Neue Implementierungsmethoden für eingebettete geberlose Motor-Controller basierend auf dem Einsatz von Multi-Core-Mikrocontrollern

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultät
der Universität des Saarlandes

von
Eric Wagner

Saarbrücken
2021

Tag des Kolloquiums: 09.12.2021

Dekan: Prof. Dr. rer. nat. Jörn Walter

Berichterstatter:
Prof. Dr.-Ing. Matthias Nienhaus
Prof. Dr. rer. nat. Martina Lehser
Prof. Dr.-Ing. Chihao Xu

Vorsitz: Prof. Dr.-Ing. Michael Vielhaber

Akad. Mitarbeiter: Dr. rer. nat. Michael Roland

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Embedded Robotics Lab der Hochschule für Technik und Wirtschaft des Saarlandes in Kooperation mit dem Lehrstuhl für Antriebstechnik der Universität des Saarlandes. An dieser Stelle möchte ich mich bei allen bedanken, die mich auf dem Weg zu dieser Arbeit begleitet und unterstützt haben.

An erster Stelle möchte ich Frau Prof. Dr. rer. nat. Martina Lehser einen ganz besonderen Dank aussprechen. Durch ihr Engagement beim Auf- und Ausbau der Forschungsgruppe des Embedded Robotics Lab wurde es mir ermöglicht, zahlreiche Forschungsprojekte mitzugestalten und daran mitzuwirken. Ohne diese Erfahrungen wäre mein Weg ein anderer gewesen und die vorliegende Arbeit nicht entstanden.

Ein ganz besonderer Dank gilt ebenso Herrn Prof. Dr.-Ing. Matthias Nienhaus für die sehr gute Betreuung während meiner Forschungsarbeiten und dem Anfertigen der Dissertation sowie für die konstruktiven Ideen und Ratschläge. Sein großes Engagement und Offenheit für gemeinsame Forschungsprojekte haben mir den Weg zur Promotion eröffnet.

Mein herzlicher Dank gilt außerdem Prof. Dr.-Ing. Steffen Knapp und meinen Kollegen Christoph Karls, Mario Korherr, Michael Sauer und Philip Hoffmann, auf die ich mich immer verlassen konnte und die mir in stressigen Phasen stets den Rücken freigehalten haben. Ich danke auch allen anderen Mitarbeitern und Hilfskräften des Embedded Robotics Lab, die mich mit Rat und Tat unterstützt haben.

Ebenfalls möchte ich mich bei Martin Becker, Dr.-Ing. Emanuele Grasso, Stephan Kleen, Niklas König, Chris May, Klaus Schuhmacher, Robert Schwartz und Elke Zarbock vom Lehrstuhl für Antriebstechnik für ihre vielfältige Unterstützung und Geduld bei meinen zahlreichen Fragen bedanken.

Ein weiterer Dank gilt Horst Lehser und Dr. rer. net. Willi Theiß für die mir am Telefon gewidmete Zeit und die fachlichen Diskussionen.

Ebenso bedanke ich mich bei Sebastian Barth für die Durchsicht der englischsprachigen Veröffentlichungen.

Nicht zuletzt möchte ich meinen Eltern, meiner Schwester Silke und meiner Freundin Anne danken, die mich in der langen Zeit des Forschens und Schreibens stets unterstützt und mir mit großer Geduld zur Seite gestanden haben.

Saarbrücken, Dezember 2021 Eric Wagner

Kurzfassung

Diese Arbeit behandelt den Einsatz von Multi-Core-Mikrocontrollern in softwarebasierten Motor-Controllern zur geberlosen Stromregelung von PMSM. Hierbei werden die Steigerung der Motor-Controller-Performance durch Parallelisierung und die Auswirkungen von Cross-Core-Interferenzen auf das zeitliche Verhalten von Motor-Controllern fokussiert.
Es wird eine Generalisierung von geberlosen Stromregelungen durchgeführt, um die allgemeine Parallelisierbarkeit dieser Anwendungen zu beschreiben. Hierauf aufbauend wird gezeigt, unter welchen Rahmenbedingungen eine effektive Steigerung der Regelfrequenz durch Parallelisierung realisierbar ist.
Durch die Konsolidierung dedizierter Motor-Controller in ein Multi-Core-System kann Hardware eingespart und dadurch der Energiebedarf um bis zu 50 % gesenkt werden. Eine solche Konsolidierung verursacht regelmäßig Cross-Core-Interferenzen. Um diesem Problem entgegenzuwirken, wird ein Verfahren vorgestellt, das die negativen Einflüsse dieser Seiteneffekte auf die Laufzeiten der Motor-Controller analysiert und quantifiziert. Hierauf aufbauend werden Strategien zur Reduktion der Interferenzen beschrieben und evaluiert.
Durch die erzielten Ergebnisse werden Multi-Core-Mikrocontroller als Basis neuer Implementierungsmethoden für Motor-Controller erschlossen. Sie erweitern deren Design- und Implementierungsprozesse, um hier Multi-Core-Mikrocontroller durch Parallelisierung und Konsolidierung effektiv und effizient einzusetzen.

Abstract

This thesis addresses the use of multi-core microcontrollers in software-based motor controllers for the position sensorless control of PMSM. The focus is on increasing the motor controller performance through parallelization and on the effects of cross-core interferences on the temporal behaviour of motor controllers.
A generalization of position sensorless current controls is carried out in order to describe the general possibilities to parallelize these applications. Building on this, it is shown under which conditions an effective increase in the control frequency can be achieved through parallelization.
By consolidating dedicated motor controllers into one multi-core system, hardware can be saved to reduce energy costs by up to 50 %. Such consolidation causes cross-core interference on a regular basis. To address this problem, a procedure is presented that analyses and quantifies the negative influences of these side effects on the runtimes of the motor controllers. Based on this, strategies for reducing interference are described and evaluated.
The results of this work open up multi-core microcontrollers as a base for new implementation methods for motor controllers. They expand the design and implementation processes of motor controllers in order to use multi-core microcontrollers effectively and efficiently through parallelization and consolidation.

Inhaltsverzeichnis

1 Einleitung **1**

2 Stand der Technik **3**

2.1 Motor-Controller für geberlose Antriebe 3

2.1.1 Implementierungsmethoden von Motor-Controllern 3

2.1.2 Feldorientierte Stromregelung 6

2.1.3 Geberlose Rotorlagedetektion 8

2.1.4 Parallele Systemarchitekturen in der Antriebstechnik 10

2.2 Multi-Core-Systeme 13

2.2.1 General-Purpose Multi-Core-Prozessoren 15

2.2.2 Graphics Processing Units 15

2.2.3 Eingebettete Systeme 16

2.2.4 Cross-Core-Interferenzen 18

2.3 Zusammenfassung und Ableitung offener Forschungsfragen 20

2.3.1 Untersuchung der Parallelisierbarkeit von geberlosen Stromregelungen 21

2.3.2 Untersuchung von Cross-Core-Interferenzen bei konsolidierten Motor-Controllern 22

3 Zielsetzung und Aufbau der vorliegenden Arbeit **24**

3.1 Zielsetzung 25

3.2 Betrachtete Algorithmen und Technologien 27

3.2.1 Algorithmen zur geberlosen Stromregelung 27

3.2.2 Multi-Core-Plattform 28

3.3 Abgrenzung und Beitrag zum Stand der Technik 29

3.4 Aufbau dieser Arbeit 30

4 Multi-Core Motor-Controller **32**

4.1 Multi-Core-Architekturen für eingebettete Motor-Controller 32

4.2 Software-Plattform des Multi-Core Motor-Controllers 36
4.2.1 Softwarestrukturen zur Parallelisierung 37
4.2.2 Softwarestruktur der Konsolidierung 40
4.2.3 Multi-Core-Betriebssystem 40
4.3 Zeitliche Kosten der parallelen Ausführung 42
4.4 Zusammenfassung der Ergebnisse 43

5 Energiebetrachtung des Multi-Core-Motor-Controllers 46
5.1 Bestimmung der Leistungsaufnahme 46
5.1.1 Systemmodell des Multi-Core-Systems 47
5.1.2 Ermittlung der statischen Leistungsaufnahme 48
5.1.3 Ermittlung der anwendungsabhängigen Leistungsaufnahme . . 48
5.2 Direkter Vergleich der betrachteten Systeme 49
5.3 Leistungsaufnahme der Multi-Core Nutzungsszenarien 51
5.3.1 Leistungsaufnahme nach Konsolidierung 51
5.3.2 Leistungsskalierung bei Parallelisierung 53
5.4 Zusammenfassung der Ergebnisse 55

6 Parallelisierung geberloser Antriebsregelungen 56
6.1 Taskmodell der geberlosen Stromregelung 56
6.1.1 Allgemeine Form des Taskmodells 56
6.1.2 Integration der GRB in die Stromregelung 57
6.1.3 Datenfluss des Taskmodells 59
6.2 Parallelität im Taskmodell 62
6.2.1 Parallele Ausführung der GRB 62
6.2.2 Datenströme innerhalb der Tasks 65
6.2.3 Parallelisierung von Rechenoperationen 66
6.2.4 Parallelisierung durch Pipelining 68
6.3 Evaluation der Parallelisierung von geberlosen Stromregelungen . . . 68
6.3.1 Parallele Ausführung mit zeitkontinuierlicher GRB 69
6.3.2 Parallele Ausführung mit zeitdiskreter GRB 70
6.3.3 Parallelisierung einzelner Tasks nach Datenströmen 72
6.3.4 Parallele Ausführung mittels einer Pipeline 75
6.3.5 Parallelisierung von Rechenoperationen 75
6.4 Interferenzen bei Parallelisierung 77
6.5 Zusammenfassung der Ergebnisse 78

7 Motor-Controller Konsolidierung und Cross-Core-Interferenzen 80
7.1 Zugriffe auf globale Module 81
7.2 Konkurrierende Zugriffe auf gemeinsame Module 83
7.2.1 Abläufe konkurrierender Zugriffe 83
7.2.2 Einflussfaktoren auf die Abstände aufeinanderfolgender Modulzugriffe 85
7.3 Parallele Zugriffe auf globale I/O-Module 86
7.3.1 Konsolidierung gleichartiger Motor-Controller 86
7.3.2 Konsolidierung unterschiedlicher Motor-Controller 89
7.4 Parallele Zugriffe auf globalen Arbeitsspeicher 90
7.4.1 Kommunikation zwischen Motor-Controllern 90
7.4.2 Erweiterung des lokalen Speichers 92
7.4.3 Einflüsse auf den zeitlichen Versatz 92
7.5 Messung von Cross-Core-Interferenzen 92
7.5.1 Maximale Laufzeiterhöhung durch Interferenzen 93
7.5.2 Laufzeiterhöhung bei aufeinanderfolgenden Zugriffen 95
7.5.3 Veränderung zeitlicher Abstände aufeinanderfolgender Modulzugriffsversuche 98
7.5.4 Verschieben von Startzeitpunkten 100
7.6 Zusammenfassung der Ergebnisse 102

8 Analytische Betrachtung von Cross-Core-Interferenzen 105
8.1 Aufbau des Instruction-Trace 107
8.2 Aufbau des Modulzugriffstrace 108
8.2.1 Zugriffe auf Speicher und Caches 109
8.2.2 Transaktionen einer Intra-Chip-Verbindung 110
8.2.3 Beschreibung der Store-Buffer-Zustände 111
8.3 Erzeugung des Modulzugriffstrace 113
8.3.1 Behandlung von Load-Anweisungen 114
8.3.2 Behandlung von Store-Anweisungen 117
8.3.3 Verarbeitung des Store-Buffer 118
8.3.4 Integration zusätzlicher Adressbereiche 118
8.4 Betrachtung der parallelen Ausführung 119
8.4.1 Abbildung der Traces auf eine gemeinsame Zeitbasis 120
8.4.2 Berechnung von Wartezeiten bei konkurrierenden Modulzugriffen 120
8.4.3 Integration der Wartezeit in Instruktion- und Modul-Timings . 123

8.5 Evaluation der Interferenzanalyse 125
8.5.1 Bestimmung der Laufzeiterweiterung 125
8.5.2 Optimierung der parallelen Ausführung 127
8.6 Zusammenfassung der Ergebnisse 129

9 Diskussion und Übertragbarkeit der Ergebnisse 131
9.1 Leistungsaufnahme von Multi-Core Motor-Controllern 131
9.2 Parallelisierung von Motor-Controllern 132
9.2.1 Zeitkontinuierliche geberlose Rotorlageberechnung 133
9.2.2 Zeitdiskrete geberlose Rotorlageberechnung 133
9.2.3 Parallelisierung einzelner Tasks 135
9.3 Analyse von Cross-Core-Interferenzen 136
9.3.1 Strukturelle Untersuchung von Cross-Core-Interferenzen . . . 136
9.3.2 Analytische Untersuchung von Cross-Core-Interferenzen 137
9.4 Übertragbarkeit der Ergebnisse 138
9.4.1 Übertragbarkeit auf andere Multi-Core-Plattformen 139
9.4.2 Anwendbarkeit der Parallelisierung auf alternative Algorithmen 139
9.4.3 Anwendbarkeit der Berechnung von Interferenzen auf alternative Multi-Core-Mikrocontroller 140
9.4.4 Anwendbarkeit der Energiebetrachtung auf alternative Multi-Core-Mikrocontroller 140
9.4.5 Übertragbarkeit der gemessenen Laufzeiten 140

10 Fazit und Ausblick 142

Literaturverzeichnis 146

Abbildungsverzeichnis 165

Tabellenverzeichnis 167

Algorithmenverzeichnis 168

Abkürzungen und Symbole 169

Kapitel 1

Einleitung

Multi-Core-Architekturen haben sich bereits in vielen Anwendungsbereichen als Standard durchgesetzt, da sie eine hohe Rechenleistung bei vergleichsweise geringer Leistungsaufnahme bieten. Die Steigerung der Anwendungsperformance durch Parallelisierung von Algorithmen sowie die Kostenreduktion durch eine Systemkonsolidierung sind die treibenden Anwendungsszenarien für den Einsatz von Multi-Core-Systemen. Vor allem Anwendungen im Bereich eingebetteter Systeme, die unter eingeschränkten Ressourcen bezüglich Energie und Integrationsplatz realisiert werden müssen, können von der Verfügbarkeit mehrerer Prozessorkerne auf einem Mikrocontroller profitieren.

Eines der wenigen Anwendungsgebiete im Bereich eingebetteter Systeme, in dem sich Multi-Core-Mikrocontroller noch nicht etabliert haben, liegt in der elektrischen Antriebstechnik im Bereich der Regelung elektrischer Antriebe. Der zunehmende Einsatz geberlos geregelter bürstenloser Antriebe in hochintegrierten Systemen, beispielsweise in sogenannten Smart Drives, befördert auch in diesem Anwendungsgebiet den Einsatz von Multi-Core-Systemen. Bei einer geberlosen Antriebsregelung, insbesondere für permanenterregte Synchronmotoren (PMSM), werden diskrete Winkelsensoren durch eine softwareseitige Auswertung von Motorsignalen ersetzt, um für die Regelung des Antriebs notwendige Rotorlageinformationen zu gewinnen. Der Verzicht auf Winkelsensoren dient der Optimierung des Antriebssystems als auch der Reduktion des Platzbedarfs und der Kosten.

Die zusätzlichen Algorithmen zur Gewinnung der Rotorlage, eine steigende Integrationsdichte und hohe Effizienzansprüche bedingen eine Erhöhung der Rechenleistung seitens der eingesetzten Motor-Controller. Alternativ zum Einsatz hochgetakteter Mikrocontroller oder hardwarebasierter Implementierungen auf Basis von Field Programmable Gate Arrays (FPGA) können die parallelen Prozessorkerne ei-

nes Multi-Core-Mikrocontrollers prinzipiell die Rechenleistung von Motor-Controllern steigern, ohne die Anforderungen an Energie und Integrationsplatz signifikant zu erhöhen. Dies ist vor allem für batteriebetriebene Kleinantriebe interessant, da hier Motor-Controller einen erheblichen Anteil der Kosten und des Bauraums einnehmen können.

