HAW HAMBURG

Masterarbeit

Hannah Kirstein

Entwicklung und Leistungsanalyse einer Betriebssystem-Abstraktion für eingebettete symmetrische Multiprozessorsysteme in der Raumfahrt

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

Hannah Kirstein

Entwicklung und Leistungsanalyse einer Betriebssystem-Abstraktion für eingebettete symmetrische Multiprozessorsysteme in der Raumfahrt

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Automatisierung*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Paweł Buczek
Zweitgutachter: M.Sc. Jan-Gerd Meß

Eingereicht am: 27. April 2022

Hannah Kirstein

Thema der Arbeit

Entwicklung und Leistungsanalyse einer Betriebssystem-Abstraktion für eingebettete symmetrische Multiprozessorsysteme in der Raumfahrt

Stichworte

Echtzeitbetriebssystem, Multiprozessorsystem, RTEMS, LEON3, GR712RC, Raumfahrt

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Erweiterung einer Betriebssystem-Abstraktion des Echtzeitbetriebssystems RTEMS in der Softwareplattform OUTPOST um die Unterstützung von symmetrischen Multiprozessorsystemen. In einer Leistungsanalyse der Erweiterung werden anhand eines Benchmarks auf dem SoC GR712RC in verschiedenen Szenarien die Steigerung des Durchsatzes bzw. das Senken der Auslastung des einzelnen Prozessors durch den Einsatz von zwei Prozessoren belegt.

Hannah Kirstein

Title of Thesis

Development and performance analysis of an operating system abstraction for embedded symmetric multiprocessor systems in space flight

Keywords

real-time operating system, multiprocessor system, RTEMS, LEON3, GR712RC, space flight

Abstract

This work deals with the extension of an operating system abstraction of the real-time operating system RTEMS in the software platform OUTPOST to support symmetric multiprocessor systems. In a performance analysis of the extension, using a benchmark on the SoC GR712RC in various scenarios, the increase in throughput and the reduction in the load on the individual processors are proven through the use of two processors.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mir die Anfertigung dieser Arbeit ermöglicht und mich auf dem Weg unterstützt haben.

Allen voran danke ich Prof. Dr. Paweł Buczek für die fachliche, finanzielle und materielle Unterstützung sowie das Vertrauen in mich während dieser Arbeit und aller vorangegangenen Projekte.

Jan-Gerd Meß danke ich für die Einführung in die Welt der Raumfahrttechnik, die spannende Themenstellung, die Betreuung und die fachliche Unterstützung während dieser Arbeit.

Ich danke dem DLR in Bremen, das mir die Arbeit in diesem Bereich ermöglicht hat.

Bei dem Team von embedded brains möchte ich mich für den interessanten Austausch bedanken, der mir weitere Denkanstöße für diese Arbeit geliefert hat.

Zu guter Letzt bedanke ich mich bei meinen Freunden, meiner Familie und insbesondere meiner Mutter für jegliche Unterstützung und den seelischen Beistand.

Inhaltsverzeichnis

Abbildungsverzeichnis viii

Tabellenverzeichnis ix

Abkürzungen x

1 Einleitung 1

1.1 Motivation 2
1.2 Zielsetzung 3
1.3 Aufbau der Arbeit 4

2 Theoretische Grundlagen 5

2.1 Anforderungen an Raumfahrtsysteme 5
2.2 OUTPOST 6
2.3 Eingebettete Systeme 7
2.3.1 GR712RC 8
2.4 Echtzeitbetriebssysteme 9
2.4.1 RTEMS 10
2.5 Multiprozessorsysteme 11
2.5.1 Multiprozessorsystem-Architekturen 11
2.5.2 Leistungssteigerung durch Multiprozessorsysteme 12
2.5.3 Cache-Speicher 14
2.5.4 Unterschiede zwischen Software für Uni- und Multiprozessorsysteme 17

3 Erweiterung von OUTPOST um RTEMS SMP 21

3.1 Anforderungen 21
3.2 RTEMS SMP Analyse 22
3.2.1 Scheduler 22
3.2.2 Zuweisen von Tasks zu Prozessoren 23
3.2.3 Schutz kritischer Sektionen in RTEMS SMP 23

3.3 Implementierung 25
3.3.1 Zuweisung von Tasks zu Prozessoren 25
3.3.2 Weitere Anpassungen 28

4 Benchmark-Applikation zur Leistungsanalyse 29
4.1 Definition der Leistungskriterien 29
4.2 Anforderungen 30
4.3 Hard- und Softwareumgebung 30
4.4 Methoden zur Leistungsanalyse 32
4.4.1 Durchsatz 32
4.4.2 Auslastung 32
4.5 Softwarearchitektur 34
4.5.1 Dateisystem 34
4.5.2 Build-Prozess Konfiguration 35
4.5.3 RTEMS Systemkonfiguration 35
4.5.4 Verhalten der Software und Klassen 36
4.6 Testszenarien 41

5 Messungen und Leistungsanalyse 43
5.1 Prognose der Messergebnisse 43
5.2 Darstellung und Gemeinsamkeiten der Messergebnisse 47
5.3 Messung des Durchsatzes 49
5.4 Messung der Auslastung 51
5.4.1 MemoryScrubber 51
5.4.2 CPU Usage Statistics 55
5.4.3 RTEMS Tracing Framework 57
5.4.4 Vergleich und Bewertung der Methoden zur Messung der Auslastung 60

6 Fazit 62
6.1 Ausblick 64

Literaturverzeichnis 65

A Vor- und Nachteile der Messwerkzeuge 67

B Software-Versionen 68

C Digitaler Anhang 69

Selbstständigkeitserklärung **70**

Abbildungsverzeichnis

2.1 OUTPOST im Gesamtsystem [3] 7
2.2 Blockdiagramm des GR712RC [10] 8
2.3 Speedup nach Amdahl und Gustafson 14
2.4 Globales, Clustered und Partitioniertes Scheduling 18

3.1 Auswertung der Affinität in dem Konstruktor des OUTPOST-Threads . . 26
3.2 Vererbung der Thread-Affinität 27
3.3 Affinitätsvererbung in der run-Methode des OUTPOST-Threads 27

4.1 Software- und Hardwareumgebung [4] 31
4.2 Ordnerstruktur des Benchmarks 34
4.3 Vereinfachtes Sequenzdiagramm des Benchmarks 37
4.4 Affinitäten der Threads in den Testszenarien 41

5.1 Zeitverhalten der Szenarien 46
5.2 Darstellung Durchsatz 48
5.3 Darstellung Auslastung 48
5.4 Durchsatz der Szenarien 49
5.5 Auslastung der CPU0, ermittelt durch den MemoryScrubber 52
5.6 Durchsatz mit und ohne MemoryScrubber, SleepTime = 500μs 54
5.7 Ausgabe der RTEMS CPU Usage Statistics (Szenario 2, SleepTime = $500\mu s$) 55
5.8 Auslastung der CPU0, ermittelt durch CPU Usage Statistics 57
5.9 Auswertung des Event Recordings im Eclipse Trace Compass (Szenario 1, SleepTime = 0) 58
5.10 Auslastung der CPU0, ermittelt durch das Event Recording 59
5.11 Vergleich der Methoden zum Messen der Auslastung, SleepTime = 500μs . 60

Tabellenverzeichnis

3.1 Anforderungen an die SMP-Unterstützung in OUTPOST 21

4.1 Anforderungen an die Benchmark-Applikation 30

A.1 Vor- und Nachteile der Methoden zur Messung der Auslastung 67

Abkürzungen

API Application Programming Interface.

ASMP Asymmetric Multiprocessing.

BSP Board Support Package.

DCT DataCollectingThread.

DLR Deutsches Zentrum für Luft- und Raumfahrt.

DPT DataProcessingThread.

DRT DataReceivingThread.

EDF Earliest Deadline First.

FDT FlightsoftwareDummyThread.

FIFO First In - First Out.

HAL Hardware Abstraction Layer.

LFU Least Frequently Used.

LRU Least Recently Used.

MrsP Multiprocessor Resource Sharing Protocol.

MSCR MemoryScrubber.

OBC OccupyCpuThread.

OCT OccupyCpuThread.

OMIP O(m) Independence-Preserving Protocol.

OUTPOST Open Modular Software Platform for Spacecraft.

RTEMS Real-Time Executive for Multiprocessor Systems.

SEL Single Event Upset.

SEU Single Event Latch-Up.

SMP Symmetric Multiprocessing.

SoC System-on-a-Chip.

TT TestingThread.

1 Einleitung

Raumfahrtsysteme unterliegen in Zeiten wachsender Digitalisierung und stetigen Forschungs- und Fortschrittsdrangs steigenden Ansprüchen. Durch das wachsende Pensum an Aufgaben wird das Herzstück eines Raumfahrtsystems, der On-Board Computer (OBC), immer stärker belastet und die Anforderungen an die Rechenleistung steigen.

Früher lag das Hauptaugenmerk zur Steigerung der Rechenleistung von Systemen auf der Leistungssteigerung des einzelnen Prozessors. Diese Methode wird jedoch durch einen damit einhergehenden steigenden Stromverbrauch und höhere Verluste begrenzt.[16] Heute bildet der Einsatz mehrerer Prozessoren in Form von Multiprozessorsystemen und die zu deren Verwaltung nötige Software eine zukunftsträchtige Herausforderung der Raumfahrt.

Der Einsatz von Systemen mit mehreren Prozessoren oder Prozessorkernen gehört für die Nutzer moderner Kommunikations- und Unterhaltungselektronik bereits zum Standard. In der Raumfahrt entstehen durch die besondere Umgebung jedoch missions- oder sicherheitskritische Anforderungen an die Systeme, die den Einsatz von Echtzeitbetriebssystemen erfordern. Die Verwaltung einer echtzeitfähigen Software stellt auf einem Prozessor bereits eine Herausforderung hinsichtlich der Planung des zeitlichen Ablaufs dar. Durch den Einsatz mehrerer Prozessoren steigt die Komplexität aufgrund der zusätzlichen Dimension, die es zu berücksichtigen gilt. Software, die für die Anwendung auf einem Prozessor entwickelt wurde, kann nicht ohne weiteres auf die Prozessoren eines Multiprozessorsystems aufgeteilt werden. Hardware, Betriebssystem und Flugsoftware müssen für die Multiprozessoranwendung ausgelegt sein.

Im Rahmen dieser Arbeit wird eine zwischen Betriebssystem und Flugsoftware liegende Softwareplattform an die Anwendung auf eingebetteten symmetrischen Multiprozessorsystemen angepasst und eine Leistungsanalyse durchgeführt.

1.1 Motivation

Die Rechenleistung ist durch hohe Datenverarbeitungsraten, die Ausführung komplexer Regelungsaufgaben und platz- und gewichtstechnische Einschränkungen der eingebetteten Systeme eine knappe Ressource von Raumfahrtsystemen. Eine Herausforderung bei der Entwicklung solcher Systeme ist demnach die optimale Nutzung der zur Verfügung stehenden Rechenleistung.

Am Deutschen Zentrum für Luft- und Raumfahrt (DLR) in Bremen am Institut für Raumfahrtsysteme werden zukünftige Raumfahrtsysteme und Raumfahrtmissionen entworfen, analysiert und bewertet [2]. Dabei werden zum Teil bereits Systeme mit zusätzlicher Rechenleistung in Form von Multiprozessorsystemen als On-Board Computer für Raumfahrtsysteme eingesetzt, jedoch nicht im vollen Umfang genutzt. Die Flugsoftware wird nur für die Verwendung eines Prozessors entwickelt. Hintergrund ist zum einen die analyseintensive Umstellung bestehender Flugsoftware und der zugrundeliegenden Softwareplattform OUTPOST auf den Multiprozessorbetrieb. Zum anderen ist dafür eine zertifizierte Unterstützung des Multiprozessorbetriebs seitens der genutzten Betriebssysteme notwendig. Ein für die Uniprozessor-Software genutztes Betriebssystem verfügt mittlerweile über entsprechende zertifizierte Unterstützung von Multiprozessorsystemen.

Die nötigen Instrumente für den Multiprozessorbetrieb sind also vorhanden und das volle Potential der gegebenen Strukturen sollte genutzt werden. Durch die zusätzliche Rechenleistung können Aufgaben mit höherer Frequenz oder zusätzliche Aufgaben parallel ausgeführt werden.

Während On-Board Computer und Betriebssystem bereits den Einsatz mehrerer Prozessoren ermöglichen, sollen diese beiden Komponenten nun beim DLR in Bremen erstmals zusammengebracht und auf Basis dieser Arbeit in einer Flugsoftware angewendet werden können. Bestehende Strukturen zur Entwicklung von Flugsoftware müssen dafür an die Multiprozessorunterstützung angepasst und die Auswirkungen auf die Flugsoftware untersucht werden.

Die im Rahmen dieser Arbeit genutzten Systemkomponenten orientieren sich dabei an dem vom DLR entwickelten Satelliten Eu:CROPIS. Für diesen existiert bereits eine fertige Uniprozessor-Software, basierend auf dem Multiprozessorsystem GR712RC und dem Multiprozessorsystem-unterstützenden Echtzeitbetriebssystem RTEMS. Die Ergebnisse dieser Arbeit sollen anschließend dabei helfen, die Multiprozessorunterstützung erstmals

mit der realen Flugsoftware von Eu:CROPIS zu testen und den Zugewinn an verfügbarer Rechenleistung abzuschätzen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, zum einen eine theoretische, und zum anderen eine praktische Grundlage für die Entwicklung von Flugsoftware für Multiprozessorsysteme in der Raumfahrt zu schaffen.

Die praktische Grundlage, die als Ergebnis aus dieser Arbeit hervorgehen soll, bezieht sich auf die Entwicklung der Betriebssystem-Abstraktion für Multiprozessorsysteme. Dabei wird eine in der Softwareplattform OUTPOST integrierte Abstraktion des Echtzeitbetriebssystems RTEMS um die symmetrische Multiprozessorunterstützung erweitert. Die theoretische Grundlage soll erste Erfahrungen in der Anwendung der Multiprozessorsystem-unterstützenden Betriebssystem-Abstraktion liefern. Dazu wird eine Leistungsanalyse durchgeführt.

Das Vorgehen der Entwicklung und der Leistungsanalyse der Betriebssystem-Abstraktion orientiert sich an folgenden Leitfragen:

1. **Wie kann die Multiprozessorunterstützung von RTEMS sinnvoll in OUTPOST integriert werden?**

Zur Anpassung der Betriebssystem-Abstraktion an den Multiprozessorbetrieb ist zunächst eine Analyse der Multiprozessorsystem-Unterstützung des Betriebssystems RTEMS nötig. Darauf aufbauend soll die Multiprozessorsystem-Unterstützung durch die Softwareplattform OUTPOST erarbeitet, begründet und implementiert werden.

2. **Wie kann die Leistungssteigerung gemessen werden?**

Die Verwendung mehrerer Prozessoren soll eine Leistungssteigerung von Raumfahrtsystemen ermöglichen. Es soll ein Benchmark entwickelt werden, mit dessen Hilfe bestimmte Leistungskenngrößen und Kriterien zuverlässig und repräsentativ gemessen werden können. Diese Leistungskenngrößen und Kriterien gilt es im Voraus zu definieren.

3. **Wie groß ist die Leistungssteigerung durch die Multiprozessorunterstützung?**

Mithilfe des Benchmarks sollen anschließend Messungen zur Leistungsanalyse durchgeführt werden und im Bezug auf die Leistungssteigerung beurteilt werden.

4. **Welche Vor- und Nachteile bringt die Multiprozessorunterstützung mit sich?**

Der Einsatz eines Multiprozessorsystems soll zudem kritisch betrachtet werden. Es sollen Vorteile sowie Risiken und zu berücksichtigende Sachverhalte ermittelt und deren Bedeutung für die Entwicklung von Flugsoftware erläutert werden.

5. **Wie lässt sich die Multiprozessorunterstützung in realer Software einsetzen?**

Anhand der Ergebnisse der vorherigen Leitfragen soll eine Beurteilung stattfinden, wie der Einsatz des Multiprozessorbetriebs sinnvoll in realer Flugsoftware umgesetzt werden kann.

1.3 Aufbau der Arbeit

In **Kapitel 2** werden zunächst die theoretischen Grundlagen erläutert, die zum Verständnis dieser Arbeit beitragen. Dazu gehören neben der Einführung von Begriffen und der Vorstellung verwendeter Systemkomponenten auch die Erläuterung von Sachverhalten und Theorien zum Thema Multiprozessorsysteme, die im Rahmen dieser Arbeit von Relevanz sind.

Kapitel 3 behandelt die Erweiterung der Softwareplattform OUTPOST um die Unterstützung des symmetrischen Multiprocessings durch das Echtzeitbetriebssystem RTEMS.

Kapitel 4 befasst sich mit der Entwicklung eines Benchmarks zur Leistungsanalyse des Multiprozessorbetriebs. Es werden Methoden zur Leistungsanalyse erläutert, bewertet und darauf basierend ein Benchmark entwickelt. Es werden verschiedene Szenarien zur Anwendung des Benchmarks im Multiprozessorbetrieb erläutert.

In **Kapitel 5** werden die in den Szenarien ermittelten Messergebnisse dargestellt, ausgewertet und verglichen. Zusätzlich werden die eingesetzten Methoden zur Leistungsanalyse bewertet.

In **Kapitel 6** werden die in dieser Arbeit gewonnenen Ergebnisse zusammengefasst und ein Ausblick auf die zukünftige Anwendung der Ergebnisse gegeben.

2 Theoretische Grundlagen

In diesem Kapitel werden die zum Verständnis der Arbeit nötigen Grundlagen erläutert. Zur Einführung in die Thematik beginnt Abschnitt 2.1 mit den Anforderungen, denen Raumfahrtsysteme im Allgemeinen unterliegen. In Abschnitt 2.2 wird die vom DLR entwickelte Softwareplattform OUTPOST vorgestellt. Abschnitt 2.3 befasst sich mit eingebetteten Systemen und stellt das Systen-on-a-Chip (SoC) GR712 vor. Abschnitt 2.4 erläutert das Konzept von Echtzeitbetriebssystemen und gibt eine Einführung in das Betriebssystem RTEMS. Den Abschluss dieses Kapitels bilden die Multiprozessorsysteme als zentrales Element dieser Arbeit. Abschnitt 2.5 geht neben hard- und softwaretechnischen Eigenschaften und den Unterschieden zu Uniprozessorsystemen auch auf Theorien zur Leistungssteigerung durch den Einsatz von Multiprozessorsystemen ein.

2.1 Anforderungen an Raumfahrtsysteme

Die Erde bietet mit ihren verhältnismäßig stabilen Temperaturen, dem konstanten Gravitationsfeld und ihrem vor kosmischen Strahlungen schützenden Magnetfeld günstige oder zumindest gut bekannte Umgebungsbedingungen für technische Systeme. Im Weltraum hingegen herrschen andere Bedingungen. Die Schwerelosigkeit, das Vakuum und große Temperaturschwankungen stellen den Einsatz technischer Systeme vor große Herausforderungen. Hinzu kommen kosmische Strahlungen, die gravierende Schäden an technischen Systemen verursachen können. So können von der Sonne ausgesandte, hochenergetische ionisierte Teilchen Single Event Upsets (SEUs) auslösen und beispielsweise zum Kippen einzelner Bits führen oder durch Single Event Latch-Ups (SELs) einen Kurzschluss und Hardwareschäden auf der mikroelektronischen Ebene verursachen. [20]

Eine Reparatur technischer Systeme im Weltraum ist aufwändig, kostspielig, dadurch oft nicht lohnenswert und in den meisten Fällen schlichtweg unmöglich. So kann der Defekt einer Forschungssonde während einer Mission den Erfolg jahrelanger Arbeit und

Vorbereitung gefährden und immense finanzielle Verluste mit sich bringen. Auch die für uns selbstverständliche und alltägliche technische Infrastruktur wie das Internet basiert auf der einwandfreien Funktion von Satelliten und verlangt ihnen ein hohes Maß an Zuverlässigkeit ab. Bei anderen Raumfahrtsystemen kann ein Defekt oder Fehlverhalten oder schlichtweg eine zu späte Reaktion des Systems auf ein Ereignis im schlimmsten Fall Menschenleben gefährden. Systeme, deren Ausfall solch schwerwiegende Folgen mit sich ziehen können, zählen zu den missionskritischen Systemen.

Durch die harschen Bedingungen des Weltraums unterliegen sowohl die Hardware als auch die Software dieser Systeme besonderen Anforderungen. Softwareseitig sind bei missionskritischen Systemen eine hinreichend schnelle Reaktion eines Systems, ein prädiktives Systemverhalten und eine hohe Zuverlässigkeit von zentraler Bedeutung. Diese Anforderungen können durch den Einsatz von Echtzeitbetriebssystemen wie RTEMS gewährleistet werden. Auch die Hardware des OBCs muss bestimmte Kriterien erfüllen, um Vakuum, hohen Temperaturen und hohen Strahlungen zu trotzen. Die Firma Cobham Gaisler entwickelt speziell für den Einsatz im Weltraum konzipierte Komponenten, die diesen Bedingungen standhalten. Dazu gehören die Prozessoren der LEON-Familie wie auch das Multiprozessorsystem GR712RC.

2.2 OUTPOST

OUTPOST (**O**pen mod**U**lar sof**T**ware **P**latf**O**rm for **S**pacecraf**T**) ist eine vom DLR entwickelte Open Source Software Plattform. Sie dient als Grundlage für modulare portierbare Flugsoftware und kommt in eingebetteten, missionskritischen Systemen wie Satelliten aber auch Raketen zum Einsatz.[3] OUTPOST abstrahiert eine Auswahl an Treibern, verschiedene Board Support Packages (BSP) und Betriebssysteme wie beispielsweise RTEMS und stellt eine C++ API (Application Programming Interface) zur Entwicklung von Flugsoftware zur Verfügung (Abbildung 2.1).

Durch den modularen Aufbau kann die Nutzung von OUTPOST als Grundlage für die Flugsoftware individuell auf die Hardware und die zu entwickelnde Anwendung angepasst werden. OUTPOST ist aufgeteilt in outpost-core und outpost-satellite. Zusätzlich stellt OUTPOST verschiedene Hardware Abstraction Layers (HAL) zur Unterstützung verschiedener Target-Systeme zur Verfügung. Für diese Arbeit wird outpost-platform-leon verwendet, welches Systeme auf Basis der LEON Prozessoren unterstützt.

