyperthreading) migriert bei höchster Frequenz. Wie in Anhang in der Abbildung 9.4 zu erkennen ist, ist die Abweichung minimal. Der Kreiszahlberechnungstask verlängert seine Ausführungszeit kurzzeitig von 0,039889s zu 0,039918s, was einer Verlängerung von 29$\mu$ entspricht. Die Ausführungszeit des Kantenerkennungstasks ändert sich kurzzeitig von von 0,127095s zu 0,1287s, was einem Delta von 1,6ms entspricht. Beide Ausführungszeiten sind im nächsten Zyklus wieder gleich mit den normalen Ausführungszeiten. Somit ist die Miteinbeziehung dieser Hardware Eigenschaft wichtig für ein rekonfigurierbares System und wird mit der Formel 5.10 und dem Algorithmus 7 bei der HAMS KB Erstellung berücksichtigt. So wird im HAMS System verhindert, dass durch die kurzzeitige Änderung der Ausführungszeit ein Auslastungsfehler auftritt.

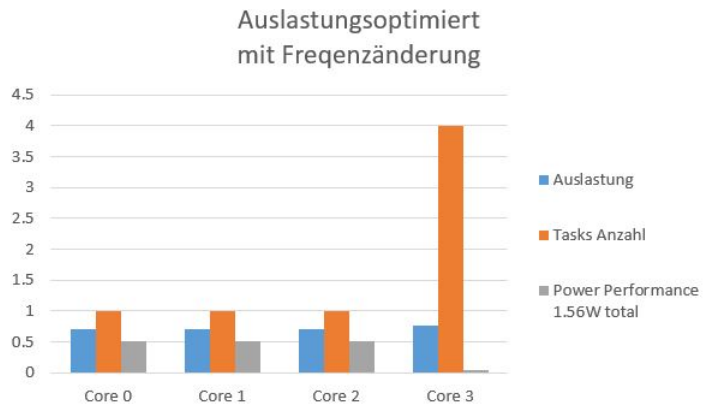



Abbildung 7.11: Frequenzänderung bei der Auslastungsoptimierung

## Fehlerfallbehandlung

Um die Überwachung des HAMS Systems mit der KB mit aktuellen Systemen zu vergleichen, müssen die Methoden zur Laufzeitüberwachung gegenübergestellt werden. Wie in Kapitel 7.1 beschrieben, kann nicht jedes aktuelle OS mit dem HAMS System in Bezug auf Tasküberwachung verglichen werden, da viele die gleichen Eigenschaften besitzen. So sind z.B. Linux, VxWorks und andere auf Unix und POSIX basierende Systeme in ihrem Umgang mit der Tasküberwachung gleich. Eine Ausnahme und zugleich das OS mit den aktuellsten Techniken der Tasküberwachung ist der AUTOSAR Standard mit der TIMEX Erweiterung.
Um aufzuzeigen wie in einer Situation mit Tasks- und Hardwarefehlern der HAMS Scheduler auf Basis der KB reagiert und wie das in anderen Systemen behandelt wird, müssen verschiedenste Fehlerfälle simuliert und gegenübergestellt werden. Mit den Rückschlüssen aus der Simulation wird das Evaluationskonzept aus Kapitel 7.1 auf das HAMS System vervollständigt.

**Deadline Fehlerfallbehandlung** Um in einem RMS oder EDF Schedulingalgorithmus einen Auslastungsfehler zu erkennen, müssen die Deadlines mit überwacht werden. Ein Auslastungsfehler kann dann simuliert werden, wenn ein höherpriorer Task den Core nicht mehr abgibt. In einem Linux System oder auf einem AUTOSAR Standard basierenden OS wird der höher priore Task weiter ausgeführt. Das Fehlverhalten des Tasks wird toleriert und das System so unbrauchbar. In einem Echtzeitschedulingsystem, wie dem HAMS System mit KB, führt die Überschreitung der Deadline zu einem Auslösen des Übergangs in den Fail-Operational Zustand. Das Wissen über die Task Deadlines kann der HAMS Scheduler aus der KB entnehmen. Hier sind, wie in Abbildung 6.3 gezeigt, für jeden Task die phasenabhängigen Deadlines vorhanden. Sollte wie in Anhang in der Abbildung 9.5 die Task Deadline eines Tasks überschritten werden, z.B. durch eine simulierte Überlast, kann der HAMS Scheduler somit einen Fehler in der Auslastung zur Laufzeit erkennen. In einem harten Echtzeitsystem terminiert der HAMS Scheduler den defekten Task.

**Cyclische Fehlerfallbehandlung** In einem cyclischen Schedulingalgorithmus unter POSIX und dem AUTOSAR Standard mit Timex können Tasks nicht auf ihre Deadline hin überprüft werden. Sobald der Task am Ende seines Slots angekommen ist, wird dieser automatisch vom Core entfernt und der Task für den nächsten Slot geladen. Somit ist das Überschreiten von Deadlines und anderen Zeiten nicht möglich und wird von vornherein unterbunden. Andere Tasks werden durch das Fehlverhalten nicht beeinflusst.
In einem HAMS System mit HAMS KB können auch im cyclischen Scheduling Deadline Überschreitungen erkannt werden. Sollte ein simulierter Task die Deadline in seinem Slot überschreiten, so wird dieses erkannt. Dies ist aber nur möglich, wenn die Deadline kleiner ist als die Periode bzw. der cyclische Slot. Ist dies nicht der Fall, muss der Task selbst durch eine interne Fehlererkennung einen Fehler bekannt geben. Hierfür werden die im nächsten Abschnitt näher beschriebenen Checkpunktparameter $CP_{UE.n.m}$ benützt.

**Task Fehlerfallbehandlung** Um einen Taskfehler zu erkennen, muss der Task selbst eine Fehlerdiagnose durchführen. Dies ist in POSIX, AUTOSAR Standard und HAMS gleich. Der Task kann durch BIST erfolgen, die periodisch ausgeführt werden, oder durch bestimmte Verzweigungen im Ablauf des Tasks. So können bestimmte Bedingungen im Task u.U. in einem Fehlerfall nicht erreicht werden und so ein Fehler getriggert werden.
Ausschließlich im HAMS System mit der HAMS KB hat ein Task eine gewisse Anzahl von Checkpunkten während seiner Ausführung, wie in der Formel 5.3 mit dem Parameter $CP_{UE.n.m}$ und in Abbildung 6.3 dargestellt. Diese sind phasenabhängig, optional und in einer gewissen Zeitspanne der WCET aufgelistet. Sollte die Meldung über das Erreichen einer dieser Checkpunkte ausbleiben, so nimmt der HAMS Scheduler diesen Task als fehlerhaft an und leitet den Übergang in den Fail-Operational Modus ein.

**Hardware Fehlerfallbehandlung** Sollte in einem simulierten Task eine notwendige Hardwareschnittstelle nicht mehr vorhanden sein, so muss in POSIX und im AUTOSAR Standard der Task seine eigene Fehlerbehandlungsroutine ausführen. Im HAMS System mit HAMS KB muss die

Hardware Fehlererkennung, ebenso wie die Meldung der Checkpunkte, durch einen Task erfolgen. Dazu kann es einen gesonderten BIST Task geben, welcher periodisch läuft und via Ausbleiben eines Checkpunktes einen Fehler an den HAMS Scheduler meldet. Auch Tasks selbst können Tests über die für ihre Funktion wichtigen Schnittstellen durchführen. Sollte eine Hardware Schnittstelle nicht mehr die gewünschte Funktion erfüllen, so muss auch dieser Task die Meldung über das Erreichen von Checkpunkten auslassen. Der HAMS Scheduler nimmt dann automatisch den Fail-Operational Zustand ein.

## 7.3 Zusammenfassung

Bei der Evaluation der HAMS Knowledgebase als Teil des HAMS Schedulers zeigen sich dessen Vorteile gegenüber den aktuellen statischen Methoden und den Methoden aus der Forschung. Zur Durchführung der Evaluation wurde zuerst ein Evaluationskonzept aufgestellt. Dieses Evaluationskonzept gliedert sich in vier Abschnitte. So wurden die Bereiche Scheduling / Allokation, Rekonfiguration, Hardware und Fehlerfallbehandlung in das Evaluationskonzept aufgenommen. Für die Evaluation im Bereich Scheduling und Allokation werden Tasks und deren Eigenschaften aus den Zieldomänen ermittelt und den aktuellen Methoden gegenübergestellt. Für die Rekonfiguration werden diese Tasks ebenfalls verwendet und unter Betrachtung der ermittelten Algorithmen aus Kapitel 6 mit einem Evaluationssystem auf Durchführbarkeit verglichen. Für den Bereich Hardware werden die Multi-Core Eigenschaften der Algorithmen aus Kapitel 6 gegen die Anforderung Multi-Core zu unterstützen evaluiert. Im Fall der Fehlerbehandlung wird ermittelt wie sich das HAMS System bei einem Fehler verhält und die Auswirkungen eingegrenzt werden können.
Unterstützt wird die Evaluation durch eine Evaluationsumgebung (ARM und PowerPC), bestehen aus Evaluationstasks, Multi-Core Prozessoren und Visualisierungssoftware zur Darstellung von Scheduling Abläufen. Bei der Evaluation im Bereich Scheduling werden die Algorithmen aus Kapitel 6 auf ein gegebenes Taskset für das Evaluationssystem ARM angewandt (Tabelle 7.2). Der Output der KB Erstellung, die KB Datei, ist in Anhang 9.1 zu finden. Anhand der Auswertungen ist es

erkennbar das ein aktuelles statisches System diese Anzahl an Tasks nicht auf einem System allokieren kann (Tabelle 7.2). Ein weiteres Beispiel aus diesem Bereich zeigt ebenfalls die Vorteile der KB im Bereich Scheduling gegenüber den aktuellen Systemen auf (Abbildung 7.2). In diesem Beispiel werden aber gleichzeitig die Grenzen der KB deutlich. Sollte es zur System Designzeit nicht möglich sein, logische Abhängigkeiten zwischen Tasks zu finden, wird es nicht möglich sein Tasks auf einem System zusammenzulegen. Auch im Bereich Allokation werden die Tasks aus den Zieldomänen verwendet um sie der Allokation mit den KB Algorithmen und der Allokation mit aktuellen Algorithmen gegenüberzustellen. Damit eine Rekonfiguration durchführbar ist ohne Verletzung der Deadline, muss zur Erstellung der KB eine Rekonfigurationsprüfung durchgeführt werden. Dabei wird eine Formel benötigt, um die maximal mögliche Dauer der Rekonfiguration wiederzugeben. In der Evaluation wurde in einem HAMS System gezeigt, wie lange die einzelnen Zeitabschnitte dauern und ob diese die durch die KB errechneten Maximalwerte überschreiten. Wie die Evaluation zeigt, werden diese Werte immer eingehalten (Charts 7.7, 7.8 und 7.9). So ist die deterministische Rekonfiguration des HAMS Schedulers und die Formel, welche bei der KB Berechnung mit einbezogen wird korrekt und es kann schon bei der Erstellung der HAMS Knowledgebase evaluiert werden, ob alle Deadlines eingehalten werden. Somit sind die Anforderungen im Bereich Rekonfiguration umgesetzt. Die Vorabprüfung ist mit der HAMS Knowledgebase für alle Konfigurationen möglich, abhängig vom Schedulingalgorithmus.

Auch der Einbezug von Multi-Core Hardware Eigenschaften wird in dieser Evaluation genauer betrachtet. Insbesondere wird evaluiert, ob die benötigten Parameter und Algorithmen für die KB Erstellung ausreichen. Sowohl für die Frequenzeinstellung eines Multi-Core Prozessors als auch für die verlängerte Ausführungszeit eines Tasks nach einer Migration konnten die Parameter und Algorithmen erfolgreich bestätigt werden. So ist die HAMS Knowledgebase bereit, aktiv Multi-Core Systeme in deren Hardware Eigenschaften zu unterstützen. Insbesondere sind hier die Bereiche Migrationseinschränkungen der Cores, I/O Schnittstellen und Peripherie aus den Anforderungen zu nennen.