Die parallele Ausführung von Software auf mehreren Prozessorkernen kann negativen Seiteneffekten, sogenannten Cross-Core-Interferenzen, unterliegen. Interferenzen entstehen durch die Verwendung geteilter Ressourcen innerhalb des Multi-Core-Mikrocontrollers. Seiteneffekte können in Form von Laufzeiterhöhungen die zeitlichen Eigenschaften der Anwendungen beeinflussen und durch eine kernübergreifende Fehlerausbreitung die Integrität von Daten verletzen. Dies bedeutet eine Beeinträchtigung der Zuverlässigkeit des Motor-Controllers, wenn zu einer Multi-Core-Architektur gewechselt wird.

Um den Einsatz von Multi-Core-Mikrocontroller zur Realisierung von Motor-Controllern zu bewerten, untersucht diese Arbeit die Möglichkeiten, welche sich durch parallele Prozessorkerne für die Parallelisierung geberloser feldorientierter Stromregelungen sowie für die Konsolidierung entsprechender Motor-Controller ergeben.

Kapitel 2

Stand der Technik

Die geberlose Regelung von PMSM mittels digitaler, auf Mikrocontrollern basierenden Motor-Controllern bildet das Anwendungsgebiet für den in dieser Arbeit betrachteten Einsatz von Multi-Core-Systemen. In den folgenden Abschnitten werden die fokussierten Motor-Controller und Algorithmen zur geberlosen Regelung von PMSM beschrieben. Gleichermaßen wird für Multi-Core-Systeme verfahren.

2.1 Motor-Controller für geberlose Antriebe

Motor-Controller sind oftmals Teil eines Motion-Control-Systems, dessen Aufgaben in Datenerfassung und Datenverarbeitung, Systemlogik und Algorithmen, Schnittstellen zu Leistungswandlern und Kommunikation klassifiziert werden können [98]. Solche Systeme umfassen oftmals mehrere Recheneinheiten, die sich unter anderem aus unterschiedlichen Anforderungen bezüglich einer zeit- und sicherheitskritischen Integration der verschiedenen Funktionen ergeben. In Abhängigkeit von der Architektur des Motion-Control-Systems, der konkret zu realisierenden Anwendung und nicht zuletzt durch bevorzugte Hersteller können Motor-Controller in unterschiedlichen Bauformen realisiert sein. Unter anderem als PC- oder PLC-Erweiterungsmodule bis hin zu eigenständigen eingebetteten Systemen [17]. Durch kontinuierliche Fortschritte in der Miniaturisierung und Integration elektrischer Systeme stellen vollständig in das Antriebssystem eingebettete Lösungen eine heute oft bevorzugte Systemarchitektur dar.

2.1.1 Implementierungsmethoden von Motor-Controllern

Unabhängig von der konkreten Architektur eines Motor-Controllers können zur Funktionsimplementierung unterschiedliche Methoden genutzt werden. Diese un-

terteilen sich in die hardwarebasierte Implementierung, die Verwendung von per Software programmierbarer Mikrocontroller und hybride Ansätze. Die Wahl einer Methode ist von Anforderungen an die zu erreichende Performance der Antriebsregelung, an die Wart- und Erweiterbarkeit des Systems und nicht zuletzt vom verfügbaren technischen Know-how abhängig.

2.1.1.1 Hardwarebasierte Implementierung

Werden Algorithmen per Hardware realisiert, so geschieht dies vorzugsweise mittels FPGA. Sie realisieren Algorithmen in Form von logischen Schaltungen. Aufgrund ihrer inhärenten Parallelität ermöglichen sie eine sehr schnelle Anwendungsausführung [26][71]. Dies ermöglicht die Umsetzung von High-Speed- und High-Performance-Systemen, bei denen eine sehr hohe Bandbreite seitens der Regelung gefordert wird. Bei einem entsprechenden Design können auch anspruchsvolle Algorithmen im niedrigen einstelligen Mikrosekundenbereich ausgeführt werden [71]. Hierdurch wird eine quasi verzögerungsfreie Ausführung möglich, die der Regelperformance analoger Systeme nahekommt [113]. Die hardwarebasierte Implementierung verlangt die Anwendung entsprechender Programmiersprachen wie Very High Speed Integrated Circuit Hardware Description Language oder Verilog. Deren Anwendung bedingt zum einen tiefes Wissen über das zugrundeliegende digitale System, zum anderen ist die performante Implementierung komplexer Regelanwendungen zeitaufwendig [3]. Zwar wird eine solche Implementierung durch leistungsstarke Software-Tools zunehmend vereinfacht [134], jedoch reichen diese noch nicht an die Anwendungsentwicklung für Mikrocontroller mittels Hochsprachen heran. Weiterhin steigt mit der Komplexität der Anwendungen auch die benötigte Chipfläche, die einen signifikanten Kostenfaktor darstellt.

2.1.1.2 Softwarebasierte Implementierung

Zur softwarebasierten Implementierung werden typischerweise Mikrocontroller [38] in Form von digitalen Signalprozessoren (DSP) beziehungsweise System-on-Chips mit Prozessortaktraten bis zu 200 MHz eingesetzt [104][55]. Der Einsatz von Floating-Point Units und auf digitale Signalverarbeitung angepasste Instruktionssätze, die Integration einer Vielzahl leistungsfähiger Peripheriemodulen wie Kommunikationsschnittstellen, Timer, Analog-Digital-Converter (ADC) und Module zur Erzeugung einer mehrkanaligen Pulsweitenmodulation (PWM) ermöglichen die Umsetzung anspruchsvoller Anwendungen [23]. Die Verwendung etablierter Hochsprachen ermöglicht eine flexible Anwendungsimplementierung und Integration, wodurch sich

neben kurzen Entwicklungszeiten eine hohe Wart- und Erweiterbarkeit der Systeme ergibt [72].

2.1.1.3 Hybride Implementierung

Hybride Systeme vereinen die Fähigkeiten und Eigenschaften der beiden oben genannten Ansätze. Sie können als Multiprozessorsysteme realisiert sein, die FPGA und vollständige Mikrocontroller innerhalb des Motor-Controllers integrieren [111]. Weitere Ansätze realisieren programmierbare Mikroprozessoren direkt auf der Chipfläche des FPGA. In Form von Hard-Cores werden diese Prozessoren nativ durch zusätzliche Schaltungen integriert, während Soft-Cores durch den FPGA synthetisiert werden [14]. Die Abbildung komplexer Prozessorarchitekturen beansprucht jedoch einen hohen Bedarf an Logikressourcen des FPGA und bedeutet damit einen Anstieg der Kosten. Im Vergleich mit nativen Mikroprozessoren erreichen Soft-Cores eine geringere Rechenleistung und zeigen einen höheren Energieverbrauch [106].

2.1.1.4 Abgrenzung der Implementierungsmethoden

Performance und Kostenfaktoren bezüglich der Entwicklung, Erweiterbarkeit und Wartung des zu entwickelnden Systems sind die zu betrachtenden Faktoren zur Abgrenzung der Verwendung von hard- und softwarebasierten Integrationsansätzen. Die Komplexität einer Anwendung und die Abhängigkeit ihrer Daten voneinander wirken sich auf die Kosten- und Performancefaktoren aus und können als Metriken zur Abgrenzung der Paradigmen verwendet werden [113]. Abbildung 2.1 zeigt die Einordnung der Systeme bezüglich der Anwendungskomplexität und der Datenabhängigkeit [115]. Durch ihre inhärente Fähigkeit, parallele Verarbeitungsschritte abbilden zu können, ist die hardwarebasierte Implementierung vor allem für Anwendungen mit geringen Datenabhängigkeiten geeignet. Entsprechende Algorithmen können somit eine sehr kurze Ausführungszeit erreichen. Mit steigender Datenabhängigkeit steigt auch der Anteil an seriell auszuführendem Code. Die Instruktionssätze von Mikrocontrollern sind auf eine performante Ausführung von solchem Code ausgelegt, sodass hier ein nur geringfügiger Performance-Unterschied gegenüber FPGA zu erwarten ist. Anwendungen mit vielen und heterogenen Operationen bedeuten bei der hardwarebasierten Integration einen signifikanten Anstieg der Komplexität der zu entwerfenden Schaltung und damit auch einen erhöhten Bedarf an Chipfläche. Die Schnittmenge beider Ansätze zeigt den Bereich, der für beide Paradigmen geeignet ist. Seitens FPGA ergibt sich dieser unter anderem durch den Einsatz von Soft-Cores. Andererseits sind auch Mikrocontroller durch komplexe Befehlssätze in

der Lage, Datenabhängigkeiten in Form von Instruction-Level-Parallelität zu einem gewissen Maß auszunutzen.

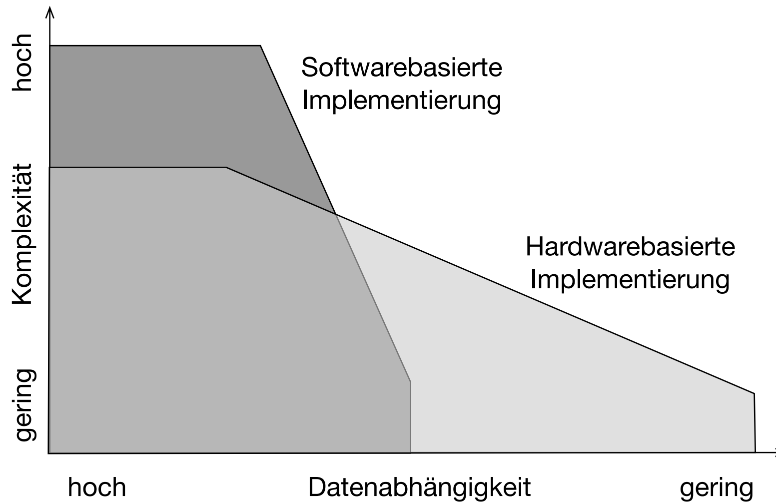


Abbildung 2.1: Abgrenzung von hard- und softwarebasierten Integrationsmethoden

2.1.2 Feldorientierte Stromregelung

Die grundlegenden Funktionen des Motor-Controllers liegen in der Antriebsregelung, die für PMSM standardmäßig auf einer feldorientierten Stromregelung (FOS) nach [19] basiert. Ziel dieser vektorbasierten Regelung ist es, Drehfeldmaschinen über eine feldbildende und eine drehmomentbildende Stromkomponente zu regeln. Hierzu werden die sinusförmigen Wechselgrößen beider Komponenten mittels geeigneter Transformationen stationär als Gleichgrößen betrachtet und separat geregelt. Abbildung 2.2 beschreibt den grundlegenden Aufbau einer FOS. Über eine Clark-Transformation werden die Wechselgrößen zunächst in ein statorfestes α-β-Koordinatensystem überführt. Anschließend werden die α-β-Ströme mittels einer Park-Transformation und dem Winkel der elektrischen Rotorlage in das rotorfeste d-q-Koordinatensystem transformiert. Der aus den d-q-Komponenten resultierende Raumzeiger beschreibt die Flussrichtung und den Betrag das Statorfeldes zum Zeitpunkt der Messung der Strangströme. Wird die d-Achse entlang des Rotorflusses angeordnet, so beschreibt sie den feldbildenden Anteil des Raumzeigers. Die um 90° zur Rotorachse angeordnete q-Achse repräsentiert den drehmomentbildenden Anteil des Zeigers. Innerhalb des rotierenden d-q-Koordinatensystems können beide Stromkomponenten als Gleichgrößen betrachtet und über PI(D)-Regelkreise separat geregelt werden. Zur Regelung von Geschwindigkeit und Position der PMSM kann die FOS durch entsprechende Regelkreise erweitert werden, die letztendlich

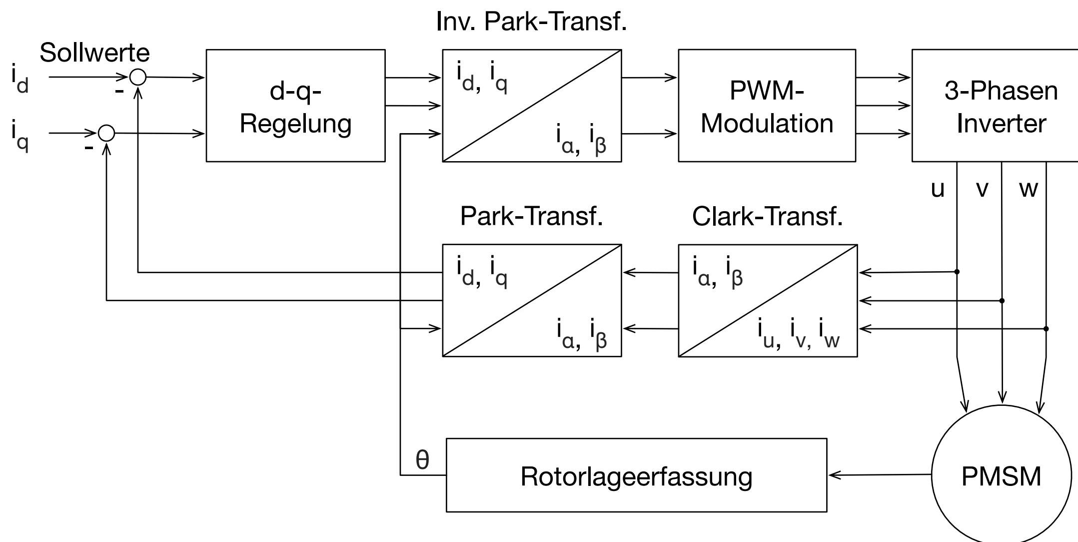


Abbildung 2.2: Aufbau der feldorientierten Stromregelung

die Führungsgrößen für die Stromregelung bereitstellen. Die aus der Regelung resultierenden d-q-Sollgrößen werden über eine inverse Park-Transformation in das α-β-Koordinatensystem zurücktransformiert. Der aus den α-β-Komponenten resultierende Raumzeiger repräsentiert den Spannungsvektor, der zur Realisierung des maximalen Drehmoments erzeugt werden muss. Hierzu wird eine Raumzeigermodulation eingesetzt [161], welche die Schaltmuster der dreiphasigen PWM-Signale zur Erzeugung der Spannungen an den Motorsträngen berechnet.

Die Regelfrequenz des zu regelnden Antriebs ist ein maßgeblicher Faktor für die Bandbreite und Stabilität der Regelung. Ausschlaggebend ist hier das Verhältnis zwischen der PWM-Frequenz und der Frequenz der Strangströme, das mindestens einen Wert von 10 aufweisen sollte [175]. Ein zu gering gewählter Wert kann die Performance der Regelung beeinträchtigen [137] und insbesondere bei hohen Drehzahlen oder kleinen Zeitkonstanten zur Instabilität des Regelkreises führen [99]. Gründe hierfür sind unter anderem Stromoberschwingungen und daraus resultierende Drehmomentrippel, die durch eine Erhöhung des Frequenzverhältnisses reduziert werden können [100]. Für High-Performance-Anwendungen, die neben hohen Drehzahlen auch eine hohe Regeldynamik oder eine unmittelbare Rückkehr in den stationären Zustand nach einem Lastwechsel fordern können, sind PWM-Frequenzen im Bereich von 100 kHz keine Seltenheit [147].

Durch die digitale Umsetzung der Regelung entsteht eine Verzögerung zwischen der Signalerfassung und dem Aktivieren des für den nächsten Regelzyklus berechneten Schaltmusters [113]. Idealerweise ist diese Verzögerung möglichst klein. Durch das Weiterdrehen des Rotors während der Rechenzeit der Regelung entsteht eine

Differenz zwischen dem α-β-Referenzrahmen zu Beginn und Ende des Regelintervalls. Diese Differenz wirkt als Phasen- und Amplitudenfehler auf den von der Regelung berechneten Spannungsvektor [9]. Qualitativ können sich hierdurch, verstärkt bei höheren Geschwindigkeiten, das Drehmoment, der stabile Bereich [58] sowie die Bandbreite und Dynamik der Regelung verringern [72].

Abbildung 2.3 zeigt die typischen zeitlichen Abläufe der Regelung auf einem Mikroprozessor. Die Ausführung richtet sich nach dem Takt der PWM. Die Periodendauer der PWM definiert dabei den zeitlichen Rahmen, in dem alle Aktionen durchgeführt werden müssen. Die Konfiguration der Hardwaremodule zum Schalten der PWM-Muster muss vor dem Beginn des neuen Zyklus abgeschlossen sein.