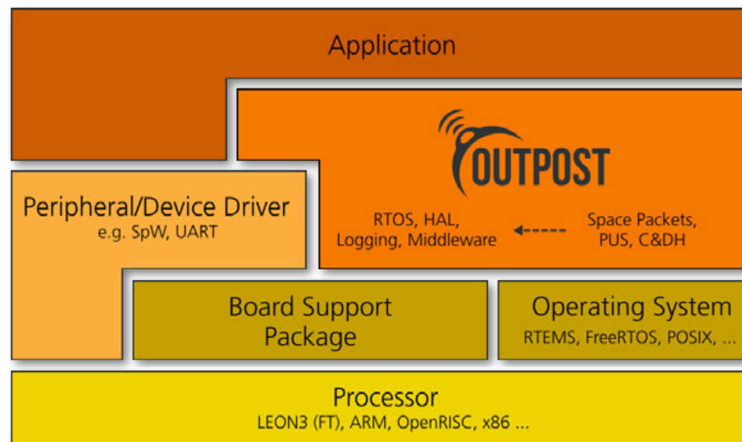


Abbildung 2.1: OUTPOST im Gesamtsystem [3]

- **outpost-core** stellt die Open Source Basisumgebung von OUTPOST zur Verfügung. Diese beinhaltet grundlegende Softwaremodule wie Funktionen für das Zeitmanagement, eine Hardware-Abstraktionsschicht für Kommunikationsschnittstellen und ein Testframework, aber auch die Betriebssystem-Abstraktion, die das Multitasking ermöglicht und im Rahmen dieser Arbeit erweitert werden soll. [5]
- **outpost-satellite** beinhaltet satellitenspezifische Closed Source Module, die unter anderem zur Unterstützung des Command and Data Handling und verschiedener Protokolle dienen.[7]
- **outpost-platform-leon** ist eine Hardware-Abstraktionsebene, welche Systeme unterstützt, die auf den LEON Prozessoren basieren. Dazu gehört das in diesem Projekt verwendete GR712RC. [6]

2.3 Eingebettete Systeme

Als eingebettete Systeme werden aus Hard- und Softwarekomponenten bestehende Systeme bezeichnet, die in einen technischen Kontext integriert sind. Sie kommunizieren mittels Sensorik und Aktorik mit der Umgebung, steuern und regeln diese und führen datenverarbeitende Aufgaben durch. Durch häufige platz- und gewichtstechnische Limitation durch das umgebende System sind kleine Programm- und Datenspeicher typisch für eingebettete Systeme. Zudem wird meist ein geringer Stromverbrauch vorausgesetzt,

um die Versorgung durch mobile Stromquellen oder von autarken Systemen zu gewährleisten. Der Einsatz eingebetteter Systeme erstreckt sich von Systemen des täglichen Gebrauchs, wie elektrische Küchengeräte oder Heizungsanlagen, über Fabrik-Roboter bis hin zu Flugzeugen, Raketen und Satelliten [1]. Je nach Einsatzbereich entstehen dabei unterschiedliche Anforderungen an die Hard- und Software eingebetteter Systeme.

Besonders in der Raumfahrt unterliegen eingebettete Systeme strengen Anforderungen. Äußere Störeinflüsse und der missionskritische Hintergrund müssen berücksichtigt werden und erfordern unter anderem eine redundante Speicherauslegung und größere Fertigungstechnologien auf der Mikroelektronischen ebene, wodurch wiederum eine Leistungsminderung entsteht. Ein typischer Vertreter für den OBC in Kleinsatelliten in Europa ist das SoC GR712RC.

2.3.1 GR712RC

Das GR712RC ist ein von dem Unternehmen Cobham Gaisler entwickeltes Dual-Core LEON3FT System-on-a-Chip (Abbildung 2.2). Als speziell für den Einsatz in der Raumfahrt entwickeltes, hoch zuverlässiges System bietet das GR712RC eine hohe Strahlenresistenz und Fehlertoleranzfunktionen. [10]

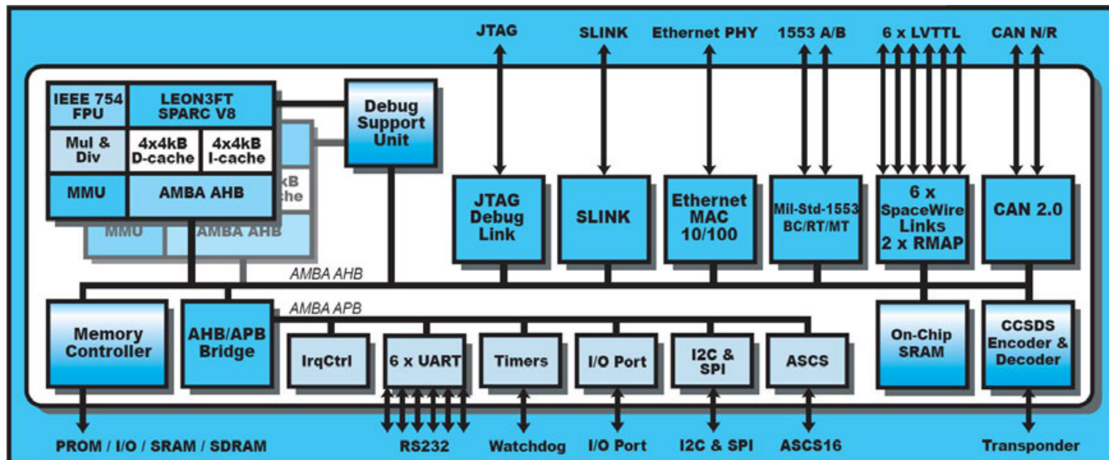


Abbildung 2.2: Blockdiagramm des GR712RC [10]

Das GR712RC verfügt über zwei in SMP-Konfiguration implementierte LEON3FT Prozessoren. Der LEON3FT ist ein fehlertoleranter 32-Bit Prozessor, entworfen nach der SPARC V8 Architektur.

Als für den Einsatz in der Raumfahrt entwickelter Prozessor liegt ein Hauptaugenmerk der Fehlertoleranz des LEON3 auf dem Auftreten von SEU-Fehlern. Die Implementierung der On-Chip Register unter Verwendung dreifach modularer Redundanz (Triple Modular Redundancy, TMR) ermöglicht das Entfernen von SEU-Registerfehlern innerhalb eines Taktzyklusses. Der externe Speicher wird durch das Fehlerkorrekturverfahren EDAC (error-detection and correction) geschützt. Die Caches verfügen über eine einfache Fehlererkennung in Form von Paritätsbits.[18] Eine PLL-Watchdog-Funktion bietet die Möglichkeit, den Ausfall eines PLLs und somit des Systemtakts zu erkennen[13].

Die LEON3FT Prozessoren besitzen jeweils einen eigenen Level 1 Cache. Der Harvard-Architektur entsprechend ist der Cache des LEON3FT in einen 16 KiB Daten-Cache und einen 16 KiB Instruction-Cache aufgeteilt. Der Daten-Cache nutzt die write-through Policy mit der no-allocate Policy bei einem write-miss und die LRU replacement-Policy (siehe Abschnitt 2.5.3). Die Caches beider LEON3FT Prozessoren sind über einen gemeinsamen AHB-Bus mit dem Hauptspeicher verbunden. [4]

2.4 Echtzeitbetriebssysteme

Echtzeitbetriebssysteme werden eingesetzt, wenn Systemen eine schnelle Reaktion, eine hohe Zuverlässigkeit und gleichzeitig ein weitgehend deterministisches Verhalten abverlangt wird. Im Bereich der Raumfahrt sind sie daher unabkömmlich.

Echtzeitbetriebssysteme zeichnen sich gegenüber allgemeinen Betriebssystemen dadurch aus, dass sie die Reaktion eines Systems auf ein auftretendes Ereignis innerhalb eines definierten Zeitrahmens garantieren. Die für die Reaktion erlaubte Dauer, die zum Einhalten der harten Echtzeit nicht überschritten werden darf, kann durch Deadlines definiert werden, die den Aufgaben zugewiesen sind. Das Grundprinzip von Echtzeitbetriebssystemen liegt in der Aufteilung eines gewünschten Verhaltens in Aufgaben und deren Priorisierung. Wichtigen, sicherheitskritischen Aufgaben, die eine schnelle Reaktion erfordern und die bei zu später Reaktion zu fatalen Folgen, wie z.B. dem Ausfall des Gesamtsystems, führen können, wird eine hohe Priorität zugewiesen. Weniger wichtige Aufgaben erhalten eine niedrigere Priorität. Tritt ein Ereignis auf, das die Ausführung einer Aufgabe mit einer höheren Priorität als die der aktiven Aufgabe erfordert, ermöglicht das präemptive Scheduling die Unterbrechung der aktiven Aufgabe zur Ausführung der höher priorisierten.

Im Folgenden werden einige im Kontext von Echtzeitbetriebssystemen relevante Begriffe erläutert.

- **Task:** Ein Programm wird in mehrere kleinere Aufgaben, die sogenannten Tasks, zerlegt. Ein Task kann dabei als kleines Programm mit einer bestimmten Aufgabe betrachtet werden. Ein Task kann sich in einem von drei Zuständen befinden[1]. Im Zustand **Ready** ist er bereit und kann vom Scheduler aufgerufen und ausgeführt werden. Im Zustand **Blocked** ist der Task blockiert und kann nicht vom Scheduler aufgerufen werden. In diesem Zustand verweilt er, bis beispielsweise ein Ereignis eintritt, auf welches der Task wartet. Befindet sich ein Task im Zustand **Executing**, wird er gerade auf einer CPU ausgeführt. Synchronisierung und Nachrichtenaustausch zwischen Tasks kann über Mutexe, Semaphoren, Queues und Shared Memory erfolgen.
- **Scheduler:** Zentrale Verwaltungseinheit eines Echtzeitbetriebssystems ist der Scheduler. Der Scheduler entscheidet anhand eines Scheduling Algorithmus, welcher Task der CPU zugewiesen und ausgeführt wird.
- **Dispatcher:** Der Dispatcher ist für den Kontext Switch zuständig. Ein Kontext Switch findet statt, wenn ein Task von der CPU verdrängt und ein anderer Task ihr zugewiesen wird. Der Dispatcher speichert den Zustand des von der CPU verdrängten Tasks und lädt den Zustand des Tasks, der auf der CPU ausgeführt werden soll.

2.4.1 RTEMS

Im Rahmen dieser Arbeit wird das Open Source Echtzeitbetriebssystem RTEMS (Real-Time Executive for Multiprocessor Systems) verwendet. RTEMS zeichnet sich durch eine hohe Zuverlässigkeit, eine harte Echtzeitfähigkeit und geringen Ressourcenverbrauch aus, wodurch es sich als geeignetes Betriebssystem für den Einsatz in Raumfahrtsystemen erweist[14]. Zudem bietet RTEMS Unterstützung für eine Vielzahl von Prozessor-Architekturen und SoCs, darunter auch in der Raumfahrt genutzte SoCs wie das GR712RC. Für einige Multiprozessorsysteme unterstützt RTEMS zusätzlich das Symmetric Multiprocessing (SMP). Seit Anfang 2022 ist die SMP Konfiguration von RTEMS 5[2] von der

[1] Je nach Betriebssystem können sich die Zustände unterscheiden. Neben abweichenden Bezeichnungen können zusätzliche Zustände vorkommen

[2] RTEMS 4 verfügt lediglich über eine experimentelle SMP Konfiguration

European Space Agency (ESA) für die ECSS Criticality Category C und D zertifiziert [8].

Für das Multiprozessorsystem GR712RC bietet RTEMS drei Board Support Packages: für den Uniprozessor-Betrieb, das Asymmetric Multiprocessing und das Symmetric Multiprocessing. Das SMP-BSP gilt es in dieser Arbeit hinsichtlich seiner Eigenschaften und Auswirkungen auf OUTPOST und eine Flugsoftware zu analysieren.

2.5 Multiprozessorsysteme

Als Multiprozessorsysteme werden Systeme bezeichnet, die über zwei oder mehr Prozessoren verfügen. Im Gegensatz zu der Quasiparallelität im Sinne des Multitaskings ermöglicht das Multiprocessing durch das Ausführen von Tasks auf mehreren Prozessoren die echte zeitliche Parallelität. Multiprozessorsysteme können verschiedene Architekturen aufweisen (Unterabschnitt 2.5.1). Ziel des Einsatzes mehrerer Prozessoren ist häufig die Steigerung der Leistung eines Systems (Unterabschnitt 2.5.2). Es können mehr Aufgaben in einem bestimmten Zeitrahmen abgearbeitet werden, während gleichzeitig die Einhaltung des Echtzeitkriteriums garantiert werden kann. In Unterabschnitt 2.5.3 werden die Cache-Speicher behandelt und deren Einfluss auf die Leistung von Multiprozessorsystemen erläutert. Bei der Softwareentwicklung für Multiprozessorsysteme müssen einige Unterschiede gegenüber der Software für Uniprozessorsysteme beachtet werden (Unterabschnitt 2.5.4).

2.5.1 Multiprozessorsystem-Architekturen

Bei Multiprozessorsystemen wird zwischen symmetrischen und asymmetrischen Systemen unterschieden.

- Bei symmetrischen Multiprozessorsystemen sind alle Prozessoren des Systems gleichartig. Beim Symmetric Multiprocessing (SMP) sind die Prozessoren gleichwertig und es kann die Ausführung jedes Programmabschnitts auf jedem der Prozessoren stattfinden. Alle Prozessoren werden von einer einzigen Betriebssysteminstanz verwaltet und besitzen einen gemeinsamen Adressraum. [17]Unter bestimmten Umständen kann beim Symmetric Multiprocessing eine statische Zuweisung von Tasks zu Prozessoren sinnvoll sein, um beispielsweise ein deterministischeres Verhalten

des Systems zu erzielen. Zudem gibt es symmetrische Multiprozessorsysteme, die das Asymmetric Multiprocessing unterstützen.

- Bei asymmetrischen Multiprozessorsystemen können unterschiedliche Prozessoren mit unterschiedlichen Möglichkeiten im Bezug auf Speicher- oder Peripheriezugriff vorliegen. Beim Asymmetric Multiprocessing (ASMP) kann eine Aufgabe nicht auf jedem beliebigen Prozessor laufen. Die Prozessoren sind dabei nicht gleichwertig sondern ein Master-Prozessor hat die Kontrolle über die weiteren Slave-Prozessoren und übernimmt die Aufgabenverteilung. Die Prozessoren können von unterschiedlichen Betriebssystemen oder mehreren Instanzen desselben Betriebssystems verwaltet werden. [17]

2.5.2 Leistungssteigerung durch Multiprozessorsysteme

Multiprozessorsysteme verfügen über mehrere Instanzen der zentralen Recheneinheiten. Der Gedanke, dass eine Verdopplung der Anzahl der Prozessoren auch zur Verdopplung der Rechenleistung eines Systems führt, ist naheliegend. Dass diese Annahme eines linearen, direkt proportionalen Zusammenhangs zwischen Leistung und Anzahl der Prozessoren nicht uneingeschränkt der Realität entspricht, begründete Amdahl anhand eines Gesetzes, welches die Höhe der Beschleunigung, des sogenannten Speedups, eines Programmes durch parallele Ausführung unter Berücksichtigung limitierender Faktoren thematisiert.

Speedup

Der Speedup beschreibt das Verhältnis der Leistung zweier Systeme, die dasselbe Problem bearbeiten. Der Speedup kann dabei anhand der Ausführungszeit oder des Durchsatzes definiert werden.

- Der Speedup der Ausführungszeit beschreibt das Verhältnis der Zeit zweier Systeme, die sie zum Bearbeiten derselben Problemgröße benötigen.
- Der Speedup des Durchsatzes beschreibt das Verhältnis der in einem festgelegten Zeitrahmen bearbeiteten Problemgröße zweier Systeme.

Der Speedup S lässt sich mit der Ausführungszeit T_1 bzw. T_2 und dem Durchsatz D_1 bzw. D_2 von System 1 bzw. System 2 folglich berechnen durch

$$S = \frac{T_1}{T_2} = \frac{D_2}{D_1} \tag{1}$$

Gesetz von Amdahl

Das Gesetz von Amdahl widerlegt die Annahme, dass der Speedup eines Systems linear von der Anzahl der zum Lösen eines Problems eingesetzten Prozessoren abhängt. Nach Amdahl wird der Speedup eines Problems fester Größe durch dessen sequentiellen Anteil begrenzt.[19]

Ein Problem P lässt sich in einen sequentiellen Anteil P_S und einen parallelen Anteil P_P aufteilen. Der sequentielle Anteil kann auch durch Einsatz mehrerer Prozessoren nicht beschleunigt und die Laufzeit nicht verringert werden. Sequentielle Anteile eines Programmes können Initialisierungsprozesse sowie Zugriffe auf externe Ressourcen wie der in Unterabschnitt 2.5.3 beschriebene Zugriff auf den Hauptspeicher darstellen. Der parallele Anteil eines Problems lässt sich durch Einsatz mehrerer Prozessoren beschleunigen. Der maximale Speedup wird somit durch die Größe des sequentiellen Anteils begrenzt und konvergiert gegen $S_{max} = \frac{P}{P_S}$ (Abbildung 2.3, *Amdahl*, $P_P = 0,75$ konvergiert gegen $\frac{1}{0{,}25} = 4$).[19]

Nach Amdahl berechnet sich der Speedup durch Einsatz von n Prozessoren zu

$$S = \frac{P}{P_S + \frac{P_P}{n_P}} \tag{2}$$

Abbildung 2.3 (*Amdahl*) zeigt den Verlauf des Speedups nach Amdahl. Für kleine Prozessorzahlen verläuft der Speedup annähernd linear. Bei steigender Prozessorzahl hängt der Speedup immer stärker von dem sequentiellen Anteil ab und nähert sich asymptotisch dem maximalen Speedup.

Gesetz von Gustafson

Das Gesetz von Gustafson basiert auf dem Gesetz von Amdahl. Im Gegensatz zu Amdahl betrachtet Gustafson jedoch nicht den Speedup der Laufzeit eines Problems, sondern

die innerhalb eines festgelegten Zeitfensters lösbare Größe eines Problems.[19] Wie bei Amdahl wird das Problem in einen parallelen Anteil P_P und einen sequentiellen Anteil $P_S = (1 - P_P)$ aufgeteilt. Der sequentielle Anteil bleibt auch bei steigender Anzahl an Prozessoren und somit wachsender Problemgröße konstant. Unter Verwendung mehrerer Prozessoren ergibt sich der Speedup S zu

$$S = P_S + n \cdot P_P \tag{3}$$

Abbildung 2.3 (*Gustafson*)zeigt den Verlauf des Speedups nach Gustafson, der durch den konstanten sequentiellen Anteil langsamer als der ideale Speedup, aber ebenfalls linear steigt.

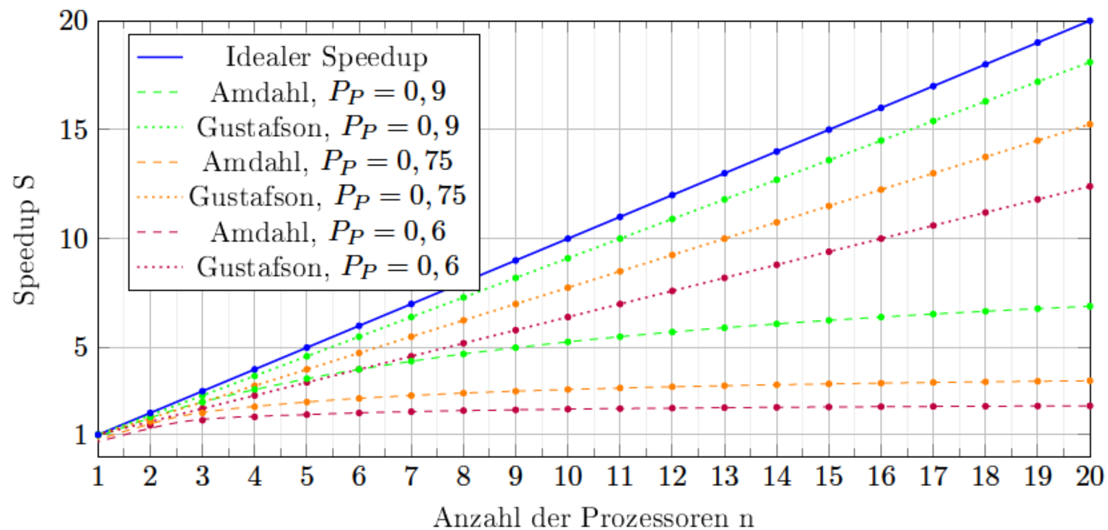


Abbildung 2.3: Speedup nach Amdahl und Gustafson

2.5.3 Cache-Speicher

Cache-Speicher sind schnelle on-Chip Speichereinheiten, die eine Teilmenge des Hauptspeichers enthalten und dem Prozessor einen schnellen Zugriff auf diese Daten ermöglichen. Der Cache liegt zwischen der CPU und dem Hauptspeicher. Welche Daten sich im Cache befinden, wird durch bestimmte Algorithmen, die sogenannten Policies, geregelt. Der Cache vieler Systeme ist hierarchisch aufgebaut und besteht aus verschiedenen Levels. Die Caches mit niedrigem Level liegen dabei nah am Prozessor, mit aufsteigendem Level befinden sich die Caches näher am Hauptspeicher. Mit steigender Entfernung

zum Prozessor werden die Caches größer, aber auch langsamer. Der Level 1 (L1) Cache ist demnach der schnellste, aber auch der kleinste. Die Caches beinhalten jeweils eine Teilmenge des Inhalts des Caches, der ein Level näher am Hauptspeicher liegt. Bei Multiprozessorsystemen verfügt in der Regel jeder Prozessor über einen eigenen L1 Cache.[15]

Cache-Verwaltung

Welche Daten sich im Cache befinden, hat Einfluss auf die Zugriffszeit, wenn diese Daten vom Prozessor benötigt werden. Wie der Cache und der Hauptspeicher verwendet werden, wird durch die Cache-Policies festgelegt. Unterschieden wird beim Zugriff des Prozessors auf den Speicher zwischen einem Cache Hit und einem Cache Miss. Bei einem Cache Hit befinden sich die angefragten Daten bereits im Cache, bei einem Cache Miss nicht.[22]

1. **Lesezugriff:** Bei einem Lesezugriff des Prozessors auf den Speicher kann ein Read Hit oder ein Read Miss auftreten. Bei einem Read Hit befinden sich die angeforderten Daten bereits im Cache und werden direkt aus diesem gelesen. Bei einem Read Miss befinden sich die angeforderten Daten nicht im Cache. Sie werden aus dem Hauptspeicher gelesen und dabei, um sie für weitere Zugriffe schnell erreichbar zu machen, im Cache hinterlegt.[22]

2. **Schreibzugriff:** Beim Schreibzugriff stehen für ein Cache Hit zwei Policies zur Auswahl.

   - **write-through Policy:** Die Daten werden in den Cache und gleichzeitig in den Hauptspeicher geschrieben.

   - **write-back Policy:** Die Daten werden zunächst nur in den Cache und nicht in den Hauptspeicher geschrieben. Erst wenn der Cache voll ist und neue Daten geladen werden sollen, werden Daten aus dem Cache entfernt und in den Hauptspeicher übertragen. Welche Daten ersetzt werden, wird durch die Cache Replacement Policy entschieden.

   Tritt bei einem Schreibzugriff ein Cache Miss auf, kommt eine der folgenden write-miss Policies zum Einsatz.

- **write-allocate Policy:** Die Daten werden in den Hauptspeicher geschrieben und die angeforderten Daten in den Cache geladen.
- **non-write allocate Policy:** Die Daten werden nur in den Hauptspeicher geschrieben, die angeforderten Daten werden nicht in den Cache geladen.