Im Bereich der Fehlerfallbehandlung und Auflösung ist das HAMS System zur Evaluation den aktuellen Systemen gegenübergestellt worden. Durch eine Vielzahl an Fehlererkennungsmöglichkeiten, die allesamt durch die KB ermöglicht wurden, ist das HAMS System im Verbund mit der HAMS KB führend in der Fehlererkennung. So sind unterschiedlichste Fehler, vom Taskfehler bis hin zum Hardwarefehler erkennbar, was in aktuellen Systemen nicht oder nur bedingt möglich ist.

# 8 Ergebnisse und Ausblick

Scheduling und die optimale Verteilung von Tasks auf Systemen ist ein weitreichendes und sehr vielfältiges Forschungsgebiet. Mit den anfänglichen und auch heute noch gültigen Formeln von Liu und Layland [89] wurde die Grundlage für die Berechnung für auslastungsgerichtetes und deterministisches Scheduling gelegt. Neue Hardware in Form von Multi-Core Prozessoren erweiterte das Forschungsgebiet von vorwiegend auslastungsbezogen hin zu auslastungs- und verteilungsspezifisch. Nun war es nicht nur von Bedeutung, eine höchstmögliche Auslastung zu bekommen, sondern diejenige Verteilung von Funktionen zu finden, welche eine höchstmögliche und sichere Auslastung über den gesamten Multi-Core Prozessor hinweg darstellt.
Durch die hohe Rechenleistung von Multi-Core Prozessoren auf einem geringen Raum wurden auch die Zieldomänen Automotive und Avionik auf diese aufmerksam. Viele Projekte wurden aufgesetzt, um Multi-Core Prozessoren zusammen mit dem statischen Ansatz in den Zieldomänen zu integrieren (Kapitel 3), wie IMA im Avionik Bereich. Auch im Automotive Bereich wird eine rekonfigurierbare Plattform gesucht, welche versucht, mehr Software auf einem Steuergerät zu allokieren mit Hilfe von Multi-Core Prozessoren. Auch hier hat das IMA Prinzip und dessen Partitionierung es geschafft, in Forschungssystemen angewandt zu werden.
Um mehr Funktionen auf einem Steuergerät zu laden als bisher möglich, gilt es den statischen Ansatz aufzuweichen, aber in den Grundzügen beizubehalten. Ziel ist es, vom aktuellen statischen Ansatz in einen semi-statischen Ansatz in den Zieldomänen überzugehen. Um diesen Übergang erstmalig zu definieren, wurde die HAMS Knowledgebase für das HAMS System umgesetzt. Damit die HAMS Knowledgebase aus dem HAMS System in den Zieldomänen erfolgreich eingesetzt werden kann, müssen diese folgende Anforderungen erfüllen (Kapitel 4.2):

**Scheduling / Allokation** Um mehr Tasks mit der HAMS Knowledgebase zu laden, werden im KB Konzept für Scheduling und Allokation zwei elementare Parameter eingeführt. Zum einen können Tasks eigene Phasen zugeteilt werden und zum anderen sind logische Abhängigkeiten zwischen Tasks eingeführt worden. Das KB Konzept für Phasen erlaubt es, dass ein Task je nach aktueller Situation seine Rechenzeit anpassen kann. Im Falle der Einparkhilfe kann der Task so seine Rechenzeit reduzieren, wenn aktuell kein Einparkvorgang möglich ist, wie z.B. auf der Autobahn. Für den Parameter der logischen Abhängigkeit können Tasks untereinander logisch verknüpft werden und so Rechenzeit untereinander freigeben. Ein Beispiel hierfür sind die Funktionen Einparkhilfe und Geschwindigkeitsregelanlage. Diese schließen sich gegenseitig aus. So wird während des Einparkvorgangs die Geschwindigkeitsregelanlage nicht benötigt und konträr wird auf der Autobahn nur die Geschwindigkeitsregelanlage und nicht die Einparkhilfe angeschaltet.
Diese zwei Grundgedanken des Konzeptes im Bereich Scheduling und Allokation wurden im Verlauf der Arbeit weiter untergliedert. Um dieses Konzept in den Zieldomänen einsetzten zu können, sind Anforderungen aus den Zieldomänen erarbeitet worden. Diese Anforderungen beziehen sich unter anderem darauf, cyclische und periodische Schedulingalgorithmen und bekannte Feasibility Tests zu verwenden sowie alle Berechnungen bezüglich Scheduling und Allokation schon vorab und nicht während der Laufzeit durchzuführen. Daraus abgeleitet wurden Parameter eingeführt, die das Verhalten eines Tasks in Bezug auf phasenabhängige Rechenzeit, logische Abhängigkeiten und Basis Parameter beschreiben.
Mit diesen Parametern und den zuvor ermittelten Anforderungen wurden Algorithmen entwickelt, welche die Berechnungen im Verlauf der Knowledgebase Erstellung durchführen. Diese Berechnungen stützen sich auf bekannte Feasibility Tests für cyclische und periodische Schedulingalgorithmen, die um das phasenorientierte Scheduling erweitert wurden. Ebenso wird ein Brute Force Algorithmus angewandt, welcher alle möglichen Allokationen an Tasks ermittelt und überprüft. Ist die KB Erstellung mit diesen Algorithmen durchlaufen, müssen keine weiteren Schedulingberechnungen zur Laufzeit ausgeführt werden.
In der Evaluation ist der Vorteil semi-statischer Systeme mit der Knowledgebase gegenüber aktuellen statischen Systemen ersichtlich. Mit der

Knowledgebase ist es möglich mehr Funktionen auf einem semi statischen System zu laden als mit den bisherigen aktuellen statischen Systemen.
So ist zu erkennen, dass ein System aus den Tasks Einparkhilfe, Geschwindigkeitsregelanlage und Regensensor nur zusammen mit der Knowledgebase erfolgreich auf einem System geschedult werden kann. Aktuelle Systeme benötigen dafür eine extra ECU. Auch hat sich in der Evaluation gezeigt, dass die KB Erstellung effektiver wird, je mehr logische Abhängigkeiten zwischen Tasks gefunden werden können. So konnten erfolgreich sieben Task auf einer Quad-Core CPU in einem semi-statischen System mit Knowledgebase allokiert werden (Abbildung 7.2), was mit aktuellen statischen Systemen nur mit mehreren ECUs gelingt.
So wurden nicht nur die Anforderungen an die Knowledgebase im Bereich Scheduling und Allokation konsequent umgesetzt, sondern auch aufgezeigt welche Vorteile die Knowledgebase in Form von besserer Auslastung hat.

**Rekonfiguration** Eine Anforderungen für die Rekonfiguration eines Systems aus den Zieldomänen ist es, vorab die Rekonfiguration zu evaluieren und auf Machbarkeit zu überprüfen. Um zu erkennen, ob eine Rekonfiguration zu lange dauert, wurde ein Parameter für jeden Task eingeführt. Eine weitere Anforderung war es, alle möglichen Konfigurationen auf Rekonfigurierbarkeit hin zu überprüfen und daraus die optimale Konfiguration auszuwählen. Um dies zu ermöglichen wurde der Brute Force Algorithmus im Konzept der KB Erstellung aufgenommen. Die durch den Brute Force Algorithmus erstellten Konfigurationen müssen zusätzlich auf zeitliche Dauer mit einer Formel überprüft werden. So wendet die HAMS Knowledgebase bei der Rekonfigurationsprüfung die Formel an, welche sich aus dem Rekonfigurationsschema des HAMS Schedulers und den Eigenschaften des zugrunde liegenden Schedulingalgorithmus, periodisch oder cyclisch, ergeben. Die zeitliche Dauer des Migrationsvorgangs kann somit berechnet und gegen Vorgaben verglichen werden.
In der Evaluation der Rekonfiguration ist zu erkennen, dass die zeitlichen maximalen Vorgaben, die aus dem HAMS Migrationsschema entstehen, immer eingehalten werden. Eine Rekonfigurationsprüfung

in der KB kann somit vorab evaluieren, ob die zeitlichen Vorgaben eingehalten werden. Diese wurde sowohl für cyclische als auch periodische Schedulingalgorithmen überprüft. Für den Fall dass diese nicht eingehalten werden ermittelt der Brute Force Algorithmus noch andere Verteilungen um dennoch unter Einhaltung der zeitlichen Parameter zu rekonfigurieren. Damit ist es möglich, Konfigurationen und invalide Rekonfigurationsübergänge vorab zu erkennen und auszuschließen, sofern die Vorgaben nicht erfüllt werden.

**Multi-Core Hardware** Das KB Konzept sieht vor Multi-Core Prozessoren und deren Fähigkeit, viel Leistung auf kleinem Raum zu bringen, zu unterstützen. Folglich zeiht die HAMS Knowledgebase während der Erstellung die Eigenschaften von Multi-Core Prozessoren mit ein. Hier sind vor allem die unterschiedlichen Taktraten eines Multi-Core Prozessors anzugeben und die erforderliche Veränderung der Ausführungszeit eines Tasks die damit einhergehen. Ebenso werden die I/O Schnittstellen des Prozessors und deren Core Verfügbarkeit mit einbezogen, wie auch die manuelle Allokation eines Tasks auf einem Core ausgehend von Interrupts und ähnlichen lokalen Eigenschaften. All diese Eigenschaften spiegeln sich in Task Parametern sowie in den eigens entwickelten Multi-Core Parametern wieder. Die Ergebnisse dieser Unterstützung sind in den Algorithmen zur Allokation von Tasks auf Cores ersichtlich. So wird hier der Brute Force Algorithmus eingeschränkt aufgrund der Hardware Eigenschaften.
In der Evaluation ist anschaulich dargestellt, dass Multi-Core Eigenschaften mit einbezogen werden und so nicht nur ein schedulbares System hervorbringen, sondern auch eine bessere Performance bieten können, indem die Taktraten mit einbezogen werden. Aber werden Tasks oder Cores mit zu vielen Einschränkungen versehen, wird eine Schedulefindung erschwert oder sogar verhindert.

**Fehlerfallbehandlung** Auch in durchgeplanten Systemen mit einer ist es möglich, dass Tasks sich fehlverhalten, auch wenn eine HAMS Knowledgebase vorhanden ist. Um dies zu erkennen, wurden in dem HAMS Knowledgebase XML Format für den HAMS Scheduler nicht nur die WCET und Deadline der Tasks eingetragen. Zusätzlich können auch

Checkpunkte, die ein Task verwenden kann, um Fehlerfälle schneller zu erkennen der XML Datei hinzugefügt werden. Auf Basis der HAMS Knowledgebase XML Datei wertet der HAMS Scheduler zur Laufzeit aus, ob ein korrektes Verhalten eines Tasks vorliegt. Sollte ein Fehlverhalten erkannt worden sein, wird die in der HAMS Knowledgebase eingetragene Fail Operational Funktion hergestellt, um eine minimale Funktion des Steuergeräts zu gewährleisten.

Zusammenfassend ist in dieser Arbeit die HAMS Knowledgebase für ein HAMS System von der Idee über die Anforderungen und die unterschiedlichen Algorithmen bis zur Evaluation im Bereich von Multi-Core Steuergeräten komplett umgesetzt worden. Das Ziel aus Kapitel 1, mehr Funktionen auf einer ECU zu allokieren, wird durch die HAMS Knowledgebase erreicht, wie im Kapitel 7 Evaluation dargestellt ist.