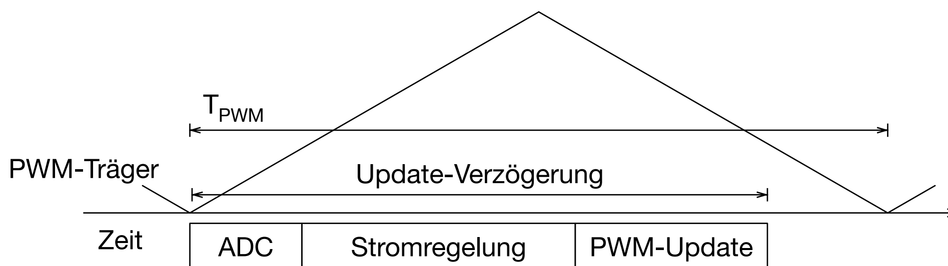


Abbildung 2.3: Abläufe der Regelung auf einem Mikroprozessor

2.1.3 Geberlose Rotorlagedetektion

Zum Betrieb und zur Regelung von PMSM sind Informationen über die elektrische Rotorlage der Maschine notwendig. Diese Informationen werden typischerweise durch physikalische Lagegeber, beispielsweise Resolver oder Encoder, gewonnen. Der Einsatz solcher Hardware ist grundsätzlich mit mechanischen und elektrischen Aufwänden verbunden, welche die Systemkomplexität, die Wahrscheinlichkeit von Fehlern und die Kosten erhöhen. Zur Vermeidung dieser Nachteile wurden zahlreiche Verfahren veröffentlicht, um die benötigten Rotorlageinformationen ohne die Verwendung diskreter Messeinrichtungen zu gewinnen. Entsprechende Techniken beruhen hauptsächlich auf der Ausnutzung von zwei physikalischen Effekten: die induzierte Gegen-EMK (elektromotorische Kraft) und das Vorhandensein von magnetischer Anisotropie im elektrischen Antrieb [176]. Entsprechende Algorithmen erweitern die FOS um zusätzliche Analysen von Strangströmen als auch Strangspannungen.

2.1.3.1 Verfahren basierend auf Gegen-EMK

Die ersten Ansätze zur geberlosen Ansteuerung von PMSM basieren auf der Auswertung des Gegen-EMK-Signals, dessen Spannung proportional zur Rotordrehzahl ist. Nichtadaptive Verfahren beruhen auf der direkten Auswertung der Gegen-EMK. Eine vergleichsweise einfache Methode kann durch das Beobachten des Nulldurchgangs der rückinduzierten Spannung zur Realisierung einer Blockkommutierung für Brushless DC (BLDC) Antriebe eingesetzt werden [123]. Adaptive Ansätze basieren auf Model Reference Adaptive System Techniken [140], auf Zustandsbeobachter wie Luenberger-Beobachter [105], Sliding-Mode Observer (SMO) [174] oder auf Kalman-Filter [71]. Zustandsbeobachter stellen mathematische Modelle dar, die das Verhalten nichtlinearer realer Referenzsysteme nachbilden und die zur Berechnung der Rotorlage benötigten Informationen schätzen. Ziel ist es, den Schätzfehler über die Zeit zu minimieren und das Modell gegen das reale System konvergieren zu lassen. Im Gegensatz zur deterministischen Zustandsschätzung führen Kalman-Filter eine stochastische Schätzung der Zustände durch. Dies wird mittels Wissen über System- und Messdynamiken, statistische Beschreibungen von Störungen, Messfehlern und Ungenauigkeiten im Systemmodell sowie Informationen über die initialen Zustände des Systems ermöglicht [34].

2.1.3.2 Verfahren basierend auf magnetischer Anisotropie

Der Hauptnachteil der auf Gegen-EMK basierenden Verfahren ist ihre Nichtanwendbarkeit bei niedrigen Geschwindigkeiten beziehungsweise im Stillstand. Diesen Nachteil haben Verfahren basierend auf magnetischer Anisotropie prinzipiell nicht. Magnetische Anisotropien verursachen in charakteristischer Art eine Veränderung der magnetischen Eigenschaften des PMSM, die sich durch messbare Änderung ihrer Induktivitäten ausdrückt. Eine wesentliche Eigenschaft der magnetischen Anisotropie ist die Korrelation ihrer räumlichen Ausprägung mit der Rotorposition. Als konstruktive Größe ist der Einfluss der Induktivität auf die elektrischen Parameter der Maschine drehzahlunabhängig. Die hier etablierten Verfahren können nach PWM ausnutzenden und nicht-ausnutzenden Ansätzen gruppiert werden.

Erste Ansätze zur Nutzung anisotroper Effekte sind durch das Verfahren Indirect Flux-detection by Online Reactance Measurement (INFORM) beschrieben [143] [144]. Es nutzt zur Berechnung von Rotorlageinformationen die Abhängigkeit der Phasenreaktanzen von der Rotorposition. INFORM führt eine Online-Messung der Motorreaktanzen durch die Messung von Strömen durch, die sich aus der Einspeisung von Testimpulsen ergeben. Die Testimpulse basieren auf der Verwendung eines

modifizierten PWM-Treibersignals. Aus diesem Grund kann INFORM als PWM ausnutzende Technik angesehen werden.

Verfahren, die nicht auf der Ausnutzung der PWM zur Erregung der Maschine beruhen, basieren auf dem Ansatz High Frequency Current Injection (HFCI) [85][28]. Kern dieser Verfahren ist die Modellierung der Abhängigkeit der Spuleninduktivität von der Rotorposition, die sich durch eine hochfrequente Anregung der Maschine ergibt. Hierzu werden hochfrequente Trägersignale in den Motor eingespeist. Die Frequenz dieser Signale liegt typischerweise um den Faktor 10 über der Drehzahlfrequenz. Die aus der Erregung resultierenden hochfrequenten Ströme werden durch die Rotorposition moduliert. Daher können nach einer entsprechenden Strommessung eine Demodulation durchgeführt und mittels Beobachter der Rotorwinkel rekonstruiert werden.

Im Fall von PMSM mit Sternverschaltung und zugänglichem Sternpunkt können Stromsensoren zur Rotorlagebestimmung vermieden werden. Das Verfahren Direct Flux Control (DFC) gewinnt Informationen zur Online-Bestimmung der rotorlageabhängigen Motorinduktivitäten durch Messungen des Spannungspotenzials zwischen dem realen und einem künstlichen Sternpunkt [152][158]. Hierzu ermittelt DFC für jede Phase die Induktivität. Die Generierung der dazu notwendigen Testsignale erfolgt direkt über das zur Ansteuerung des Inverters verwendete PWM-Muster. Die Erregung der Maschine erfolgt dabei durch die Sprünge der PWM, wodurch deren Ansteuerung, im Gegensatz zu INFORM, nicht unterbrochen werden muss. Um die Signale der einzelnen Phasen am Sternpunkt voneinander zu separieren, wird das in Abbildung 2.4 gezeigte PWM-Muster verwendet. In jeder PWM-Periode wird die zu messende Phase vor den beiden anderen aktiv geschaltet. Die Messung des Sternpunktpotentials wird vor und nach dem PWM-Sprung durchgeführt. Bei einem dreiphasigen Motor sind entsprechend viele PWM-Perioden erforderlich, um einen vollständigen Messdatensatz zur Bestimmung der Rotorlage zu erhalten. Dementsprechend beträgt die Frequenz der Stromregelung unter Verwendung von DFC ein Drittel der PWM-Frequenz. Durch das angepasste Pulsmuster kann DFC als PWM ausnutzende Technik betrachtet werden.

2.1.4 Parallele Systemarchitekturen in der Antriebstechnik

Im Bereich der Ansteuerung und Regelung elektrischer Antriebe konzentrieren sich Forschung und Entwicklung primär auf den Entwurf und die Optimierung entsprechender Verfahren und weniger auf deren Implementierung auf Mikrocontrollern. Multi-Core-Mikrocontroller finden im Bereich der Regelung von PMSM üblicher-

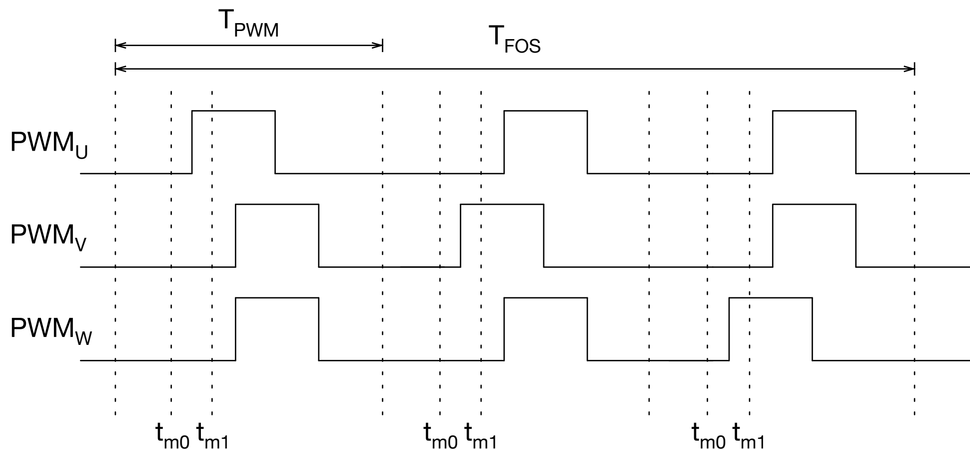


Abbildung 2.4: PWM-Muster zur Messung der Spannungspotenziale zwischen realem und künstlichem Sternpunkt [55]

weise keinen Einsatz. Wird hier allgemein der Einsatz paralleler Recheneinheiten betrachtet, so finden sich vor allem hybride Systeme, die FPGA und Mikrocontroller zur Funktionsintegration kombinieren. Die Motivation zur Verwendung mehrerer Recheneinheiten liegt hier in der Möglichkeit, gleichzeitig komplexe Funktionen mittels Software effizient implementieren und die kurzen Reaktionszeiten hardwarebasierter Implementierungen nutzen zu können. Letztere werden weniger durch die explizite Parallelisierung der Algorithmen erreicht, sondern durch die Verwendung parallelisierter Rechenoperationen [5].

Eine Aufteilung der zu implementierenden Funktionen auf die Recheneinheiten geschieht in Abhängigkeit der jeweils verwendeten Hard- und Softwaresysteme als auch von den zu erfüllenden Anforderungen an die Anwendung. In [4] implementiert ein DSP die Strom- und Geschwindigkeitsregelung für zwei dreiphasige BLDC Antriebe. Das PWM-Modul des Mikrocontrollers erzeugt ein dreiphasiges Signalmuster. Der FPGA dient als Schnittstelle zwischen dem PWM-Modul und den Invertern der Antriebe. Er führt, gesteuert durch den DSP, das Routing des Signalmusters zwischen den Invertern durch. In [62] übernimmt der FPGA eine Weiterverarbeitung des durch einen DSP erzeugten PWM-Musters. Hierbei werden Boolesche Operatoren auf die Muster angewendet und die weitere Übertragung der resultierenden Signale zeitlich gesteuert. In [70][24] werden die in Hardware realisierten Funktionen durch die PWM-Generierung und die Steuerung von Peripherie wie ADC und digitale I/O erweitert. Software übernimmt hier weiterhin die Implementierung komplexer Algorithmen. Der Austausch von Daten geschieht hier auf Basis von dual-port RAM-Modulen [86], die durch den FPGA bereitgestellt werden. Zwischen DSP und

FPGA ist hierbei ein Datenaustausch mit Übertragungsraten bis zu 2,5 Gbit/s möglich [87]. In [35] realisiert der FPGA eine Hybrid-PWM, die mehrere Modulationsmethoden einsetzt. Der DSP übernimmt neben einer FOS die Parameterberechnung für die Hybrid-PWM und implementiert einen Zustandsautomaten zum Umschalten zwischen den einzelnen Methoden. Die Wahl der DSP-FPGA-Architektur ist hier durch das bessere zeitliche Verhalten und einer höheren Auflösung seitens der PWM-Generierung durch den FPGA motiviert. Geringere Kosten und eine effizientere Wartung bei der Implementierung komplexer Funktionen motiviert hier die Verwendung des DSP. Um den FPGA effizienter zu nutzen, wird in [33] neben der PWM-Erzeugung und Abtastung der Motorsignale auch die Stromregelung in Hardware realisiert. Der DSP implementiert eine Geschwindigkeitsregelung. Entsprechende Vorgaben erhält er durch eine Interaktion mit externen Systemen. In [89] ist der DSP um eine Positionsregelung erweitert. Weiterhin wird die Verwendung eines Echtzeitbetriebssystems zur nebenläufigen Implementierung der Regelkreise beschrieben.

In [107] wird eine hybride geberlose Regelung vorgestellt, die vollständig auf einem FPGA implementiert wird. Dies beinhaltet neben der FOS eine geberlose Rotorlagebestimmung mittels HFCI und eines EMK-Observer. Ein zusätzlicher, auf dem FPGA synthetisierter Soft-Core wird zum Debuggen der implementierten Algorithmen eingesetzt. In [97] realisiert der Soft-Core eine adaptive Geschwindigkeitsregelung, während die FOS und PWM-Generierung in Hardware realisiert sind. In [64] werden ausschließlich Soft-Cores eingesetzt, welche die Aktoren eines mehrbeinigen Roboters steuern. Fünf der insgesamt sechs Soft-Cores steuern jeweils die Antriebe eines Roboterbeins. Hierzu übernehmen sie die Auswertung benötigter Sensordaten und realisieren die PID-Regelkreise der Beinsteuerungen. Der sechste Soft-Core koordiniert die einzelnen Roboterbeine, um ihn fortzubewegen.

In [148] wird ein FPGA explizit zur Parallelisierung eines Algorithmus eingesetzt. Hier wird eine geberlose Rotorlageberechnung unter Verwendung eines Marginalized Particle Filter parallelisiert. Die Parallelisierung wird in Form einer Pipeline mit 61 Stufen auf dem FPGA realisiert. Nach der Berechnung des ersten Partikels mit einer Laufzeit von 72 Taktzyklen werden für jeden zusätzlichen Partikel jeweils nur zwei weitere Taktzyklen benötigt. Im Rahmen einer mehrphasigen Stromerfassung mittels eines Delta-Sigma-Modulators wird in [93] die parallele Nutzung eines digitalen Dezimierungsfilters zur effizienten Umsetzung des Datenstromes in digitale Datenworte gezeigt. Ein FPGA führt hierbei zwei Instanzen des Filters parallel aus und führt die Ergebnisse einem PI-Regelkreis und einer Überstromerkennung zu.

Weiterhin werden die P- und I-Komponenten des Reglers parallel zueinander berechnet. Auch in [177] wird die Parallelisierung eines Regelkreises betrachtet. Hier wird die parallele Ausführung der P-, I- und D-Komponenten des Regelkreises auf Basis eines FPGA und eines Multi-Core-Mikrocontrollers untersucht und verglichen. Die Parallelisierung ermöglicht eine Erhöhung der Regelfrequenz um 50 % auf dem FPGA und um 37,5 % auf dem Mikrocontroller.

In [22] werden General-Purpose Multi-Core-Systeme für Motion-Control-Systeme im Bereich der Steuerungstechnik betrachtet. Unter Verwendung von Virtualisierungsansätzen werden Strategien zur Systemkonsolidierung untersucht, um Hardwareaufwände und insgesamt Systemkosten zu senken. Eine Bewertung der Systemkonsolidierung zeigt die Einflüsse der Virtualisierung auf die Performance entsprechender Anwendungen. Weiterhin werden die Auswirkungen von Cross-Core-Interferenzen auf die zeitlichen Eigenschaften der konsolidierten Anwendungen betrachtet. In Abhängigkeit der Konstellation von parallel ausgeführten Anwendungen wird eine potenzielle Verlängerung der Reaktionszeit nachgewiesen, die 40-60 % von der durchschnittlichen Reaktionszeit beträgt. Weiterhin wird eine Variation der Reaktionszeit bestimmt, die regelmäßig 0,01 % und 4 % der durchschnittlichen Reaktionszeit beträgt. Das Maximum der Variation wird durch die parallele Ausführung eines Benchmarks mit sehr hohen Ansprüchen an Systemressourcen hervorgerufen und liegt bei 21,5 %. Aufgrund fehlender Möglichkeiten einer zuverlässigen Isolation von sicherheits- und zeitkritischen Funktionen wird in [22] für den Anwendungsbereich der Steuerungstechnik geraten, lediglich Anwendungen mit sehr geringen Anforderungen an das zeitliche Verhalten zu virtualisieren. Weiterhin werden zur Steigerung der Werkstückgüte und Verringerung der Bearbeitungszeiten Methoden zur Parallelisierung der Firmware entsprechender Maschinen vorgestellt. Auf Basis aufgezeichneter Laufzeitdaten wird hierbei das zu optimierende System modelliert. Mittels genetischer Algorithmen werden diese Modelle zur Exploration und Evaluation möglicher Parallelisierungen herangezogen. Eine resultierende Dekomposition und Verteilung der Firmware ermöglicht bei einem Einsatz von vier Prozessorkernen eine Verkürzung der Laufzeit der Firmware um bis zu 82 %.