3. **Replacement Policy:** Die Replacement Policy kommt zum Einsatz, wenn Daten in einem vollen Cache platziert und vorhandene Daten somit verdrängt werden sollen. Welche Daten aus dem Cache verdrängt werden sollen, wird durch eine Verdrängungsstrategie festgelegt. Hier kommen bekannte Strategien wie *First In - First Out* (FIFO), *Least Recently Used* (LRU) oder *Least Frequently Used* (LFU) zum Einsatz. [22]

Bedeutung für Multiprozessorsysteme

Bei einem Multiprozessorsystem bzw. der SMP-Konfiguration stellt der gemeinsam genutzte Bus zwischen den Prozessoren und dem Hauptspeicher einen Bottleneck dar. Im Uniprozessor-Modus greift nur ein Prozessor auf die Busleitung zu, wenn ein Schreib- oder Lesevorgang im Hauptspeicher stattfinden soll. In der SMP-Konfiguration nutzen beide Prozessoren den Bus. Durch den Bus-Zugriff eines Prozessors kann es demnach zu zeitlichen Verzögerungen auf dem anderen Prozessor kommen, wenn dieser ebenfalls den Bus nutzen will. Selbst wenn auf den Prozessoren zwei voneinander unabhängige Softwareteile laufen, mindert der gleichzeitige Betrieb beider Prozessoren die Geschwindigkeit des einzelnen. Die Gesamtleistung ist demnach niedriger als die Summe der Leistungen der Prozessoren im Einzelbetrieb.

Die Menge und der Zeitpunkt der Zugriffe auf den Hauptspeicher können durch die Cache-Policies und die Art des Programmes beeinflusst werden. Wird ein Task beispielsweise statisch einer CPU zugewiesen, so befinden sich die Daten des Tasks mit hoher Wahrscheinlichkeit im Cache und müssen theoretisch nur dort aktualisiert werden. Die write-thorugh Policy schreibt jedoch bei jeder Datenaktualisierung in den Hauptspeicher und greift auf den Bus zu. Bei der write-back Policy sorgt nur ein voller Cache für einen Schreibvorgang auf den Hauptspeicher. Die write-through Policy, wie sie beim GR712RC angewendet wird, führt somit zu einer höheren Belastung des Busses als die write-back Policy.

Bei Multiprozessorsystemen entsteht außerdem bei steigender Prozessorzahl ein höherer Synchronisiserungsaufwand, um die Cache-Kohärenz[3] zu gewährleisten. Bei der write-through Policy ist dieser durch die Aktualisierung des Hauptspeichers bei jedem Schreibvorgang geringer als bei der write-back Policy. Die Synchronisierung kann beispielsweise durch das Bus-Snooping realisiert werden, bei dem die Caches einen Zugriff auf eine sich im Cache befindende Speicheradresse über den Bus registrieren.

2.5.4 Unterschiede zwischen Software für Uni- und Multiprozessorsysteme

Die Software für Uniprozessorsysteme kann nicht ohne Weiteres beliebig instanziiert oder aufgeteilt und auf einem Multiprozessorsystem ausgeführt werden. Zum einen stellt sich die Frage, wie die Verwaltung der Prozessoren seitens des Schedulers vorgenommen werden soll. Zum anderen sind zusätzliche Mechanismen nötig, um zu verhindern, dass sich die Prozessoren gegenseitig behindern und es zu einem Fehlverhalten des Systems kommt. Dabei geht es im Allgemeinen um den Schutz des Zugriffes auf gemeinsam genutzte Ressourcen innerhalb kritischer Sektionen.

Scheduling

Das Verwalten mehrerer Prozessoren eines Systems durch den Scheduler beeinflusst die Scheduling Algorithmen, die vom Betriebssystem zur Verfügung gestellt werden. Zudem kann während der Softwareentwicklung auf Basis eines Betriebssystems Einfluss auf die Verwaltung der Prozessoren genommen werden.

Scheduling Algorithmen Das Scheduling-Problem, also die Entscheidung, wann welcher Task ausgeführt wird, sodass das System keine Deadline verletzt, stellt schon bei Uniprozessorsystemen eine Herausforderung dar, die mit Hilfe verschiedener Scheduling-Algorithmen bewältigt werden kann. Verfügt ein Scheduler über mehrere Prozessoren, denen er einen Task zur Ausführung zuweisen kann, entsteht eine zusätzliche Dimension, die von den Algorithmen berücksichtigt werden muss und durch die das Lösen des Scheduling-Probelms erheblich komplexer wird. Seitens des Betriebssystems sind daher eigene Scheduling-Algorithmen für den Einsatz auf Multiprozessorsystemen nötig.

[3] Caches und Hauptspeicher sind kohärent, wenn bei einem Lesezugriff immer der aktuelle Zustand der Daten gelesen wird.

Clustered Scheduling Der Einsatz von Multiprozessorsystemen ermöglicht das Clustered Scheduling. Beim Clustered Scheduling wird die Gesamtmenge der Prozessoren in einzelne, sich nicht überschneidende Teilmengen, die sogenannten Cluster, eingeteilt. Jeder Prozessor ist genau einem Cluster zugeordnet. Ein Cluster mit nur einem Prozessor wird als Partition bezeichnet. Gehören alle Prozessoren eines Systems dem selben Cluster an, spricht man von globalem Scheduling, andernfalls von partitioniertem Scheduling. Jedes Cluster wird von einer eigenen Scheduler-Instanz verwaltet (Abbildung 2.4). Ein Task kann einer Scheduler-Instanz und somit einem Cluster, also einer Teilmenge der Prozessoren zugewiesen werden. Der Task kann somit nur von der Scheduler-Instanz und nur auf der zur Instanz gehörenden Teilmenge von Prozessoren ausgeführt werden. Die Synchronisierung zwischen Clustern kann über Events, Queues, Semaphoren und Mutexe erfolgen. [9]

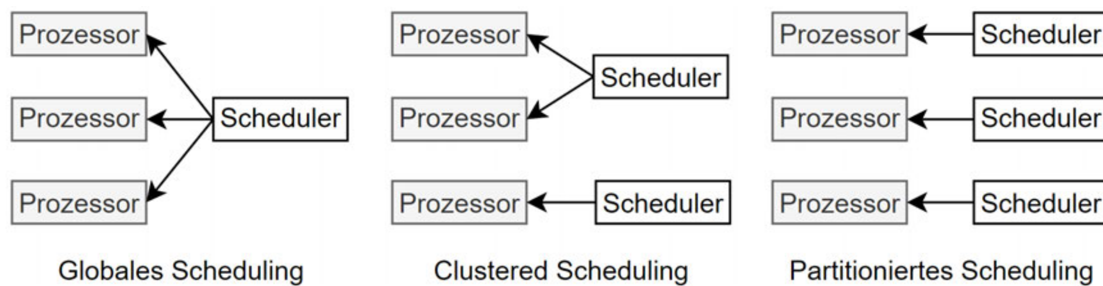


Abbildung 2.4: Globales, Clustered und Partitioniertes Scheduling

Für den Einsatz von Clustered Scheduling gibt es verschiedene Gründe. Das Clustered Scheduling kann verwendet werden, um auf den Prozessoren eines Systems parallel verschiedene Betriebssysteme anzuwenden oder sicherheitskritische Programmteile mit Echtzeitanspruch von nicht-sicherheitskritischen Programmteilen zu trennen und durch verschiedene Scheduler zu verwalten. Zudem kann mit Hilfe des Clustered Scheduling die Zugriffszeit eines Tasks auf Daten positiv beeinflusst werden. Die Task-Migration beschreibt den Wechsel eines Tasks von einem Prozessor zu einem anderen. Wechselt der Task den Prozessor, sinkt die Cache-Lokalität, also die Wahrscheinlichkeit, dass sich die vom Task benötigten Daten bereits im Cache befinden. Je mehr Prozessoren ein System hat und je mehr unterschiedliche Tasks auf einem Prozessor laufen können, desto niedriger wird die Cache-Lokalität. Durch Clustered Scheduling kann die Anzahl an Tasks, die auf einem Prozessor laufen können und die Anzahl an Prozessoren, auf denen ein Task laufen

kann, eingegrenzt werden. Dadurch steigt die Cache-Lokalität und die durchschnittliche Zugriffszeit auf Daten wird verringert.[9]

Kritische Sektionen

Die Besonderheit von Multiprozessorsystemen liegt in der parallelen Ausführung von Tasks. Versuchen jedoch mehrere Tasks gleichzeitig auf eine Ressource zuzugreifen, kann es zu fehlerhaften Zuständen kommen. Führen zwei Tasks beispielsweise einen Schreibzugriff auf denselben Speicherbereich aus, ist der Wert dieses Bereiches anschließend nicht eindeutig. Der Abschnitt des Zugriffs, der nicht parallel durch einen anderen Task ausgeführt werden darf, wird als kritischer Abschnitt bezeichnet. Der gleichzeitige Zugriff kann bei Uniprozessorsystemen als quasi-paralleler Zugriff auftreten, wenn ein Task vorzeitig unterbrochen wird, der den Zugriff noch nicht beendet hat und der unterbrechende Task ebenfalls auf die Ressource zugreift. Bei Multiprozessorsystemen kann hingegen der echt-parallele Zugriff auftreten.

Den gleichzeitigen Zugriff auf gemeinsame Ressourcen gilt es zwingend zu verhindern. Das kann zum einen durch Schützen des kritischen Abschnitts durch einen Mechanismus wie den Mutex realisiert werden. Es existieren aber auch andere gängige Mechanismen, die bei der Softwareentwicklung für Uniprozessorsysteme Anwendung finden. Diese Mechanismen können für Multiprozessorsysteme ungeeignet sein und zu einem Fehlverhalten des Systems führen.

Bei Uniprozessorsystemen kann der Schutz von Ressourcen erfolgen, indem die Präemption, also das Unterbrechen eines Tasks durch einen anderen, sowie die Interrupts deaktiviert werden, bis der Zugriff des Tasks auf die Ressource abgeschlossen ist. Bei Multiprozessorsystemen würde dieser Mechanismus nicht zum Erfolg führen. Auch wenn der Task auf einem Prozessor nicht unterbrochen werden kann, kann ein Task auf einem anderen Prozessor trotzdem zeitgleich auf die selbe Ressource zugreifen.[9]

Auch ist das Verhalten des höchstprioren Tasks zu beachten. Bei Uniprozessorsystemen läuft der höchstpriore Task allein und muss unter bestimmten Umständen den Zugriff auf eine kritische Ressource nicht schützen, da kein Task mit einer höheren Priorität existiert, der ihn verdrängen könnte. Bei Multiprozessorsystemen läuft der höchstpriore Tasks nicht allein. Ein niedriger priorisierter Task kann auf einem anderen Prozessor ausgeführt werden und auf die gleiche Ressource zugreifen. Der höchstpriore Task muss den Zugriff auf eine Ressource also ebenfalls schützen.[9]

Ein weiterer Effekt, der beim Zugriff auf gemeinsame Ressourcen auftreten kann, ist die Prioritätsinversion. Dabei wird die Ausführung eines Tasks durch einen niedriger priorisierten Task verhindert. Bei Uniprozessorsystemen kann die Prioritätsinversion durch Prioritätsvererbung verhindert werden. Dazu wird einem Task, der auf eine Ressource zugreift, für die Zeit des Zugriffes eine höhere Priorität zugeordnet als die aller Tasks, die im Laufe ihrer Ausführung auf die gleiche Ressource zugreifen können. Bei Multiprozessorsystemen ist das Verhindern der Prioritätsinversion durch Prioritätsvererbung unter Einsatz globalen Schedulings möglich, bei partitioniertem Scheduling hingegen nicht. Werden mehrere Scheduler zur Verwaltung der Prozessoren verwendet und gleichzeitig eine Ressource von Tasks verschiedener Cluster genutzt, kennen die Scheduler nicht die Prioritäten der Tasks der anderen Scheduler. Es sind zusätzliche Mechanismen nötig, um bei partitioniertem Scheduling die Prioritätsinversion zu verhindern. So stellen für diesen Zweck konzipierte Protokolle zum Schutz globaler Ressourcen beispielsweise einen Mutex zur Verfügung, der für jede Schedulerinstanz eine eigene, für die Instanz höchste Priorität besitzt, die einem Task der Instanz bei Akquirieren des Mutex zugeordnet wird.[9]

3 Erweiterung von OUTPOST um RTEMS SMP

Dieses Kapitel behandelt die Erweiterung der Betriebssystemabstraktion in OUTPOST um die Multiprozessorunterstützung durch RTEMS. In Abschnitt 3.1 werden die Anforderungen an die Erweiterung in OUTPOST erläutert. In Abschnitt 3.2 wird anschließend die SMP-Unterstützung von RTEMS analysiert, um Änderungen gegenüber der Uniprozessor-Version darzulegen und Einflüsse auf aktuelle Implementierungen von OUTPOST und Flugsoftware abzuschätzen. Nach der Analyse wird in Abschnitt 3.3 die Implementierung der SMP-Unterstützung in OUTPOST erläutert und begründet.

3.1 Anforderungen

Zu Beginn werden Anforderungen an die Implementierung der Multiprozessorunterstützung in OUTPOST aufgestellt.

Tabelle 3.1: Anforderungen an die SMP-Unterstützung in OUTPOST

Nr.	Anforderung
A1	Die Abwärtskompatibilität für bestehende Flugsoftware basierend auf einem Prozessor muss gewährleistet sein.
A2	Über die OUTPOST-API soll eine Zuweisung von OUTPOST-Threads zu Prozessoren möglich sein.
A3	Es soll eine einheitliche, definierte Zuweisung von OUTPOST-Threads zu Prozessoren stattfinden, wenn keine Zuweisung über die API vorgenommen wird.
A4	Unterschiede zwischen der API von RTEMS in der Uniprozessor-Konfiguration und der SMP-Konfiguration sollen in OUTPOST berücksichtigt werden.

3.2 RTEMS SMP Analyse

Vor der Implementierung der Unterstützung symmetrischer Multiprozessorsysteme durch RTEMS in OUTPOST werden die Unterschiede zwischen der Uniprozessor- und der SMP-Konfiguration von RTEMS analysiert. Hauptaugenmerk liegt dabei auf Unterschieden, die für die Implementierung in OUTPOST und darauf basierende Flugsoftware relevant sind.

3.2.1 Scheduler

RTEMS stellt für das symmetrische Multiprocessing vier Scheduler zur Verfügung. Wie für Uniprozessorsysteme existieren der *Earliest Deadline First Scheduler* (*EDF*), der *Simple Priority Scheduler* und der *Deterministic Priority Scheduler*. Der *EDF-Scheduler* erlaubt die Zuweisung einer Deadline und einer Priorität zu einem Task. Das Einhalten der Deadlines ist beim Scheduling vorrangig. Wird einem Task keine Deadline zugeordnet, findet das Scheduling anhand der vergebenen Prioritäten statt. Der *Deterministic Priority Scheduler* und der *Simple Priority Scheduler* sind prioritätsbasierte Scheduler ohne Deadline, die sich in ihrer Implementierung unterscheiden. Der *Deterministic Priority Scheduler* garantiert dabei eine vorhersehbare und feste Ausführungszeit von Tasks. Die Namen der speziell auf Multiprozessorsysteme angepassten Varianten dieser Scheduler grenzen sich durch den Zusatz *_SMP* von denen der Scheduler für Uniprozessorsysteme ab.[9]

Zusätzlich steht in der SMP-Konfiguration der ebenfalls prioritätsbasierte *Arbitrary Processor Affinity Scheduler* zur Verfügung, der die Zuweisung von Affinitäten zu Tasks ermöglicht. Der *Constant Bandwith Server (CBS)* steht in der SMP-Konfiguration nicht zur Verfügung. Für Uniprozessorsysteme stellt dieser eine Erweiterung des *EDF-Schedulers* dar, die eine von anderen Tasks unabhängige Rechenzeit eines Tasks sicherstellen soll. [9]

Der Standard-Scheduler in der SMP-Konfiguration ist der *EDF-Scheduler* und in der Uniprozessor-Konfiguration der *Deterministic Priority Scheduler*.[9]

3.2.2 Zuweisen von Tasks zu Prozessoren

Die Zuweisung von Tasks zu Prozessoren kann auf zwei Wegen erfolgen. Zum einen kann einem RTEMS-Task eine Prozessor-Affinität zugewiesen werden. Zum anderen unterstützt RTEMS das Clustered Scheduling. Diese Methoden können ebenfalls kombiniert angewendet werden.

- **Prozessor-Affinität:** Eine RTEMS-basierte Anwendung besteht aus RTEMS-Tasks. Werden mehrere Prozessoren verwendet, kann einem Task eine Affinität zugewiesen werden. Die Affinität gibt an, auf welcher Teilmenge von Prozessoren ein Task ausgeführt werden kann. Wird einem Task keine Affinität zugewiesen, kann er standardmäßig auf allen Prozessoren ausgeführt werden. Der *EDF-Scheduler* und der *Arbitrary Processor Affinity Scheduler* unterstützen die Zuweisung von Task-Prozessor-Affinitäten. Der *EDF-Scheduler* ermöglicht die Zuweisung eines Tasks zu genau einem Prozessor (one-to-one) oder zu allen Prozessoren (one-to-all). Der *Arbitrary Processor Affinity Scheduler* ermöglicht die willkürliche Zuweisung einer Prozessor-Affinität zu einem Task. Die Zuweisung einer Prozessor-Task-Affinität kann vor dem Start eines Tasks oder während dessen Laufzeit stattfinden.[9]
- **Clustered Scheduling:** RTEMS unterstützt das in Unterabschnitt 2.5.4 beschriebene Clustered Scheduling. Durch das Clustered Scheduling kann die Zuweisung von Tasks zu Clustern, und somit zu Prozessoren, auch mit Schedulern erfolgen, die keine Affinitätszuweisung unterstützen.
- **Kombination von Clustered Scheduling und Affinität:** Bei Anwendung des Clustered Scheduling kann einem Task zusätzlich eine Affinität zugewiesen werden. Auf Prozessoren, denen ein Task durch eine Affinität zugewiesen ist, die aber nicht zur Menge der Prozessoren des Schedulers gehören, kann ein Task nicht ausgeführt werden. [9]

3.2.3 Schutz kritischer Sektionen in RTEMS SMP

Aufgrund der in Unterabschnitt 2.5.4 beschriebenen Unterschiede beim Schutz kritischer Sektionen bei Uniprozessor- und Multiprozessorsystemen weist RTEMS in der SMP-Konfiguration entsprechende Änderungen gegenüber der Uniprozessor-Konfiguration auf.

- **Deaktivieren von Präemption:** Das Deaktivieren der Präemption (*rtems_-NO_PREEMPT*) ist in RTEMS-SMP nicht erlaubt und führt zu Laufzeitfehlern. So wird verhindert, dass der Mechanismus genutzt wird, um kritische Sektionen zu schützen.[9]

- **Deaktivieren von Interrupts:** Auch das Deaktivieren von Interrupts dient bei Uniprozessorsystemen dem Schutz kritischer Sektionen, führt bei Multiprozessorsystemen aber nicht zum gewünschten Erfolg. Die Makros *rtems_interrupt_disable*, *rtems_interrupt_enable*, und *rtems_interrupt_flash* sind daher in RTEMS SMP deaktiviert und führen zu Warnungen während des Kompilierens und zu Fehlern während des Linkens. Wenn das Deaktivieren von Interrupts trotzdem nötig sein sollte, stehen die Macros *rtems_interrupt_local_disable* und *rtems_interrupt_local_enable* zur Verfügung. Diese deaktivieren bzw. aktivieren die Interrupts für den jeweiligen Prozessor. Zum Schutz der kritischen Sektionen von Threads und Interrupts müssen die Makros *rtems_interrupt_lock_acquire* / *rtems_interrupt_-lock_release* bzw. *rtems_interrupt_lock_acquire_isr* / *rtems_interrupt_lock_release_isr* genutzt werden.[9]

- **Prioritätsinversion:** Zum Vermeiden ungewollter Prioritätsinversion bei dem Clustered Scheduling kann das *Priority Ceiling Protocol* durch das *Multiprocessor Resource Sharing Protocol* (MrsP) ersetzt werden. Damit kann unter anderem die Prioritätsinversion bei zwischen mehreren Clustern geteilten Ressourcen verhindert werden. Bei dem *Priority Ceiling Protocol* verfügt ein Mutex über eine Prioritätsgrenze. Dem Besitzer des Mutex wird diese Prioritätsgrenze als vorübergehende Priorität zugeordnet. Das MrsP Protokoll macht diesen Ansatz für das Clustered Scheduling tauglich, indem einem Mutex für jede Scheduler-Instanz eine eigene Prioritätsgrenze zugeordnet wird, die einem Task der jeweiligen Instanz bei Aquirieren des Mutex vorübergehend zugewiesen wird. Das *Priority Inheritance Protocol* in der RTEMS SMP-Konfiguration kann beim Clustered Scheduling durch das *O(m) Independence-Preserving Protocol* (OMIP)[1] ersetzt werden. Das *Priority Inheritance Protocol* ordnet dem Besitzer eines Mutex eine höhere Priorität als der Tasks zu, die zur gleichen Zeit versuchen, auf den Mutex zuzugreifen. Das OMIP Protokoll verallgemeinert dieses Verfahren für das Clustered Scheduling. [9]

[1]Protokoll erhältlich ab RTEMS 5.1

3.3 Implementierung

Nach der Analyse von RTEMS SMP kann OUTPOST an die Mehrprozessorunterstützung angepasst werden. Die Zuweisung von Tasks zu Prozessoren muss implementiert und die Auswirkungen der genannten Unterschiede auf OUTPOST müssen berücksichtigt werden.

3.3.1 Zuweisung von Tasks zu Prozessoren

Zur Erweiterung der Betriebssystemabstraktion von RTEMS in OUTPOST um die Multiprozessorunterstützung stehen nun drei Optionen zur Auswahl: Die Zuweisung von Tasks zu Prozessoren kann über die Affinität, das Clustered Scheduling oder eine Kombination der Methoden erfolgen. Das Clustered Scheduling erfordert einen zusätzlichen Aufwand bei der Softwareentwicklung und der Anpassung bestehender Software auf den Multiprozessorbetrieb, da von den Clustern gemeinsam genutzte Ressourcen zusätzlich bzw. durch ein anderes Protokoll geschützt werden müssen. Zudem ist die Zuweisung der Tasks zu Prozessoren beim Clustered Scheduling bei der Verwendung von zwei Prozessoren (also partitioniertem Scheduling) statisch und es könnte keine Migration der Tasks stattfinden. In dieser Arbeit sollen in Kapitel 4 jedoch alle möglichen Zuweisungen von Tasks zu Prozessoren untersucht werden. Daher wird die Zuweisung von Tasks zu Prozessoren durch die Affinitätszuweisung umgesetzt.

Die Integration der Multiprozessorunterstützung in OUTPOST greift an dem Modul outpost-core an. In diesem Modul wird unter anderem das Echtzeitbetriebssystem RTEMS abstrahiert. Die RTEMS-Tasks werden im Kontext eines OUTPOST-Threads erstellt. Die Klasse der OUTPOST-Threads muss demnach angepasst werden, um die Threads, und damit die RTEMS-Tasks, über die OUTPOST-API einem Prozessor zuweisen zu können.