**Nächste Schritte** Durch die Erfüllung dieser Anforderungen hat die HAMS Knowledgebase das semi-statische Scheduling in den Zieldomänen Automotive und Avionik eingeführt. Das HAMS System stellt somit eine rekonfigurierbare phasenabhängige Plattform der weichen Echtzeit für diese Domänen dar.
Eine Erweiterung dieses HAMS Systems ist es, die Rekonfiguration über Steuergeräte hinweg zu erlauben. So sollten Tasks nicht nur zwischen Cores rekonfiguriert werden, sondern auch auf andere ECUs gerechnet werden können. Auch hier ist es dringend erforderlich, eine HAMS Knowledgebase einzusetzen. Die HAMS Knowledgebase muss ebenso die Rekonfiguration bzw. den Transfer von Tasks betrachten, die zwischen zwei Steuergeräten vorgenommen werden. Wie in Multi-Core Systemen wird auch diese Rekonfiguration zeitlichen Anforderungen unterliegen, die eingehalten werden müssen. Auch die Architekturen von ECUs können unterschiedlich sein, genauso wie deren I/O Schnittstellen. So können Tasks nicht beliebig von ECU zu ECU transferiert werden. Eine offline Validierung und Überprüfung wie in der HAMS Knowledgebase muss auch hier stattfinden.
Sollte es möglich sein, Systeme über deren Core Grenze hinweg zu rekonfigurieren, können auch neue Formen des Fail Operational Modus eingeführt werden. So kann es z.B. möglich sein, Steuergeräte bei Ausfall komplett oder teilweise zu ersetzen indem nicht notwendige Funktionen

nicht mehr ausgeführt werden. Der Fail Operational Modus kann auf diese Weise umfangreicher untergliedert werden als es in der HAMS Knowledgebase aktuell auszulesen ist und ermöglicht so eine optimierte Betriebsstrategie im Fehlerfall.

# 9 Anhang

Listing 9.1: RMS Knowledgebase mit 3 Tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<KB_result>
  <MC_Configuration>
    <Nr_Cores>2</Nr_Cores>
    <Synchron_Cores>1</Synchron_Cores>
    <Common_Clock>1</Common_Clock>
    <Frequencys>
      <Frequency>1000</Frequency>
    </Frequencys>
    <Peripherals>
      <Peripheral_Name>serial0</Peripheral_Name>
      <Peripheral_Interrupt_Name>
      </Peripheral_Interrupt_Name>
      <Is_Interrupt_Needed>0</Is_Interrupt_Needed>
      <Allowed_Interrupt_Cores>
        <AC>0</AC>
        <AC>1</AC>
      </Allowed_Interrupt_Cores>
    </Peripherals>
  </MC_Configuration>
  <Configuration>
    <ID>1</ID>
    <Core>
      <Core_ID>0</Core_ID>
        <Sched_Class>Pure RMS</Sched_Class>
        <Use_Only_RMS>1</Use_Only_RMS>
        <RMS_Tick>1</RMS_Tick>
        <Frequency>1000</Frequency>
```

```
<Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P1</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkegeschwindigkeit</Sub-Phase>
  <RMS_Prio>0</RMS_Prio>
  <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Lin_Init</Task_Name>
  <Task_ID>3</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>3</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Init_Norm</Sub-Phase>
  <RMS_Prio>1</RMS_Prio>
  <RMS_Max_WCET_Task>100</RMS_Max_WCET_Task>
  <RMS_Period_Task>60000</RMS_Period_Task>
  <RMS_Deadline>60000</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>HAMS_SLS</Task_Name>
  <Task_ID>6</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>6</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>SLS_Norm</Sub-Phase>
  <RMS_Prio>2</RMS_Prio>
  <RMS_Max_WCET_Task>1</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
```

```
    <Tasks_on_Core>
      <Task_Name>Linux_Kworker0</Task_Name>
      <Task_ID>4</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>4</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker0_Norm</Sub-Phase>
      <RMS_Prio>3</RMS_Prio>
      <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
      <RMS_Period_Task>60000</RMS_Period_Task>
      <RMS_Deadline>60000</RMS_Deadline>
    </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>1</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>3</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Scheibe_Nass</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>GRA</Task_Name>
    <Task_ID>0</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
```

```
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Parkegeschwindigkeit</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
      <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Lin_Kworker1</Task_Name>
      <Task_ID>5</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>5</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker1_Norm</Sub-Phase>
      <RMS_Prio>2</RMS_Prio>
      <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
      <RMS_Period_Task>60000</RMS_Period_Task>
      <RMS_Deadline>60000</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>Parkgeschwindigkeit/Init_Norm/
  Scheibe_Nass/Kworker0_Norm/Kworker1_Norm/
  SLS_Norm/</Beschreibung>
</Configuration>
<Configuration>
  <ID>2</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>5</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
```

```
  <Leading_Task_ID>6</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>SLS_Norm</Sub-Phase>
  <RMS_Prio>0</RMS_Prio>
  <RMS_Max_WCET_Task>1</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <RMS_Prio>1</RMS_Prio>
  <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Regensensor</Task_Name>
  <Task_ID>2</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>2</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Scheibe_Trocken</Sub-Phase>
  <RMS_Prio>2</RMS_Prio>
  <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Lin_Init</Task_Name>
  <Task_ID>3</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>3</Leading_Task_ID>
```

```
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Init_Norm</Sub-Phase>
    <RMS_Prio>3</RMS_Prio>
    <RMS_Max_WCET_Task>100</RMS_Max_WCET_Task>
    <RMS_Period_Task>60000</RMS_Period_Task>
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <RMS_Prio>4</RMS_Prio>
    <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
    <RMS_Period_Task>60000</RMS_Period_Task>
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>5</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Einparkhilfe</Task_Name>
    <Task_ID>101/0-P1</Task_ID>
    <Is_Leading_Task>0</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
```

```
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Lin_Kworker1</Task_Name>
      <Task_ID>5</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>5</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker1_Norm</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
      <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
      <RMS_Period_Task>60000</RMS_Period_Task>
      <RMS_Deadline>60000</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Parkgeschwindigkeit
  /Scheibe_Trocken/Init_Norm/Kworker0_Norm
  /Kworker1_Norm/</Beschreibung>
</Configuration>
<Configuration>
  <ID>3</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <RMS_Prio>0</RMS_Prio>
      <RMS_Max_WCET_Task>1</RMS_Max_WCET_Task>
```

```
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <RMS_Prio>1</RMS_Prio>
  <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Lin_Init</Task_Name>
  <Task_ID>3</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>3</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Init_Norm</Sub-Phase>
  <RMS_Prio>2</RMS_Prio>
  <RMS_Max_WCET_Task>100</RMS_Max_WCET_Task>
  <RMS_Period_Task>60000</RMS_Period_Task>
  <RMS_Deadline>60000</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Linux_Kworker0</Task_Name>
  <Task_ID>4</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>4</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Kworker0_Norm</Sub-Phase>
  <RMS_Prio>3</RMS_Prio>
  <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
  <RMS_Period_Task>60000</RMS_Period_Task>
```

```
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>1</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>3</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Einparkhilfe</Task_Name>
    <Task_ID>101/0-P2</Task_ID>
    <Is_Leading_Task>0</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P2</Active_Phase_Name>
    <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Scheibe_Nass</Sub-Phase>
    <RMS_Prio>1</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Kworker1</Task_Name>
    <Task_ID>5</Task_ID>
```

```
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>5</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker1_Norm</Sub-Phase>
      <RMS_Prio>2</RMS_Prio>
      <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
      <RMS_Period_Task>60000</RMS_Period_Task>
      <RMS_Deadline>60000</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Fahrtgeschwindigkeit
  /Init_Norm/Kworker0_Norm/Scheibe_Nass
  /Kworker1_Norm/</Beschreibung>
</Configuration>
<Configuration>
  <ID>4</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <RMS_Prio>0</RMS_Prio>
      <RMS_Max_WCET_Task>1</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>GRA</Task_Name>
```

```
    <Task_ID>0</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P2</Active_Phase_Name>
    <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
    <RMS_Prio>1</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Init</Task_Name>
    <Task_ID>3</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>3</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Init_Norm</Sub-Phase>
    <RMS_Prio>2</RMS_Prio>
    <RMS_Max_WCET_Task>100</RMS_Max_WCET_Task>
    <RMS_Period_Task>60000</RMS_Period_Task>
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <RMS_Prio>3</RMS_Prio>
    <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
    <RMS_Period_Task>60000</RMS_Period_Task>
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
```

```
<Sched_Class>Pure RMS</Sched_Class>
<Use_Only_RMS>1</Use_Only_RMS>
<RMS_Tick>1</RMS_Tick>
<Frequency>1000</Frequency>
<Tasks_Nr_on_Core>3</Tasks_Nr_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P2</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <RMS_Prio>0</RMS_Prio>
  <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Regensensor</Task_Name>
  <Task_ID>2</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>2</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Scheibe_Trocken</Sub-Phase>
  <RMS_Prio>1</RMS_Prio>
  <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
  <RMS_Period_Task>35</RMS_Period_Task>
  <RMS_Deadline>35</RMS_Deadline>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Lin_Kworker1</Task_Name>
  <Task_ID>5</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>5</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Kworker1_Norm</Sub-Phase>
  <RMS_Prio>2</RMS_Prio>
```

```
      <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
      <RMS_Period_Task>60000</RMS_Period_Task>
      <RMS_Deadline>60000</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Fahrtgeschwindigkeit/Init_Norm
  /Kworker0_Norm/Scheibe_Trocken/
      Kworker1_Norm/</Beschreibung>
</Configuration>
<Fail_Operation>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>3</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <RMS_Prio>0</RMS_Prio>
      <RMS_Max_WCET_Task>1</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Lin_Init</Task_Name>
      <Task_ID>3</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>3</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Init_Norm</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
```

```
    <RMS_Max_WCET_Task>100</RMS_Max_WCET_Task>
    <RMS_Period_Task>60000</RMS_Period_Task>
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <RMS_Prio>2</RMS_Prio>
    <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
    <RMS_Period_Task>60000</RMS_Period_Task>
    <RMS_Deadline>60000</RMS_Deadline>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>1</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P2</Active_Phase_Name>
    <Sub-Phase>Fail</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
```

```
        <Task_Name>Lin_Kworker1</Task_Name>
        <Task_ID>5</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>5</Leading_Task_ID>
        <Active_Phase_Name>P1</Active_Phase_Name>
        <Sub-Phase>Kworker1_Norm</Sub-Phase>
        <RMS_Prio>1</RMS_Prio>
        <RMS_Max_WCET_Task>30</RMS_Max_WCET_Task>
        <RMS_Period_Task>60000</RMS_Period_Task>
        <RMS_Deadline>60000</RMS_Deadline>
      </Tasks_on_Core>
    </Core>
  </Fail_Operation>
</KB_result>
```

Listing 9.2: RMS mit Polling Server Ausschnitt aus KB

```
<Configuration>
  <ID>1</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <RMS_Prio>3</RMS_Prio>
      <RMS_Max_WCET_Task>1</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Einparkhilfe</Task_Name>
      <Task_ID>101/0-P1</Task_ID>
      <Is_Leading_Task>0</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
      <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Polling_Server>
        <PS_Max_WCET_Task>130</RMS_Max_WCET_Task>
```

```
      <PS_Period_Task>60000</RMS_Period_Task>
      <PS_Deadline>60000</RMS_Deadline>
      <PS_Tasks>
        <Task_Name>Lin_Init</Task_Name>
        <Task_ID>3</Task_ID>
      </PS_Tasks>
      <PS_Tasks>
        <Task_Name>Linux_Kworker0</Task_Name>
        <Task_ID>4</Task_ID>
      </PS_Tasks>
    </Polling_Server>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>1</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>3</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Scheibe_Nass</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Polling_Server>
    <PS_Max_WCET_Task>30</RMS_Max_WCET_Task>
    <PS_Period_Task>60000</RMS_Period_Task>
    <PS_Deadline>60000</RMS_Deadline>
    <PS_Tasks>
```