2.2 Multi-Core-Systeme

Konventionelle Ansätze zur Steigerung der Rechenleistung bestehen in einer Anhebung der Anzahl ausgeführter Prozessorinstruktionen pro Takt der Central Processing Unit (CPU) [21]. Neben der Verwendung immer komplexer werdender Instruk-

tionssätze und Mikroarchitekturen zur Erhöhung der Instruction-Level-Parallelität durch parallele und tiefe Pipelines, spekulative und out-of-order Befehlsausführung sowie mehrere Ebenen von Cache-Speichern [112][149][118] stand eine stetige Erhöhung der Taktrate des Mikroprozessors im Vordergrund. Aufgrund physikalischer Grenzen [110][1] und eines signifikanten Anstiegs des Leistungsbedarfs und der Wärmeabstrahlung von Transistoren sind beide Strategien nicht beliebig fortsetzbar [160]. Durch ihren quadratischen Anteil an der Schaltleistung der Transistoren ist vor allem die Versorgungsspannung der Prozessorkerne, die zusammen mit der Taktrate erhöht werden muss, ein Treiber für den wachsenden Energiebedarf [25].

Eine Alternative zu hohen Taktraten liegt in Multi-Core-Systemen, die mehrere parallele niedriger getaktete Prozessorkerne auf einem Chip integrieren. Beispielsweise kann die Verwendung von zwei Prozessorkernen, deren Taktrate jeweils um 20 % geringer ist als die eines Hochleistungskerns, den Energiebedarf um bis zu 50 % reduzieren [136]. Die Rechenleistung beider Prozessorkerne in Summe ist gleichzeitig um 60 % höher als die des einzelnen Hochleistungskerns. Allgemein ermöglichen parallele Prozessorkerne vielseitige Möglichkeiten zum Einsparen von Energie ohne signifikante Verluste der Rechenleistung. Die voneinander unabhängigen Prozessorkerne mit gegebenenfalls unterschiedlichen Architektur- und Leistungsmerkmalen [59][167] ermöglichen eine Variabilität, die bezüglich der dynamischen Frequenz- und Spannungsskalierung [95] erweiterte Möglichkeiten zur Erhöhung der Energieeffizienz bietet [7]. Die Voraussetzung hierzu ist, dass Prozessorkerne entsprechende Möglichkeiten zur Skalierung beider Parameter bereitstellen. Zudem muss durch das Betriebssystem eine Strategie implementiert werden, die mittels Informationen über die auszuführenden Anwendungen und deren Scheduling beziehungsweise durch eine Anpassung der Scheduling-Entscheidungen die Skalierung effektiv umsetzt [181]. Insbesondere bei zeitkritischen Anwendungen sind erweiterte Algorithmen notwendig, um durch eine Skalierung der Rechenleistung das Echtzeitverhalten der Anwendungen nicht zu gefährden [11].

Multi-Core-Architekturen sind heute in nahezu allen Anwendungsgebieten angesiedelt:

- General-Purpose- und Serveranwendungen
- Hochleistungsanwendungen
- Eingebettete Systeme
  - Multimedia und mobile Anwendungen

- Digitale Signalverarbeitung
- Sicherheitskritische Anwendungen

Durch die jeweiligen Anforderungen sind eine Vielzahl unterschiedlicher Multi-Core-Architekturen entstanden, die sich neben der Art der zu realisierenden Anwendungen durch die in den Multi-Core-Systemen eingesetzten Mikroarchitekturen und Instruktionssätzen, dem Speichersystem, der Intra-Chip-Verbindung und dem Verhältnis der Rechenleistung zum Energiebedarf abgrenzen lassen [18].

2.2.1 General-Purpose Multi-Core-Prozessoren

General-Purpose (GP) Multi-Core-Prozessoren werden in gängigen PC-, Laptop- und Serversystemen eingesetzt. Die Prozessorkerne werden hier zur gleichzeitigen Ausführung mehrerer Anwendungen bis hin zur Virtualisierung vollständiger Systeme und zur parallelen Ausführung rechenintensiver Anwendungen genutzt [20][2]. Etablierte GP Multi-Core-Prozessoren setzen hierzu homogene Prozessorkerne ein, die auf traditionellen Architekturen entsprechender Single-Core-Prozessoren basieren [61]. Die typische Anzahl an Prozessorkernen liegt hier im einstelligen Bereich. Die auf Flexibilität und eine hohe durchschnittliche Rechenleistung ausgelegten Systeme setzen standardmäßig dreistufige Cache-Hierarchien und interne Verbindungssysteme mit Bandbreiten im Gigabit-Bereich ein [182].

2.2.2 Graphics Processing Units

Hochleistungsanwendungen liegen in den Bereichen der Grafik- und Videoverarbeitung, Big Data, der künstlichen Intelligenz und wissenschaftlicher Berechnungen. Die gemeinsame Eigenschaft solcher Anwendungen liegt in einer sehr hohen Datenparallelität, die von massiv parallelen Architekturen zum Erreichen eines extrem hohen Rechendurchsatzes ausgenutzt wird. Eine Graphics Processing Unit (GPU) integriert hierzu mehrere Hundert sogenannter Streaming-Prozessorkerne. Jeder dieser Kerne realisiert eine Vielzahl paralleler Hardware-Threads, sodass eine GPU insgesamt weit über eintausend homogene parallele Recheneinheiten realisieren kann [101]. Die ursprünglich für Grafik- und Videoverarbeitung entwickelten Architekturen haben sich im Laufe der Jahre hard- als auch softwareseitig hin zur Ausführung von Algorithmen entwickelt, die nicht in diesen Bereichen liegen [124]. Die GPU wird hierbei als Co-Prozessor angesehen, der den Hauptprozessor des Computersystems entlastet. Zur effektiven Verwendung von GPU werden Programmiersprachen wie

C/C++ und Python durch Methoden zur Abbildung rechenintensiver Anwendungen auf den parallelen Recheneinheiten der GPU erweitert [120][116]. Die als Kernel bezeichneten Anwendungen werden zur Ausführung auf die GPU ausgelagert und dort in Form paralleler Threads auf den Streaming-Prozessorkernen beziehungsweise den Rechenkernen ausgeführt. Die parallele Ausführung entspricht dem Prinzip der Single Instruction Multiple Data Ausführung [46] beziehungsweise der Single Instruction Multiple Threads Ausführung [121]. Hierbei führen die Recheneinheiten jeweils die gleiche Instruktion aus, wenden diese aber auf unterschiedliche Datenströme an. Folglich müssen, um die Leistungsfähigkeit der parallelen Recheneinheiten ausnutzen zu können, entsprechend viele unabhängige Datenströme vorhanden sein. Als Co-Prozessoren bekommen GPU die auszuführenden Kernel durch das Hauptsystem mittels entsprechender Hardwaretreiber zugewiesen. Dies fokussiert das parallele Ausführen eines einzelnen Kernels, der standardmäßig versucht, die GPU bis zum Beenden seiner Ausführung zu belegen. Ansätze für Multitasking erweitern Programmiermodelle [150] und Treiber [169], um auch mehrere Kernel effizient als auch unter weichen [91] sowie harten [54] Echtzeitbedingungen auszuführen.

2.2.3 Eingebettete Systeme

Im Bereich eingebetteter Systeme existieren eine Vielzahl unterschiedlicher Architekturen [90], die aus der Spezialisierung auf unterschiedliche Anwendungsdomänen entstanden sind [173]. Analog zu Multi-Core-Architekturen in den Bereichen GP- und Hochleistungsanwendungen werden in eingebetteten Systemen homogene Prozessorkerne eingesetzt, wenn eine Vielzahl unterschiedlicher Funktionen flexibel integriert oder einzelne Funktionen parallelisiert werden sollen. Entsprechende Multi-Core-Mikrocontroller, beispielsweise in Form von Multiprocessor Systems-on-Chip, können mehrere GP-Prozessorkerne, die für Anwendungen eingebetteter Systeme ausgelegt sind [68], bis hin zu einer Vielzahl spezialisierter DSP integrieren [139]. Im Sinne des Architecture/Algorithm Co-Design werden heterogene Multi-Core-Architekturen eingesetzt [31], um durch eine Abstimmung zwischen den eingesetzten Prozessorarchitekturen und den umzusetzenden Aufgaben ein optimiertes Verhältnis zwischen Rechenleistung und Energieverbrauch zu realisieren [8]. So entstehen Architekturen, die unterschiedliche Kombinationen aus GP-Mikroprozessorkernen, DSP-Kernen, GPUs und spezialisierten Hardwarebeschleunigern integrieren, um zeitkritische und rechenintensive Aufgaben auf dafür spezialisierte Prozessorkerne auszulagern [155]. Eines der größten Anwendungsgebiete für solche Systeme liegt in den Bereichen Multimedia [94] und Mobilfunk [16]. Entsprechende Architekturen

setzen regelmäßig auf ARM-Architekturen [51] basierende GP-Mikroprozessoren ein [30][37], die durch DSPs [27] und GPU [168] zur Funk-, Audio- und Videoverarbeitung unterstützt werden.

Zur digitalen Signalverarbeitung werden die heterogenen Recheneinheiten regelmäßig in Form mehrstufiger paralleler Pipelines organisiert, um einen höheren Durchsatz der typischerweise seriellen Algorithmen zu erzielen [131][170]. Neben den Architekturen der eingesetzten Recheneinheiten sind bei eingebetteten Multi-Core-Systemen die Hierarchien globaler und lokaler Speicher, deren Konsistenz- und Kohärenzmodelle sowie die Art der Intra-Chip-Verbindungen zwischen Prozessorkernen, Speichermodulen und I/O-Peripherie weitere Abgrenzungsmerkmale [18]. Vor allem Letzteres ist ein zu beachtender Faktor bezüglich der erreichbaren Bandbreite und Latenz von Anwendungen sowie des Energie- und Platzbedarfs auf dem Chip [96]. Verbreitete Architekturen sind auf Broadcast basierende Bussysteme [56], paarweise über Crossbars verbundene Module [10] sowie paketorientierte Networks-on-Chip [15].

Ein noch junges Anwendungsgebiet für eingebettete Multi-Core-Systeme ist der Bereich sicherheitskritischer Anwendungen, beispielsweise aus dem Automotive-Umfeld. Entsprechende Architekturen setzen leistungsstarke I/O-Peripherie mit einer Vielzahl flexibel konfigurierbarer PWM- und ADC-Kanäle sowie bis zu sechs überwiegend homogene Prozessorkerne zur digitalen Signalverarbeitung mit Taktraten zwischen 100 MHz und 300 MHz ein [145][48]. Die Mikroarchitektur der Prozessorkerne ist hier zur Unterstützung harter Echtzeit optimiert [74]. Die parallele Ausführung von Software steigert die Systemkomplexität und damit die Herausforderungen an die Systemintegration [130]. Letzteres umfasst steigenden Aufwand bezüglich der Zertifizierbarkeit der Systeme, der Eingrenzung der Fehlerausbreitung und der Beibehaltung des zeitlichen Determinismus zur Einhaltung harter Echtzeit [39]. Ein typischer Ansatz, diesen Aufwand zu verringern, ist die Partitionierung des Multi-Core-Systems, wodurch exklusive Zugriffe auf On-Chip-Hardwarekomponenten wie Prozessorkerne, Speicher und I/O-Peripherie garantiert werden [103]. Die Realisierung einer Partitionierung beruht auf speziellen Softwarearchitekturen und Betriebssystemen, die unter Verwendung von hardwarebasierten Sicherheitsfunktionen die Kapselung sicherheitskritischer Funktionen sowie das Erkennen und Verfolgen von Fehlern ermöglichen [138]. Hardwareseitig werden hierzu Memory Protection Units (MPU) [63], Lockstep-Architekturen, Funktionen zur Erkennung und Korrektur von Speicherfehlern und Module zur Überwachung von Systemparametern eingesetzt [108].

2.2.4 Cross-Core-Interferenzen

In einem Multi-Core-System können parallele Zugriffe auf On-Chip-Hardwaremodule wie Speicher und I/O-Peripherie stattfinden. In Abhängigkeit des parallel zugegriffenen Moduls und der Intra-Chip-Verbindung können sogenannte Cross-Core-Interferenzen entstehen, die als zusätzliche und variierende Latenzen auf die Zugriffszeit der Hardwaremodule wirken. Darüber hinaus können gemeinsam genutzte Datenstrukturen, beispielsweise innerhalb von Betriebssystemen, Latenzen durch zusätzliche Synchronisierungsmaßnahmen bedingen [49]. Neben einer reduzierten Performance bedeuten Interferenzen für Echtzeitanwendungen den Verlust der Vorhersagbarkeit und des Determinismus ihres zeitlichen Verhaltens [29]. Dementsprechend sind Multi-Core-Systeme als nicht Time-Composable zu betrachten [138], wodurch die parallele Integration von Echtzeitanwendungen besonderer Aufmerksamkeit bedarf [13]. Dies gilt vor allem dann, wenn auf Single-Core-Systemen basierende Anwendungen auf einem Multi-Core-System integriert werden. Die auf Basis des Single-Core-Systems verifizierten Anwendungseigenschaften verlieren nach der Integration ihre Gültigkeit.

Die Betrachtung von Interferenzen ist insbesondere zur Berechnung der Worst Case Execution Time (WCET) notwendig, um zuverlässige und enge zeitliche Schranken für zeit- und sicherheitskritische Multi-Core-Systeme berechnen zu können. Hier werden typischerweise statische Analysemethoden eingesetzt, um Laufzeitschranken unabhängig einer konkreten Verteilung der Anwendungen zu ermitteln. Zur Einbeziehung der Cross-Core-Interferenzen müssen unterschiedliche Ressourcen betrachtet werden, die Interferenzen hervorrufen können [171].

Bandbreitenressourcen, beispielsweise Bussysteme, serialisieren parallel Zugriffsversuche, wodurch Wartezeiten bis zu dem tatsächlichen Zugriff entstehen können. Typische Ansätze zur Berechnung der WCET basieren hier auf einer klassischen Laufzeitanalyse, die um eine Berechnung der Wartezeiten ergänzt wird. Entsprechende Ansätze unterscheiden nach zeitgesteuerten [141], ereignisgesteuerten [126][127] und hybriden [142] Arbitrierungsstrategien der Interferenzquelle. Zusätzlich werden optimierte Zugriffsstrategien betrachtet, um die Vorhersagbarkeit der Laufzeiten und der Interferenzen zu erhöhen [135]. In [92] wird ein Modell zur Berechnung einer oberen Grenze der Interferenzen hinsichtlich DRAM-Arbeitsspeicher beschrieben. Das Modell betrachtet hierzu unterschiedliche Speicherbänke, den Speicherbus sowie den Speichercontroller. In [132][125] werden zur Vermeidung von Zugriffslatenzen und deren Abhängigkeit von der Reihenfolge, in der Anwendungen auf den Kernen ausgeführt werden, Optimierungen bezüglich des Speichercontrollers betrach-

tet. Diese beziehen sich unter anderem auf vorhersagbare Arbitrierungsstrategien und angepasste DRAM-spezifische Zugriffsmuster auf die Speichermodule [117]. Für Echtzeitanwendungen beschreibt [32] ein Framework zur Berechnung der WCET, das zur Betrachtung von Interferenzen unterschiedliche Arbitrierungsstrategien bei Zugriffen auf geteilten Speicher einbeziehen kann.