Die Zuweisung einer Affinität wird vorerst für die Nutzung von maximal zwei Prozessoren ausgelegt. Über die Enumeration *ProcessorAffinity* werden vier Konfigurationsmöglichkeiten für die Affinität eines Threads zur Verfügung gestellt:

- **cpu0**: Der Task kann nur auf der ersten CPU ausgeführt werden.
- **cpu1**: Der Task kann nur auf der zweiten CPU ausgeführt werden.
- **cpu0cpu1**: Der Task kann auf der ersten und der zweiten CPU ausgeführt werden.

- **inheritAffinity**: Der neue Task erbt die Affinität des Tasks, der den neuen Task startet.

Die Affinität soll einem Thread bei der Instanziierung zugewiesen werden können. Die Zuweisung der Affinität soll optional sein, damit bestehende Uniprozessor-Software weiterhin Threads ohne Affinität-Angabe erstellen können und nicht angepasst werden müssen. Der Konstruktor der Klasse *Thread* wird dazu um den Parameter *affinity* des Typs *ProcessorAffinity* ergänzt. Da sich dieser am Ende der Parameterliste des Konstruktors befindet, kann er optional angegeben werden. Als Standard-Wert wird dem Parameter *affinity* der Wert *inheritAffinity* zugewiesen. So kann gewährleistet werden, dass bei OUTPOST-internen Thread-Instanziierungen eine definierte Affinität der Threads vorliegt und über die OUTPOST-API direkt oder indirekt Einfluss auf die Affinität jedes Threads genommen werden kann.

Die *ProcessorAffinity*-Elemente *cpu0*, *cpu1* und *cpu0cpu1* werden direkt im Konstruktor der Klasse *Thread* ausgewertet (Abbildung 3.1). Die Auswertung wird nur eingebunden, wenn das Makro *RTEMS_SMP* durch das SMP-BSP definiert ist. In der Uniprozessor-Konfiguration hat die Affinität keine Auswirkung auf die Tasks. Die Variable *cpuset* wird je nach vergebener Affinität gesetzt (Abbildung 3.1, Z. 4-14) und anschließend dem Task zugewiesen (Abbildung 3.1, Z. 15).

```
1  #ifdef RTEMS_SMP
2      cpu_set_t cpuset;   //CPU set to which a task can be assigned
3      CPU_ZERO(&cpuset);  //removes all CPUs from cpuset
4      if (mAffinity == cpu0)
5      {
6          CPU_SET(0, &cpuset);
7      }else if (mAffinity == cpu1)
8      {
9          CPU_SET(1, &cpuset);
10     }else if (mAffinity == cpu0cpu1)
11     {
12         CPU_SET(0, &cpuset);
13         CPU_SET(1, &cpuset);
14     }
15     status = rtems_task_set_affinity(mTid, sizeof(cpuset), &cpuset);
16 #endif
```

Abbildung 3.1: Auswertung der Affinität in dem Konstruktor des OUTPOST-Threads

Das Element *inheritAffinity*, also die Vererbung der Affinität, wird erst beim Start des Threads ausgewertet, da der Konstruktor der Klasse im Kontext eines anderen Threads

aufgerufen werden kann, als der Kontext, in dem der Thread gestartet werden soll. Beispiel siehe Abbildung 3.2: Der MainThread instanziiert ThreadA. Im Konstruktor von ThreadA wird ThreadB instanziiert. Der Konstruktor des ThreadA und dadurch ebenfalls der Konstruktor des ThreadB werden im Kontext einer MainThread-Instanz aufgerufen. Würde die Affinitätsvererbung im Konstruktor ausgewertet werden, würde der Thread B1 die Affinität der MainThread-Instanz erhalten. Der Start des Threads B1 kann hingegen im Kontext von Klasse A1 ausgeführt werden, sodass der Thread B1 die Affinität des Threads A1 erbt.

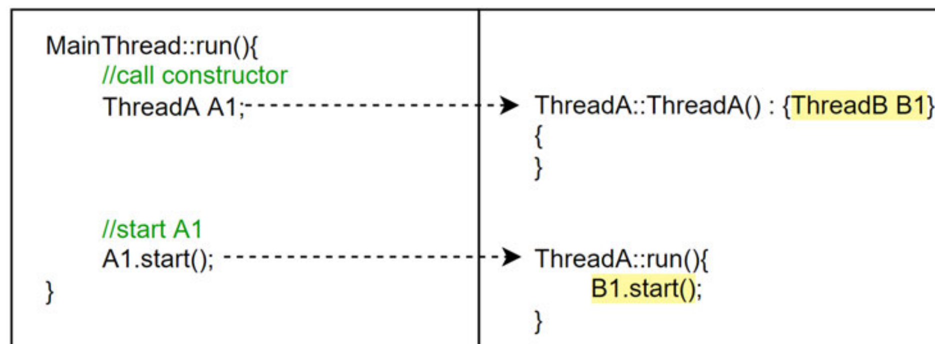


Abbildung 3.2: Vererbung der Thread-Affinität

Abbildung 3.3 zeigt die Umsetzung der Affinitätsvererbung in der run-Methode des OUTPOST-Threads. Wie bei der Umsetzung im Konstruktur wird die Affinität der Variable *cpuset* zugewiesen. Zur Vererbung der Affinität wird die Affinität des Tasks, der die run-Methode des Tasks aufruft, dem zu startenden Task zugewiesen (Abbildung 3.3, Z.8-11).

```
1  #ifdef RTEMS_SMP
2   if (mAffinity == inheritedFromCaller)
3      {
4          rtems_status_code status;
5          cpu_set_t cpuset;   //CPU set to which a task can be assigned
6          CPU_ZERO(&cpuset);  //removes all CPUs from cpuset
7          rtems_id pTid = rtems_task_self();
8          rtems_task_get_affinity(
9                  pTid,
10                 sizeof(cpuset),
11                 &cpuset);
12         status = rtems_task_set_affinity(mTid, sizeof(cpuset), &cpuset);
13     }
14 #endif
```

Abbildung 3.3: Affinitätsvererbung in der run-Methode des OUTPOST-Threads

Um nach der Instanziierung eines Threads dessen Affinität ändern zu können, wird der Thread-Klasse die Methode *setAffinity* hinzugefügt. So kann auch OUTPOST-Klassen, die vom OUTPOST-Thread erben, und die in dieser Arbeit nicht berücksichtigt wurden, eine Affinität zugewiesen werden. Zudem wird die Methode *getAffinity* zum Abfragen der aktuellen Affinität ergänzt.

Affinitätszuweisung für Timer
Neben dem RTEMS-Task, der in OUTPOST durch die Thread-Klasse abstrahiert wird, existiert der RTEMS-Timer. Dieser wird in OUTPOST durch eine Timer-Klasse abstrahiert. Einem RTEMS-Timer kann wie einem Task eine Affinität zugewiesen werden. Die Zuweisung der OUTPOST-Timer-Affinität wird in der Timer-Methode *startTimerDaemonThread* implementiert. Die Parameterliste der Methode wird um die Affinität erweitert und diese nach der Initialisierung des RTEMS-Timer-Servers ausgewertet.

Affinitätszuweisung für den *DataProcessorThread*
Für das in Kapitel 4 beschriebene Benchmark wird die Klasse *DataProcessorThread* benötigt. Um diesem bei der Instanziierung eine Affinität zuweisen zu können, wird der Konstruktor um den Affinitäts-Parameter ergänzt.

3.3.2 Weitere Anpassungen

Die Wahl des Schedulers ist applikationsspezifisch und erfordert somit keine Anpassung von OUTPOST. Die Funktionen zum Deaktivieren von Präemption und Interrupts werden im für diese Arbeit genutzten Teil von OUTPOST nicht verwendet. Das Protokoll zur Prioritätsvererbung kann weiterhin genutzt werden. Eine Anpassung auf das OMIP-Protokoll wäre nur bei Clustered Scheduling nötig.

4 Benchmark-Applikation zur Leistungsanalyse

Dieses Kapitel behandelt die Entwicklung eines Benchmarks zur Leistungsanalyse der im vorherigen Kapitel beschriebenen Betriebssystemabstraktion zur Multiprozessorunterstützung von OUTPOST. In Abschnitt 4.1 werden zunächst geeignete Kriterien zur Leistungsanalyse definiert und in Abschnitt 4.2 die Anforderungen an den Benchmark aufgestellt. Abschnitt 4.3 beschreibt die Hard- und Softwareumgebung des Benchmarks. Abschnitt 4.4 stellt Methoden vor, mit denen die zuvor aufgestellten Leistungskriterien ermittelt werden können. Abschnitt 4.5 beschreibt die Architektur und das Verhalten des Benchmarks. In Abschnitt 4.6 werden die Testszenarien erläutert, die mit Hilfe des Benchmarks durchgeführt werden können.

4.1 Definition der Leistungskriterien

Die Leistungsanalyse der Multiprozessorunterstützung von OUTPOST soll anhand von drei Leistungskriterien durchgeführt werden.

- **Durchsatz:** Zum einen soll der Durchsatz des Benchmarks gemessen werden, um den Speedup beim Nutzen des zweiten Prozessors berechnen zu können.
- **Auslastung des ersten Prozessors:** Gleichzeitig soll die Auslastung des ersten Prozessors in der Uniprozessor- und der Multiprozessorkonfiguration gemessen werden, um zu zeigen, dass die Aufteilung der Aufgaben auf zwei Prozessoren den ersten Prozessor entlastet.

4.2 Anforderungen

Zum Messen der Leistungskriterien soll die Benchmark-Applikation folgende Anforderungen erfüllen:

Tabelle 4.1: Anforderungen an die Benchmark-Applikation

Nr.	**Anforderung**
B1	Der Benchmark soll die SMP-Unterstützung von RTEMS verwenden.
B2	Der Benchmark soll auf OUTPOST basieren.
B3	Der Benchmark soll die SMP-Unterstützung von OUTPOST nutzen.
B4	Der Benchmark soll die Auslastung des ersten Prozessors messen bzw. ausgeben.
B5	Der Benchmark soll den Durchsatz des Gesamtsystems messen bzw. ausgeben.
B6	Die Struktur des Benchmarks soll der einer realen Flugsoftware möglichst ähnlich sein.
B7	Der Benchmark soll in der Uniprozessor-Konfiguration und der Multiprozessor-Konfiguration ausführbar sein.
B8	Der Benchmark soll in der Uniprozessor-Konfiguration und der Multiprozessor-Konfiguration ein vergleichbares Verhalten aufweisen.
B9	Der Benchmark soll modular aufgebaut sein und verschiedene Szenarien bezüglich der Verteilung der Aufgaben auf die Prozessoren ermöglichen.

4.3 Hard- und Softwareumgebung

Die Hard- und Softwareumgebung zur Entwicklung des Benchmarks entspricht der, die beim DLR für Projekte wie dem Eu:CROPIS Satelliten eingesetzt wird (Abbildung 4.1).

Der Entwicklungsrechner läuft mit Windows10 (64 Bit) als Host-Betriebssystem. Mit Hilfe der Virtualisierungssoftware VirtualBox wird das Gast-Betriebssystem Debian10 (32Bit) als virtuelle Maschine bereitgestellt. Die im Rahmen dieser Arbeit durchgeführten Schritte auf dem Entwicklungsrechner finden ausschließlich innerhalb der virtuellen Maschine statt. Als Build Tool wird das Open Source Software Construction Tool *SCons* genutzt. Dieses wird über die vom DLR entwickelten *scons-build-tools* bereitgestellt, die bereits Konfigurationsdateien für bestimmte Hardware-Plattformen wie das GR712RC

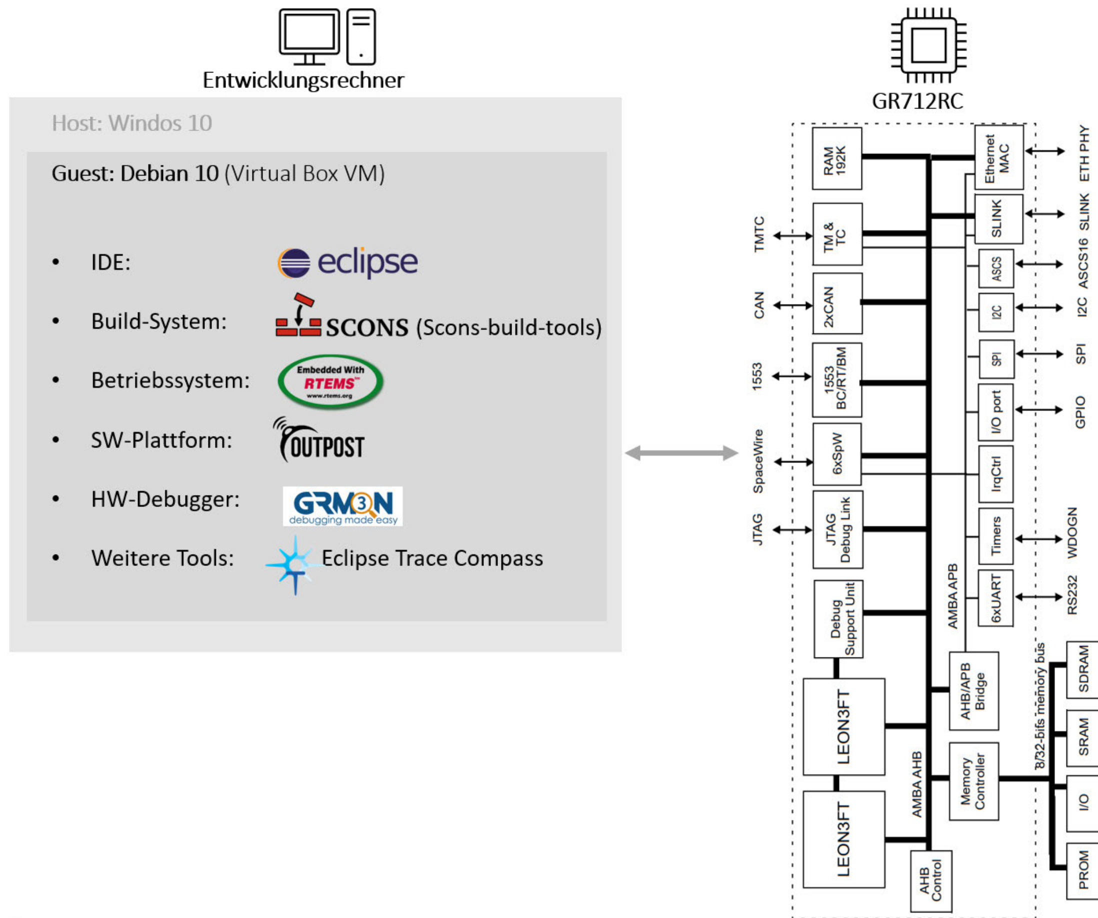


Abbildung 4.1: Software- und Hardwareumgebung [4]

enthalten. Die Entwicklung der Software findet mit Hilfe der integrierten Entwicklungsumgebung *Eclipse* statt. Die Entwicklung wird dabei durch den Debug-Monitor GRMON für die LEON3 Prozessoren unterstützt. Zur Leistungsanalyse wird zusätzlich der Eclipse Trace Compass genutzt.

Die Ziel-Hardware stellt das SoC GR712RC mit den zwei LEON3 Prozessoren dar. Die Software für das GR712RC basiert auf dem Echtzeitbetriebssystem RTEMS und der Softwareplattform OUTPOST.

4.4 Methoden zur Leistungsanalyse

Für die Leistungsanalyse der SMP-Konfiguration werden die Auslastung und der Durchsatz untersucht. Im Folgenden werden die Methoden zum Aufnehmen der Messwerte erläutert. Zur Ausgabe der Messwerte über das Eclipse-Terminal ist dabei die Konfiguration von GRMON, die eine Ausgabe über die Debug-Schnittstelle erlaubt, vorausgesetzt(*grmon -ftdi -ucli -gdb*).

4.4.1 Durchsatz

Zur Messung des Durchsatzes soll eine Softwarekomponente dienen, die mit bestimmter Frequenz Daten verarbeitet. Die Menge der verarbeiteten Daten stellt den Durchsatz des Benchmarks dar. Die Softwarekomponente wird in Abschnitt 4.5 beschrieben.

4.4.2 Auslastung

Zur Auslastung des Prozessors werden verschiedene Methoden genutzt, da diese jeweils ihre Vor- und Nachteile mit sich bringen. Die Vor- und Nachteile werden bei der Anwendung der Methoden in Kapitel 5 erläutert.

Memory Scrubbing

Beim Memory Scrubbing wird nacheinander jede Adresse eines Speichers gelesen, geprüft und bei Auftreten eines Fehlers korrigiert. Der Memory Scrubber kann dabei als Hardwarekomponente auf dem Chip eines Systems vorliegen oder als Softwarekomponente im Hintergrund einer Software laufen. In dieser Arbeit wird zur Messung der Auslastung ein Software Memory Scrubber verwendet. Dieser läuft als Thread mit niedrigster Priorität im Hintergrund der Software. Er ist also immer dann aktiv, wenn der Prozessor ohne den Memory Scrubber keine Aufgabe abzuarbeiten hätte und sich somit im Idle-Zustand befinden würde. In einem Zeitfenster läuft der Memory Scrubber über eine bestimmte Anzahl an Speicher-Blöcken. Anhand der Anzahl der Blöcke, die der Memory Scrubber in einem Zeitraum überprüft, kann ein Rückschluss darauf gezogen werden, wie lange sich der Prozessor ohne den Memory Scrubber im Idle-Zusand befinden würde und somit wie hoch die Auslastung des Prozessors wäre. Die Anzahl der Blöcke kann über die Konsole ausgegeben werden.

RTEMS CPU Usage Statistics

Der CPU Usage Statistics Manager von RTEMS ermöglicht es, den prozentualen und absoluten zeitlichen Anteil eines Threads an der Gesamtlaufzeit der Prozessoren anzuzeigen. Um die CPU Usage Statistic auszugeben, muss die entsprechende Datei *<rtems/cpuuse.h>* eingebunden sein und die Funktion *rtems_cpu_usage_report()* aufgerufen werden. Die Funktion *rtems_cpu_usage_reset()* dient zum Zurücksetzen der Statistik.

RTEMS Tracing Framework

Mit Hilfe des RTEMS Tracing Frameworks kann das Verhalten von Software während oder nach der Laufzeit analysiert werden. Das Framework unterstützt das Event Recording, mit dem hochfrequente Events, wie beispielsweise der Wechsel zwischen Threads, aufgezeichnet werden können. Die aufgezeichneten Events können entweder während der Laufzeit einer Software über eine TCP-Schnittstelle an einen Host Computer übertragen werden oder in einem Buffer auf der Target-Hardware gespeichert und zu einem späteren Zeitpunkt ausgelesen werden.

Das Event Recording wird in der RTEMS System-Konfiguration aktiviert (siehe Unterabschnitt 4.5.3). Zum Speichern der Events wird pro Prozessor ein Buffer einer bestimmten Größe definiert. Ist das Event Recording aktiviert, führt RTEMS auf der Ziel-Hardware während der Laufzeit beim Auftreten von Events Funktionen aus, um Informationen zu den Events zu speichern. Die aufgezeichneten Events liegen innerhalb der reservierten Buffer als Binärdaten vor. Für dieses Projekt werden die aufgezeichneten Events am Ende eines Testlaufes ausgegeben, um den Test während der Laufzeit möglichst wenig zu beeinflussen. Der Inhalt der Event-Buffer wird durch die Funktion *rtems_record_dump_base64(rtems_put_char, NULL)*[1] in der Base64-Codierung[2] ausgegeben. Der über das Terminal ausgegebene Text wird anschließend manuell in eine Text-Datei kopiert. Mit Hilfe des *Linux Trace Toolkit: next generation* (LTTng) können über den Kommandozeilenbefehl *rtems-record-lttng -e <application>.elf -b <trace-records>.txt* die im Base64 Format in der Text-Datei vorliegenden Events in das Common Trace Format (CTF) übersetzt werden. Dabei wird eine Metadaten-Datei und pro CPU eine CTF-Datei erzeugt. Diese Dateien können in den Eclipse Trace Compass importiert und als Grafik dargestellt

[1] Einbinden folgender Dateien nötig: rtems/record.h, rtems/recorddump.h, rtems/score/io.h, rtems/bspIo.h

[2] Die Base64-Codierung stellt Binärdaten als ASCII-Zeichenfolgen dar.

werden. Der Eclipse Trace Compass bietet mehrere Darstellungsformen zur Analyse der aufgezeichneten Events. Unter Anderem können die CPU-Auslastung und die Anteile eines Threads an der Prozessorlaufzeit prozentual dargestellt werden. Auf diese Weise kann die Auslastung der Prozessoren mit Hilfe des Event-Recordings über den Eclipse Trace Compass ermittelt werden.

4.5 Softwarearchitektur

Im Folgenden werden das Datei-System des Benchmarks, die Konfiguration des Build-Prozesses, die RTEMS System-Konfiguration und das Verhalten des Benchmarks beschrieben.

4.5.1 Dateisystem

Das Dateisystem des Benchmarks besteht zum einen aus externen Dateien wie der OUTPOST-Plattform (Ordner *ext*) und zum anderen aus den applikationsspeziefischen Softwarekomponenten, die in diesem Projekt entwickelt werden (Ordner *smp_example*) (Abbildung 4.2). Der Ordner *ext* enthält neben outpost-core, outpost-sattelite und

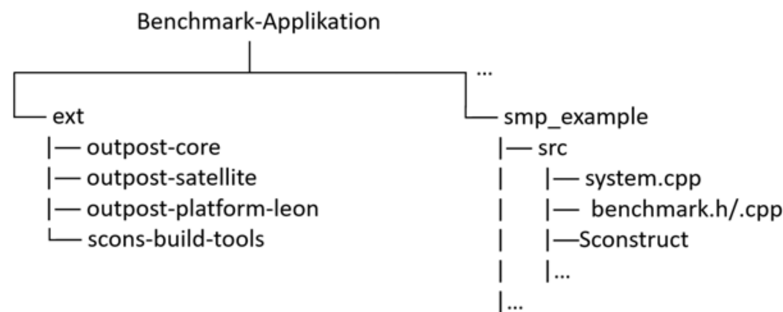


Abbildung 4.2: Ordnerstruktur des Benchmarks

outpost-platform-leon zudem die scons-build-tools. Der applikationsspezifische Softwareteil enthält Dateien für den Benchmark-Code und Konfigurationsdateien für den Build-Prozess.

4.5.2 Build-Prozess Konfiguration

Der Build-Prozess des Benchmarks wird über die SConstruct Datei beschrieben. Um die zertifizierte SMP-Fähigkeit von RTEMS nutzen zu können, müssen die Version RTEMS5 und das zur Ziel-Hardware passende BSP (*gr712rc_smp*) eingebunden werden.