```
        <Task_Name>Linux_Kworker1</Task_Name>
        <Task_ID>5</Task_ID>
      </PS_Tasks>
    </Polling_Server>
    <Tasks_on_Core>
      <Task_Name>GRA</Task_Name>
      <Task_ID>0</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
      <RMS_Prio>2</RMS_Prio>
      <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
</Configuration>
```

Listing 9.3: EDF Knowledgebase mit 3 Tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<KB_result>
  <MC_Configuration>
    <Nr_Cores>2</Nr_Cores>
    <Synchron_Cores>1</Synchron_Cores>
    <Common_Clock>1</Common_Clock>
    <Frequencys>
      <Frequency>1000</Frequency>
    </Frequencys>
    <Peripherals>
      <Peripheral_Name>serial0</Peripheral_Name>
      <Peripheral_Interrupt_Name>
      </Peripheral_Interrupt_Name>
      <Is_Interrupt_Needed>0</Is_Interrupt_Needed>
      <Allowed_Interrupt_Cores>
        <AC>0</AC>
        <AC>1</AC>
      </Allowed_Interrupt_Cores>
    </Peripherals>
  </MC_Configuration>
  <Configuration>
    <ID>1</ID>
    <Core>
      <Core_ID>0</Core_ID>
      <Sched_Class>Pure EDF</Sched_Class>
      <Use_Only_EDF>1</Use_Only_EDF>
      <EDF_Tick>1</EDF_Tick>
      <Frequency>1000</Frequency>
      <Tasks_Nr_on_Core>5</Tasks_Nr_on_Core>
      <Tasks_on_Core>
        <Task_Name>HAMS_SLS</Task_Name>
        <Task_ID>6</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>6</Leading_Task_ID>
        <Active_Phase_Name>P1</Active_Phase_Name>
        <Sub-Phase>SLS_Norm</Sub-Phase>
```

```
  <EDF_Max_WCET>1</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>2</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P1</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>25</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Lin_Init</Task_Name>
  <Task_ID>3</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>3</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Init_Norm</Sub-Phase>
  <EDF_Max_WCET>100</EDF_Max_WCET>
  <EDF_Deadline>60000</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
```

```
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <EDF_Max_WCET>30</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure EDF</Sched_Class>
  <Use_Only_EDF>1</Use_Only_EDF>
  <EDF_Tick>25</EDF_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Scheibe_Nass</Sub-Phase>
    <EDF_Max_WCET>25</EDF_Max_WCET>
    <EDF_Deadline>35</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
   <Tasks_on_Core>
    <Task_Name>Lin_Kworker1</Task_Name>
    <Task_ID>5</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>5</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker1_Norm</Sub-Phase>
```

```
      <EDF_Max_WCET>30</EDF_Max_WCET>
      <EDF_Deadline>60000</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Parkgeschwindigkeit
  /Init_Norm/Kworker0_Norm/worker1_Norm/
  Scheibe_Nass/</Beschreibung>
</Configuration>
<Configuration>
  <ID>2</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure EDF</Sched_Class>
    <Use_Only_EDF>1</Use_Only_EDF>
    <EDF_Tick></EDF_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>6</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <EDF_Max_WCET>1</EDF_Max_WCET>
      <EDF_Deadline>35</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>GRA</Task_Name>
      <Task_ID>0</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
      <EDF_Max_WCET>2</EDF_Max_WCET>
```

```
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P1</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>25</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Regensensor</Task_Name>
  <Task_ID>2</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>2</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Scheibe_Trocken</Sub-Phase>
  <EDF_Max_WCET>2</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Lin_Init</Task_Name>
  <Task_ID>3</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>3</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Init_Norm</Sub-Phase>
  <EDF_Max_WCET>100</EDF_Max_WCET>
  <EDF_Deadline>60000</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
```

```
      <Task_Name>Linux_Kworker0</Task_Name>
      <Task_ID>4</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>4</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker0_Norm</Sub-Phase>
      <EDF_Max_WCET>30</EDF_Max_WCET>
      <EDF_Deadline>60000</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
  </Core>
  <Core>
    <Core_ID>1</Core_ID>
    <Sched_Class>Pure EDF</Sched_Class>
    <Use_Only_EDF>1</Use_Only_EDF>
    <EDF_Tick>0</EDF_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>Lin_Kworker1</Task_Name>
      <Task_ID>5</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>5</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker1_Norm</Sub-Phase>
      <EDF_Max_WCET>30</EDF_Max_WCET>
      <EDF_Deadline>60000</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Parkgeschwindigkeit/
  Scheibe_Trocken/Init_Norm/Kworker0_Norm
  /Kworker1_Norm/</Beschreibung>
</Configuration>
<Configuration>
  <ID>3</ID>
  <Core>
```

```
<Core_ID>0</Core_ID>
<Sched_Class>Pure EDF</Sched_Class>
<Use_Only_EDF>1</Use_Only_EDF>
<EDF_Tick></EDF_Tick>
<Frequency>1000</Frequency>
<Tasks_Nr_on_Core>5</Tasks_Nr_on_Core>
<Tasks_on_Core>
  <Task_Name>HAMS_SLS</Task_Name>
  <Task_ID>6</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>6</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>SLS_Norm</Sub-Phase>
  <EDF_Max_WCET>1</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>25</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P2</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>2</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
```

```
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Init</Task_Name>
    <Task_ID>3</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>3</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Init_Norm</Sub-Phase>
    <EDF_Max_WCET>100</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <EDF_Max_WCET>30</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure EDF</Sched_Class>
  <Use_Only_EDF>1</Use_Only_EDF>
  <EDF_Tick>25</EDF_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
```

```
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Scheibe_Nass</Sub-Phase>
      <EDF_Max_WCET>25</EDF_Max_WCET>
      <EDF_Deadline>35</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Lin_Kworker1</Task_Name>
      <Task_ID>5</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>5</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Kworker1_Norm</Sub-Phase>
      <EDF_Max_WCET>30</EDF_Max_WCET>
      <EDF_Deadline>60000</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Fahrtgeschwindigkeit/Init_Norm
  /Kworker0_Norm/Kworker1_Norm/
  Scheibe_Nass/</Beschreibung>
</Configuration>
<Configuration>
  <ID>4</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure EDF</Sched_Class>
    <Use_Only_EDF>1</Use_Only_EDF>
    <EDF_Tick></EDF_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>6</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
```

```
  <Sub-Phase>SLS_Norm</Sub-Phase>
  <EDF_Max_WCET>1</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>25</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P2</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <EDF_Max_WCET>2</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Regensensor</Task_Name>
  <Task_ID>2</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>2</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Scheibe_Trocken</Sub-Phase>
  <EDF_Max_WCET>2</EDF_Max_WCET>
  <EDF_Deadline>35</EDF_Deadline>
  <EDF_Period>0</EDF_Period>
```

```
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Init</Task_Name>
    <Task_ID>3</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>3</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Init_Norm</Sub-Phase>
    <EDF_Max_WCET>100</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <EDF_Max_WCET>30</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure EDF</Sched_Class>
  <Use_Only_EDF>1</Use_Only_EDF>
  <EDF_Tick>0</EDF_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Kworker1</Task_Name>
    <Task_ID>5</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>5</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
```

```
      <Sub-Phase>Kworker1_Norm</Sub-Phase>
      <EDF_Max_WCET>30</EDF_Max_WCET>
      <EDF_Deadline>60000</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Fahrtgeschwindigkeit/
  Scheibe_Trocken/Init_Norm/Kworker0_Norm
  /Kworker1_Norm/</Beschreibung>
</Configuration>
<Fail_Operation>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure EDF</Sched_Class>
    <Use_Only_EDF>1</Use_Only_EDF>
    <EDF_Tick></EDF_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <EDF_Max_WCET>1</EDF_Max_WCET>
      <EDF_Deadline>35</EDF_Deadline>
      <EDF_Period>0</EDF_Period>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Regensensor</Task_Name>
      <Task_ID>2</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>2</Leading_Task_ID>
      <Active_Phase_Name>P2</Active_Phase_Name>
      <Sub-Phase>Fail</Sub-Phase>
      <EDF_Max_WCET>2</EDF_Max_WCET>
```

```
    <EDF_Deadline>35</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Init</Task_Name>
    <Task_ID>3</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>3</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Init_Norm</Sub-Phase>
    <EDF_Max_WCET>100</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Linux_Kworker0</Task_Name>
    <Task_ID>4</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>4</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Kworker0_Norm</Sub-Phase>
    <EDF_Max_WCET>30</EDF_Max_WCET>
    <EDF_Deadline>60000</EDF_Deadline>
    <EDF_Period>0</EDF_Period>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure EDF</Sched_Class>
  <Use_Only_EDF>1</Use_Only_EDF>
  <EDF_Tick>0</EDF_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Lin_Kworker1</Task_Name>
    <Task_ID>5</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
```

```
        <Leading_Task_ID>5</Leading_Task_ID>
        <Active_Phase_Name>P1</Active_Phase_Name>
        <Sub-Phase>Kworker1_Norm</Sub-Phase>
        <EDF_Max_WCET>30</EDF_Max_WCET>
        <EDF_Deadline>60000</EDF_Deadline>
        <EDF_Period>0</EDF_Period>
      </Tasks_on_Core>
    </Core>
  </Fail_Operation>
</KB_result>
```

Listing 9.4: Cyclische Knowledgebase mit 3 Tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<KB_result>
  <MC_Configuration>
    <Nr_Cores>2</Nr_Cores>
    <Synchron_Cores>1</Synchron_Cores>
    <Common_Clock>1</Common_Clock>
    <Frequencys>
      <Frequency>1000</Frequency>
    </Frequencys>
    <Peripherals>
      <Peripheral_Name>serial0</Peripheral_Name>
      <Peripheral_Interrupt_Name>
      </Peripheral_Interrupt_Name>
      <Is_Interrupt_Needed>0</Is_Interrupt_Needed>
      <Allowed_Interrupt_Cores>
        <AC>0</AC>
        <AC>1</AC>
      </Allowed_Interrupt_Cores>
    </Peripherals>
  </MC_Configuration>
  <Configuration>
    <ID>1</ID>
    <Core>
      <Core_ID>0</Core_ID>
      <Sched_Class>Cyclic</Sched_Class>
      <Use_Only_Cyc>1</Use_Only_Cyc>
      <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
      <Cyclic_Major>35</Cyclic_Major>
      <Cyclic_Minor>1</Cyclic_Minor>
      <Frequency>1000</Frequency>
      <Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
      <Tasks_on_Core>
        <Task_Name>HAMS_SLS</Task_Name>
        <Task_ID>6</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>6</Leading_Task_ID>
```

```
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>SLS_Norm</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>0</Cyclic_Start>
    <Cyclic_End>0</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>1</Cyclic_Start>
    <Cyclic_End>2</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P1</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>3</Cyclic_Start>
     <Cyclic_End>27</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Cooperative</Task_Name>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
```

```
    <Cycle_Times>
      <Cyclic_Start>28</Cyclic_Start>
      <Cyclic_End>34</Cyclic_End>
    </Cycle_Times>
    <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
  </Tasks_on_Core>
    </Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Cyclic</Sched_Class>
  <Use_Only_Cyc>1</Use_Only_Cyc>
  <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
  <Cyclic_Major>35</Cyclic_Major>
  <Cyclic_Minor>1</Cyclic_Minor>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Scheibe_Nass</Sub-Phase>
    <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
    <Cycle_Times>
      <Cyclic_Start>0</Cyclic_Start>
      <Cyclic_End>24</Cyclic_End>
    </Cycle_Times>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Cooperative</Task_Name>
    <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
    <Cycle_Times>
      <Cyclic_Start>25</Cyclic_Start>
      <Cyclic_End>34</Cyclic_End>
    </Cycle_Times>
    <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
```

```
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Parkgeschwindigkeit
  /Scheibe_Nass</Beschreibung>
</Configuration>
<Configuration>
  <ID>2</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Cyclic</Sched_Class>
    <Use_Only_Cyc>1</Use_Only_Cyc>
    <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
    <Cyclic_Major>35</Cyclic_Major>
    <Cyclic_Minor>1</Cyclic_Minor>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>5</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
      <Cycle_Times>
        <Cyclic_Start>0</Cyclic_Start>
        <Cyclic_End>0</Cyclic_End>
      </Cycle_Times>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>GRA</Task_Name>
      <Task_ID>0</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
      <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
```

```
  <Cycle_Times>
    <Cyclic_Start>1</Cyclic_Start>
    <Cyclic_End>2</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P1</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>Parkgeschwindigkeit</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>3</Cyclic_Start>
    <Cyclic_End>27</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Regensensor</Task_Name>
  <Task_ID>2</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>2</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Scheibe_Trocken</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>27</Cyclic_Start>
    <Cyclic_End>28</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Cooperative</Task_Name>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>29</Cyclic_Start>
    <Cyclic_End>34</Cyclic_End>
```