Speicherressourcen beschreiben gemeinsame Arbeits- und Cachespeicher. Neben Interferenzen durch die Serialisierung der Speicherzugriffe entstehen hier Konflikte bezüglich der Speicherinhalte. Diese treten beispielsweise dann auf, wenn ein Kern Speicherblöcke innerhalb eines geteilten Cache verdrängt, die von einem anderen Kern genutzt werden. Als Resultat sind zeitintensive Zugriffe auf Speicher einer höheren Hierarchieebene notwendig, um die entsprechenden Inhalte neu zu laden. Typische Strategien zur Begegnung dieser Problematik ist die Partitionierung geteilter Speicher, die den Prozessorkernen exklusiven Zugriff auf bestimmte Speicherbereiche ermöglicht. Ansätze zur Partitionierung werden nach Cache-Set [178][154] und Cache-Way [119][128] basierten Ansätzen unterschieden. Hierbei ist die Partitionsgröße ein maßgeblicher Faktor, um die vergleichsweise kleinen Cache-Speicher weiterhin effektiv für eine hohe Systemperformance zu nutzen. Zur Bestimmung der Partitionsgröße werden Performance-Metriken wie das Cache-Miss-Verhältnis [178], Blockierzeiten bei Cache-Zugriffen [156] oder die benötigten Taktzyklen pro Instruktion [128] herangezogen. Zur weiteren Optimierung der Performance und Verringerung von Speicherinterferenzen ziehen angepasste Scheduling-Strategien Nutzungs- und Konkurrenzparameter geteilter Caches [179] als auch Partitionsgrößen [57] in ihre Scheduling-Entscheidungen ein.

Neben statischen Analysemethoden können die Einflüsse paralleler Zugriffe auf geteilte Ressourcen mittels Performance-Messungen bestimmt werden. Konkurrenz wird dabei durch das gezielte parallele Ausführen sogenannter Resource-Stressing Benchmarks hervorgerufen. Dies sind synthetische Anwendungen, die gezielt Zugriffe auf geteilte Ressourcen verursachen. Die Bestimmung der Interferenzen beziehungsweise die damit verbunden Seiteneffekte auf die Laufzeit und Performance werden typischerweise durch entsprechende Performance-Counter der eingesetzten Hardware ermittelt [146]. In [129] generieren die Benchmarks Zugriffe auf geteilte Funktionseinheiten des Prozessors und die Speicherhierarchie, insbesondere Caches. Maximale Interferenzen werden durch das parallele Ausführen mehrerer Resource-Stressing Benchmarks hervorgerufen. In [122][45] werden unterschiedliche Typen von Arbeitsspeicher und Cache-Kohärenzprotokolle zur Charakterisierung der Interferenzen unterschiedlicher Multi-Core-Prozessoren betrachtet. Es zeigt sich, dass

Kohärenzprotokolle insbesondere bei schreibenden Zugriffen einen signifikanten Anstieg an Interferenzen und damit zusätzliche Laufzeiten verursachen. Durch Hintergrundaktionen zur Beibehaltung der Kohärenz treten selbst dann Interferenzen auf, wenn keine expliziten Speicherzugriffe durch eine Anwendung durchgeführt werden. In [44] wird ein Vorgehen gezeigt, um die maximalen Blockierzeiten bei Zugriffen auf einen per Round-Robin-Algorithmus arbitrierten Bus zu ermitteln und diese zur Berechnung der resultierenden Laufzeiten einzusetzen. Hierbei werden keine spezifischen zeitlichen Eigenschaften der Bus-Architektur vorausgesetzt. Zusätzlich zu hardwarebasierten Performance-Countern werden in [146] Datenstrukturen aus dem Betriebssystemkernel mit Informationen über die parallel ausgeführten Prozesse und deren Cache-Nutzung verwendet. Aus diesen Informationen werden Strategien zur Verteilung abgeleitet, um Cache-Interferenzen zu reduzieren. In [109] wird ein direkter Vergleich zwischen der Laufzeit in Isolation und unter Interferenzen durchgeführt. Hierzu wird ein Framework vorgestellt, das Resource-Stressing Benchmarks dynamisch zur Laufzeit einer zu testenden Anwendung aktiviert und kontinuierlich Performance-Daten sammelt. Hierüber wird ein Profil erstellt, das die Sensitivität der Anwendung gegenüber Interferenzen beschreibt. Dies ermöglicht eine Anpassung von Scheduling-Entscheidungen, sodass die parallele Ausführung von interferenzsensitivem Code reduziert wird.

2.3 Zusammenfassung und Ableitung offener Forschungsfragen

Die vorliegende Arbeit untersucht den Einsatz von Multi-Core-Mikrocontrollern als Mikrocontrollerarchitektur für Motor-Controller mit Fokus auf Systeme zur geberlosen Regelung von PMSM. Motor-Controller stellen hierbei das Anwendungsgebiet dar, das durch den beschriebenen Stand der Technik abgegrenzt wird. Dies umfasst die hier typischerweise verwendeten Algorithmen, deren Anforderungen sowie den standardmäßigen Ansätzen zu deren Implementierung. Ergänzt durch die Betrachtung von Multi-Core-Systemen lassen sich potenzielle Möglichkeiten abgrenzen, um Multi-Core-Mikrocontroller als Recheneinheit von Motor-Controllern einzusetzen. Diese Möglichkeiten liegen in der Parallelisierung von Motor-Controllern und der Konsolidierung mehrerer Motor-Controller auf einem Multi-Core-System.

Aus diesen Einsatzszenarien lassen sich folgende Forschungsfragen ableiten, die im Rahmen der vorliegenden Arbeit behandelt werden:

- Kann die geberlose Regelung von PMSM ausreichend parallelisiert werden, um ihre Ausführungszeiten effektiv zu verkürzen?
- Gefährden Cross-Core-Interferenzen eine Integration mehrerer Motor-Controller in ein Multi-Core-System?

2.3.1 Untersuchung der Parallelisierbarkeit von geberlosen Stromregelungen

Die Ausführung einer geberlosen Regelung von PMSM muss unter Einhaltung von Echtzeitbedingungen erfolgen. Diese Bedingungen definieren ein Zeitfenster, innerhalb dessen eine Iteration der Regelung ausgeführt werden muss. Das Nichteinhalten der durch das Fenster definierten zeitlichen Schranken kann die Qualität der Regelung beeinflussen und den zuverlässigen Betrieb des Antriebs verhindern. Die maximale Größe des Zeitfensters richtet sich nach der Periodendauer der dreiphasigen PWM, die zur Ansteuerung des Motors genutzt wird und gleichzeitig den Takt der Stromregelung vorgibt. Insbesondere bei Kleinantrieben werden vergleichsweise hohe PWM-Frequenzen eingesetzt, die regelmäßig in einem Intervall von 10 bis 100 kHz liegen, wodurch sich für die Zeitfenster ein Bereich von 100 und 10 μs ergibt. Die Realisierung unterschiedlicher Qualitätsfaktoren fordern zunehmend PWM-Frequenzen in der oberen Hälfte des Intervalls mit entsprechend kleinen Zeitfenstern.

Der ausschlaggebende Faktor, ob auch solch kleine Zeitfenster eingehalten werden können, ist die Rechenleistung des im Motor-Controller eingesetzten Mikrocontrollers. Die beste Performance bietet eine Implementierung in Hardware. Algorithmen werden hierbei in Form digitaler Schaltungen realisiert, die optimal auf den Algorithmus zugeschnitten werden können. Diese Leistungsfähigkeit wird jedoch durch hohe Aufwände seitens der Implementierung und einer vergleichsweise geringen Flexibilität bezüglich der Erweiterbarkeit und Wartbarkeit der Systeme erkauft. Software-basierte Implementierungen ermöglichen durch den Einsatz flexibler Hochsprachen eine signifikante Reduktion der Aufwände und bieten durch Software-Updates komfortable Wartungs- und Erweiterungsmöglichkeiten. Diese Vorteile werden jedoch durch höhere Laufzeiten erkauft, die prinzipiell einen geringeren Spielraum zur Umsetzung von Qualitätsanforderungen zulassen. Für viele Anwendungen, vor allem aus dem Bereich Forschung und Entwicklung, überwiegen die Vorteile der software-basierten Implementierung gegenüber der möglichen Leistungsfähigkeit einer hard-

wareseitigen Realisierung. Es müssen jedoch regelmäßig Mikrocontroller aus dem oberen Leistungsbereich der für dieses Anwendungsgebiet geeigneten Systeme eingesetzt werden, um eine ausreichende Rechenleistung zur Verfügung zu haben. Diese setzen typischerweise DSPs mit Prozessortaktraten zwischen 100 und 200 MHz ein, die eine Umsetzung von geberlosen Regelungen mit PWM-Frequenzen im Bereich von 10 bis 20 kHz ermöglichen. Sollen höhere PWM-Frequenzen realisiert werden, müssen Motor-Controller mehr Rechenleistung bereitstellen.

Eine merkliche Erhöhung der Rechenleistung durch eine Steigerung der Taktraten der Mikrocontroller ist hier jedoch nicht zu erwarten, da dies gleichzeitig einen signifikanten Anstieg des Energiebedarfs und der Wärmeentwicklung bedeutet. Eine effiziente Alternative ist das Verwenden mehrerer in den Mikrocontroller integrierter Prozessorkerne. Um von diesen zu profitieren, muss eine Anwendung in ausreichendem Maße parallelisierbar sein. Dies geschieht durch die Nutzung voneinander unabhängiger Datenströme, die parallel bearbeitet werden können. Bei einer dreiphasigen PWM entstehen durch Strom- und Spannungsmessungen jeweils drei Datenkanäle. Erfasste Daten werden überwiegend in aufeinander aufbauenden Prozessschritten weiterverarbeitet. Daher stellt sich die Frage, ob geberlose Regelungen von PMSM prinzipiell ausreichend parallelisiert werden können, um ihre Ausführungszeiten zu verkürzen. Ist dies der Fall, stellen Multi-Core-Mikrocontroller eine neue Möglichkeit dar, um die Leistungsfähigkeit und Effizienz von softwarebasierten Motor-Controllern zur Regelung von PMSM zu erhöhen.

2.3.2 Untersuchung von Cross-Core-Interferenzen bei konsolidierten Motor-Controllern

Ein weiteres Szenario für einen potenziellen Einsatz von Multi-Core-Mikrocontrollern in Motor-Controllern leitet sich aus den Anwendungsbereichen Server- und General-Purpose-Anwendungen ab. Hier werden parallele Prozessorkerne genutzt, um separate Teilsysteme zu bilden. Jedes Teilsystem wird verwendet, um die Anwendungen eines dedizierten Single-Core-Systems zu integrieren. Auf das Anwendungsgebiet übertragen ergibt dieses Vorgehen die Möglichkeit, mehrere separate Single-Core Motor-Controller als jeweils ein Teilsystem auf einem gemeinsamen Multi-Core-Mikrocontroller zu integrieren. Als Resultat kann die Hardware mehrerer Motor-Controller eingespart und dadurch der Energie- sowie Platzbedarf reduziert werden.

Mit Multi-Core-Mikrocontrollern aus dem Automotive-Bereich stehen Systeme zur Verfügung, die ausreichend I/O-Peripherie zur Integration mehrere Motor-Con-

troller bieten. Zudem integrieren diese Mikrocontroller Safety-Mechanismen, um Teilsysteme zu einem gewissen Maße voneinander zu isolieren. Den Grad der Isolation separater Systeme können diese Mechanismen zwar nicht vollständig wiederherstellen, jedoch kann die Durchsetzung von Zugriffsrechten mittels MPUs eine Fehlerausbreitung zwischen Teilsystemen verhindern. Dies schützt jedoch nicht vor negativen Seiteneffekten, welche die Laufzeit der Teilsysteme beeinflussen. Dies sind sogenannte Cross-Core-Interferenzen, die durch zeitgleiche Zugriffe auf geteilte Hardwaremodule des Multi-Core-Mikrocontrollers entstehen. Das Resultat sind zusätzliche und variierende Latenzen, welche die Performance sowie das zeitliche Verhalten der Teilsysteme beeinflussen. Um die Einflüsse von Interferenzen zu berücksichtigen, werden Methoden zur WCET-Analyse als auch Performance-Messungen durch Modelle von potenziell konkurrierenden Zugriffen auf Bus-Systeme, geteilte Arbeitsspeicher und Cache-Speicher erweitert. Darauf aufbauend werden Scheduling-Verfahren und Ressourcenzugriffe hinsichtlich der Reduktion von Interferenzen sowie ihrer besseren Vorhersagbarkeit optimiert.

Für die Integration mehrerer Motor-Controller in ein Multi-Core-System bedeuten Interferenzen, dass sie die Leistungsfähigkeit der Controller bezüglich realisierbarer PWM-Frequenzen mindern können. Darüber hinaus gefährden sie das zuverlässige Einhalten der zur Verfügung stehenden Zeitfenster, sodass ein zuverlässiger Betrieb der Antriebe gefährdet ist. In Anbetracht beider Faktoren eröffnet sich die Frage, ob Interferenzen eine Integration mehrerer Motor-Controller in ein Multi-Core-System gefährden.

Kapitel 3

Zielsetzung und Aufbau der vorliegenden Arbeit

Elektrische Antriebe unterliegen einer stetigen Weiterentwicklung und Optimierung. Hierbei sind bürstenlose Antriebe hervorzuheben, die durch den Verzicht auf Schleifkontakte mehrere Vorteile gegenüber bürstenbehafteten Antrieben bieten: einen besseren Wirkungsgrad, eine erhöhte Lebensdauer, eine verminderte Abwärme und Geräuschbelastung sowie höhere Drehzahlen. Insbesondere sinuskommutierte PMSM lösen in nahezu allen Anwendungsbereichen, unter anderem in der Automobiltechnik, der industriellen Automatisierung und Messtechnik, zunehmend bürstenbehaftete Gleichstromantriebe ab. Gegenüber blockkommutierten BLDC-Antrieben erlauben PMSM selbst bei geringen Drehzahlen eine präzise Drehzahl- und Drehmomentregelung sowie einen flachen Drehmomentverlauf.

Zur Optimierung der Antriebssysteme werden zunehmend Verfahren eingesetzt, die Rotorlageinformationen, zur Realisierung der bei PMSM standardmäßig eingesetzten feldorientierten Stromregelung, ohne den Einsatz diskreter Winkelsensoren gewinnen. Durch das Einsparen solcher Sensorik und entsprechender Bauteile am Antrieb können die mechanische Stabilität sowie die Rotordynamik erhöht und gleichzeitig der Platzbedarf und die Kosten gesenkt werden. Insbesondere Kleinantriebe profitieren von den Vorteilen geberlos angesteuerter PMSM. Gleichzeitig gewinnen Kleinantriebe im Zuge der voranschreitenden Digitalisierung, beispielsweise in Form sogenannter Smart Drives, zunehmend an Relevanz, welche die Weiterentwicklung und Optimierung der genannten Technologien antreiben.

Der Weg vom bürstenbehafteten Antrieb zum geberlosen Motor umfasst die Weiterentwicklung und Optimierung der Systeme, beginnend bei konstruktiven Ansätzen bis hin zu komplexen Algorithmen zur Signalanalyse und Signalerzeugung. Eine

bisher nur wenig beachtete Komponente der Antriebssysteme sind die in Motor-Controllern eingesetzten Mikrocontroller. Insbesondere solche, die Algorithmen per Software realisieren. Die hier standardmäßig eingesetzten Single-Core-Mikrocontroller stoßen derzeit an ihre Leistungsgrenzen. Das Überschreiten dieser Grenzen durch eine Erhöhung der Rechenleistung führt jedoch auch zu einem signifikanten Anstieg des Energiebedarfs. Eine Alternative kann der Einsatz von Multi-Core-Mikrocontrollern in Motor-Controllern bieten. Mehrere parallele Prozessorkerne können die Rechenleistung steigern, ohne einen signifikanten Anstieg des Energiebedarfs zu erhöhen. Gleichermaßen können sie den Energiebedarf im Vergleich zu ähnlich leistungsfähigen Single-Core-Mikrocontrollern senken. Diese Eigenschaften sind insbesondere für Anwendungen mit Kleinantrieben im niedrigen Leistungsbereich interessant. Die oftmals über Batterien versorgten Systeme bedingen aufgrund kleiner Zeitkonstanten vergleichsweise hohe Regelfrequenzen, deren Realisierung eine hohe Rechenleistung veranschlagt. Gleichzeitig verursachen hier Motor-Controller beziehungsweise die darin eingesetzten Mikrocontroller einen relativ großen Anteil des Energiebedarfes des Gesamtsystems. Vor diesem Hintergrund bieten Multi-Core-Mikrocontroller eine vielversprechende Möglichkeit zur Optimierung der für Kleinantriebe eingesetzten Motor-Controller.