4.5.3 RTEMS Systemkonfiguration

Die RTEMS System-Konfiguration wird in der Datei *system.cpp* vorgenommen. Dazu gehören folgende Einstellungen:

- **Scheduler:** Es wird kein Scheduler definiert, sodass der Standard-Scheduler zum Einsatz kommt. In der Uniprozessor-Konfiguration wird somit der *Deterministic Priority Scheduler* und in der Multiprozessorkonfiguration der Affinitäten-unterstützende *EDF-Scheduler* verwendet. Den Tasks des Benchmarks wird keine Deadline zugewiesen, sodass sich der *EDF-Scheduler* wie ein prioritätsbasierter Scheduler verhält [9].
- **Anzahl der Prozessoren:** Über das Macro CONFIGURE_MAXIMUM_PROCESSORS wird die Anzahl der Prozessoren definiert, die die Applikation maximal verwenden soll. Wird die RTEMS Uniprozessor-Konfiguration verwendet, wird das Makro ignoriert[9].
- **Event Recording** (optional)**:** Das Event Recording wird über das Makro CONFIGURE_RECORD_EXTENSIONS_ENABLED aktiviert und der Buffer zum Speichern der aufgezeichneten Events über das Makro CONFIGURE_RECORD_-PER_PROCESSOR_ITEMS auf die Größe $1024 \cdot 4$ gesetzt.
- **Tick-Länge**: Die Länge eines Ticks definiert die Auflösung aller zeitabhängigen RTEMS-Aktionen [9]. Die Tick-Länge wird über das Makro CONFIGURE_MICROSECONDS_PER_TICK auf 500μs festgelegt.
- **Time-Slice:** Ein Time-Slice gibt an, wie lange ein Task aktiv sein darf, wenn ein Task gleicher Prioriät ausführbereit ist. Das Timeslicing ist in OUTPOST standardmäßig aktiviert. Die Länge eines Time-Slices wird durch das Makro CONFIGURE_TICKS_PER_TIMESLICE auf 5 Ticks gesetzt. Ein Task wird also nach 2,5ms unterbrochen, wenn ein Task gleicher Priorität ausführbereit ist.

4.5.4 Verhalten der Software und Klassen

Mit Hilfe des Benchmarks sollen die Unterschiede der definierten Leistungskriterien zwischen Ausführung einer Applikation auf einem und auf zwei Prozessoren untersucht werden. Ein Ziel ist es, anhand der durch den Benchmark aufgenommenen Messergebnisse eine Beurteilung vornehmen zu können, wie eine reale Flugsoftware auf zwei Prozessoren aufgeteilt werden kann. Wichtig ist daher ein modularer Aufbau des Benchmarks, der im Multiprozessorbetrieb verschiedene Szenarien ermöglicht, die sich in der Verteilung der Aufgaben auf die Prozessoren unterscheiden.

Im Wesentlichen besteht die Benchmark-Applikation aus drei Teilen: Ein Teil stellt die Flugsoftware dar, die in realen Anwendungsfällen auf dem OBC ausgeführt wird. Davon abgekoppelt wird für dieses Benchmark die Datenverarbeitung, welche somit den zweiten Teil bildet. Die Datenverarbeitung ist die Softwarekomonente, deren Eigenschaften in diesem Benchmark in verschiedenen Testszenarien variiert werden. Den dritten Teil bilden die Softwarekomponenten, die für die Messungen und zur Analyse des Benchmarks nötig sind. Abbildung 4.3 veranschaulicht anhand eines vereinfachten Sequenzdiagramms das Verhatlen der Benchmark-Applikation mit den Softarekomponenten, die im Folgenden beschrieben werden. Dargestellt sind eine von zwei Datenverarbeitungseinheiten und der TestingThread. Die FlightsoftwareDummyThreads und der MemoryScrubber, welche kontinuierlich parallel zu dem dargestellten Verhalten ausgeführt werden können, sind nicht abgebildet.

Teil 1: Die Flugsoftware

Die Flugsoftware wird vereinfacht durch eine Klasse dargestellt. Relevant ist hier nicht das Verhalten der Flugsoftware, sondern die Inanspruchnahme des Prozessors.

FlightsoftwareDummyThread (FDT): Der FlightsoftwareDummyThread ersetzt in der Benchmark-Applikation den Last-Anteil der Flugsoftware und dient ausschließlich der Beschäftigung der CPU. Um dabei auch den gemeinsamen Bus zum Hauptspeicher zu belasten, wie es eine reale Flugsoftware täte, führt ein aktiver FDT regelmäßig Schreibvorgänge aus. Ein FDT soll einen bestimmten Anteil der CPU-Zeit beanspruchen. Zur Umsetzung wechselt der FDT zwischen aktiven und inaktiven Phasen. Die Dauer der

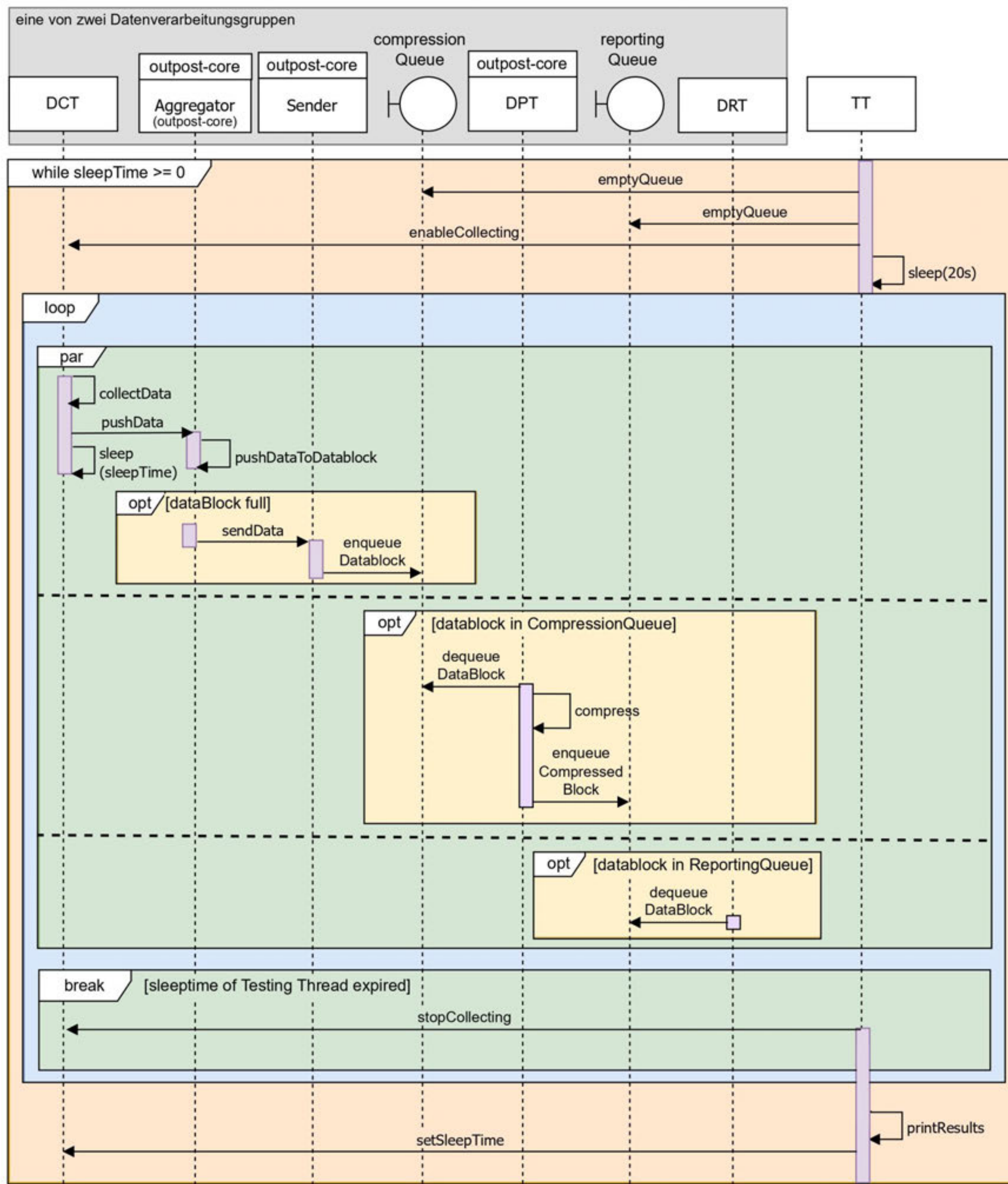


Abbildung 4.3: Vereinfachtes Sequenzdiagramm des Benchmarks

inaktiven Phase wird auf 60ms festgelegt. Die Dauer der aktiven Zeit wird in ein entsprechendes Verhältnis gesetzt. Das Verhältnis von aktiver zu inaktiver Phase ist bei der Instanziierung des FDT anzugeben.

Der FlightsoftwareDummyThread wird dreifach instanziiert. Dabei werden den Instanzen, in Anlehnung an eine reale Flugsoftware mit vielen, unterschiedlich wichtigen Aufgaben, verschiedene Prioritäten zugewiesen und der prozentuale Anteil der CPU-Zeit, den ein FDT maximal in Anspruch nehmen soll, definiert.

Teil 2: Datenverarbeitung

Die Datenverarbeitung orientiert sich an der Datenverarbeitung von bestehender Flugsoftware, wie sie auch in der Software der Eu:CROPIS zur Anwendung kommt. Die Datenverarbeitung besteht dabei aus dem Messen von Daten, deren Kompression und deren Weiterleitung. Statt Daten zu messen, werden in der Benchmark-Applikation durch einen Thread zufällige Daten generiert, um den Benchmark unabhängig von Peripherie zu halten. Zur Kompression der Daten wird eine in OUTPOST bereits implementierte Datenkompression genutzt. Eine Weiterleitung der komprimierten Daten, wie das Senden zu einer Bodenstation, findet in diesem Fall nicht statt. Die Daten werden lediglich durch einen Thread empfangen und verworfen, um den Buffer zu leeren und das Speichern weiterer Daten zu ermöglichen.

Die im Folgenden beschriebenen Threads zum Sammeln, Komprimieren und Empfangen der komprimierten Daten werden jeweils zweifach instanziiert. Dabei bilden je ein Thread jedes Datenverarbeitungsschrittes eine Datenverarbeitungseinheit. Jeder Datenverarbeitungseinheit wird eine eigene Queue für die Rohdaten (compressionQueue), eine Queue für die komprimierten Daten (reportingQueue) sowie eine Affinität zugewiesen.

DataCollectingThread (DCT): Der DataCollectingThread generiert mit einer bestimmten Frequenz zufällige Werte, um die Messung von beispielsweise Spannungswerten zu simulieren. Die anschließende Handhabung der Daten ist durch die in OUTPOST implementierte Datenkompression vorgegeben. Der DCT instanziiert dafür einen DataAggregator und einen oneTimeQueueSender. Nach dem Generieren eines Wertes wird dieser einem DataAggregator übergeben. Der Aggregator reserviert einen Bereich des MemoryPools und legt einen DataBlock an. In diesem Block wird das generierte Datum gespeichert. Die Größe eines Datenblocks wird für den Benchmark auf 16 Daten festgelegt. Ist ein Datenblock voll, sendet der Aggregator ihn an den OneTimeQueueSender. Der Sender schiebt den Datenblock in eine Queue (compressionQueue). Nachdem ein Datum generiert und in den Aggregator geschoben wurde, wechselt der DCT für eine

bestimmte Zeit, die SleepTime, in den suspendet state und beginnt anschließend die generierung des nächsten Datums. Der DCT bietet Methoden, um die SleepTime zu ändern und das Sammeln von Daten zu starten und zu stoppen.

DataprocessingThread (DPT): Der DataProcessingThread gehört zum *compression*-Modul von outpost-core. Ein DataProcessorThread empfängt Datenblöcke aus einer ihm bei der Instanziierung zugewiesenen CompressionQueue. Ein empfangener Datenblock wird mit Hilfe der Wavelet-Transformation komprimiert und in eine weitere, ihm zugewiesene ReportingQueue geschoben.

DataReceivingThread (DRT): Der DataReceivingThread empfängt komprimierte Datenblöcke aus einer ReportingQueue und verhindert somit ein Überlaufen der ReportingQueue.

Teil 3: Messungen und Analyse

Zum Durchführen und Ausgeben der Messungen werden drei Threads eingeführt: der MemoryScrubber, der OccupyCpuThread und der TestingThread. Das Event-Recording und die CPU Usage Statistics sind bereits in RTEMS enthalten und müssen lediglich eingebunden und entsprechende Funktionen aufgerufen werden, es werden also keine zusätzlichen Threads benötigt. Die Threads für Messungen und Analyse des Benchmarks werden allesamt auf der ersten CPU ausgeführt.

MemoryScrubber (MSCR): Der MemoryScrubber wird aus der Eu:CROPIS-Software übernommen und auf die Benchmark-Applikation angepasst. Dafür wird die Affinität des MemoryScrubber-Threads eingefügt und so gewählt, dass der MemoryScrubber auf der ersten CPU ausgeführt wird. Die Blockgröße, also die Anzahl an zu einem Block gehörenden Speicheradressen, wird herabgesenkt, um die Genauigkeit der Aussage über die Auslastung zu erhöhen, die aus der Anzahl an im Zeitfenster geprüften Blöcken hergeleitet werden kann.

OccupyCpuThread (OCT): Der OCT kommt beim Aufnehmen von Messergebnissen mit Hilfe der CPU Usage Statistics zum Einsatz. Die Ausgabe der Statistik nach Beenden eines Testszenarios über ein Terminal dauert mehrere Sekunden. Unter der Nutzung beider Prozessoren kann der zweite Prozessor weiterhin Aufgaben ausführen oder sich im Idle-Zustand befinden, während der erste Prozessor mit der Ausgabe der Statistik beschäftigt ist. Die Aktivitäten des zweiten Prozessors während dieser Zeit fließen in

die Statistik mit ein. Um zwischen den Aktivitäten des zweiten Prozessors während des Testlaufes und denen danach differenzieren zu können, wird der OCT nach Beenden eines Testlaufes auf der zweiten CPU ausgeführt. Der OCT führt eine leere Verzögerungsschleife aus, bis er gestoppt wird. Der OCT wird nur in der SMP-Konfiguration eingebunden, da er in der Uniprozessor-Konfiguration die erste CPU blockieren würde.

TestingThread (TT): Der TestingThread unterstützt die Analyse der Benchmark-Applikation. Er startet und beendet die Messungen eines Testszenarios. Die Messungen eines Testszenarios bestehen dabei aus einzelnen Teildurchläufen, bei denen jeweils die SleepTime der DCTs verändert wird.

Der TestingThread wird mit höchster Priorität auf der ersten CPU ausgeführt. Zum Vorbereiten eines Teildurchlaufes leert der TT alle in der Applikation genutzten Queues. Durch Aktivieren der Datensammlung durch die DCTs und anschließendes Wechseln in den Suspended-Zustand für eine bestimmte Zeit startet der TestingThread einen Teildurchlauf und überlässt den FDTs, den Datenverarbeitungs-Threads und gegebenenfalls dem MemoryScrubber die CPU. Nach Ablauf der Zeit wird der TestingThread wieder aktiv und beendet einen Teildurchlauf des Testszenarios. Je nachdem, welche Messwerte durch welche Methode aufgenommen werden sollen, gibt er die Anzahl der vom MemoryScrubber geprüften Blöcke, die Anzahl der von den DataProcessingThreads verarbeiteten Daten, die durch das Event-Recording aufgezeichneten Daten oder die CPU Usage statistics aus. Zusätzlich wird die im Teildurchlauf genutzte DCT-SleepTime ausgegeben. Nach abgeschlossener Ausgabe der Messergebnisse ändert der TestingThread die SleepTime, die der DCT nach Sammeln eines Datums im Suspendet-Zustand verbringt. Anschließend beginnt der TestingThread die Vorbereitung des nächsten Teildurchlaufes für das Testszenario.

Relevante Parameter, die innerhalb des TestingThreads festgelegt sind, sind die Dauer eines Teildurchlaufes eines Szenarios und die SleepTime der DCTs. Für die Laufzeit eines Teildurchlaufes wird eine Dauer von 20 Sekunden gewählt. Diese Zeitspanne ist groß genug, um das Messen eines aussagekräftigen Durchsatzes auch bei hoher SleepTime zu ermöglichen und hält dabei trotzdem den zeitlichen Messaufwand möglichst gering. Die SleepTime wird, beginnend mit einer SleepTime von einer Sekunde, für jeden Teildurchlauf verringert. Bei kleiner werdender SleepTime wird die Differenz zur vorherigen SleepTime geringer, um die Genauigkeit der Messergebniskurve eines Szenarios in diesem Bereich zu steigern. Die SleepTime wird in diesen Bereichen zusätzlich abhängig von der RTEMS-Tick-Länge gewählt (vgl. Abschnitt 5.2).

4.6 Testszenarien

Zur Leistungsanalyse des Multiprozessorbetriebs werden verschiedene Szenarien betrachtet, bei denen die Affinitäten der datenverarbeitenden Thread-Gruppen variiert werden (Abbildung 4.4). Allen Szenarien gemein ist die Affinität der FDTs, des TestingThreads und des MemoryScrubbers zu CPU0. Die Threads sind von oben nach unten mit absteigender Priorität[3] angeordnet.

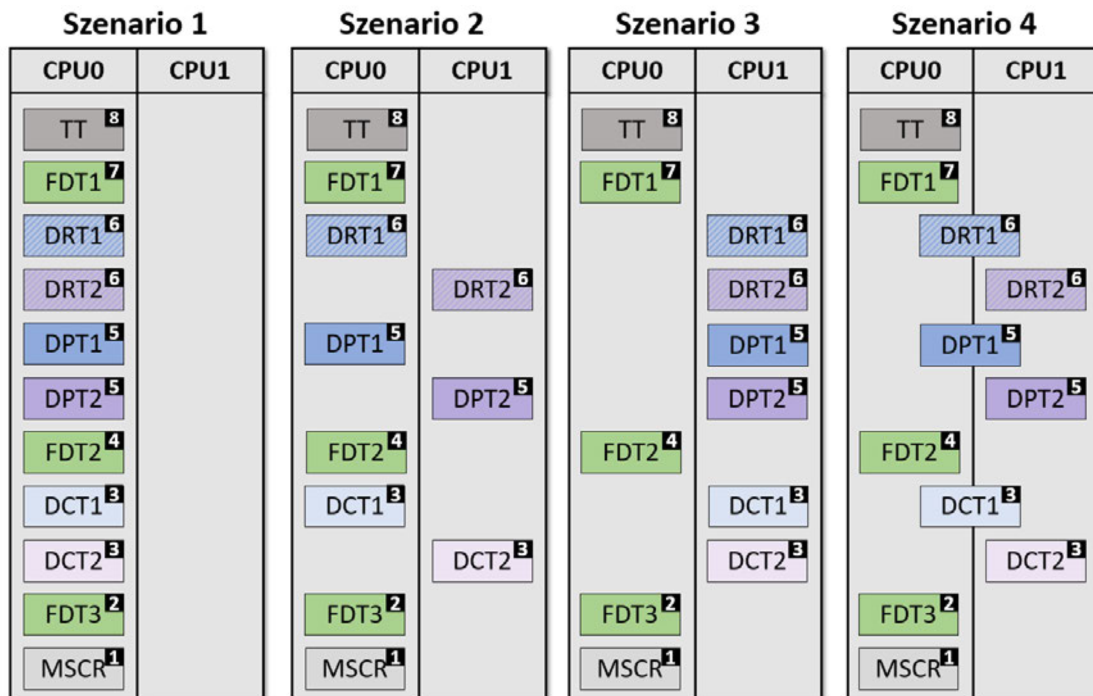


Abbildung 4.4: Affinitäten der Threads in den Testszenarien

- **Szenario 1:** In Szenario 1 werden alle Threads auf der ersten CPU ausgeführt. Dieses Szenario kann mit der RTEMS SMP-Konfiguration, aber auch mit der Uniprozessor-Konfiguration, also deaktivierter CPU1, ausgeführt werden.
- **Szenario 2:** In Szenario 2 wird die zweite datenverarbeitende Thread-Gruppe auf die CPU1 ausgelagert. Die erste Gruppe wird weiterhin auf der CPU0 ausgeführt.

[3]Hier werden die OUTPOST-Prioritäten verwendet: 1→ niedrigste Priorität, 255→ höchste Priorität. Die Prioritäten in RTEMS werden gegenläufig gewertet: 1→ höchste Priorität, 255→ niedrigste Priorität.

- **Szenario 3:** In Szenario 3 wird ebenfalls die erste Gruppe auf CPU1 ausgelagert. Auf der CPU0 laufen während eines Szenarios somit nur die FDTs und gegebenenfalls der MemoryScrubber.
- **Szenario 4:** In Szenario 4 wird die zweite Gruppe weiterhin auf CPU1 ausgeführt. Die erste Gruppe kann hingegen auf beiden Prozessoren ausgeführt werden, sodass der Scheduler während der Laufzeit dynamisch die Threads einer CPU zuweist.

5 Messungen und Leistungsanalyse

In diesem Kapitel werden die mit Hilfe des Bencharks aufgenommenen Messungen der Auslastung und des Durchsatzes in den verschiedenen Szenarien analysiert. In Abschnitt 5.1 wird zunächst eine Prognose der Ergebnisse der Leistungsmessungen in den Szenarien aufgestellt. Abschnitt 5.2 erläutert die Darstellung der Messergebnisse. In Abschnitt 5.3 werden die Messergebnisse des Durchsatzes und in Abschnitt 5.4 die Messergebnisse der Auslastung diskutiert. Zur Messung der Auslastung kommen drei verschiedene Methoden zum Einsatz.

Für die Messungen der Leistungskriterien werden die vier Szenarien in der SMP-Konfiguration und zusätzlich eines der Szenarien in der Uniprozessor-Konfiguration ausgeführt. Das Szenario für die Uniprozessor-Konfiguration kann beliebig gewählt werden, da sich die vier Szenarien in der Uniprozessor-Konfiguration gleich verhalten. Die Verteilung der Tasks auf die Prozessoren in der Uniprozessor-Konfiguration entspricht Szenario 1. Das Ändern des BSP kann, trotz Nutzung von nur einer CPU in der SMP-Konfiguration, zu anderen Messergebnissen führen als in der Uniprozessor-Konfiguration und wird daher separat betrachtet.

5.1 Prognose der Messergebnisse

Vor dem Analysieren der Messergebnisse wird das Verhalten des Benchmarks in den vier Szenarien genauer betrachtet und eine Prognose hinsichtlich der Leistungssteigerung aufgestellt.

In Abbildung 5.1 sind die Szenarien in vereinfachter Form abgebildet. Die FlightsoftwareDummyThreads sind als ein Thread mit höchster Priorität dargestellt. Der FDT belegt die CPU zu etwa einem Drittel der Zeit. In Szenario 1 bis 4 sammeln zwei DataCollectingThreads mit einer SleepTime von zwei Zeiteinheiten (ZE) Daten, in Szenario 4.1 ist die SleepTime auf $0\mu s$ gesetzt. Wenn ein DCT drei Mal aktiv war und Daten gesammelt

hat, werden diese von dem jeweils zugehörigen DPT verarbeitet und vom DRT empfangen (DPT und DRT gemeinsam dargestellt). Die gelben Flächen stellen die Zeiteinheiten dar, die die CPU0 nicht belastet wird. Beim Messen der Auslastung mit Hilfe des MemoryScrubbers ist dieser während der markierten Zeiteinheiten aktiv. Nach 65 ZE werden Durchsatz und Auslastung der Szenarien verglichen.