```
      </Cycle_Times>
      <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
    </Tasks_on_Core>
  </Core>
  <Core>
    <Core_ID>1</Core_ID>
    <Sched_Class>Cyclic</Sched_Class>
    <Use_Only_Cyc>1</Use_Only_Cyc>
    <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
    <Cyclic_Major>35</Cyclic_Major>
    <Cyclic_Minor>1</Cyclic_Minor>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>Cooperative</Task_Name>
      <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
      <Cycle_Times>
        <Cyclic_Start>0</Cyclic_Start>
        <Cyclic_End>34</Cyclic_End>
      </Cycle_Times>
      <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Parkgeschwindigkeit/
  Scheibe_Trocken</Beschreibung>
</Configuration>
<Configuration>
  <ID>3</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Cyclic</Sched_Class>
    <Use_Only_Cyc>1</Use_Only_Cyc>
    <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
    <Cyclic_Major>35</Cyclic_Major>
    <Cyclic_Minor>1</Cyclic_Minor>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>4</Tasks_Nr_on_Core>
```

```
<Tasks_on_Core>
  <Task_Name>HAMS_SLS</Task_Name>
  <Task_ID>6</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>6</Leading_Task_ID>
  <Active_Phase_Name>P1</Active_Phase_Name>
  <Sub-Phase>SLS_Norm</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>0</Cyclic_Start>
    <Cyclic_End>0</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>GRA</Task_Name>
  <Task_ID>0</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>1</Cyclic_Start>
    <Cyclic_End>25</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P2</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>26</Cyclic_Start>
    <Cyclic_End>27</Cyclic_End>
```

```
    </Cycle_Times>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Cooperative</Task_Name>
    <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
    <Cycle_Times>
      <Cyclic_Start>28</Cyclic_Start>
      <Cyclic_End>34</Cyclic_End>
    </Cycle_Times>
    <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Cyclic</Sched_Class>
  <Use_Only_Cyc>1</Use_Only_Cyc>
  <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
  <Cyclic_Major>35</Cyclic_Major>
  <Cyclic_Minor>1</Cyclic_Minor>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Regensensor</Task_Name>
    <Task_ID>2</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>2</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Scheibe_Nass</Sub-Phase>
    <Cycle_Times>
      <Cyclic_Start>0</Cyclic_Start>
      <Cyclic_End>24</Cyclic_End>
    </Cycle_Times>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Cooperative</Task_Name>
    <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
    <Cycle_Times>
```

```
        <Cyclic_Start>24</Cyclic_Start>
        <Cyclic_End>34</Cyclic_End>
      </Cycle_Times>
      <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Fahrtgeschwindigkeit/
  Scheibe_Nass</Beschreibung>
</Configuration>
<Configuration>
  <ID>4</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Cyclic</Sched_Class>
    <Use_Only_Cyc>1</Use_Only_Cyc>
    <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
    <Cyclic_Major>35</Cyclic_Major>
    <Cyclic_Minor>1</Cyclic_Minor>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>5</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>HAMS_SLS</Task_Name>
      <Task_ID>6</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>6</Leading_Task_ID>
      <Active_Phase_Name>P1</Active_Phase_Name>
      <Sub-Phase>SLS_Norm</Sub-Phase>
      <Cycle_Times>
        <Cyclic_Start>0</Cyclic_Start>
        <Cyclic_End>0</Cyclic_End>
      </Cycle_Times>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>GRA</Task_Name>
      <Task_ID>0</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
```

```
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <Cycle_Times>
    <Cyclic_Start>1</Cyclic_Start>
    <Cyclic_End>25</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Einparkhilfe</Task_Name>
  <Task_ID>101/0-P2</Task_ID>
  <Is_Leading_Task>0</Is_Leading_Task>
  <Leading_Task_ID>0</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Fahrtgeschwindigkeit</Sub-Phase>
  <Cycle_Times>
    <Cyclic_Start>26</Cyclic_Start>
    <Cyclic_End>27</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Regensensor</Task_Name>
  <Task_ID>2</Task_ID>
  <Is_Leading_Task>1</Is_Leading_Task>
  <Leading_Task_ID>2</Leading_Task_ID>
  <Active_Phase_Name>P2</Active_Phase_Name>
  <Sub-Phase>Scheibe_Trocken</Sub-Phase>
  <Cycle_Times>
    <Cyclic_Start>28</Cyclic_Start>
    <Cyclic_End>29</Cyclic_End>
  </Cycle_Times>
</Tasks_on_Core>
<Tasks_on_Core>
  <Task_Name>Cooperative</Task_Name>
  <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
  <Cycle_Times>
    <Cyclic_Start>30</Cyclic_Start>
    <Cyclic_End>34</Cyclic_End>
```

```
      </Cycle_Times>
      <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
    </Tasks_on_Core>
  </Core>
  <Core>
    <Core_ID>1</Core_ID>
    <Sched_Class>Cyclic</Sched_Class>
    <Use_Only_Cyc>1</Use_Only_Cyc>
    <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
    <Cyclic_Major>35</Cyclic_Major>
    <Cyclic_Minor>1</Cyclic_Minor>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>Cooperative</Task_Name>
      <Cyclic_Nr_In_Major>1</Cyclic_Nr_In_Major>
      <Cycle_Times>
        <Cyclic_Start>0</Cyclic_Start>
        <Cyclic_End>34</Cyclic_End>
      </Cycle_Times>
      <Tasks_Nr_in_Coop>0</Tasks_Nr_in_Coop>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SLS_Norm/Fahrtgeschwindigkeit/
  Scheibe_Trocken</Beschreibung>
</Configuration>
<Fail_Operation>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Cyclic</Sched_Class>
    <Use_Only_Cyc>1</Use_Only_Cyc>
    <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
    <Cyclic_Major>35</Cyclic_Major>
    <Cyclic_Minor>1</Cyclic_Minor>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
    <Tasks_on_Core>
```

```
        <Task_Name>HAMS_SLS</Task_Name>
        <Task_ID>6</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>6</Leading_Task_ID>
        <Active_Phase_Name>P1</Active_Phase_Name>
        <Sub-Phase>SLS_Norm</Sub-Phase>
        <Cycle_Times>
          <Cyclic_Start>0</Cyclic_Start>
          <Cyclic_End>0</Cyclic_End>
        </Cycle_Times>
      </Tasks_on_Core>
      <Tasks_on_Core>
        <Task_Name>Regensensor</Task_Name>
        <Task_ID>2</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>2</Leading_Task_ID>
        <Active_Phase_Name>P2</Active_Phase_Name>
        <Sub-Phase>Fail</Sub-Phase>
        <Cycle_Times>
          <Cyclic_Start>1</Cyclic_Start>
          <Cyclic_End>2</Cyclic_End>
        </Cycle_Times>
      </Tasks_on_Core>
    </Core>
    <Core>
      <Core_ID>1</Core_ID>
      <Sched_Class>Cyclic</Sched_Class>
      <Use_Only_Cyc>1</Use_Only_Cyc>
      <Cyclic_Tick_Time>1</Cyclic_Tick_Time>
      <Cyclic_Major>35</Cyclic_Major>
      <Cyclic_Minor>1</Cyclic_Minor>
      <Frequency>1000</Frequency>
      <Tasks_Nr_on_Core>0</Tasks_Nr_on_Core>
    </Core>
  </Fail_Operation>
</KB_result>
```

Listing 9.5: RMS Knowledgebase mit 7 Tasks

```
<?xml version="1.0" encoding="UTF-8"?>
<KB_result>
  <MC_Configuration>
    <Nr_Cores>2</Nr_Cores>
    <Synchron_Cores>1</Synchron_Cores>
    <Common_Clock>1</Common_Clock>
    <Frequencys>
      <Frequency>1000</Frequency>
    </Frequencys>
  </MC_Configuration>
  <Configuration>
    <ID>1</ID>
    <Core>
      <Core_ID>0</Core_ID>
      <Sched_Class>Pure RMS</Sched_Class>
      <Use_Only_RMS>1</Use_Only_RMS>
      <RMS_Tick>1</RMS_Tick>
      <Frequency>1000</Frequency>
      <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
      <Tasks_on_Core>
        <Task_Name>LaunchControl</Task_Name>
        <Task_ID>103/0-P1</Task_ID>
        <Is_Leading_Task>0</Is_Leading_Task>
        <Leading_Task_ID>0</Leading_Task_ID>
        <Active_Phase_Name>P1</Active_Phase_Name>
        <Sub-Phase>Parken</Sub-Phase>
        <RMS_Prio>0</RMS_Prio>
        <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
        <RMS_Period_Task>35</RMS_Period_Task>
        <RMS_Deadline>35</RMS_Deadline>
      </Tasks_on_Core>
      <Tasks_on_Core>
        <Task_Name>Einparkassistent</Task_Name>
        <Task_ID>0</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>0</Leading_Task_ID>
```

```
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Parken</Sub-Phase>
    <RMS_Prio>1</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>1</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>Berganfahrassistent</Task_Name>
    <Task_ID>101/0-P1</Task_ID>
    <Is_Leading_Task>0</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Parken</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Abbiegeassistent</Task_Name>
    <Task_ID>102/0-P1</Task_ID>
    <Is_Leading_Task>0</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P1</Active_Phase_Name>
    <Sub-Phase>Parken</Sub-Phase>
    <RMS_Prio>1</RMS_Prio>
    <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
```

```
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>Parken/</Beschreibung>
</Configuration>
<Configuration>
  <ID>2</ID>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>Abbiegeassistent</Task_Name>
      <Task_ID>102/0-P2</Task_ID>
      <Is_Leading_Task>0</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P2</Active_Phase_Name>
      <Sub-Phase>LangsamFahrt</Sub-Phase>
      <RMS_Prio>0</RMS_Prio>
      <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>LaunchControl</Task_Name>
      <Task_ID>103/0-P2</Task_ID>
      <Is_Leading_Task>0</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P2</Active_Phase_Name>
      <Sub-Phase>LangsamFahrt</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
      <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
```

```
    </Tasks_on_Core>
  </Core>
  <Core>
    <Core_ID>1</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>Einparkassistent</Task_Name>
      <Task_ID>0</Task_ID>
      <Is_Leading_Task>1</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P2</Active_Phase_Name>
      <Sub-Phase>LangsamFahrt</Sub-Phase>
      <RMS_Prio>0</RMS_Prio>
      <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Berganfahrassistent</Task_Name>
      <Task_ID>101/0-P2</Task_ID>
      <Is_Leading_Task>0</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P2</Active_Phase_Name>
      <Sub-Phase>LangsamFahrt</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
      <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>LangsamFahrt/</Beschreibung>
</Configuration>
<Configuration>
```

```
<ID>3</ID>
<Core>
  <Core_ID>0</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
  <RMS_Tick>1</RMS_Tick>
  <Frequency>1000</Frequency>
  <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
  <Tasks_on_Core>
    <Task_Name>LaunchControl</Task_Name>
    <Task_ID>103/0-P3</Task_ID>
    <Is_Leading_Task>0</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P3</Active_Phase_Name>
    <Sub-Phase>SchnellStart</Sub-Phase>
    <RMS_Prio>0</RMS_Prio>
    <RMS_Max_WCET_Task>25</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
  <Tasks_on_Core>
    <Task_Name>Einparkassistent</Task_Name>
    <Task_ID>0</Task_ID>
    <Is_Leading_Task>1</Is_Leading_Task>
    <Leading_Task_ID>0</Leading_Task_ID>
    <Active_Phase_Name>P3</Active_Phase_Name>
    <Sub-Phase>SchnellStart</Sub-Phase>
    <RMS_Prio>1</RMS_Prio>
    <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
    <RMS_Period_Task>35</RMS_Period_Task>
    <RMS_Deadline>35</RMS_Deadline>
  </Tasks_on_Core>
</Core>
<Core>
  <Core_ID>1</Core_ID>
  <Sched_Class>Pure RMS</Sched_Class>
  <Use_Only_RMS>1</Use_Only_RMS>
```