3.1 Zielsetzung

Die vorliegende Arbeit konzentriert sich auf Motor-Controller zur Regelung von PMSM, deren zentrale Recheneinheit durch einen Multi-Core-Mikrocontroller mit homogenen, per Software programmierbaren Prozessorkernen realisiert wird. Der Fokus liegt auf Motor-Controllern in Form eingebetteter Systeme, die als Teil des elektrischen Antriebes oder sehr nahe bei diesem integriert sind. Zentrale Systeme, beispielsweise in Form von PC- oder PLC-Systemen zur Antriebsregelung werden nicht betrachtet.

Die in Kapitel 2 vorgenommene Abgrenzung des betrachteten Anwendungsgebietes in Kombination mit der Betrachtung von Multi-Core-Systemen beschreibt die Möglichkeiten, Multi-Core-Mikrocontroller als zentrale Recheneinheit von Motor-Controllern einzusetzen. Konkret ergeben sich Chancen sowohl in Form einer Performance-Steigerung der Motor-Controller durch eine Parallelisierung von Algorithmen als auch im Einsparen von Hardware- und Energieaufwänden durch eine Konsolidierung von Motor-Controllern. Beide Punkte werden im Rahmen dieser Arbeit betrachtet, um den Einsatz von Multi-Core-Mikrocontrollern in Motor-Con-

trollern zu evaluieren. Die Abgrenzung führt zur Formulierung der für diese Arbeit zentralen Forschungsfragen.

1. Kann die geberlose Regelung von PMSM ausreichend parallelisiert werden, um ihre Ausführungszeiten effektiv zu verkürzen?

Effektiv bedeutet hier, dass durch Parallelisierung die Laufzeit dermaßen verkürzt werden kann, damit höhere PWM- und Regelfrequenzen erzielt werden können. Als übergeordnetes Ziel zur Beantwortung der Forschungsfrage sollen Methoden entstehen, mit deren Hilfe sich die Effektivität einer Parallelisierung abschätzen und gegebenenfalls realisieren lässt. Hierzu sind folgende Arbeitsziele definiert:

- Abstraktion geberloser Stromregelungen in Form eines Taskmodells.
- Analyse und Evaluation der Parallelisierbarkeit geberloser Stromregelungen.

2. Gefährden Cross-Core-Interferenzen eine Integration mehrerer Motor-Controller in ein Multi-Core-System?

Im Rahmen dieser Fragestellung wird untersucht, ob Interferenzen die Laufzeiten von Motor-Controllern in einem Maße erhöhen, das zu einer Überschreitung der von der Stromregelung definierten Zeitfenster führen kann. Hierbei soll eine Methode entstehen, mit deren Hilfe Laufzeiterhöhungen infolge von Cross-Core-Interferenzen für eine Multi-Core-Integration mehrere Motor-Controller bestimmt werden können. Eine solche Bestimmung ist idealerweise ohne eine reale Ausführung der Motor-Controller auf dem Multi-Core-System möglich, um die Systemsicherheit nicht zu gefährden. Hierzu sind folgende Arbeitsziele definiert:

- Modellierung konkurrierender Zugriffe auf gemeinsame Hardwaremodule.
- Analyse und Quantifizierung von Laufzeiterhöhungen aufgrund konkurrierender Zugriffe.
- Methoden zur Reduktion von Cross-Core-Interferenzen.

3. Welchen Einfluss hat der Einsatz mehrerer Prozessorkerne auf den Energiebedarf von Motor-Controllern?

Die Betrachtung des Energiebedarfs eines Multi-Core Motor-Controllers vervollständigt die beschriebenen Fragestellungen und Ziele. Es wird untersucht, welchen Einfluss der Einsatz zusätzlicher Prozessorkerne im Zuge einer Parallelisierung auf den

Energiebedarf des Motor-Controllers hat. Entsprechendes gilt für die potenzielle Einsparung für den Anwendungsfall der Konsolidierung. Es sind folgende Arbeitsziele definiert:

- Aufbau eines Modells zur feingliedrigen Bestimmung der Leistungsaufnahme des Multi-Core-Systems.
- Experimentelle Bestimmung und Bewertung der Leistungsaufnahme für die Parallelisierung und Konsolidierung.

3.2 Betrachtete Algorithmen und Technologien

Zur Erarbeitung der in den Arbeitszielen beschriebenen Modelle und zur Durchführung der experimentellen Bewertungen konzentriert sich diese Arbeit auf ausgewählte Algorithmen zur geberlosen Regelung von PMSM und verwendet eine Multi-Core-Plattform bestehend aus einem Multi-Core-Mikrocontroller und einem Echtzeitbetriebssystem.

3.2.1 Algorithmen zur geberlosen Stromregelung

Als Stromregelung wird eine standardmäßige FOS nach dem in Abbildung 2.2 beschriebenen Modell eingesetzt. Dieses wird mittels unterschiedlicher Algorithmen zur geberlosen Bestimmung der Rotorlage erweitert. Konkret werden ein Sliding-Mode Observer und ein Algorithmus basierend auf High Frequency Current Injection nach den Modellen aus [43] sowie das Verfahren Direct Flux Control nach [159] eingesetzt. Unterschiedliche Faktoren haben zur Auswahl dieser Algorithmen geführt. Dies sind Anforderungen, welche die Algorithmen an ihre softwareseitige Implementierung stellen. Durch SMO werden Verfahren repräsentiert, die eine kontinuierliche Signalauswertung durchführen müssen, um die Rotorposition präzise ermitteln zu können. Den anderen Verfahren genügt eine diskrete Signalabtastung. Weiterhin bedingen sie unterschiedliche Datenströme, welche die Basis für eine Parallelisierung bilden. Aus Sicht des Motor-Controllers beziehungsweise der Antriebsregelung decken die Algorithmen insgesamt einen breiten Drehzahlbereich ab, in dem die Rotorlage zuverlässig bestimmt werden kann. Mit den auf magnetischer Anisotropie basierenden Verfahren HFCI und DFC ist dies im Stillstand bis nahezu über den gesamten Drehzahlbereich (DFC) möglich. Mittels der Auswertung der Gegen-EMK ist SMO insbesondere bei hohen Drehzahlen effektiv. Die ausgewählten Algorithmen

bieten somit ausreichend Diversität zur Bearbeitung der genannten Forschungsfragen. Zudem haben sie durch ihre Eigenschaften ausreichend Relevanz für einen produktiven Einsatz in Motor-Controller, wodurch die Verwertung der Forschungsergebnisse unterstützt wird.

3.2.2 Multi-Core-Plattform

Für die Durchführung der geplanten Arbeiten wird eine Plattform bestehend aus aufeinander abgestimmten Hardware- und Softwarekomponenten verwendet. Infineon Tricore™ Prozessorkerne [74] sind für die Steuerung und Regelung mechatronischer Systeme ausgelegt und liegen mit Taktraten zwischen 100 und 200 MHz in einem für Motor-Controller typischen Leistungsbereich. Infineon Aurix™ Mikrocontroller stehen mit einer unterschiedlichen Anzahl solcher Prozessorkerne zur Verfügung. Dies ermöglicht hinsichtlich der Energiebetrachtung den Einfluss zusätzlicher Prozessorkerne zu untersuchen. Die Integration mehrkanaliger und flexibel konfigurierbarer Timer-Module [133][78] ermöglicht im Rahmen der Konsolidierung die Erzeugung der notwendigen PWM-Signale, um mehrere PMSM gleichzeitig ansteuern zu können. Eine Besonderheit dieser Module ist deren Fähigkeit, PWM-Kanäle in Gruppen zu gliedern, die unabhängig voneinander konfiguriert, betrieben und mit ADC-Kanälen synchronisiert werden können. Bezüglich der Signalabtastung und Signalerzeugung entstehen somit keine Abhängigkeiten zwischen den integrierten Motor-Controllern, welche deren Funktionsweise sowie die durchzuführenden Untersuchungen beeinflussen könnten. Zusätzlich ermöglicht eine MPU, die von den Motor-Controllern verwendeten Adressräume sicher voneinander zu kapseln, um eine gegenseitige Beeinflussung im Sinne fehlerhafter Zugriffe auf Speicher und I/O-Module zu verhindern. Diese Safety-Funktionen sind zur Durchführung dieser Arbeit nicht zwingend erforderlich. Sie werden jedoch als notwendig erachtet, um eine Konsolidierung von Motor-Controllern im Rahmen einer Verwertung der Forschungsergebnisse praxistauglich durchführen zu können. Deshalb wurden sie bei dem Aufbau der Plattform berücksichtigt.

Als nicht-technische Eigenschaften sprechen eine direkte Verfügbarkeit der Infineon Aurix™ Mikrocontroller in Form von Evaluationskits für deren Verwendung in dieser Arbeit. Die Evaluationskits bieten ausreichende Möglichkeiten, um Signalleitungen sowie die Spannungs- und Stromsensoren mehrerer Motoren mit den entsprechenden I/O-Modulen des Mikrocontrollers zu verbinden. Sehr detaillierte Dokumentationen der Tricore™ Prozessorkerne, der internen Bussysteme und der I/O-Module unterstützen eine präzise Modellbildung für die analytische Betrachtung von

Interferenzen.

Softwareseitig wird die für Forschungseinrichtungen frei verfügbare PXROS-HR Tricore Development Plattform [67][66] eingesetzt, die sowohl auf Single- als auch Multi-Core-Mikrocontroller der Infineon Aurix^{TM} Familie mit Tricore^{TM} Prozessorkernen spezialisiert ist. Neben einem Tricore^{TM} GCC-Compiler stellt die Plattform das Safety-Echtzeitbetriebssystem PXROS-HR zur Verfügung. PXROS-HR wird seit weit über einem Jahrzehnt für die Implementierung von Motor-Controllern basierend auf Tricore^{TM} Single-Core-Mikrocontrollern eingesetzt. Zudem ist eines der ersten zertifizierten Betriebssysteme für Infineon Aurix^{TM} Multi-Core-Mikrocontroller. Für die Implementierung und Integration parallelisierter Algorithmen sowie für eine sichere Integration mehrerer vollständiger Motor-Controller-Anwendungen stellt es unterschiedliche Task- und Interrupt-Modelle zur Verfügung. Diese erlauben wahlweise schnelle Kontextwechsel, eine flexible Verteilung und Synchronisation von Funktionen sowie deren sichere Kapselung. Durch seine Zertifizierung nach den Safety-Standards SIL3 (IEC61508) und ASIL D (ISO 26262) wird zudem eine praxistaugliche Verwertbarkeit der in dieser Arbeit entstehenden Software zum Aufbau von Multi-Core Motor-Controllern (MCMC) ermöglicht.

3.3 Abgrenzung und Beitrag zum Stand der Technik

MCMC erweitern die standardmäßigen Implementierungsmethoden von Motor-Controllern. Wie auch hybride Ansätze stellen Multi-Core-Mikrocontroller Mehrprozessorsysteme dar. Es bestehen jedoch signifikante Unterschiede zwischen beiden Ansätzen. Hybride Ansätze werden hinsichtlich konkreter Anwendungen entwickelt und entsprechend optimiert. Sie stellen heterogene Systeme dar, deren Prozessorkerne für die Ausführung bestimmter Algorithmen optimiert sind. Die betrachteten Multi-Core-Mikrocontroller stellen homogene Prozessorkerne bereit und sind im Sinne der softwarebasierten Implementierung für die Ausführung einer Vielzahl von Algorithmen geeignet. Für ihren effektiven Einsatz in der Antriebstechnik eröffnen sie bezüglich der Implementierung und Integration von Anwendungen neue Problemstellungen. Daher müssen sie als Basis für neue Implementierungsmethoden betrachtet werden.

Die Ergebnisse dieser Arbeit tragen dazu bei, Multi-Core-Mikrocontroller als Basis neuer Implementierungsmethoden für die Realisierung von Motor-Controllern zur geberlosen Regelung von PMSM zu erschließen und zu etablieren. Hierzu wer-

den die Design- und Implementierungsprozesse von Motor-Controllern erweitert. Im Rahmen der Parallelisierung geschieht dies durch die zu erarbeitenden Methoden zur

- Beurteilung der Effektivität der Parallelisierung eines softwarebasierten Motor-Controllers und zur
- Realisierung der Parallelisierung von Motor-Controllern auf einem Multi-Core-Mikrocontroller.

Durch die Betrachtung von Cross-Core-Interferenzen wird der Implementierungsprozess durch Methoden erweitert, welche die Integration mehrerer Motor-Controller auf einem Multi-Core-Mikrocontroller ermöglichen und die Systeme optimieren. Konkret geschieht dies durch die zu erarbeitenden Methoden zur

- Analyse von Cross-Core-Interferenzen und deren Auswirkungen auf die integrierten Motor-Controller und zur
- Reduktion von Cross-Core-Interferenzen.

3.4 Aufbau dieser Arbeit

Die folgenden Kapitel dieser Arbeit sind wie folgt gegliedert:

Kapitel 4 ergänzt die beschriebene Abgrenzung der Implementierungsmethoden von Motor-Controllern um Multi-Core Motor-Controller. Ergänzend wird die für MCMC verwendete Multi-Core-Plattform abgegrenzt und beschrieben.

Kapitel 5 umfasst das Vorgehen zur Messung der Leistungsaufnahme und eine Evaluation der Leistungsaufnahme des MCMC.

Kapitel 6 untersucht die Parallelisierung von Motor-Controllern bezüglich der geberlosen feldorientierten Stromregelung. Die Möglichkeiten der Parallelisierung werden erörtert und experimentell für den MCMC evaluiert.

Kapitel 7 beschreibt die durch eine Konsolidierung von Motor-Controllern entstehenden Cross-Core-Interferenzen. Die Abläufe konkurrierender Modulzugriffe werden beschrieben und deren Auftreten eingegrenzt. Weiterhin werden die durch solche Zugriffe verursachte Laufzeiterhöhung und deren Einflussfaktoren beschrieben und experimentell bestimmt.

Kapitel 8 zeigt ein Verfahren zur Berechnung der Laufzeiterhöhung bei konsolidierten Motor-Controllern. Basierend auf der Ausführung der Motor-Controller auf

Single-Core-Systemen werden die nach einer angenommenen Konsolidierung entstehenden parallelen Modulzugriffe abgeleitet. Anschließend werden deren Einflüsse auf die Laufzeiten der Motor-Controller berechnet.

In Kapitel 9 werden die Ergebnisse dieser Arbeit und deren Übertragbarkeit diskutiert. Kapitel 10 gibt ein abschließendes Fazit und einen Ausblick auf Anschlussarbeiten.

Kapitel 4

Multi-Core Motor-Controller

Für den Aufbau eines MCMC wird eine Multi-Core-Plattform bestehend aus einem per Software programmierbaren Multi-Core-Mikrocontroller und einem Echtzeitbetriebssystem betrachtet. Der Mikrocontroller erweitert die typischen Single-Core Motor-Controller um zusätzliche Prozessorkerne. Hierdurch wird die in Abschnitt 2.1.1.4 beschriebene Abgrenzung der Implementierungsmethoden von Motor-Controllern ergänzt. Abbildung 4.1 beschreibt diesbezüglich die Erweiterung des durch Mikrocontroller abgedeckten Anwendungsbereichs in horizontaler Richtung. Aufgrund der Programmierbarkeit des MCMC per Software bleibt der maximal abgedeckte Komplexitätsbereich unverändert. Mehrere parallele Kerne vergrößern die Fähigkeit des Mikrocontrollers, Datenabhängigkeiten durch Parallelisierung auszunutzen. Die ungleichmäßige Ausbreitung pro Prozessorkern in horizontaler Richtung wird durch die Kosten der Parallelisierung bedingt.