- **Szenario 1:** Die DataCollecting- und DataProzessingThreads können nur in den Pausen des FDT auf der CPU0 ausgeführt werden und müssen sich diese freie CPU-Zeit teilen. Nach 65 ZE sind acht Datenblöcke verarbeitet. Die CPU0 ist 59 von 65 Zeiteinheiten ausgelastet.

- **Szenario 2:** Die zweite Datenverarbeitungs-Gruppe läuft ohne weitere konkurrierende Threads auf CPU1. Somit steigt der Durchsatz von DCT2 und DPT2 und damit auch der Gesamtdurchsatz. Der Durchsatz nach 65 ZE beträgt elf Datenblöcke und ist höher, als in Szenario 1. Die CPU0 ist 45 ZE und somit weniger als in Szenario 1 ausgelastet.

- **Szenario 3:** In Szenario 3 wird auch die erste Datenverarbeitungs-Gruppe auf CPU1 ausgeführt. Obwohl die Threads auf CPU1 dadurch mehr Konkurrenz bekommen, steigt der Durchsatz. DPT1 und DCT1 müssen nicht mehr auf die Pausen des FDT warten sondern können, abwechselnd mit DCT2 und DPT2, ohne große Wartezeiten ausgeführt werden. Der Durchsatz von DCT2 und DPT2 verringert sich zwar etwas, die Steigerung des Durchsatzes von DCT1 und DPT1 überwiegt jedoch. Der Durchsatz ist gegenüber Szenario 2 um einen Datenblock auf zwölf Blöcke gestiegen. Die CPU0 ist 22 ZE, also etwa zu einem Drittel, ausgelastet. Die Auslastung ist geringer als in Szenario 1 und 2.

- **Szenario 4:** In Szenario 4 werden DCT2 und DPT2 weiterhin statisch der CPU1 zugewiesen. DCT1 und DPT1 können hingegen auf beiden CPUs ausgeführt werden. Durch die dynamische Zuweisung können DPT1 und DCT1 in den Pausen auf der CPU0 ausgeführt werden und behindern somit nicht DCT2 und DPT2 auf der CPU1. DCT2 und DPT2 können in den Pausen des FDT wie in Szenario 2 ohne Verzögerung ausgeführt werden. Dadurch kann der Durchsatz erneut gesteigert werden. Der Durchsatz nach 65 ZE beträgt 13 Datenblöcke und ist damit der höchste der vier Szenarien. Die Auslastung der CPU0 ist mit 32 ZE höher als bei Szenario 3 und niedriger als in Szenario 1 und 2.

- **Szenario 4.1:** In Szenario 4.1 wird die SleepTime der DCTs auf $0 \mu s$ gesetzt, sodass beide Prozessoren fast vollständig ausgelastet sind und der Durchsatz maximiert wird. So kommt es zwar einerseits zu einer stärkeren Behinderung der Threads untereinander, die CPUs werden jedoch bestmöglich für das Szenario ausgenutzt.

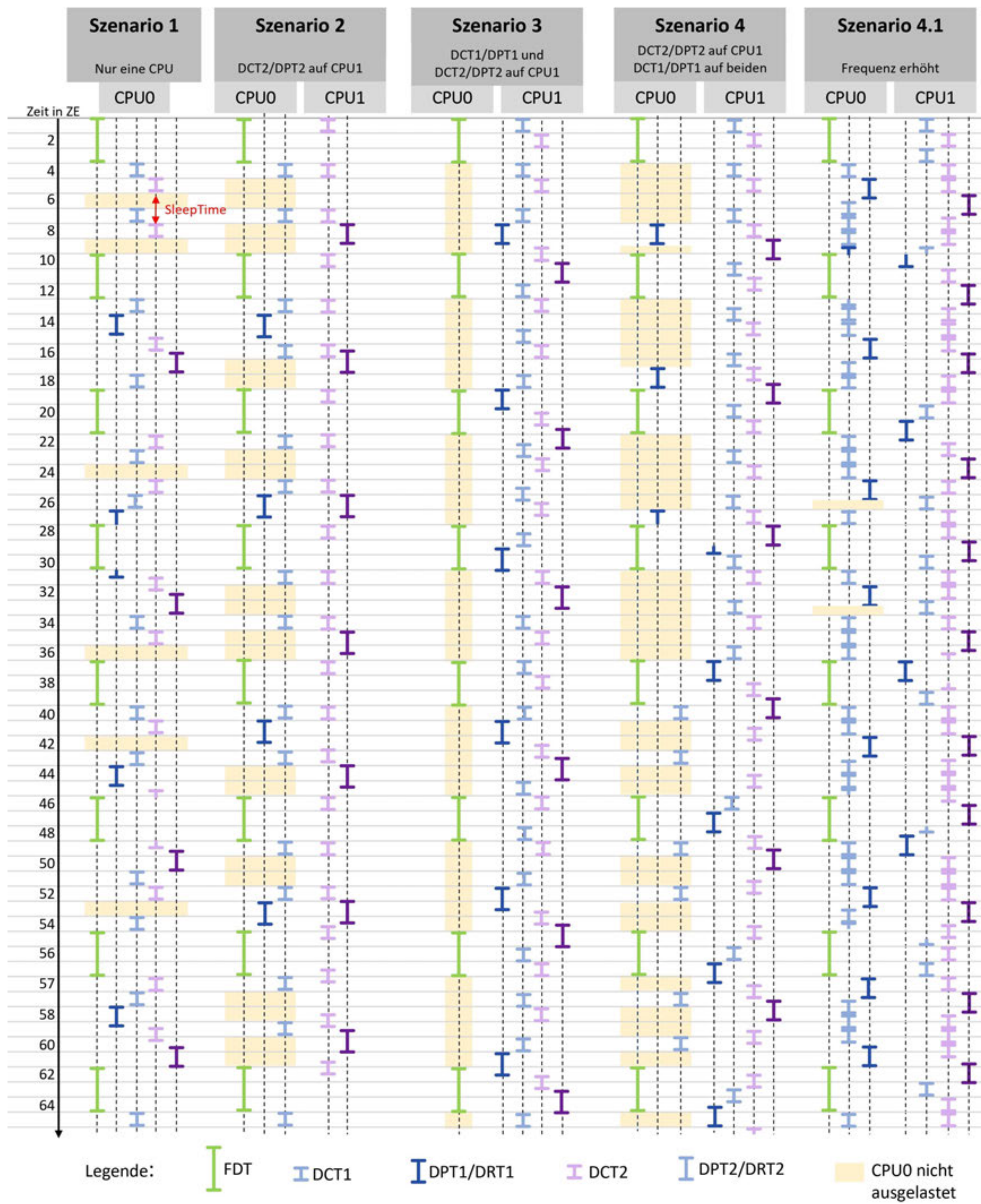


Abbildung 5.1: Zeitverhalten der Szenarien

Die Szenarien eins bis vier wurden zuvor mit einer SleepTime größer $0\mu s$ betrachtet. Wird, wie in Szenario 4.1 gezeigt, die SleepTime auf $0\mu s$ gesetzt, können sich die beschriebenen Leistungsverhältnisse und der Speedup gegenüber Szenario 1 ändern. Allgemein ergibt sich der maximale Durchsatz D_n, der sich unter Verwendung von n zusätzlichen Prozessoren, einem zu parallelisierenden Anteil P und einem Einprozessor-Durchsatz D_0, erzielen lässt, zu

$$D_n = \left(\frac{n}{P} + 1\right) \cdot D_0 \tag{4}$$

Der Speedup ergibt sich entsprechend zu

$$S_n = \left(\frac{n}{P} + 1\right) \tag{5}$$

Der maximale Durchsatz in Szenario 2 und Szenario 4 wird als annähernd gleich erwartet, da beide Prozessoren nahezu vollständig ausgelastet sind. Der Speedup von Szenario 2 und 4 gegenüber Szenario 1 ergibt sich bei den vereinfachten Verhältnissen zu $\left(\frac{1}{\frac{2}{3}} + 1\right) = 2,5$. Der Durchsatz in Szenario 3 wird bei einer SleepTime von $0\mu s$ um $\frac{1}{3}$ höher als bei Szenario 1 erwartet (entspricht einem Speedup von 1,33), da alle datenverarbeitenden Threads auf der zweiten CPU ausgeführt werden und, anders als in Szenario 1, nicht mit dem FDT konkurrieren.

5.2 Darstellung und Gemeinsamkeiten der Messergebnisse

Die folgenden Abschnitte behandeln die mithilfe des Benchmarks aufgenommenen Messergebnisse. Die beiden Messgrößen Auslastung und Durchsatz werden abhängig von der SleepTime in Diagrammen dargestellt. Abbildung 5.2 zeigt die Darstellung des Durchsatzes und Abbildung 5.3 die Darstellung der Auslastung der CPU0.

Die Diagramme stellen auf der y-Achse die Auslastung bzw. den Durchsatz und auf der x-Achse die logarithmierte SleepTime der DCTs dar. Das Diagramm für die Auslastung stellt diese zum einen als prozentualen Anteil der Laufzeit (linke y-Achse) und für die Messungen mittels CPU Usage Statistics und MemoryScrubber zum anderen als Anzahl an vom MemoryScrubber geprüfter Blöcke bzw. als Dauer des Idle-Zustandes von CPU0 dar (rechte y-Achse).

Der Verlauf der Messwertkurven weist, wie bei digitalen Systemen üblich, einen annähernd zeitdiskreten Verlauf auf. Die kleinste Zeiteinheit für die SleepTime ist der in der RTEMS System-Konfiguration definierte Tick. Die Länge eines Ticks beträgt $500\mu s$.

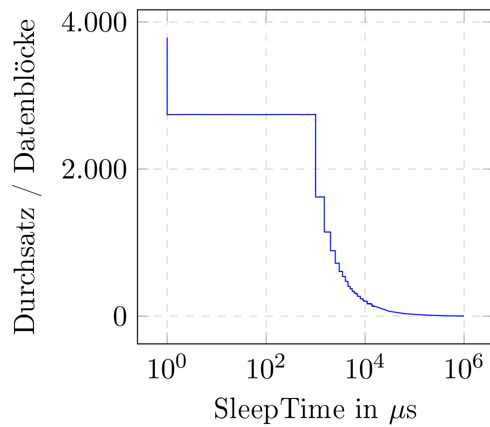


Abbildung 5.2: Darstellung Durchsatz

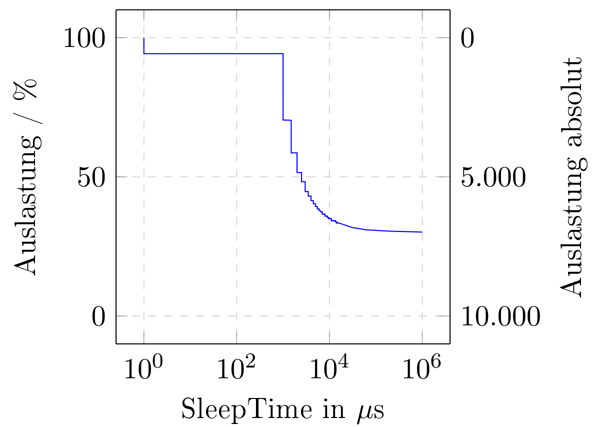


Abbildung 5.3: Darstellung Auslastung

Die SleepTime wird OUTPOST-intern auf ein Vielfaches der Tick-Länge abgerundet. Somit Ist die tatsächlich ausgeführte SleepTime im Abstand von 500μs diskretisiert. Zum Darstellen des stufenförmigen Verlaufes wird die SleepTime innerhalb des Benchmarks auf ein Vielfaches von 500μs gesetzt und anschließend um eins reduziert. Durch die Implementierung der sleep-Funktion in OUTPOST wird eine SleepTime im Bereich $0 < SleepTime < Tick - Länge$ auf die Länge eines Ticks aufgerundet. Das Maximum des Durchsatzes bwz. der Auslastung wird daher bei einer SleepTime von 0μs[1] erreicht.

Der Vergleich der Kurven des Durchsatzes und der Auslastung zeigt den proportionalen Zusammenhang der beiden Größen. Bei einem geringen Durchsatz (SleepTime = $10^6\mu s$) ist auch die Auslastung niedrig. Bei steigendem Durchsatz wird die CPU0 (abhängig von dem Szenario) stärker ausgelastet. Die Kurven des Durchsatzes und der Auslastung verlaufen in den Szenarien in etwa wie in Abbildung 5.2 und Abbildung 5.3 dargestellt. Je nach Szenario und Aufteilung der Threads auf die Prozessoren unterscheiden sich die Verläufe in der Steilheit.

Die minimale Auslastung der CPU0 befindet sich bei einer SleepTime von $10^6\mu s$ durch die Belastung der CPU durch die FDTs etwa bei 33%. Auch der Durchsatz ist bei dieser SleepTime minimal und befindet sich (durch die SleepTime von einer Sekunde, einer Teildurchlaufszeit von 20s und einer Blockgröße von 16) im Bereich von null bis zwei Datenblöcken. Das Maximum erreichen Durchsatz und Auslastung bei einer SleepTime

[1] Bei einer logarithmierten Darstellung der SleepTime kann eine SleepTime von 0μs nicht dargestellt werden. Der bei 0μs aufgenommene Wert wird daher bei 1μs abgebildet.

von $0\mu s$. Bei dieser SleepTime werden die DCTs, wenn der prioritätsbasierte Scheduler es erlaubt, ohne Pause ausgeführt.

5.3 Messung des Durchsatzes

Bei der Messung des Durchsatzes ist zu beachten, dass der MemoryScrubber nicht im Betrieb ist. Wird der MemoryScrubber auf der ersten CPU und die Datenverarbeitung auf der zweiten CPU ausgeführt, können die Tasks auf der zweiten CPU durch den von beiden CPUs genutzten Bus und den MemoryScrubber ausgebremst werden, wordurch die Menge der verarbeiteten Daten sinkt (siehe Unterabschnitt 5.4.1). Die Messergebnisse der Szenarien sind in Abbildung 5.4 dargestellt.

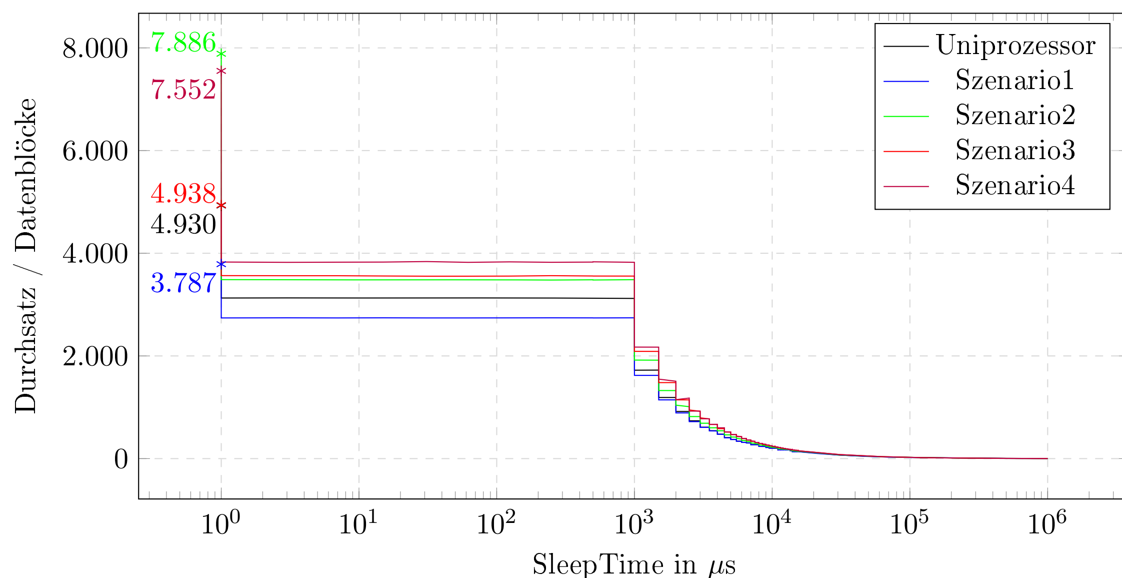


Abbildung 5.4: Durchsatz der Szenarien

Vergleich von Uniprozessor- und SMP-Konfiguration anhand von Szenario 1: Der Vergleich der Kurven zeigt, dass der Durchsatz in der SMP-Konfiguration in Szenario 1 niedriger ist, als der Durchsatz in der Uniprozessor-Konfiguration. Die Differenz entsteht durch die Änderungen[2] im SMP-BSP gegenüber dem Uniprozessor-BSP,

[2] Um RTEMS SMP-fähig zu machen, wurden Teile des Betriebssystems von Grund auf neu entwickelt, wodurch die genauen Ursachen der Unterschiede des Verhaltens zwischen Uniprozessor- und SMP-BSP und der Leistungsabweichung nicht ohne weiteres erkennbar sind.[21]

die beispielsweise durch den höheren Synchronisierungsaufwand für das Betriebssystem bei Verwendung mehrerer Prozessoren bedingt sind. Der Vergleich von Szenario 2 und der Uniprozessor-Konfiguration zeigt, dass bereits das Auslagern eines niederfrequenten Tasks (hohe SleepTime) auf die zweite CPU die Durchsatzminderung durch das SMP-BSP ausgleicht und zu einer Steigerung des Durchsatzes führt.

Vergleich der Szenarien, SleepTime $> 0\mu s$:
Bei einer SleepTime größer $0\mu s$ steigt der Durchsatz, wie in Abschnitt 5.1 erwartet, von Szenario 1 (niedrigster Durchsatz) bis Szenario 4 (höchster Durchsatz). Der Durchsatz von Szenario 2, 3 und 4 ist etwa ein Drittel höher als der Durchsatz von Szenario 1, da die Datenverarbeitung zusätzlich oder ausschließlich auf CPU1 ausgeführt wird und diese nicht durch die FDTs belastet wird. Die Differenz der Durchsätze von Szenario 2 und 3 ist nur gering, da die Laufzeit der DCTs gegenüber der SleepTime von 500μs relativ klein ist[3]. So kommt es kaum zu einer Behinderung der Threads untereinander. Die Durchsatzsteigerung gegenüber Szenario 1 entsteht größtenteils durch die vollständig der Datenverarbeitung zur Verfügung stehenden CPU1 und nur wenig durch das parallele Ausführen der Datenverarbeitung auf zwei CPUs. Der Durchsatz in Szenario 4 ist etwas höher, da die erste Datenverarbeitungsgruppe, je nachdem, ob die CPU0 durch die FDTs bzw. die CPU1 durch die zweite Datenverarbeitungsgruppe belegt ist, zwischen den CPUs migrieren kann.

Vergleich der Szenarien, SleepTime $= 0\mu s$:
Bei einer SleepTime von $0\mu s$ ändern sich die Leistungsverhältnisse. Der Durchsatz von Szenario 2 ist deutlich höher als der von Szenario 3. Der Durchsatz von Szenario 3 ist gegenüber Szenario 1 um den Faktor 1,30 höher und entspricht somit etwa dem in Abschnitt 5.1 berechneten Speedup von 1,33. Der Durchsatz von Szenario 2 und Szenario 4 ist um den Faktor 2,08 bzw. 1,99 höher als in Szenario 1 und die Steigerung somit geringer als durch den erwarteten Speedup von 2,5. Die Differenz zu den angenommenen Werten entsteht zum einen durch die Vereinfachung der Betrachtung aus Abschnitt 5.1. Hier werden die FDTs als ein Thread behandelt. In der Benchmark-Applikation liegen hingegen drei FDTs mit unterschiedlichen Prioritäten vor. Da die datenverarbeitenden Threads eine höhere Priorität als der FDT3 und zum Teil eine höhere als der FDT2 haben, nehmen diese bei kleinerer SleepTime mehr Laufzeit in Anspruch und verdrängen die FDTs. Somit wird der Anteil der Laufzeit der beiden niederprioren FDTs niedriger und weicht von dem angenommenen Anteil von $\frac{1}{3}$ ab. Zum anderen kann der Bus, der die

[3] zu ermitteln mit Hilfe des Tracing Tools, Laufzeit beträgt etwa 200 μs

beiden Prozessoren mit dem Hauptspeicher verbindet, als Bottleneck in den Szenarien 2, 3 und 4 zu einer Durchsatzminderung führen (vgl. Unterabschnitt 5.4.1).

5.4 Messung der Auslastung

In diesem Kapitel wird die Auslastung der CPU0 für die verschiedenen Szenarien gemessen. Zur Messung der Auslastung stehen drei Methoden zur Verfügung: das Memory Scrubbing, die CPU Usage Statistics und das Event-Recording inklusive Analyse mit Hilfe des Eclipse Trace Compasses. Jede dieser Methoden bringt ihre Vor- und Nachteile mit sich und kann zur Verfälschung der in den Testszenarien aufgenommenen Werte führen.

5.4.1 MemoryScrubber

Beim Messen der Auslastung des ersten Prozessors mit Hilfe des MemoryScrubbers wird die Höhe der Auslastung durch die Anzahl der Blöcke, die der MemoryScrubber in einem bestimmten Zeitfenster prüft, bestimmt. Abbildung 5.5 zeigt die mithilfe des MemoryScrubbers ermittelte Auslastung der CPU0 in Abhängigkeit der SleepTime. Um anhand der vom MemoryScrubber geprüften Blöcke eine Aussage über die prozentuale Auslastung der CPU treffen zu können, wird in einem separaten Testdurchlauf die Anzahl der vom MemoryScrubber geprüften Blöcke ohne konkurrierende Threads gemessen. Dazu wird ein Szenario ohne Datenverarbeitungs-Threads und ohne FDTs ausgeführt. Die Achse in Abbildung 5.5 zur Angabe der Auslastung in Prozent ist an der 0%-Linie des SMP-BSP ausgerichtet.

Vergleich von Uniprozessor- und SMP-Konfiguration anhand von Szenario 1: Das Verwenden des SMP-BSP anstelle des Uniprozessor-BSPs hat auch auf die Auslastung der CPU0 einen Einfluss. Unter Verwendung des SMP-BSPs ist, wie nach den Messungen des Durchsatzes zu erwarten, die für die Null-Linie gemessene MemoryScrubber-Blockzahl niedriger als in der Uniprozessor-Konfiguration (0%-Linien). Durch die Unterschiede der BSPs ist die Auslastung in der Uniprozessor-Konfiguration mit Referenz auf die dazugehörige Null-Linie niedriger als die in der SMP-Konfiguration gemessene Auslastung im Bezug auf die SMP-Null-Linie.