```
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
    <Tasks_Nr_on_Core>2</Tasks_Nr_on_Core>
    <Tasks_on_Core>
      <Task_Name>Berganfahrassistent</Task_Name>
      <Task_ID>101/0-P3</Task_ID>
      <Is_Leading_Task>0</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P3</Active_Phase_Name>
      <Sub-Phase>SchnellStart</Sub-Phase>
      <RMS_Prio>0</RMS_Prio>
      <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
    <Tasks_on_Core>
      <Task_Name>Abbiegeassistent</Task_Name>
      <Task_ID>102/0-P3</Task_ID>
      <Is_Leading_Task>0</Is_Leading_Task>
      <Leading_Task_ID>0</Leading_Task_ID>
      <Active_Phase_Name>P3</Active_Phase_Name>
      <Sub-Phase>SchnellStart</Sub-Phase>
      <RMS_Prio>1</RMS_Prio>
      <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
      <RMS_Period_Task>35</RMS_Period_Task>
      <RMS_Deadline>35</RMS_Deadline>
    </Tasks_on_Core>
  </Core>
  <Beschreibung>SchnellStart/</Beschreibung>
</Configuration>
<Fail_Operation>
  <Core>
    <Core_ID>0</Core_ID>
    <Sched_Class>Pure RMS</Sched_Class>
    <Use_Only_RMS>1</Use_Only_RMS>
    <RMS_Tick>1</RMS_Tick>
    <Frequency>1000</Frequency>
```