4.1 Multi-Core-Architekturen für eingebettete Motor-Controller

Der Fokus auf eingebettete Systeme schließt GP Multi-Core-Prozessoren und GPU aus den infrage kommenden Architekturen aus. Neben dem Formfaktor und der benötigten externen Elektronik zum Betrieb solcher Systeme sind fehlende On-Chip-Peripherie zur Signalverarbeitung sowie der vergleichsweise sehr hohe Energieverbrauch weitere Ausschlusskriterien. Zusätzlich zeigen diese Systeme unzureichende Fähigkeiten zur zuverlässigen Einhaltung zeitlicher Anforderungen. GP-Prozessoren beziehungsweise ihre Mikroarchitekturen sind auf eine hohe durchschnittliche Rechenleistung ausgelegt. Architekturmerkmale zur Steigerung der Performance durch Instruction-Level-Parallelität, mehrstufige Speicherhierarchien und Memory Mana-

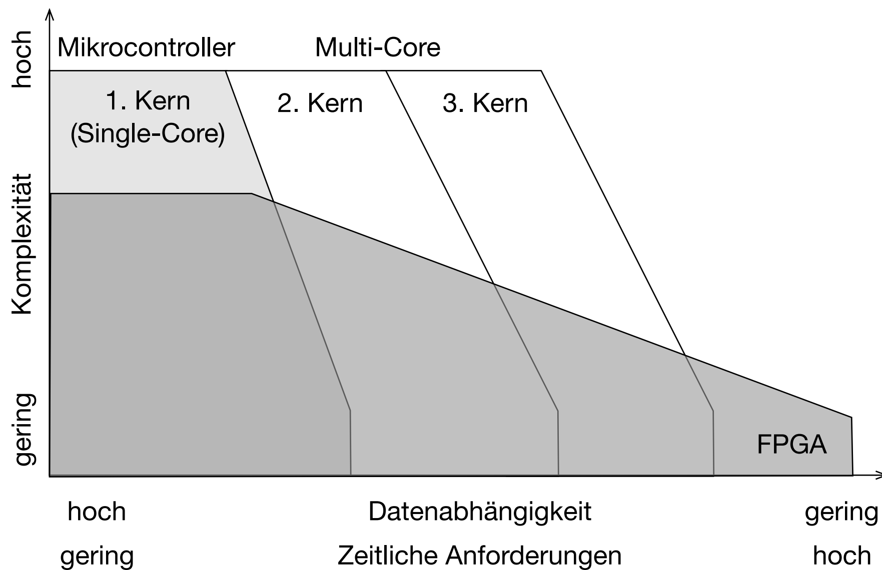


Abbildung 4.1: Erweiterung der hard- und softwarebasierten Integrationsmethoden durch Multi-Core-Systeme

gement Units verursachen variierende Laufzeiten, die nur schwer bis unmöglich vorhersagbar sind [172]. Insbesondere bei der Ausführung von Interrupts können aufgrund der komplexen Architekturen Latenzen entstehen. Diese liegen regelmäßig in einem Bereich von 2-5 μs und können um bis zu 20 % vom durchschnittlichen Latenzwert abweichen [22][84]. Ähnliches gilt für den Einsatz von GPU. Würde die Problematik des Echtzeit-Scheduling der GPU-Kernel außer Acht gelassen werden, so verbleiben für die minimale Antwortzeit eines Kernels Latenzen in einer Größenordnung von durchschnittlich 20 μs [69]. Maßgeblich ist hier der Datentransfer zwischen Hauptspeicher und GPU [50]. Mit Latenzen und Jittern in den beschriebenen Größenordnungen sind die genannten Architekturen für die betrachteten Regelanwendungen nicht geeignet. Hier werden durch die PWM-Frequenz der Stromregelung Zeitfenster zwischen 10 und 100 μs definiert, innerhalb derer sämtliche Berechnungen der Regelung durchgeführt werden müssen. Somit können bereits die bei GP Multi-Core-Prozessoren und GPU auftretenden Latenzen zu einer Überschreitung dieser Zeitfenster führen. Selbst unter der Annahme, dass die Regelung bei ausreichend kleinen Latenzen innerhalb der Zeitfenster ausgeführt werden könnte, können Jitter zu einer Überschreitung der zeitlichen Grenzen führen. Um die Zeitfenster ideal ausnutzen zu können, ist eine exakte Synchronisation der Softwareausführung mit jeder Periode der PWM notwendig. Jitter können diese Synchronisation verhindern, sodass die Regelung verspätet ausgeführt wird und dadurch ihr Zeitfenster überschreitet.

Die in der Antriebstechnik fokussierten Algorithmen setzen sich aus vergleichswei-

se einfachen Berechnungen auf kleinen Datenmengen zusammen. Parallel ausführbare Berechnungen, die aus einer Parallelisierung oder Konsolidierung entstehen, sind bezüglich der Komplexität ihrer Instruktionen als gleichartig zu betrachten. Dementsprechend erbringen hier heterogene Multi-Core-Architekturen mit spezialisierten Recheneinheiten keinen nennenswerten Vorteil. Als Basis zur Wahl einer Multi-Core-Architektur werden somit Architekturen betrachtet, die auf den im Anwendungsgebiet regelmäßig verwendeten Mikrocontrollern, DSP oder System-on-Chips basieren, jedoch mehrere dieser Recheneinheiten in Form eines homogenen Multi-Core-Systems integrieren.

Die Konsolidierung mehrerer Motor-Controller stellt jedoch eine neue Anforderung dar, die bei Controllern im Single-Core-Bereich bisher nicht aufgetreten ist. Durch die Verwendung separater eingebetteter Systeme zur Regelung von jeweils einem Motor sind Motor-Controller inhärent voneinander isoliert. Auf Hardwareebene arbeiten die Systeme unabhängig voneinander, sodass keine gegenseitige Beeinflussung, beispielsweise im Fehlerfall, stattfinden kann. Diese Isolation wird durch die gemeinsamen Hardwareressourcen eines Multi-Core-Systems zunächst aufgehoben. So kann beispielsweise durch einen gemeinsam zugreifbaren Speicher ein Motor-Controller im Falle eines Speicherzugriffsfehlers die Regelparameter des benachbarten Controllers verändern. Dies kann zu unvorhersehbaren Reaktionen des Antriebs führen, wodurch eine Gefahr für das System und die Umgebung entstehen kann. Hier empfiehlt sich die Verwendung entsprechender Schutzmaßnahmen, um die räumliche Isolation zwischen den parallel ausgeführten Motor-Controllern wiederherzustellen.

Architekturen mit diesen Funktionalitäten finden sich vor allem im Automotive-Bereich, beispielsweise in Steuergeräten für Motoren und für automatische Getriebe oder zur Integration adaptiver Assistenzsysteme. Hier verbreitete Multi-Core-Mikrocontroller integrieren im Allgemeinen Prozessorkerne mit DSP-Funktionalitäten wie Infineon Tricore™ Architekturen [74], Power Architecture™[48] oder Arm®-Architektur [6]. Durchschnittlich werden pro Chip zwei bis drei Prozessorkerne verbaut [77][151][157]. Abbildung 4.2 zeigt eine Verallgemeinerung der hier verwendeten Architekturen am Beispiel eines Systems mit zwei Prozessorkernen.

Jeder Prozessorkern besitzt lokale Code- und Datenspeicher in Form von schnellen Scratchpad-Arbeitsspeichern. Da diese Arbeitsspeicher Zugriffszeiten im Bereich von Cache-Speichern ermöglichen, werden für Zugriffe auf diese lokalen Speicher typischerweise keine Caches eingesetzt. Zugriffe auf globale Flash- und Arbeitsspeicher können über lokale Caches durchgeführt werden. Hardwareseitig ist kein Kohärenzmechanismus zwischen den Caches und dem globalen Speicher integriert. Zur Verbin-

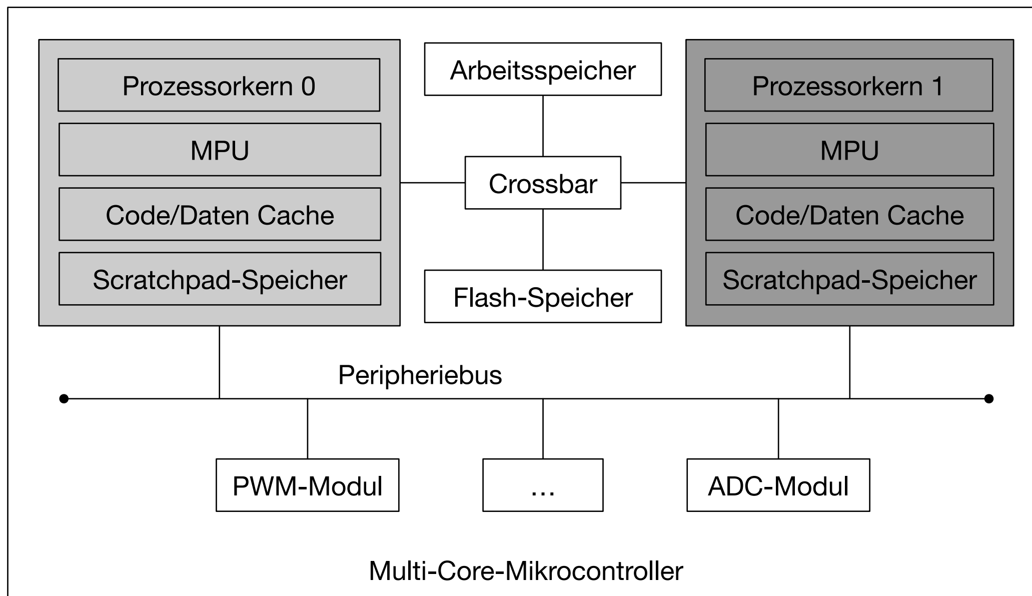


Abbildung 4.2: Allgemeiner Aufbau der betrachteten Multi-Core-Architekturen aus dem Automotive-Bereich

dung der Prozessorkerne mit globalen Speicher- und I/O-Peripheriemodulen werden zwei unterschiedliche Intra-Chip-Verbindungen eingesetzt. Eine Crossbar verbindet die Kerne mit global zugreifbarem Arbeits- und Flash-Speicher. Konkurrenz entsteht hier nur dann, wenn parallel auf ein an die Crossbar angeschlossenes Modul zugegriffen wird. Konkurrierende Zugriffe behandelt die Crossbar beziehungsweise das entsprechende Modul durch eine Arbitrierung, die regelmäßig nach einer Round-Robin-Strategie oder basierend auf statischen Prioritäten durchgeführt wird. I/O-Module sind über ein Peripheriebussystem mit den Prozessorkernen verbunden. Alle parallelen Zugriffsversuche auf den Bus müssen arbitriert werden, unabhängig davon, auf welche konkreten Module versucht wird zuzugreifen. Zur Partitionierung von Speicher und I/O-Modulen integriert jeder Prozessorkern eine MPU. Hierdurch können feingranulare Zugriffsrechte auf definierte Adressbereiche bis hin zu einzelnen Adressen realisiert und fehlerhafte Zugriffsversuche erkannt und nachverfolgt werden.

Store-Buffer realisieren eine out-of-order Ausführung von Store-Instruktionen. Diese Anweisungen resultieren nicht direkt in einem schreibenden Zugriff auf das adressierte Speicher- oder I/O-Modul. Stattdessen wird ein Job zum Schreiben der entsprechenden Daten in den Store-Buffer eingefügt. Diese Jobs werden im Hintergrund, parallel zur Ausführung von Instruktionen durch den Prozessorkern, ausgeführt. Dementsprechend wird der Kern durch den schreibenden Zugriff nicht belastet. Durch diese parallele Abarbeitung können lesende Zugriffe des Prozessorkerns

mit dem Store-Buffer um den Zugriff auf ein Modul konkurrieren. In diesem Fall werden die lesenden Zugriffe vorrangig behandelt, um Wartezeiten für den Kern aufgrund eines belegten Mediums infolge von schreibenden Zugriffen des eigenen Store-Buffer zu verhindern. Bereits aktive Zugriffe des Store-Buffer werden hierbei nicht unterbrochen, wodurch eine Wartezeit bis zum lesenden Zugriff entstehen kann. Zur Erhaltung der Konsistenz wird aus dem Store-Buffer statt aus dem Zielmedium gelesen, falls für die zu lesende Zieladresse ein Schreibauftrag im Buffer existiert.

Zum Aufbau von Multi-Core Motor-Controllern werden in dieser Arbeit Infineon Aurix™ Multi-Core-Mikrocontroller mit einem (TC23 [75]), zwei (TC26 [76]) und drei (TC27 [77], TC29 [79]) Tricore™ TC1.6.1 Prozessorkernen betrachtet. Unterschiede zwischen den Systemen bestehen in den Größen ihrer lokalen Arbeitsspeicher und lokalen Caches sowie im Umfang einiger Peripheriemodule. Jeder Prozessorkern arbeitet mit einer maximalen Taktfrequenz von 200 MHz und integriert insgesamt bis zu 200 KiB lokalen Scratchpad-Arbeitsspeicher und bis zu 24 KiB lokalen Cache-Speicher. Durch parallele Load/Store- und Integer-Pipelines ist ein Kern in der Lage, bis zu zwei Instruktionen pro Taktzyklus auszuführen. Neben globalen I/O-Modulen teilen sich alle Prozessorkerne bis zu 32 KiB große globale Arbeitsspeicher und bis zu 2 MiB große Flash-Module zur persistenten Ablage von Programmcode. Integriert sind die Mikrocontroller auf jeweils einem Infineon Application Kit TC2x7 Board [80]. Unterschiede zwischen den Boards bestehen lediglich durch den jeweils verwendeten Mikrocontroller. Die Leistungsaufnahme der Gesamtsysteme beträgt zwischen 1 W und 2 W.

4.2 Software-Plattform des Multi-Core Motor-Controllers

Die effektive und zuverlässige Nutzung eines Multi-Core-Systems setzt den Einsatz eines Betriebssystems zur Bildung nebenläufiger und paralleler Anwendungen sowie zur Verwaltung mehrere Speicherbereiche voraus. Im Rahmen der Parallelisierung stellt das Betriebssystem Werkzeuge bereit, um Anwendungen in parallel ausführbare Aktivitäten (Tasks) aufzuteilen und um diese auf unterschiedlichen Prozessorkernen synchronisiert auszuführen. Zur Konsolidierung dedizierter Systeme muss das Betriebssystem eine Partitionierung der Ressourcen des Multi-Core-Systems realisieren, um voneinander separierte Teilsysteme zu bilden. Dies umfasst eine Zugriffskontrolle auf prinzipiell global zugreifbare Speicher- und I/O-Module, um eine gegenseitige Beeinflussung der Teilsysteme im Falle von Fehlern zu unterbinden. Be-

triebssystemdienste müssen eine etwaige Interaktion zwischen Systemen, die vor der Konsolidierung über externe Kommunikationsmedien realisiert wurde, ersetzen. Aus Parallelisierung und Konsolidierung resultieren softwareseitig Vorgehensweisen und Muster für die Implementierungen und Integration der jeweiligen Anwendungen.

4.2.1 Softwarestrukturen zur Parallelisierung

Für die Parallelisierung von geberlosen feldorientieren Stromregelungen (GFOS) werden zwei Strategien betrachtet. Die Parallelisierung auf Task-Ebene zielt auf eine Reduktion der Laufzeit der Anwendung. Durch eine Erhöhung der Regelfrequenz oder eine Verkürzung der Signal-Update-Verzögerung [58] kann dies zur Verbesserung der Regelperformance genutzt werden. Die zweite Strategie führt eine Parallelisierung in Form einer Pipeline durch. Hierbei werden mehrere Iterationen eines Algorithmus parallel, jedoch zeitlich versetzt, ausgeführt. Dies ermöglicht eine Erhöhung der Regelfrequenz bei gleichbleibender Laufzeit.

Entsprechend der umzusetzenden Parallelisierungsstrategie wird die GFOS in mehrere Tasks aufgeteilt, die auf unterschiedliche Kerne verteilt werden. Um zeitliche Aufwände durch Scheduling beziehungsweise Kontextwechsel zu minimieren, wird jede Task in Form einer Interrupt-Service-Routine (ISR) implementiert. Die Ausführung von Code per Interrupt ist ein Standardmechanismus moderner Mikrocontroller.