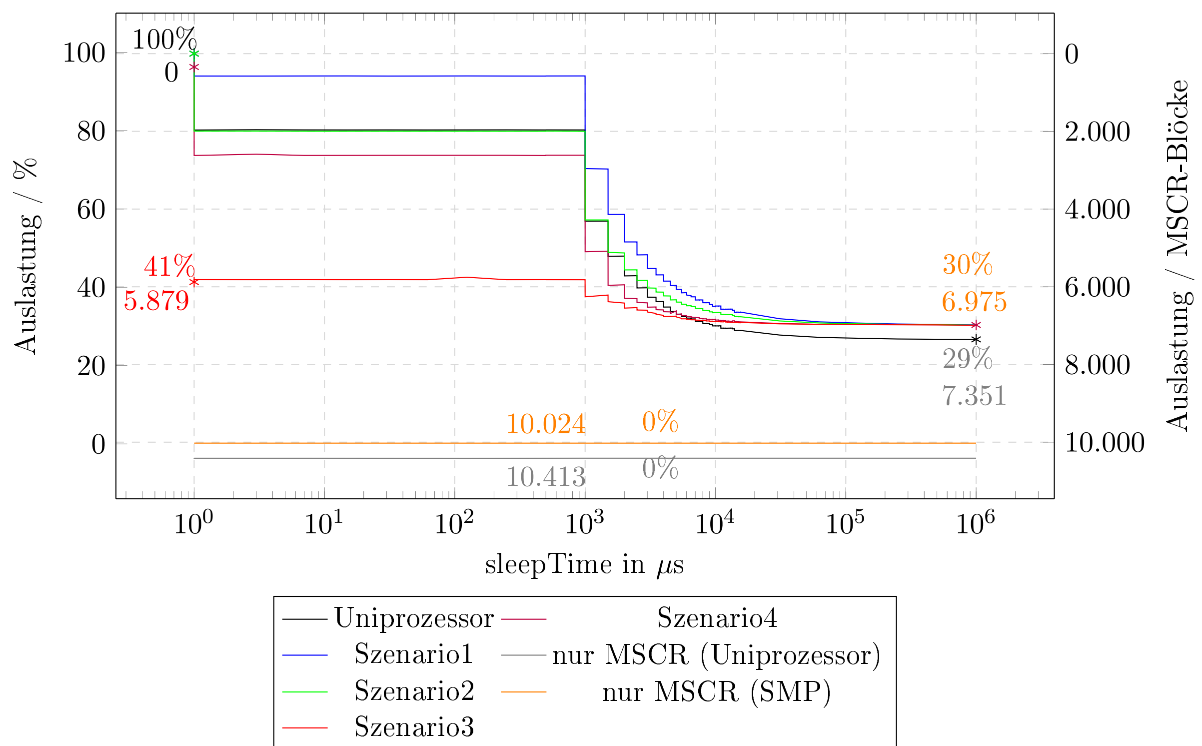


Abbildung 5.5: Auslastung der CPU0, ermittelt durch den MemoryScrubber

Vergleich der Szenarien, SleepTime $> 0\mu s$:
Bei einer hohen SleepTime ($10^6\mu$s) ist die CPU0, entsprechend der Laufzeitanteile der FDTs, etwa zu einem Drittel ausgelastet. Die höchste Auslastung der CPU0 bei steigender SleepTime weist Szenario 1 auf. In Szenario 2 sinkt die Auslastung gegenüber Szenario 1 durch Auslagern der zweiten Datenverarbeitungsgruppe auf die CPU1. In Szenario 3 sinkt die Auslastung erneut durch Auslagern beider Datenverarbeitungsgruppen. In Szenario 4 ist die Auslastung höher als in Szenario 3, da die erste Datenverarbeitungsgruppe während der Laufzeit auch CPU0 zugewiesen wird. Die Auslastung in Szenario 4 ist jedoch geringer als in Szenario 1 und 2, da die zweite Datenverarbeitungsgruppe auf CPU1 ausgeführt wird und auch die erste Datenverarbeitungsgruppe der CPU1 während der Laufzeit zugewiesen werden kann, sodass sie nicht bei jeder Ausführung die CPU0 belastet.

Vergleich der Szenarien, SleepTime $= 0\mu s$:
Bei einer SleepTime von $0\mu s$ ist die Auslastung der CPU0 in Szenario 1 und 2 maximal, da auf CPU0 datenverarbeitende Threads mit maximaler Frequenz ausgeführt werden.

In Szenario 4 ist die Auslastung geringfügig niedriger. Die Threads der ersten Datenverarbeitungsgruppe können zwischen den CPUs migrieren. Ist die CPU0 durch einen FDT belegt, kann beispielsweise der DCT1 auf der CPU1 ausgeführt werden. Wird vor dem Beenden des DCT1 auf CPU1 die CPU0 wieder frei, wird der DCT1 trotzdem auf der CPU1 zu Ende ausgeführt. In dieser Zeit ist die CPU0 nicht ausgelastet, bis der DCT1 vom Scheduler erneut aufgerufen und einer CPU zugewiesen wird. In Szenario 3 werden keine datenverarbeitenden Threads auf der CPU0 ausgeführt. Trotzdem steigt die Auslastung der CPU0 mit sinkender SleepTime kontinuierlich an. Der Anstieg der Auslastung ist auf den von CPU0 und CPU1 gemeinsam genutzten Bus zum Hauptspeicher zurückzuführen. Durch die vom GR712rc genutzte write-through Policy belastet jeder Schreibvorgang den Bus. Wird die Frequenz der DCTs auf CPU1 erhöht, führt die CPU1 mehr Schreibvorgänge aus. Da auch die FDTs auf CPU0 auf den Bus zugreifen, kommt es zur Verzögerung des Zugriffs durch CPU1 und somit zu einer längeren Laufzeit der FTDs und einer höheren Auslastung der CPU0. Neben den FDTs wird auch der auf der ersten CPU ausgeführte MemoryScrubber ausgebremst, wodurch die Anzahl der geprüften Blöcke sinkt und die daraus berechnete Auslastung steigt.

Einfluss des MemoryScrubbers auf Messungen und Benchmark-Verhalten

Anders als beim Ermitteln des Durchsatzes des Benchmarks kann die Auslastung nicht ohne direkten Eingriff in das Verhalten eines Teildurchlaufes des Benchmarks ermittelt werden. Beim Messen der Auslastung mit Hilfe des MemoryScrubbers wird dieser auf der CPU0 ausgeführt. Bei einigen Szenarien ist der zweite Prozessor ebenfalls mit der Abarbeitung von Aufgaben beschäftigt. Sowohl der erste als auch der zweite Prozessor belasten dabei den Bus zum Hauptspeicher. Der MemoryScrubber kann also durch Tasks auf der CPU1 ausgebremst werden, wenn er auf den von CPU1 belegten Bus zugreifen muss. Die Auslastung, die anhand der Blockzahl abgelesen wird, die der MemoryScrubber prüft, wird demnach durch die Aktivitäten auf dem zweiten Prozessor beeinflusst. Ebenso beeinflusst der MemoryScrubber den Durchsatz der Tasks auf der zweiten CPU. Anders als im Leerlauf werden bei aktivem MemoryScrubber hochfrequent Daten geschrieben und gelesen. Diese Vorgänge belasten den gemeinsamen Bus der Prozessoren zum Hauptspeicher und führen zur Ausbremsung der zweiten CPU und somit zur Verringerung des Durchsatzes in Szenario 2, 3 und 4. Der Einfluss des MemoryScrubbers auf den Durchsatz ist in Abbildung 5.6 beispielhaft für eine SleepTime von 500μs dargestellt.

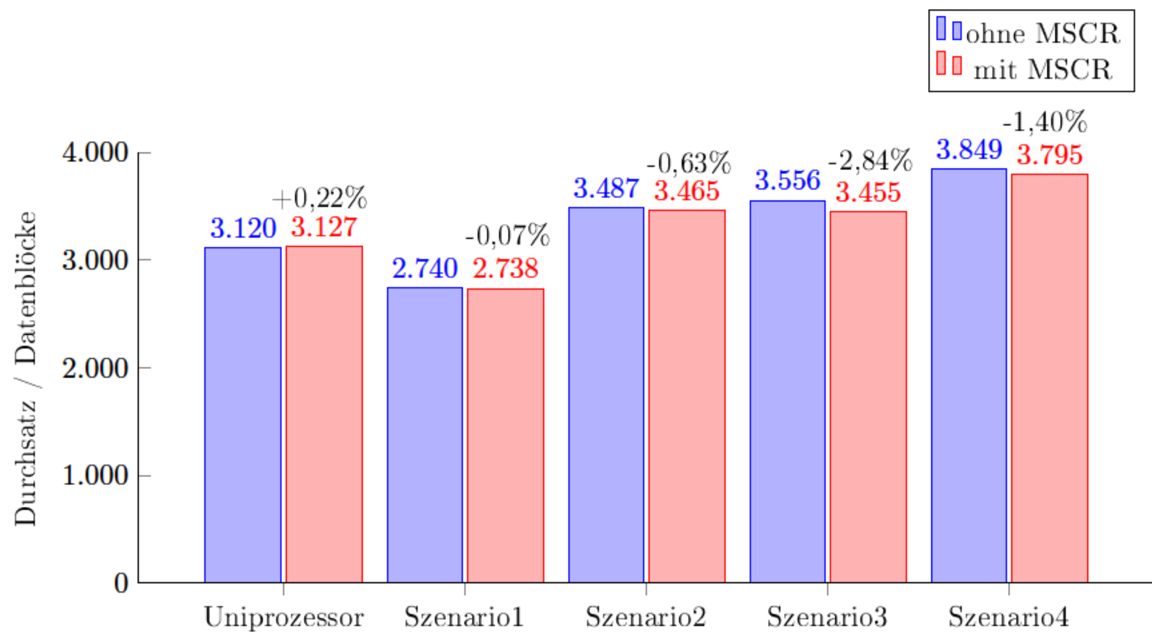


Abbildung 5.6: Durchsatz mit und ohne MemoryScrubber, SleepTime = 500μs

In der Uniprozessor-Konfiguration und dem Szenario 1 werden die Threads nur auf einer CPU ausgeführt. Die Abweichungen des mit und ohne MemoryScrubber gemessenen Durchsatzes in Abbildung 5.6 von 0,22% und 0,07% sind auf Messungenauigkeiten durch die Umsetzung des TestingThreads zurückzuführen. In Szenario 2 wird die zweite Datenverarbeitungsgruppe auf CPU1 ausgeführt. Durch den MemoryScrubber wird der Durchsatz um 0,63% herabgesenkt. In Szenario 3 werden beide datenverarbeitenden Thread-Gruppen auf CPU1 und somit parallel zu dem MemoryScrubber auf CPU0 ausgeführt. Hier ist deutlich der Einfluss des MemoryScrubbers durch den gemeinsamen Bus durch den Durchsatzverlust von 2,84% zu erkennen. In Szenario 4 ist der Verlust mit 1,40% geringer als in Szenario 3, da die CPU0 zum Teil durch die erste Datenverarbeitungsgruppe belegt ist und somit der MemoryScrubber mit einem geringeren Laufzeitanteil weniger Einfluss auf die Messungen nimmt.

Bei einer SleepTime von 0μs ist die erste CPU in Szenario 1, 2 und 4 bereits maximal ausgelastet, wodurch der MemoryScrubber nicht ausgeführt wird und somit keinen Einfluss auf den Durchsatz hat. In Szenario 3 wird neben den FDTs nur der MemoryScrubber auf der CPU0 ausgeführt, wodurch der Durchsatz auch bei einer SleepTime von 0μs durch den MemoryScrubber verringert wird. Durch die höhere Frequenz der DCTs als in

Abbildung 5.6 wird der Durchsatz bei einer SleepTime von 0μs in Szenario 3 von 4938 Datenblöcke auf 4616 Datenblöcke und damit um 6,58% verringert.

5.4.2 CPU Usage Statistics

Die CPU Usage Statistics von RTEMS dient als zweite Methode zur Messung der Auslastung der CPU0. Abbildung 5.7 zeigt die Ausgabe einer CPU Usage Statistic über das Terminal in Eclipse. Die absoluten und prozentualen Angaben der CPU-Belegung durch die Threads beziehen sich auf einen Teil-Durchlauf und beinhalten die Zeit, in der das Szenario mit einer bestimmten SleepTime durchgeführt wird und die Zeit, die der TT benötigt, um den Teildurchlauf zu starten, zu stoppen und die Messwerte auszugeben. Die prozentualen Angaben werden aus der Laufzeit des Tasks und der Gesamtlaufzeit zu dem Zeitpunkt der Ausgabe des Wertes berechnet. Da die Ausgabe einige Zeit dauert, beziehen sich die prozentualen Angaben der verschiedenen Threads auf unterschiedliche Gesamtlaufzeiten. Daher wird zum Analysieren der Auslastung die Laufzeit in Sekunden betrachtet.

```
-------------------------------------------------------------------------------
                              CPU USAGE BY THREAD
------------+----------------------------------------+---------------+---------
 ID         | NAME                                   | SECONDS       | PERCENT
------------+----------------------------------------+---------------+---------
 0x09010001 | IDLE                                   |     12.229867 |  42.947
 0x09010002 | IDLE                                   |      4.712675 |  15.607
 0x0a010001 | UI1                                    |      0.000000 |   0.000
 0x0a010002 | MSCR                                   |      0.000000 |   0.000
 0x0a010003 | DCT1                                   |      5.318681 |  14.941
 0x0a010004 | DPT1                                   |      1.660176 |   4.445
 0x0a010005 | DRT1                                   |      0.301044 |   0.770
 0x0a010006 | DCT2                                   |      6.639515 |  16.275
 0x0a010007 | DPT2                                   |      2.050688 |   4.820
 0x0a010008 | DRT2                                   |      0.411888 |   0.929
 0x0a010009 | TT                                     |     26.187125 |  56.698
 0x0a01000a | OCT                                    |     27.973012 |  58.309
 0x0a01000b | FDT1                                   |      4.000982 |   8.041
 0x0a01000c | FDT2                                   |      1.517294 |   2.941
 0x0a01000d | FDT3                                   |      1.157076 |   2.169
------------+----------------------------------------+---------------+---------
 TIME SINCE LAST CPU USAGE RESET IN SECONDS:                         53.329075
-------------------------------------------------------------------------------
```

Abbildung 5.7: Ausgabe der RTEMS CPU Usage Statistics (Szenario 2, SleepTime = 500μs)

Die IDLE-Threads repräsentieren die Zeit, in der die CPUs nicht belastet sind. Die IDLE-Threds für CPU0 und CPU1 sind anhand der Namen nicht zu unterscheiden. Zudem enthält die CPU Usage Statistics keine Information darüber, auf welcher CPU ein Task ausgeführt wurde. Für die Analyse einer Applikation mit SMP-Konfiguration ist die CPU Usage Statistics daher nur bedingt geeignet.

Die Tasks sind in der Reihenfolge angeordnet, wie sie im Benchmark erstellt werden. Aus dem Vergleich der Messungen mit dem erwarteten Benchmark-Verhalten ist ersichtlich, dass in Szenario 1, 2 und 4 der erste IDLE-Thread in der SMP-Konfiguration CPU1 und der zweite CPU0 zuzuordnen ist. In Szenario 3 ist hingegen zu beobachten, dass bei geringen SleepTimes die Summe der Laufzeiten der IDLE- plus der weiteren Threads eines Prozessors nicht die Gesamtdauer des Teildurchlaufes (20s) ergibt. Die Laufzeiten der beiden IDLE-Threads scheinen nicht eindeutig getrennt zu werden. Möglicherweise kann dies dadurch erklärt werden, dass die Funktionen, die von RTEMS zum Aufzeichnen der CPU Usage Statistics ausgeführt werden, zwischen den Prozessoren wechseln. Szenario 3 ist das einzige Szenario, in dem die CPU0 nicht durch die datenverarbeitenden Threads belastet wird. Wird die CPU Usage Statistics standardmäßig auf CPU1 ausgeführt, ist es naheliegend, dass die Ausführung auf die CPU0 verlagert wird, wenn die Auslastung der CPU1 hoch ist, die von CPU0 hingegen niedrig. Da die Namen der IDLE-Threads sich nicht unterscheiden, wird der eine IDLE-Thread der CPU zugeordnet, auf der die CPU Usage Statistics ausgeführt wird und der andere IDLE-Thread der anderen CPU. Je nachdem, wo die CPU Usage Statistics ausgeführt wird, werden Laufzeiten eines Prozessors den Daten von unterschiedlichen IDLE-Threads zugeordnet. In den Szenarien 1, 2 und 4 steigt die Auslastung von CPU1 *und* CPU0 mit sinkender SleepTime. Durch die zusätzlichen FDTs auf CPU0 ist diese dabei stets höher ausgelastet als die CPU1, wodurch es zu keinem CPU-Wechsel der CPU Usage Statistics kommt. In Szenario 3 wird daher die Auslastung aus der Laufzeit eines Teildurchlaufes abzüglich der CPU0 zugeordneten Threads, also der FDTs, berechnet.

In Abbildung 5.8 ist die Auslastung der CPU0 prozentual und als Laufzeiten des IDLE-Threads von CPU0 in Abhängigkeit der SleepTime dargestellt.

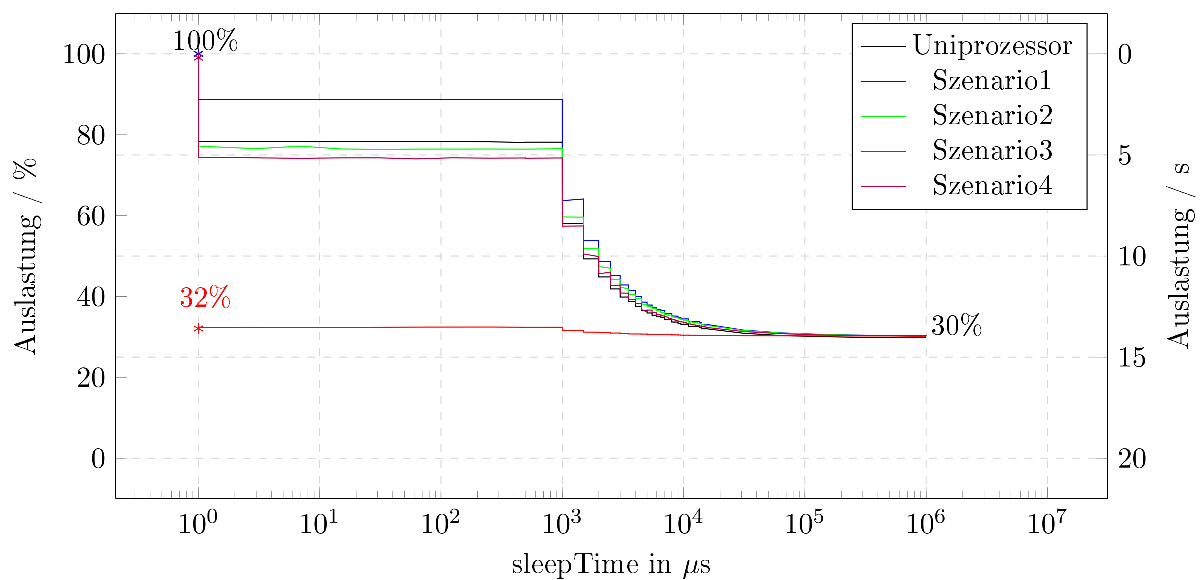


Abbildung 5.8: Auslastung der CPU0, ermittelt durch CPU Usage Statistics

Die Verhältnisse der Auslastungen in den Szenarien entsprechen derer, die mithilfe des MemoryScrubbers ermittelt wurden. Die Ergebnisse werden genauer in Unterabschnitt 5.4.4 verglichen. Eine zusätzliche Ausgabe des Durchsatzes neben der CPU Usage Statistics zeigt keine Abweichungen zu dem in Abschnitt 5.3 gemessenen Durchsatz.

5.4.3 RTEMS Tracing Framework

Als dritte Methode wird die Auslastung der CPU0 mithilfe des RTEMS Tracing Frameworks und des Eclipse Trace Compass ermittelt. Hier werden, anders als bei den vorherigen Messungen, die Szenarien nur mit wenigen SleepTime-Werten durchgeführt, da die Messungen mit mehreren Minuten bis zu über einer Stunde pro Teildurchlauf eines Szenarios relativ viel Zeit in Anspruch nehmen. Dies liegt vor allem an der langsamen Ausgabe der aufgezeichneten Events über das Terminal in Eclipse. Die Dauer eines Teildurchlaufes wird abhängig von der SleepTime gekürzt, um die Ausgabezeit zu verringern und trotzdem genügend Zyklen der Datenverarbeitung aufzuzeichnen, um aussagekräftige Messwerte zu erhalten. Die prozentuale Auslastung der CPU0 ist somit vergleichbar mit den durch MemoryScrubber und CPU Usage Statistics in längeren Testdurchläufen ermittelten Werten.

In Abbildung 5.9 ist die Darstellung der aufgenommenen Events im Eclipse Trace Compass abgebildet.

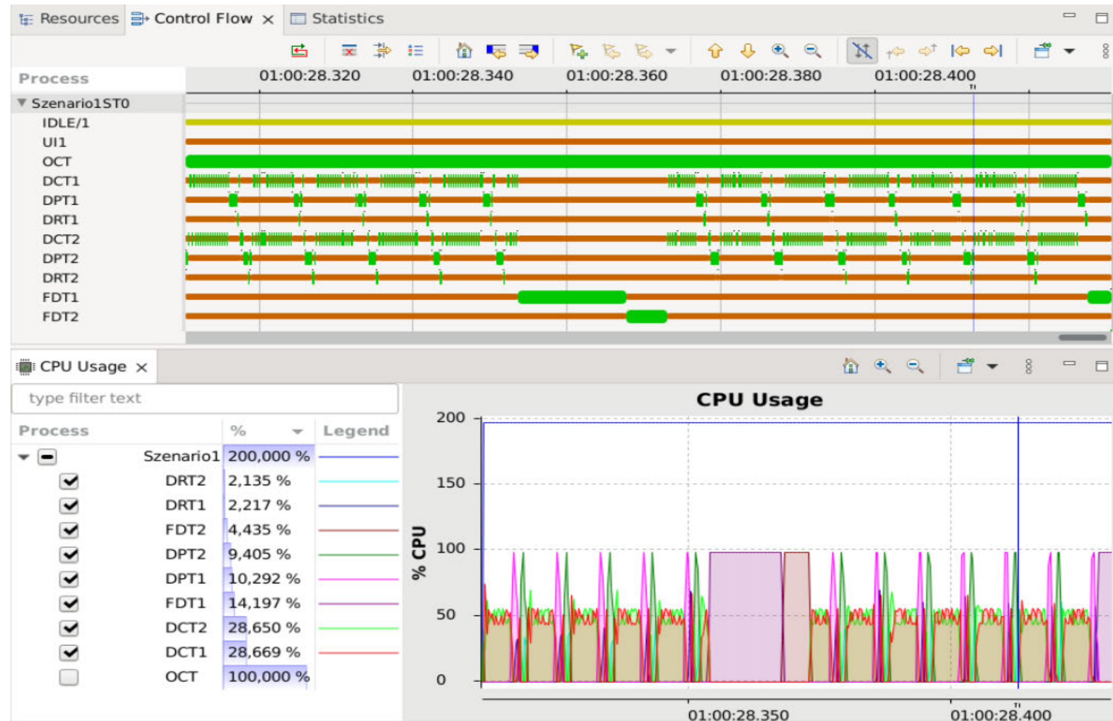


Abbildung 5.9: Auswertung des Event Recordings im Eclipse Trace Compass (Szenario 1, SleepTime = 0)

Neben dem zeitlichen Verlauf der Tasks in der oberen Grafik ist in der unteren Grafik die Belastung der CPU durch die einzelnen Tasks zu sehen. Wie bei der CPU Usage Statistics ist hier ebenfalls nicht direkt zwischen der Auslastung der ersten und zweiten CPU zu unterscheiden. Zudem wird nicht der Laufzeitanteil der IDLE-Tasks angezeigt. In Szenario 1, 2 und 3 lässt sich der Anteil der Laufzeit des IDLE-Tasks aus den Laufzeitanteilen der Tasks, die auf CPU0 ausgeführt werden, berechnen. In Szenario 4 kann die erste Datenverarbeitungsgruppe jedoch auf beiden CPUs ausgeführt und der Laufzeitanteil nicht einer CPU zugeordnet werden. Die Auslastung der CPU0 kann somit nicht ohne weiteres aus den Laufzeiten der einzelnen Threads berechnet werden. Durch eine fälschlicherweise gegeneinander verschobene Darstellung der CPU-Spuren im Eclipse Trace Compass (Erklärung in Unterabschnitt 5.4.4) gibt es jedoch Abschnitte, in denen nur die Aktivitäten

der CPU0 dargestellt werden. In diesen Abschnitten stellt die Summe der prozentualen Task-Laufzeiten die Auslastung von CPU0 dar.