```
      <Tasks_Nr_on_Core>1</Tasks_Nr_on_Core>
      <Tasks_on_Core>
        <Task_Name>Einparkassistent</Task_Name>
        <Task_ID>0</Task_ID>
        <Is_Leading_Task>1</Is_Leading_Task>
        <Leading_Task_ID>0</Leading_Task_ID>
        <Active_Phase_Name>P3</Active_Phase_Name>
        <Sub-Phase>Fail</Sub-Phase>
        <RMS_Prio>1</RMS_Prio>
        <RMS_Max_WCET_Task>2</RMS_Max_WCET_Task>
        <RMS_Period_Task>35</RMS_Period_Task>
        <RMS_Deadline>35</RMS_Deadline>
      </Tasks_on_Core>
    </Core>
    <Core>
      <Core_ID>1</Core_ID>
      <Sched_Class>Pure RMS</Sched_Class>
      <Use_Only_RMS>1</Use_Only_RMS>
      <RMS_Tick>1</RMS_Tick>
      <Frequency>1000</Frequency>
      <Tasks_Nr_on_Core>0</Tasks_Nr_on_Core>
    </Core>
  </Fail_Operation>
</KB_result>
```

Abbildung 9.1: Kernelshark der Kreiszahlerechnung von 800 Mhz zu 2500 MHz

Abbildung 9.2: Kernelshark der Kreiszahlerechnung von 2500 Mhz zu 800 MHz

Abbildung 9.3: Kernelshark der Kantenerkennung von 800 Mhz zu 2500 MHz

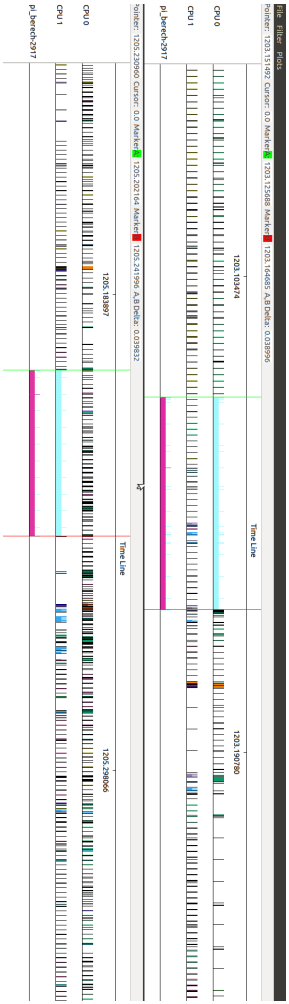



Abbildung 9.4: Kernelshark der Migration des Kreiszahlerechnungstasks

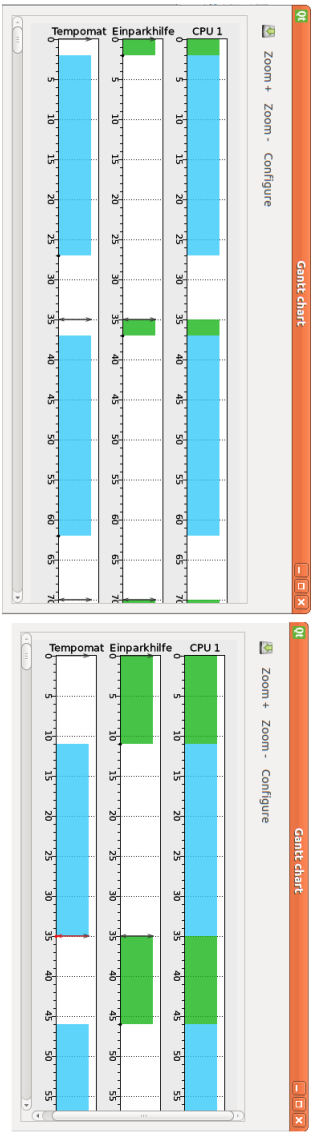



Abbildung 9.5: Gantt Charts bei Task Deadline miss ohne (links) und mit (rechts)

# Abkürzungsverzeichnis

| | |
|---|---|
| **AP** | (Available Peripherals) - Verfügbare Peripherie |
| **ACF** | (Available Core Frequencies) - Verfügbare Frequenzen des Cores |
| **ACP** | (Available Core Peripherals) - Verfügbare Peripherie des Cores |
| **AI** | (Available Core Interrupt) - Verfügbarer Interrupt des Cores |
| **AIDA** | Architecture for Independent Distributed Avionics |
| **ALU** | Arithmetic Logic Unit |
| **AMP** | Asymmetrischer Multi-Core Prozessor |
| **API** | Application Programming Interface |
| **ARAMiS** | **A**utomotive, **R**ailway and **A**vionics **M**ulti**i**core **S**ystems |
| **ARXML** | Autosar XML |
| **AUTOSAR** | **AUT**omotive **O**pen **S**ystems **AR**chitecture |
| **BIST** | (Build in Self Test) Selbstdiagnose |
| **BSW** | Basis Software |
| **CAN** | Controller Area Network |
| **CC-UMA** | Cache-Coherent Unified Memory Architecture |
| **CFS** | Completely Fair Scheduler |
| **DIANA** | **D**istributed, equipment **I**ndependent environment for **A**dvanced avio**N**ics **A**pplications |
| **CPU** | (Central Processing Unit) - Prozessor |
| **CTA** | Constant Time Admission |

| | |
|---|---|
| **CTM** | Core To Migrate to |
| **DREAMS** | **D**istributed **RE**al-time **A**rchitecture for **M**ixed Criticality **S**ystems |
| **DLR** | Deutsches Zentrum für Luft- und Raumfahrt |
| **DMA** | Direct Memory Access |
| **DMI** | Data Memory Interface |
| **DMS** | Deadline Monotonic Scheduling |
| **DSP** | Digital Signal Processor |
| **DVFS** | Dynamic Voltage and Frequency Scaling |
| **DT** | Depending Task - Abhängiger Task |
| **E/E** | Elektrik /Elektronik |
| **ECU** | (Elektronik Control Unit) - Steuergerät |
| **EDZL** | Earliest Deadline Zero Laxity |
| **EDF** | Earliest Deadline First |
| **EVA** | Eingabe Verarbeitung Ausgabe |
| **FIFO** | (First In -First Out) - der Reihe nach |
| **FLS** | First Level Scheduler |
| **FO** | Fail Operational |
| **GEDF** | Global Earliest Deadline First |
| **ggT** | größter gemeinsamer Teiler |
| **GPIO** | General-Purpose Input/Output |
| **GRA** | Geschwindigkeitsregelanlage |
| **GUI** | Graphical User Interface |
| **HAMS** | Hierarchical Asynchronous Multi-Core System |
| **HAPI** | HAMS API |
| **IMA** | Integrated Modular Avionics |
| **KB** | Knowledgebase |
| **kgV** | kleinstes gemeinsames Vielfaches |

| | |
|---|---|
| **KVM** | Kernel Based Virtual Machine |
| **LLF** | Least Laxity First |
| **LT** | (Leading Task) -Bestimmender Task |
| **LXC** | **L**inu**x** **C**ontainers |
| **MEL** | Minimum Equipment List |
| **MLLF** | Modified Least Laxity First |
| **NCP** | (Needed Core Peripherals) - Benötigte Peripherie des Cores |
| **NI** | (Needed Interrupt) - Benötigter Interrupt |
| **NM** | (Needed Memory) - Benötigter Speicher |
| **NP** | nichtdeterministisch polynomielle Zeit |
| **NPU** | Needed Peripheral Unit |
| **NUMA** | Non-Unified Memory Architecture |
| **OEM** | Original Equipment Manufacturer |
| **OIL** | OSEK Implementation Language |
| **OMAP** | Open Multimedia Application Platform |
| **OS** | (Operating System) - Betriebssystem |
| **OSEK** | **O**ffene **S**ystem und dessen Schnittstellen für die **E**lektronik im **K**raftfahrzeug |
| **OSEK/VDX** | OSEK/ Vehicle Distributed Executive |
| **PS** | Polling Server |
| **RAM** | Random-Access Memory |
| **RISC** | Reduced Instruction Set Computer |
| **RMS** | Rate Monotonic Scheduling |
| **RR** | Round Robin |
| **RTOS** | (Real Time Operating System) - Echtzeitbetriebssystem |
| **RTSA** | Real-Time Scheduling Assistant |

| | |
|---|---|
| **PMI** | Program Memory Interface |
| **PPC** | PowerPC |
| **SCARLETT** | **SCA**lable and **R**e- configurab**L**e **E**lectronics Pla**T**forms and **T**ools |
| **SimSO** | Simulation of Multiprocessor Scheduling with Overheads |
| **SLS** | Second Level Scheduler |
| **SMP** | Symetrische Multi-Core Porzessor |
| **SOC** | System-on-a-Chip |
| **TBS** | Total Bandwidth Server |
| **TDMA** | Time Division Multiple Access |
| **UAV** | Unmanned Aerial Vehicle |
| **UMA** | Unified Memory Architecture |
| **WCET** | Worst Case Execution Time |
| **XML** | (Extensible Markup Language) - Erweiterbare Auszeichnungssprache |

# Literaturverzeichnis


[1] ISO/TC 204. Intelligent transport systems – full speed range adaptive cruise control (fsra) systems – performance requirements and test procedures, September 2009.

[2] S. Ahn. Canny edge detection. `http://www.songho.ca/dsp/cannyedge/cannyedge.html`. [Online zugegriffen 10.Apr.2016].

[3] Timing Architects. Newsspeed up your automotive control unit - a synchronized tool chain for efficient multi-core development. `https://www.timing-architects.com/news/detail/speed-up-your-automotive.-control-unit-a-synchronized-tool-chain-for-/efficient-multi.-core-developme/`, October 2015. [Online zugegriffen 10.Apr.2016].

[4] Ingenieurbüro Barheine. Der Betriebssystemstandard OSEK/VDX fürs Automobil. `http://www.barheine.de/OSEK.pdf`, September 2005. [Online zugegriffen 10.Apr.2016].

[5] T. Baumann. Online-algorithmen online-packprobleme. `http://projects.laas.fr/simso/`. [Online zugegriffen 10.Apr.2016].

[6] J. Bernard. Ima1g - genesis and results. In *SCARLETT MOSCOW - 1st FORUM*, Oktober 2009.

[7] J. Bernard. Ima1g - genesis and results. In *SCARLETT*, September 2009.

[8] W. Bernhart. Automatisiertes Fahren - Evulution statt Revolution. *ATZ Extra*, 1, 2016.

[9] M. Breitinger. Gutachter sehen im Tesla-Autopiloten erhebliche Verkehrsgefährdung. `http://www.zeit.de/`

mobilitaet/2016-10/autopilot-tesla-model-s-gutachten_verkehrsministerium-zulassung, Oct 2016. [Online zugegriffen 10.Apr.2016].

[10] M. Broy. Software-Architekturen der Zukunft. *ATZ Agenda*, 1:46 – 49, November 2013.

[11] G. Buttazzo. *Hard real-time computing systems*. Springer International Publishing AG, 3 edition, 2011.

[12] Laboratoire d'Analyse et d'Architecture des Systèmes. Simso - simulation of multiprocessor scheduling with overheads. http://projects.laas.fr/simso/. [Online zugegriffen 10.Apr.2016].

[13] M.L. Dertouzos. Control robotics: The procedural control of physical processes. 1974.

[14] M. Devi. An improved schedulability test for uniprocessor periodic task systems. In *15th Euromicro Conference on Real-Time Systems*, 2003.

[15] R. Dodd. An analysis of task scheduling for a generic avionics mission computer. In Defence Science and Technology Organisation, editors, *Australian Government Department of Defence*, April 2006.

[16] C. Fielding. The design of fly-by-wire flight control systems. In *Aeronautical Journal -New Series*, 2000.

[17] Freescale Halbleiter Deutschland GmbH, Schatzbogen 7 81829 Muenchen Germany. *P4080 QorIQ Integrated Multicore Communication Processor Family - Reference Manual*, 1 edition, Januar 2012.

[18] BMW Group. Aktive Geschwindigkeitsregelung. http://www.bmw.com/com/de/insights/technology/technology_guide/articles/active_cruise_control.html, Juli 2016. [Online zugegriffen 10.Apr.2016].

[19] BMW Group. *BMW X4 Owner's Manual: Parking Assistant*. BMW Group, 2016.

[20] BMW Group. Driving Assistant Plus inkl. Lenk- und Spurführungsassistent. http://www.bmw.com/com/de/newvehicles/7series/sedan/2015/showroom/driver_assistance.html, Juli 2016. [Online zugegriffen 10.Apr.2016].

[21] BMW Group. Side view. http://www.bmw.de/de/footer/publications-links/technology-guide/side-view.html, Juli 2016. [Online zugegriffen 10.Apr.2016].

[22] F. Gruian. Hard real-time scheduling for low energy using stochastic data and dvs processors. In *International Symposium on Low-Power Electronics and Design ISLPED'01*, 2001.

[23] W. Haas. Osek/osektime os. Juli 2006.

[24] G. Höner. Preistreiber Abgasnormen. *Top Agrar*, 11:86–92, November 2013.

[25] Ambric Inc. Bric fpr multicore system-on-chip. https://i.cmpnet.com/eet/news/06/08/AMBRICS1437_PG_6.gif, August 2008. [Online zugegriffen 10.Apr.2016].

[26] Kalray Inc. http://www.kalrayinc.com. [Online zugegriffen 10.Apr.2016].

[27] Tesla Motors Inc. https://www.tesla.com/autopilot, Oct 2016. [Online zugegriffen 10.Apr.2016].

[28] Infineon Technologies AG, 81726 Munich Germany. *AURIX TC27x 32-Bit Single-Chip Microcontrollee Users's Manual*, 1.4 edition, November 2013.

[29] Intel. https://de.wikipedia.org/wiki/Datei:Intel_Core2_arch.svg. [Online zugegriffen 10.Apr.2016].

[30] Intel Corporation, 2200 Mission College Blvd. Santa Clara CA 95054-1549 U.S.A. *2nd Generation Intel® Core™ Processor Family Desktop, Intel® Pentium® Processor Family Desktop, and Intel® Celeron® Processor Family Desktop Datasheet, Volume 1*, 1.2 edition, Juni 2013.

[31] D. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[32] Aramis Konsortium. https://www.projekt-aramis.de. [Online zugegriffen 10.Apr.2016].

[33] AUTOSAR Konsortium. https://www.autosar.org/specifications/release-42/. [Online zugegriffen 10.Apr.2016].

[34] AUTOSAR Konsortium. Timing analysis - autosar release 4.2.2. https://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/methodology/auxiliary/AUTOSAR_TR_TimingAnalysis.pdf. [Online zugegriffen 10.Apr.2016].

[35] AUTOSAR Konsortium. Osek / vdx - system generation -oil: Osek implementation language. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/oil25.pdf, Juli 2004. [Online zugegriffen 10.Apr.2016].

[36] AUTOSAR Konsortium. Autosar the world automotive standart for e/e systems. *ATZ extra*, 4:6 – 12, Oktober 2013.

[37] I. Kuss. Audis zentrales Fahrerassistezsteuergerät zFAS. Den Überblick behalten. *Elektronik automotive*, 4:38–39, April 2016.

[38] LinuxContainers. https://linuxcontainers.org/. [Online zugegriffen 10.Apr.2016].

[39] Inc. Lynx Software Technologies. The world's most powerful, open-standards real-time os. http://www.lynx.com/pdf/LynxOSDatasheetFinal.pdf, 2015. [Online zugegriffen 10.Apr.2016].

[40] M. McFarland. Tesla drivers do 'crazy things' while autopilot is engaged, musk issues statement. https://www.washingtonpost.com/news/innovations/wp/2015/10/14/elon-musk-vents-about-californias-lane-markings-/confusing-teslas.-autopilot/, Oct 2015. [Online zugegriffen 10.Apr.2016].

[41] A. Mo. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[42] B. Moyer. *Real wordl multicore embedded systems*. Elsevier Inc., 1 edition, 2013.

[43] W. Nickel. Als der Citroen noch um die Ecke schaute. `https://www.welt.de/motor/article115024842/Als-der-Citroen-noch-um-die-Ecke-schaute.html`, May 2013. [Online zugegriffen 10.Apr.2016].

[44] DIANA Project White Paper. The aida system. In *DIANA Project*, 2008.

[45] J. Pikolin. Paperless cockpit lufthansa systems' mobile solutions. `http://www.caac.gov.cn/ZTZL/RDZT/XJSYY/201511/P020151126426629389439.pdf`, July 2012. [Online zugegriffen 10.Apr.2016].

[46] G. Pitcher. Cutting control complexity. *Newelectronics*, 9, 2012. [Online zugegriffen 10.Apr.2016].

[47] SCARLETT Project. `http://www.scarlettproject.eu/`. [Online zugegriffen 10.Apr.2016].

[48] J. W. Ramsey. Integrated modular avionics: Less is more. `http://www.aviationtoday.com/av/commercial/Integrated-Modular-Avionics-Less-is-More_8420.html`, Feb 2007. [Online zugegriffen 10.Apr.2016].

[49] Renesas. R-car h2. `https://www.renesas.com/en-us/solutions/automotive/products/rcar-h2.html`. [Online zugegriffen 10.Apr.2016].

[50] Airbus S.A.S. `http://www.airbus.com/innovation/proven-concepts/in-operations/brake-to-vacate/`, Oct 2016. [Online zugegriffen 10.Apr.2016].

[51] T. Schaub. Osek/vdx. Master's thesis, Universität Koblenz-Landau, 1999.

[52] O. Schied. *AUTOSAR compendium part 1 application and RTE*. AR-Compendium, 1.0.1 edition, 2015.

[53] J. Schwarz. Neue Marktsegmente erschließen. *ATZ Elektronik*, 4, 2016.

[54] P. Shelly. Stratgien zur besseren Automobilvernetzung. *ATZ Elektronik*, 6:413 – 416, Juni 2013.

[55] A. Shimpi. Barcelona architecture: Amd on the counterattack. `http://www.anandtech.com/show/2183/9`, März 2007. [Online zugegriffen 10.Apr.2016].

[56] Tesla Team. Upgrading autopilot: Seeing the world in radar. `https://www.tesla.com/blog/upgrading-autopilot-seeing-world-radar`, Nov 2016. [Online zugegriffen 10.Apr.2016].

[57] Wikipedia Team. Canny algorithmus. `https://de.wikipedia.org/wiki/Canny-Algorithmus`. [Online zugegriffen 10.Apr.2016].

[58] Wikipedia Team. Leibniz-reihe. `https://de.wikipedia.org/wiki/Leibniz-Reihe`. [Online zugegriffen 10.Apr.2016].

[59] Wikipedia Autoren Team. Brake to Vacate. `https://en.wikipedia.org/wiki/Brake_to_Vacate`, Juli 2016. [Online zugegriffen 10.Apr.2016].

[60] Texas Instruments Inc., Post Office Box 655303 Dallas Texas 75265 USA. *OMAP4430 Multimedia Device Silicon Revision 2.x - Technical Reference Manual*, an edition, April 2010.

[61] Neil C. Audsley und A. Burns und M. F. Richardson und A. J. Wellings. Deadline monotonic scheduling. 1990.

[62] M. Rodrigo und A. Coutinho. Aspects on architecture for independent distributed avionics (aida). In *IEEE/AIAA 27th Digital Avionics Systems Conference*, Oktober 2008.

[63] A. McLain und A. Eastaugh und D. Knight. Transforming the paperless cockpit into the connected aircraft. *Ascend Solutions*, 4, 2013.

[64] S. Byhlin und A. Ermedahl und J. Gustafsson und B. Lisper. Applying static wcet analysis to automotive communication software. In *Proceedings - Euromicro Conference on Real-Time Systems*, 2005.

[65] T. Hanti und A. Frey und W. Hardt. Reconfigurable multi-core scheduling for real-time functions in avionic mission systems. In IEEE/AIAA, editor, *34th Digital Avionics Systems Conference*, September 2015.

[66] D. Niedermeier und A. Lambregts. Fly-by-wire augmented manual control - basic design considerations. In ICAS, editor, *28 International Congress of the aeronautical sciences*, September 2012.

[67] S. Baruah und A. Mok und L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. In *Real-Time Systems*, 1990.

[68] S. Baruah und A. Mok und L. Rosier. Preemptively schedulinghard-real-time sporadic tasks on one processor. In *11th IEEE Real-Time Systems Symposium*, 1990.

[69] J. Lee und A. Tim und J. Yen. A fuzzy rule-based approach to real-time scheduling. In *Third IEEE Conference on Fuzzy Systems*, pages 1349 – 1399. IEEE, Juni 1994.

[70] J. How und C. Fraser. Increasing autonomy of uavs. *IEEE Robotics and Automation Magazine*, 16, Juni 2009.

[71] F. Kluge und C. Yu und J. Mische. Implementing autosar scheduling and resource management on an embedded smt processor. In ACM, editor, *Workshop on Software and Compilers for Embedded Systems*, April 2009.

[72] R. Müller und D. Danner. Multi sloth: An efficient multi-core rtos using hardware-based scheduling. In *26th Euromicro Conference on Real-Time Systems*, 2014.

[73] T. Weidner und D. Stück und S. Wender und R. Katzwinkel. Skalierbare E/E Architekturen für Fahrerassistenzfunktionen. *ATZ Elektronik*, 6:428 – 432, Juni 2013.

[74] R. Goettge und E. Brehm und C. Palczak und J. Stankovic und M. Humphrey. Knowledge-based assistance for real-time systems. In *First IEEE International Conference on Engineering of Complex Computer Systems*, pages 223 – 226. IEEE, November 1995.

[75] C. Engel und E. Jenn und P. Schmitt und R. Coutinho und T. Schoofs. Enhanced dispatchability of aircrafts using multistatic configurations. In ERTS, editor, *Embedded Real Time Software and Systems*, July 2010.

[76] P. Bieber und E. Noulard. Preliminary design of future reconfigurable ima platforms. In ACM Sigbed, editor, *2nd International Workshop on Adaptive and Reconfigurable Embedded Systems*, Oktober 2009.

[77] T. Heo und F. Mickler. Concurrency managed workqueue. `https://www.kernel.org/doc/Documentation/workqueue.txt`, 2010. [Online zugegriffen 10.Apr.2016].

[78] D. Lange und Felix Domke. The exhaust emissions scandal („dieselgate“). `https://media.ccc.de/v/32c3-7331-the_exhaust_emissions_scandal_dieselgate`, 2015. [Online zugegriffen 10.Apr.2016].

[79] K. Albers und F.Slomka. An event stream driven approximation for the analysis of real-time systems. In *16th Euromicro Conference on Real-Time Systems*, 2004.

[80] K. Albers und F.Slomka. Effcient feasibility analysis for real-time systems with edf scheduling. In *Design, Automation and Test in Europe*, 2005.

[81] E. Bini und G. C. Buttazzo. The space of rate monotonic schedulability. In *IEEE 23rd Real-Time Systems Symposium*, 2002.

[82] E. Bini und G. C. Buttazzo und G. M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, (52):933–942, 2003.

[83] J. Lelli und G. Lipari. An effcient and scalable implementation of global edf in linux. In *Operating Systems Platforms for Embedded Real-Time Applications*, 2011.

[84] M. Yu und G. Ma. 360° surround view system with parking guidance. In *SAE International Journal of Commercial Vehicles 7*, page 6. SAE, Mai 2014.

[85] N. Dewan und H. Vohra. Test scheduling of core based soc usinggreedy algorithm. 9, 2014.

[86] D. Hardy und I. Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In rtns, editor, *Real-Time Networks and Systems*, Februar 2009.

[87] M. Mollison und J. Erickson und J. Anderson und S. Baruah und J. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, pages 1864–1871, November 2010.

[88] S. Baruah und J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52:966–970, Juli 2003.

[89] C. L. Liu und J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, (20):46–61, 1973.

[90] J. Y. T. Leung und J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, Dezember 1982.

[91] J. Li und K. Agrawal. Analysis of global edf for parallel tasks. In *25th Euromicro Conference on Real-Time Systems*, Juli 2013.

[92] P. Pillai und K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, Oktober 2001.

[93] J. Lehoczky und L. Sha und Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. *IEEE Real-Time Systems Symposium*, page 166–171, 1989.

[94] P. Petrakis und M. Abuteir. On-chip networks for mixed-criticality systems. In IEEE, editor, *IEEE International Conference on Application-specific Systems, Architectures and Processors*, Juli 2016.

[95] T. Hanti und M. Ernst und A. Frey. Higher utilization of multicore processors in dynamic real-time software systems. In JEE, editor, *International Journal of Electrical Engergy*, April 2013.

[96] T. Hanti und M. Ernst und A. Frey. Phasenanalyse von funktionen in hierarchischen asynchronen multi-core system. In DLRK, editor, *Deutscher Luft- und Raumfarhtkongress*, Oktober 2014.

[97] E. Coffman und M. Garey und D. Johnson. *Approximation algorithms for bin packing: a survey.* PWS Publishing Co., 1 edition, 1997.

[98] R. Johnson und M. Loeas. Os consolidation: The next step in hypervisors. *Military embedded systems*, 6:15, Juni 2013.

[99] O. Scheickl und M. Rudorfer. Automotive real time development using a timing-augmented autosar specification.

[100] D. Gunnarsson und M. Traub und C. Pigorsch. Timing Bewertung in der E/E Architekturentwicklung. *ATZ Elektronik*, 1:36 – 41, Januar 2015.

[101] J. Liu und M. Yang. Task scheduling of real-time systems on multicore embedded processor. In *Intelligent Systems and Knowledge Engineering*, 2010.

[102] J. Liu und M. Yang. Task scheduling of real-time systems on multicore embedded processor. In *IEEE Intelligent Systems and Knowledge Engineering*, 2010.

[103] A. Monot und N. Navet und B. Bavoux und F. Simonot-Lion. Multi-source software on multicore automotive ecus - combining runnable sequencing with task scheduling. In IEEE, editor, *IEEE Transactions on Industrial Electronics*, Juli 2010.

[104] S. Kato und N. Yamasak. Semi-partitioned fixed-priority schedulingon multiprocessors. In *Intelligent Systems and Knowledge Engineering*, 2008.

[105] D. B. Stewart und P. K. Khosla. Real-time scheduling of dynamically reconfigurable systems. In *IEEE International Conference on Systems Engineering*, 1991.

[106] A.Spyridakis und P. Lalov und D. Raho. Dual-os infrastructure for mixed-criticality systems on arm devices. In ADVCOMP, editor, *International Conference on Advanced Engineering Computing and Applications in Sciences*, July 2015.

[107] G. Weiß und P. Schleiß und C. Drabek. Ausfallsichere E/E Architektur für hochautomatisierte Fahrfunktionen. *ATZ Elektronik*, 3:17 – 21, Juni 2016.

[108] A. Halilovic und P. Zimmerschitt-Halbig. Best Practices bei der Umsetzung ider ISO 26262 für die Software elektronischer Bremssysteme. *ATZ Elektronik*, 5:47 – 50, Mai 2014.

[109] H. Aydin und Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *IEEE Parallel and Distributed Processing Symposium*, 2003.

[110] S. Lauzac und R. Melhem und D. Mosse. An efficient rms admission control and its application to multiprocessor scheduling. In *IN INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM*, 2000.

[111] N. Englisch und R. Mittag und F. Hänchen und F. Ullmann und A. Masrur und W. Hardt. Application-driven evaluation of autosar basic software on modern ecus. 2015.

[112] N. Englisch und R. Mittag und F. Hänchen und O. Kahn und A. Masrur und W. Hardt. Efficiently testing autosar software based on an automatically generated knowledge base. 2016.

[113] H. Ahmadian und R. Obermaisser. Time-triggered extension layer for on-chip network interfaces in mixed-criticality systems. In IEEE, editor, *Euromicro Conference on Digital System Design*, August 2015.

[114] K. Lakshmanan und R. Rajkuma. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[115] W. Zimmerman und R. Schmidgall. *Bussysteme in der Fahrzeugtechnik*. Vieweg Teubner, 4 edition, 2011.

[116] W. Schwitzer und R. Schneider und D. Reinhardt. Tackling the complexity of timing-relevant deployment decisions in multicore-based embedded automotive software systems. In *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, pages 478 – 488, Juni 2013.

[117] A. Burns und R.I. Davis. Partitioned edf scheduling for multiprocessors using a c=d scheme. *The name of the journal*, 48(1):3–33, Januar 2010.

[118] B. Andersson und S. Baruah und J. Jansson. Static-priority scheduling on multiprocessor. In *22nd IEEE Real-Time Systems Symposium*, page 93. IEEE, Dezember 2001.

[119] A. Masrur und S. Chakraborty und G. Farber. Constant-time admission control for partitioned edf. In *22nd Euromicro Conference on Real-Time Systems*, 2010.

[120] J.Bin und S. Girbaly und D. Perezy. Studying co-running avionic real-time applicationson multi-core cots architectures. In ERTS, editor, *Embedded Real Time Software and Systems*, Februar 2014.

[121] X. Piao und S. Han. Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, April 2006.

[122] R. Grave und S. Krämer. Software optimal auf mehrere Kerne verteilen. *Elektronik automotive*, 11, 2014.

[123] S. Chakraborty und S. Künzli und L. Thiele. Approximate schedulability analysis. In *23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[124] S. Samii und S. Rafiliu und P. Eles und Z. Peng. A simulation methodology for worst-case response time estimation of distributed real-time system. In *Design, Automation and Test in Europe, 2008. DATE '08*, 2008.

[125] K. Flautner und S. Reinhardt und T. Muge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, Jul 2001.

[126] M. Sebastian und S. Schliecker. E/e and software development process for timing-aware design. *ATZ extra*, 4:96 – 99, Oktober 2013.

[127] S. Anssi und S. Tucci-Piergiovanni. Enabling scheduling analysis for autosar systems. In ISORC, editor, *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Januar 2011.

[128] D. Adam und S. Tverdyshev. Design recommendations to mitigate memory and cache non-determinisms in multi-core based ima platform of airborne systems. In IEEE, editor, *Digital Avionics Systems Conference*, September 2015.

[129] D. Adam und S. Tverdyshev. Two architecture approaches for mils systems in mobility domains. In HIPEAC, editor, *European Network on High Performance and Embedded Architecture and Compilation*, Januar 2015.

[130] S. Oh und S. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 31–36, Oct 1998.

[131] M. Ernst und T. Meier und A. Frey. Phasenmanagment eines hirarchischen asynchronen multicore schedulers. In DLRK, editor, *Deutscher Luft- und Raumfarhtkongress*, Oktober 2016.

[132] F. Wohlgemuth und T. Ringler. AUTOSAR im Entwicklungsprozess. *dSPACE-Magazin*, 2, 2008.

[133] T. Burd und W. Brodersen. Energy efficient cmos microprocessor design. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 288–297, Jan 1995.

[134] H. Jeon und W. H. Lee und S. W. Chung. A method for evaluating uncertainties in the early development phases of embedded real-time systems. In *Embedded and Real-Time Computing Systems and Applications*, November 2005.

[135] H. Jeon und W. H. Lee und S. W. Chung. Load unbalancing strategy for multicore embedded processors. In *IEEE Transactions on Computers, vol. 59*, pages 1434–1440, Oct 2010.

[136] L. Li und Z. Luo. Global edf scheduling for parallel real-time tasks. *Real-Time Systems*, 51:395–439, Juli 2014.

[137] LA. Vogt. Mit der Kraft der 1000 Herzen. Continental setzt bei der Steuerung von Verbrennungsmotoren künftig auf Multi-Core Systeme - diese Sparen Platz und Geld. *Automobilwoche*, (16):18, 2015.

[138] D. Weber. Distributed real-time architecture for mixed criticality systems. `http://www.dreams-project.eu/`, Oktober 2013. [Online zugegriffen 10.Apr.2016].

[139] Wind River Systems, Inc., 500 Wind River WayAlameda CA 94501-1153 U.S.A. *Wind River Hypervisor USER'S GUIDE 1.2*, 1.2 edition, August 2010.

[140] Wind River Systems, Inc., 500 Wind River WayAlameda CA 94501-1153 U.S.A. *VXWORKS APPLICATION PROGRAMMER'S GUIDE 6.9*, 11 edition, Januar 2015.

[141] Wind River Systems, Inc., 500 Wind River WayAlameda CA 94501-1153 U.S.A. *Vxworks 653 PROGRAMMER'S GUIDE 3.1*, 3.1 edition, März 2016.

[142] W. Wu. Embedded oder AUTOSAR - know-how für die Kommunikation von Steuergeräten. *ATZ elektronik*, 5:42 – 45, September 2014.

[143] D. Zöbel. *Echtzeitsysteme: Grundlagen der Planung*. Springer International Publishing AG, 1 edition, 2008.

## Eingebettete, selbstorganisierende Systeme im Universitätsverlag Chemnitz

8. Meisel, André (2010)
Design Flow für IP basierte, dynamisch rekonfigurierbare, eingebettete Systeme
ISBN 978-3-941003-15-6
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-201000890

9. Vodel, Matthias (2010)
Funkstandardübergreifende Kommunikation in Mobilen Ad Hoc Netzwerken
ISBN 978-3-941003-18-7
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-201001164

10. Reichelt, Toni (2012)
A Model Driven Approach for Service Based System Design Using Interaction Templates
ISBN 978-3-941003-60-6
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-85986

11. Caspar, Mirko (2013)
Lastgetriebene Validierung Dienstbereitstellender Systeme
ISBN 978-3-941003-84-2
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-110257

12. Heller, Ariane (2013)
Systemeigenschaft Robustheit: ein Ansatz zur Bewertung und Maximierung von Robustheit eingebetteter Systeme
ISBN 978-3-941003-87-3
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-113369

13. Vodel, Matthias (2014)
Energieeffiziente Kommunikation in verteilten, eingebetteten Systemen
ISBN 978-3-944640-05-1
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-131891

14. Tudevdagva, Uranchimeg (2014)
Structure Oriented Evaluation Model for E-Learning
ISBN 978-3-944640-20-4
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-146901

15. Gruber, Thomas (2016)
Prozessintegrierte Dokumentation und optimierte Wiederverwendung von Simulationsmodellen der automobilen Funktionsabsicherung
ISBN 978-3-944640-89-1
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-202845

16. Deicke, Markus (2018)
Virtuelle Absicherung von Steuergeräte-Software mit hardwareabhängigen Komponenten
ISBN 978-3-96100-039-5
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-230123

17. Hanti, Thomas (2019)
Knowledgebase basiertes Scheduling fürhierarchisch asynchrone Multi-CoreScheduler im Systembereich Automotive und Avionik
ISBN 978-3-96100-077-7
Volltext: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa2-326409