In Abbildung 5.10 ist die durch das Event Recording ermittelte Auslastung der CPU0 in den Szenarien in Abhängigkeit von drei verschiedenen SleepTimes dargestellt. Die Auslastung zu den gemessenen SleepTimes (0μs, 500μs und $10^6\mu$s) sind durch Punkte gekennzeichnet. Der Verlauf zwischen den Punkten wird interpoliert.

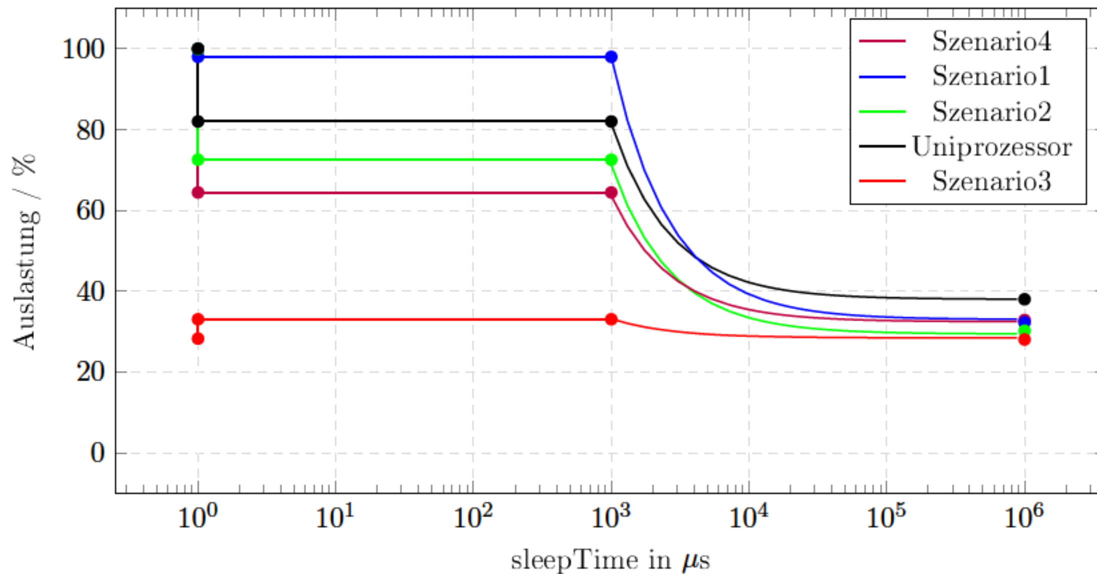


Abbildung 5.10: Auslastung der CPU0, ermittelt durch das Event Recording

Die mit Hilfe des Tracing Frameworks ermittelten Leistungsverhältnisse zwischen den Szenarien entsprechen in etwa den mit Hilfe des MemoryScrubbers und der CPU Usage Statistics ermittelten. Es sind jedoch Abweichungen in der absoluten Auslastung der einzelnen Szenarien zu erkennen.

Die Ermittlung der Auslastung anhand der Darstellungen im Eclipse Trace Compass erweist sich bei der gewählten Laufzeit eines Testdurchlaufes trotz Aufzeichnen mehrerer Datenverarbeitungszyklen als ungenau. Die Angaben der Auslastung sind abhängig von dem dargestellten Abschnitt des Zeitverhaltens (vgl. Abbildung 5.9). Die Größe des Abschnittes ist nur in groben Einstufungen zu ändern, sodass sich Threads mit einer längeren Laufzeit wie der FDT1 je nach Vorkommen in einem Abschnitt unterschiedlich stark auf die angezeigte Auslastung auswirkt.

Das Event Recording wirkt sich im Gegensatz zu der CPU Usage Statistics messbar auf den Durchsatz des Benchmarks aus. Das Aufzeichnen jedes Events belastet die CPUs und mindert den Durchsatz und erhöht die Auslastung.

5.4.4 Vergleich und Bewertung der Methoden zur Messung der Auslastung

Die Ergebnisse der Messungen der Auslastung mithilfe des MemoryScrubbers, der CPU Usage Statistics und des Tracing Frameworks sind in Abbildung 5.11 beispielhaft für eine SleepTime von $500\mu s$ dargestellt.

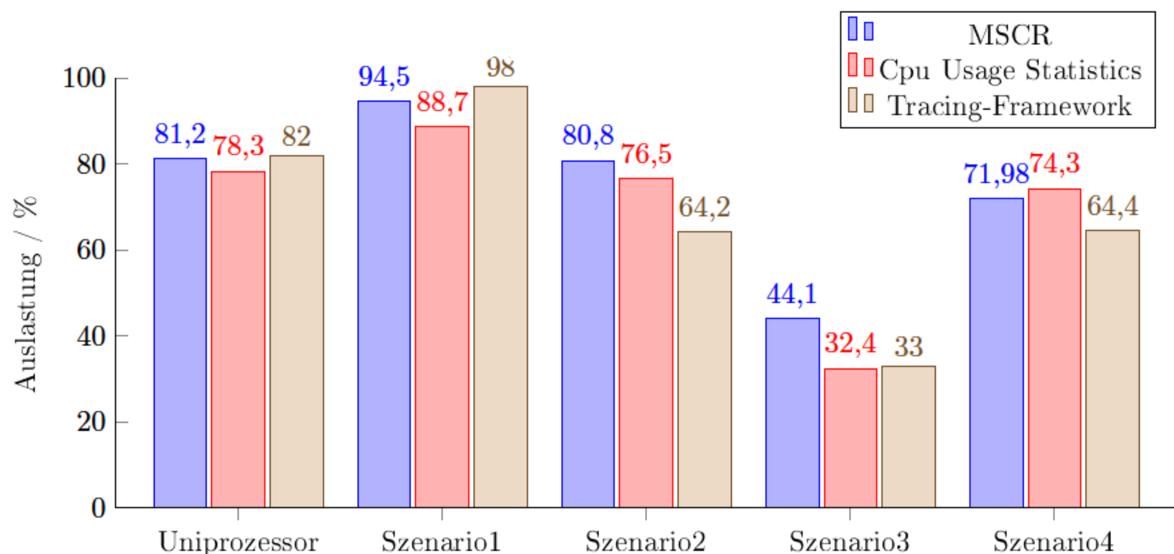


Abbildung 5.11: Vergleich der Methoden zum Messen der Auslastung, SleepTime = $500\mu s$

Mit Hilfe der genutzten Methoden zur Messung der Auslastung lässt sich jeweils das Verhältnis der Auslastungen in den Szenarien ermitteln. Die Messungen der Auslastung eines Szenarios durch verschiedene Methoden weisen jedoch Schwankungen auf. Die mit Hilfe des Memory Scrubbers gemessene Auslastung ist, mit Ausnahme von Szenario 4, durch den Einfluss des von beiden Prozessoren genutzten Busses höher, als die mit Hilfe der CPU Usage Statistics ermittelte Auslastung. In Szenario 4 kann die Zuweisung der ersten Datenverarbeitungsgruppe zu einer CPU durch den MemoryScrubber beeinflusst werden, wodurch wiederum der Durchsatz beeinflusst wird. Die mit Hilfe des Tracing Tools gemessene Auslastung wird durch die Wahl des betrachteten Zeitfensters beeinflusst

und kann nach oben und nach unten abweichen. Unter Verwendung der CPU Usage Statistics kann, im Gegensatz zu den anderen zwei Methoden, keine Abweichung des Durchsatzes festgestellt werden. Hier ist demnach der Einfluss der Messmethode auf die Messergebnisse der geringste und es ist davon auszugehen, dass diese Messwerte am ehesten der Realität entsprechen.

Neben der Messung der Auslastung sind die CPU Usage Statistics und das Tracing Framework bei der Entwicklung von Software hilfreich, da sie Informationen über das Zeitverhalten der einzelnen Tasks wiedergeben. Das Tracing Framework ist jedoch kompliziert in der Handhabung und fehleranfällig. Zum Definieren der Buffergröße ist, um alle gewollten Events aufzuzeichnen, einen Überlauf des Buffers zu vermeiden und die Ausgabezeit der aufgezeichneten Events in einem angemessenen Rahmen zu halten, eine gute Kenntnis über die Menge der aufzuzeichnenden Events nötig, die sich erst durch die Durchführung einiger Testläufe ergibt. In der SMP-Konfiguration entsteht ein zusätzliches Problem dadurch, dass für jeden Prozessor ein eigener Buffer verwendet wird. Sind diese Buffer zu Beginn bereits unterschiedlich voll, werden die Events auf CPU0 und CPU1 im Eclipse Trace Compass zeitlich verschoben angezeigt. Zudem ist diese Methode zum Messen der Auslastung mit Abstand die langwierigste.

In Anhang A sind die Vor- und Nachteile der Methoden zur Messung der Auslastung ausführlich aufgelistet.

6 Fazit

In dieser Arbeit wurde die SMP-Unterstützung durch RTEMS in die Softwareplattform OUTPOST integriert und eine Leistungsanalyse durchgeführt. Dafür wurden die Eigenschaften der SMP-Konfiguration des Echzeitbetriebssystems RTEMS und deren Unterschiede zur Uniprozessor-Konfiguration analysiert und eine entsprechende Anpassung der bestehenden Betriebssystemabstraktion von RTEMS in OUTPOST begründet und umgesetzt. Die Zuweisung von Threads zu Prozessoren kann nun in der SMP-Konfiguration über die OUTPOST-API durch die Vergabe einer Affinität an einen Thread vorgenommen werden. Die Leistungssteigerung, die sich mit der SMP-Unterstützung erzielen lässt, wurde anhand des Durchsatzes und der Auslastung des ersten Prozessors mit Hilfe eines Benchmarks auf dem Zweiprozessor-SoC GR712RC untersucht. Der Benchmark besteht dabei aus Hauptlast-Threads, die auf dem ersten Prozessor ausgeführt werden, und Nebenlast-Threads, denen zur Untersuchung der Leistungskriterien in verschiedenen Szenarien unterschiedliche Affinitäten zugeordnet sind. Dabei wurde zusätzlich die Auswirkung der Frequenz der Nebenlast-Threads auf die Leistungskriterien betrachtet. Die Messung des Durchsatzes ergibt sich aus einer zum Verhalten des Benchmarks gehörenden Datenverarbeitung und einer Ausgabe der verarbeiteten Datenmenge. Die Auslastung wurde mit Hilfe eines den Prozessor-Leerlauf repräsentierenden MemoryScrubbers, der CPU Usage Statistics und des Event Recordings von RTEMS ermittelt.

Bei der Konfiguration einer Software auf Basis der in OUTPOST umgesetzten SMP-Unterstützung kann folgendermaßen vorgegangen werden:

1. Einfügen des SMP-BSP von RTEMS in die Konfiguration des Build-Prozesses.
2. Wahl eines Schedulers, abhängig davon, ob und welche Affinitäten den Threads zugewiesen werden sollen.
3. Bei Anpassung einer bestehenden Uniprozessor-Software: Prüfen, ob die in der SMP-Konfiguration von RTEMS entfernten bzw. nicht erlaubten Funktionen und Makros (Kapitel 3) verwendet werden und diese ggf. ersetzen.

4. Festlegen, welche Threads auf welchem Prozessor laufen sollen. Den Threads entsprechend eine Affinität zuweisen. Wird keine Affinität zugewiesen, werden alle Threads auf der ersten CPU ausgeführt.

Die Leistungsanalyse hat folgende Erkenntnisse hervorgebracht:

- Wird in einer für den Uniprozessorbetrieb entwickelten Software das Uniprozessor-BSP durch das SMP-BSP ersetzt, kommt es zunächst zu einer Minderung der Leistung.
- Durch Abkoppeln und Auslagern weniger Tasks von der ersten auf die zweite CPU wird die Leistung gegenüber der Uniprozessor-Konfiguration bereits gesteigert. Der Durchsatz steigt, da die Konkurrenz der Tasks auf der zweiten CPU niedriger ist, als auf der ersten CPU. Die Auslastung auf der ersten CPU sinkt.
- Bei einer Affinität eines Tasks zu mehreren Prozessoren muss berücksichtigt werden, dass durch die Migration des Tasks die Cache-Lokalität sinkt, was zu einer längeren Laufzeit des Tasks führen kann. Zum Vermeiden von häufiger Task-Migration ist eine statische Zuweisung von Tasks zu Prozessoren empfehlenswert.
- Je nachdem, ob der Fokus auf der Erhöhung des Durchsatzes, der Reduzierung der Auslastung der ersten CPU oder einem Kompromiss liegt, können den abgekoppelten Tasks unterschiedliche Affinitäten zugeordnet werden.
- Die Leistungssteigerung ist dabei abhängig von der Frequenz, der Laufzeit und der Priorität der abgekoppelten Tasks und der Tasks, die weiterhin auf der ersten CPU ausgeführt werden.
- Die Tasks auf dem zweiten Prozessor führen zu einem Ausbremsen der Tasks auf dem ersten Prozessor. Mit steigender Auslastung des zweiten Prozessors steigt also auch die Auslastung des ersten Prozessors. Beim Anwenden einer Flugsoftware in der SMP-Konfiguration und deren Erweiterung durch Tasks, die auf dem zweiten Prozessor ausgeführt werden, muss der Slowdown berücksichtigt werden.

6.1 Ausblick

Die Ergebnisse dieser Arbeit unterstützen die Entwicklung von SMP-fähiger Flugsoftware auf Basis von OUTPOST und die Umstellung bestehender Uniprozessor-Software auf den SMP-Betrieb. Als erstes Projekt ist die Umstellung der EU:CROPIS Flugsoftware auf den SMP-Betrieb geplant, welche auf dem gleichen Betriebssystem und SoC basiert, die auch in dieser Arbeit verwendet wurden. Die Anpassung der Software an die SMP-Unterstützung von OUTPOST kann wie zuvor beschrieben vorgenommen werden. Es sind jedoch weitere Schritte nötig, um die Software SMP-sicher zu machen. Die auf OUTPOST aufbauende Flugsoftware kann über RTEMS-Funktionsaufrufe verfügen, die in der SMP-Konfiguration geändert oder entfernt wurden. Zudem muss das Interrupt-Handling in der SMP-Konfiguration angepasst werden um zu definieren, welcher Prozessor welche Interrupts ausführen können soll. Zur Qualifizierung einer Flugsoftware in SMP-Konfiguration bieten RTEMS und Partnerfirmen Unterstützung, unter anderem in Form eines Qualification Data Packages (QDP)[11] und eines Abschlussberichtes zur Qualifizierung von RTEMS[21].

Bei Verwendung von OUTPOST auf einem anderen SoC kann der Einfluss der Prozessoren aufeinander durch den gemeinsamen Bus von dem in dieser Arbeit gemessenen abweichen. Abhängig ist dies unter anderem von der Cache Policy. So verwendet beispielsweise der Nachfolger des GR712RC, das GR740, nicht die write-through Policy, sondern die write-back Policy. Hier ist der Einfluss der Prozessoren stärker durch die Affinität der Threads beeinflussbar. Bei der write-back Policy werden nur Daten in den Hauptspeicher geschrieben und somit der Bus belastet, wenn sie sich nicht im Cache befinden. Die Wahrscheinlichkeit, dass sich die Daten bereits im Cache befinden, sinkt bei der Migration von Tasks.

Bei Verwendung eines Systems mit mehr als zwei Prozessoren müssen Anpassungen in OUTPOST vorgenommen und die Zahl der möglichen Affinitäten eines Tasks erhöht werden. Bei Systemen mit einer höheren Prozessorzahl kann sich zudem die Integration des Clustered Schedulings in OUTPOST als sinnvoll erweisen. Mit steigender Prozessorzahl wird das Verhalten des Systems und gerade die dynamische Zuweisung von Tasks zu Prozessoren durch eine einzige Scheduler-Isntanz zunehmends unübersichtlichter.

Neben der Unterstützung des SMP-Betriebs durch RTEMS in OUTPOST kann in Zukunft auch die Notwendigkeit der Integration eines SMP-Betriebs durch weitere Betriebssysteme diskutiert werden. So bietet das in OUTPOST abstrahierte Echtzeitbetriebssystem FreeRTOS ebenfalls eine SMP-Konfiguration [12].

Literaturverzeichnis

[1] : *10 Real Life Beispiele für eingebettete Systeme.* – URL https://de.digi.com/blog/post/examples-of-embedded-systems

[2] : *DLR - Institut für Raumfahrtsysteme.* – URL https://www.dlr.de/irs/

[3] : *DLR - Institut für Raumfahrtsysteme - S2TEP.* – URL https://www.dlr.de/irs/desktopdefault.aspx/tabid-12525/21846_read-49985/

[4] *GR712RC Dual-Core LEON3FT SPARC V8 Processor User's Manual*

[5] : *outpost-core.* – URL https://gitlab.dlr.de/avionics-software-products/outpost-core

[6] : *outpost-platform-leon.* – URL https://gitlab.dlr.de/avionics-software-products/outpost-platform-leon

[7] : *outpost-satellite.* – URL https://gitlab.dlr.de/avionics-software-products/outpost-satellite

[8] : *RTEMS Real Time Operating System (RTOS).* – URL https://www.rtems.org/

[9] : *RTEMS Classic API Guide (5.0.0 (master)).* 2018. – URL https://ftp.rtems.org/pub/rtems/people/joel/docs-eng/c-user/index.html

[10] : *GR712RC Dual-Core LEON3FT SPARC V8 Processor.* 2021. – URL https://www.gaisler.com/index.php/products/components/gr712rc

[11] : *RTEMS SMP QDP.* 2021. – URL https://rtems-qual.io.esa.int/

[12] : *Symmetric Multiprocessing (SMP) wirh FreeRTOS.* 2021. – URL https://freertos.org/symmetric-multiprocessing-introduction.html

[13] Andersson, Jan ; Hjorth, Magnus ; Johansson, Frederik ; Habinc, Sandi: LEON Processor Devices for Space Missions. First 20 Years of LEON in Space. (2017)

[14] ARNOLD, Heinz: *Multicore-RTOS für Satelliten. RTEMS für Raumfahrt qualifiziert.* 2022. – URL https://www.elektroniknet.de/embedded/software/rtems-fuer-raumfahrt-qualifiziert.192871.html

[15] BLOOM, Gedare ; SHERILL, Joel ; HU, Tingting ; BERTOLOTTI, Ivan C.: *Real-Time Systems Development with RTEMS and Multicore Processors.* CRC Press, 2020

[16] BODE, Pradip: *Encyclopedia of Parallel Computing.* Springer US, 2011

[17] BRAUN, Christian ; BENGEL, Günther ; KUNZE, Marcel ; STUCKY, Karl-Uwe: *Rechnerarchitekturen für Parallele und Verteilte Systeme.* Springer Verlag, 2015

[18] GAISLER, Jiri: A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture / Gaisler Research. – Forschungsbericht

[19] HUANG, Tian ; ZHU, Yongxin ; QIU, Meikang ; YIN, Xiaojing ; WANG, Xu: *Extending Amdahl's law and Gustafson's law by evaluating interconnections on multi-code processors.* Springer Science+Business Media New York, 2013

[20] LEITENBERGER, Bernd: *Die Umgebungsbedingungen im Weltraum.* – URL https://www.bernd-leitenberger.de/umgebungsbedingungen.shtml

[21] SPACEBEL ; EMBEDDED BRAINS ; UNIVERSITY OF PADUA: *Development Environment for Future LEON Multi-Core. RTEMS SMP Final Report.* Spacebel, 2015

[22] VAHID, Frank ; GIVARGIS, Tony D.: *Embedded System Design. A Unified Hardware/Software Introduction.* Wiley, 2002

A Vor- und Nachteile der Messwerkzeuge

Tabelle A.1: Vor- und Nachteile der Methoden zur Messung der Auslastung

	Vorteile	**Nachteile**
MSCR	• Keine direkte Ausbremsung der Tasks auf der ersten CPU • Schnelle Ausgabe der Ergebnisse • Ausgabe der Messergebnisse über Terminal kann individuell angepasst werden	• Belastet gemeinsamen Bus • Zusätzliche Klasse nötig • Keine gemeinsame Referenz der Messergebnisse für Uniprozessor- und SMP-Konfiguration (Null-Linie)
Cpu Usage Statistics	• Laufzeit-Informationen für alle Tasks • Einbinden/Handling ist einfach	• Ausgabe dauert relativ lange • Auswertung vieler Messreihen erfordert manuelle Sortierung oder ein zusätzliches Programm • Keine explizite Differenzierung zwischen den Prozessoren
Tracing Framework	• Auswertung der Ausgabe geht schnell • Viele Informationen über Softwareverhalten (Auslastung, Laufzeiten der Threads, Statistiken...)	• Freier Speicher für Buffer nötig • Ausgabe dauert sehr lange • Ausbremsung der Tasks, da für das Event Recording Funktionen aufgerufen werden • Extra Programm zur Auswertung nötig • Aufnahmen sehr fehleranfällig

B Software-Versionen

Software	**Version**
Debian	10
Eclipse	4.21.0
Eclipse Trace Compass	7.1.0
GRMON	3
RTEMS	5
SCons	3.0.1

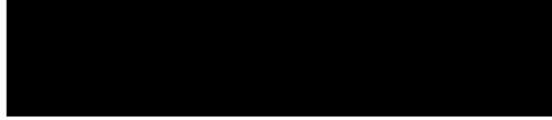
C Digitaler Anhang

Der digitale Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden. Der digitale Anhang enthält folgende Dateien:

- Die vorliegende Masterarbeit als .pdf-Datei
- Den Sourcecode der Benchmark-Applikation inklusive OUTPOST als .zip-Datei
- Die in dieser Arbeit aufgenommenen Messwerte inklusive Programm zum Sortieren der CPU Usage Statistics als .zip-Datei

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.



Ort	Datum	Unterschrift im Original