4.2.1.1 Parallelisierung auf Task-Ebene

Bei der Parallelisierung auf Task-Ebene wird eine serielle ausgeführte Anwendung in voneinander unabhängige Tasks aufgeteilt. Abbildung 4.3 zeigt dies am Beispiel der Tasks $T0_a$, $T0_b$, $T0_c$, *T1* und *T2*. Eine gültige serielle Ausführung der Tasks wäre beispielsweise $T0_a \rightarrow T0_b \rightarrow T1 \rightarrow T2 \rightarrow T_c$. Zudem wird von einer zyklischen Ausführung der Taskfolge ausgegangen. Es wird davon ausgegangen, dass alle zeitlichen Anforderungen an die Anwendung erfüllt werden und ein Zyklus erst dann gestartet wird, nachdem der Vorangegangene beendet wurde.

Alle Tasks werden statisch einem Prozessorkern zugewiesen. Zum Start von $\mathrm{T0_a}$ wird ein zeitgesteuerter Interrupt eingesetzt. Die Tasks $T0_b$, *T1* und *T2* werden parallel zueinander ausgeführt. Die Ausführung von *T1* und *T2* kann beginnen, wenn die von den Tasks benötigten Eingangsdaten durch $T0_a$ bereitgestellt wurden. Entsprechendes gilt für $T0_c$ bezüglich *T1* und *T2*. An den beiden Übergängen zwischen serieller und paralleler Ausführung findet somit eine Synchronisation statt. Hierbei werden Eingangsdaten übertragen und eine Signalisierung zum Start der von

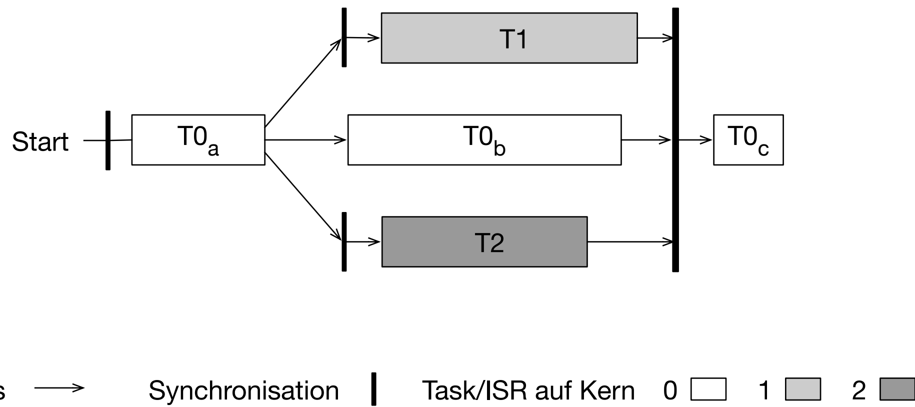


Abbildung 4.3: Parallelisierung durch Tasks

den Daten abhängigen Tasks durchgeführt. Diese Vorgänge können mit Kosten im Sinne zusätzlicher Laufzeiten verbunden sein. In Anbetracht der kurzen Zeitfenster (typischerweise zwischen 10 und 100 μs), die im Rahmen einer Stromregelung zur Verfügung stehen, sollte eine kernübergreifende Synchronisation möglichst schnell sein, um die Kosten der Parallelisierung zu minimieren.

Die Synchronisation des Übergangs von serieller zu paralleler Ausführung wird über ein Interruptsignal realisiert. Das Auslösen eines Interrupts startet die Ausführung der mit der entsprechenden ISR verbundenen Task. Es wird angenommen, dass ein Kern zum Zeitpunkt einer Taskausführung nicht belegt ist. Der Übergang von paralleler zu serieller Ausführung wird über eine Dijkstra Semaphore [36] realisiert. Die Semaphore wird durch einen Zähler und zwei atomare Funktionen repräsentiert, welche den Wert des Zählers inkrementieren und dekrementieren. Erreicht der Zähler nach dem Dekrementieren durch eine Task einen negativen Wert, so muss die Task warten, bis der Zähler wieder den Wert 0 erreicht. Dies geschieht durch parallele Tasks, die den Wert inkrementieren. Sollen die parallelen Tasks *T1* und *T2* in $T0_c$ übergehen, so wartet die Task $T0_c$ an der Semaphore, die zum Zeitpunkt des Wartens den Wert -2 besitzt. Der Wert der Semaphore wird durch *T1* und *T2* jeweils einmal inkrementiert. Erreicht er 0, kann $T0_c$ mit der Ausführung beginnen. Dieses Vorgehen ist zur Zusammenführung beliebig vieler Tasks erweiterbar.

Zur Datenübertragung zwischen Prozessorkernen wird ein gemeinsam zugreifbarer Speicherbereich auf einem globalen Speichermodul verwendet. Jeder schreibende Kern erhält einen dedizierten Bereich innerhalb dieses Moduls. Durch das Auslösen eines Interrupts oder durch das Setzen einer Semaphore wird das Zugriffsrecht auf einen solchen Bereich an den lesenden Kern übertragen.

Die Kosten der Synchronisation umfassen die Laufzeiten, welche für das Schreiben beziehungsweise das Lesen der Daten in und aus dem globalen Speicher anfallen.

Weitere Kostenfaktoren werden durch die Interruptlatenz und die Operationen der Semaphore gebildet.

4.2.1.2 Parallelisierung mittels Pipeline

Anstelle einzelne Tasks innerhalb einer Iteration eines Algorithmus parallel auszuführen, führt eine Pipeline Tasks aus unterschiedlichen Iterationen parallel aus. Abbildung 4.4 zeigt dies am Beispiel einer zweistufigen Pipeline. Jeder Prozessorkern bildet eine Stufe und integriert eine Task. Das Ausführen dieser Task entspricht einer Iteration des zu betrachtenden Algorithmus. Die Tasks der unterschiedlichen Stufen werden zeitgesteuert mit einem Versatz $\Delta Start$ zueinander ausgeführt, sodass mehrere Iterationen parallel berechnet werden. Das Aktivieren einer Stufe erfolgt nach dem Takt der Pipeline. Daraus folgt, dass bei gefüllter Pipeline nach jedem Takt eine Iteration beendet wird. Bestehen zwischen zwei Iterationen keine Datenabhängigkeiten, so entspricht bei N Prozessorkernen die minimale Periodendauer eines Taktes der Pipeline

$$T_{Pipeline} = \frac{T_{Anwendung}}{N} = \Delta Start \tag{4.1}$$

In diesem Fall ist jeder Prozessorkern vollständig ausgelastet und $\Delta Start$ entspricht der Periodendauer der Pipeline.

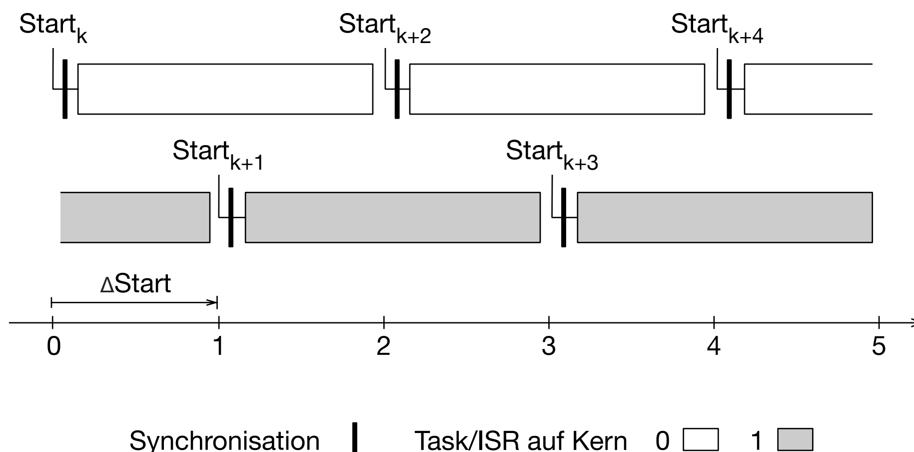


Abbildung 4.4: Parallelisierung durch eine zweistufige Pipeline

Sind Berechnungen in Iteration $k + 1$ von Ergebnissen aus k abhängig, so muss $\Delta Start$ ausreichend groß gewählt sein, dass alle Datenabhängigkeiten rechtzeitig aufgelöst werden. Abhängigkeiten zwischen Operationen in einer Pipeline können allgemein über die Distanz zwischen der abhängigen Operation j und der die Abhängigkeit auflösenden Operation i beschrieben werden [41]. Die Distanz D beschreibt

die Laufzeit zwischen beiden Operationen. Hierzu werden die Zeitpunkte t_j und t_i betrachtet, an denen beide Operationen im seriellen Code, der einer Iteration zugrunde liegt, ausgeführt werden. Die Distanz ergibt sich entsprechend Gleichung 4.2.

$$D = t_j - t_i \tag{4.2}$$

Ist $D > 0$, so wird j zeitlich nach i ausgeführt und die Abhängigkeit hat keine Auswirkung auf die Pipeline. Bei $D < 0$ wird j vor i ausgeführt. Dies bedeutet, dass zur Ausführung von j das Ergebnis von i aus der vorangegangenen Iteration verwendet werden muss. Dies wird erreicht, indem der Wert für $\Delta Start$ vergrößert und $T_{Pipeline}$ entsprechend angepasst wird. Die angepasste Periodendauer der Pipeline ergibt sich aus Gleichung 4.3.

$$T_{Pipeline} = \max(\Delta Start, |D|) \tag{4.3}$$

Für $|D| > \Delta Start$ wird die Periodendauer der Pipeline um $|D| - \Delta Start$ verlängert. $T_{Pipeline}$ wird in diesem Fall größer als $T_{Anwendung}$. Für einen Prozessorkern entsteht dadurch eine Wartezeit zwischen der Ausführung von zwei Zyklen der Pipeline.

4.2.2 Softwarestruktur der Konsolidierung

Bei der Konsolidierung werden mehrere vollständige Motor-Controller in das Multi-Core-System integriert. Jeder integrierte Controller wird als ein Teilsystem des Multi-Core-Systems betrachtet, das sich aus einer oder mehreren Tasks zusammensetzen kann. Intuitiv wird ein Teilsystem vollständig auf einen Prozessorkern abgebildet. Prinzipiell sollten die Tasks eines Teilsystems auch auf unterschiedliche Kerne verteilt werden können. Das Betriebssystem muss sicherstellen, dass jedes Teilsystem nur auf die ihm zugewiesenen Speicherbereiche und I/O-Module schreibend zugreifen kann. Dies bedeutet auch, dass ein einfacher Datenaustausch über gemeinsamen Speicher durch Funktionen des Betriebssystems realisiert werden muss, um die sichere Trennung der Teilsysteme beizubehalten. Die Form, in der Teilsysteme beziehungsweise Tasks implementiert werden, ist vom eingesetzten Betriebssystem und dessen Schutzkonzepten abhängig. Die verwendete System- und Softwarearchitektur für konsolidierte Systeme ist in Abbildung 4.5 dargestellt.

4.2.3 Multi-Core-Betriebssystem

Für die betrachteten Multi-Core-Mikrocontroller stehen derzeit drei Betriebssysteme zur Verfügung, die eine hardwaregestützte Implementierung sicherer Teilsysteme

unterstützen: Flexible Safety RTOS [40], PXROS-HR [66] und Erika Enterprise [42]. Prinzipiell können alle genannten Systeme für den MCMC verwendet werden, da sie jeweils schnelle Kontextwechsel und eine sichere Funktionsintegration basierend auf einer MPU ermöglichen. Letztere ermöglicht die Definition und Durchsetzung von Zugriffsrechten auf Adressbereiche zur Bildung sicherer Teilsysteme. Durch die Zertifizierung nach SIL 3 (IEC61508) und ASIL D (ISO 26262) von Flexible Safety RTOS und PXROS-HR sind diese Systeme vorzuziehen, um den MCMC auch in industrienahen Umgebenen einsetzen zu können. Letztendlich wird PXROS-HR für die Realisierung des MCMC und die weiteren Untersuchungen der Parallelisierung und Konsolidierung verwendet. Hierzu bietet es flexiblere Möglichkeiten zur Implementierung von Tasks und ISR als vergleichbare Betriebssysteme. Insbesondere durch komfortablere Schnittstellen zur Einbindung unterschiedlicher Adressbereiche von Speichern und I/O-Modulen in die geschützten Teilsysteme. Diese Eigenschaften schließen Alternativen nicht aus, erleichtern jedoch die Implementierung und Integration von Motor-Controller auf den Multi-Core-Mikrocontroller.

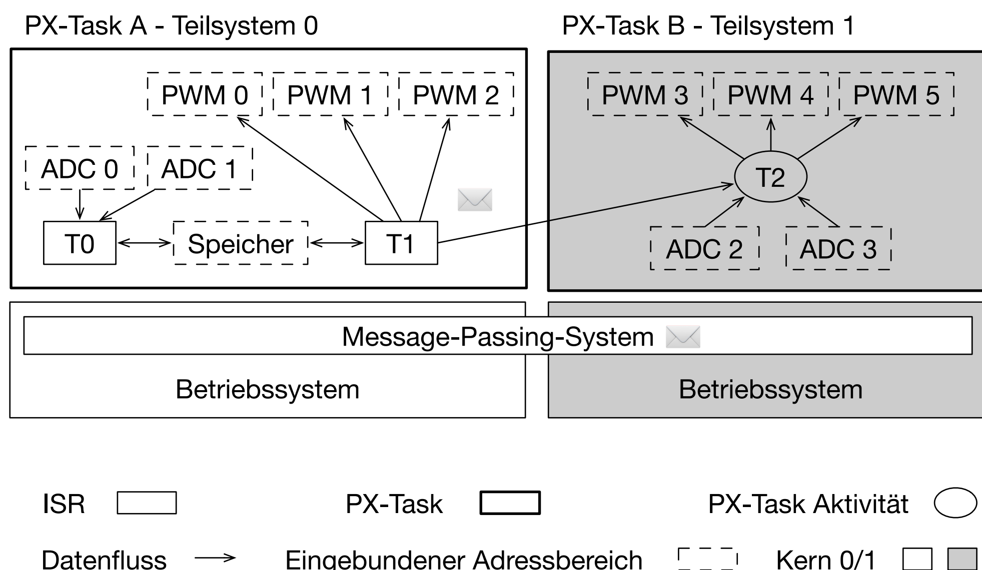


Abbildung 4.5: Softwarestruktur zur Konsolidierung

Abbildung 4.5 zeigt die beispielhafte Konsolidierung von zwei Systemen mittels PXROS-HR. Der Begriff Task beschreibt im Kontext von PXROS-HR einen gekapselten Prozess (PX-Task), der eine Aktivität und Daten umfasst. Eine PX-Task wird statisch einem Prozessorkern zugewiesen und kann mittels Message-Passing mit anderen PX-Tasks kommunizieren. Jede PX-Task wird durch den Mikrokernel des Betriebssystems unter Verwendung der MPU gekapselt, sodass sie lediglich auf Adressräume zugreifen kann, die in den Kontext der Kapsel eingebunden sind. PX-Tasks

repräsentieren somit im Sinne der Konsolidierung sichere Teilsysteme. Weiterhin ermöglicht das Betriebssystem im Kontext von PX-Tasks ISR auszuführen. Analog zu Threads unterbrechen diese ISR die eigentliche Aktivität der PX-Task und können auf den kompletten Adressraum der Kapsel zugreifen. Diesbezüglich zeigt Teilsystem 0 die Integration der Anwendungstasks *T0* und *T1* in Form von zwei ISR, die geschützt innerhalb des Kontextes von PX-Task A ausgeführt werden. Beide ISR realisieren eine schnelle Kommunikation über gemeinsamen Speicher. Zusätzlich zu diesem sind auch die Adressbereiche der PWM- und ADC-Kanäle, die Teilsystem 0 zur Ansteuerung des Motors benötigt, in den Kontext von PX-Task A eingebunden. *T2* wird direkt als Aktivität von PX-Task B ausgeführt. Ein Datenaustausch zwischen den Teilsystemen ist über das Message-Passing-System des Betriebssystems realisiert. Durch das Konzept sogenannter Mailboxes ermöglicht PXROS-HR sowohl blockierendes als auch blockierungsfreies Warten auf eine Nachricht.

4.3 Zeitliche Kosten der parallelen Ausführung

Die Synchronisation paralleler Tasks sowie der Austausch von Daten zwischen unterschiedlichen Kernen bedingt Kosten in Form von zusätzlichen Laufzeiten. Neben dem Schreiben und Lesen von Nutzdaten in und aus globalem Speicher entstehen diese Zeiten durch die Vorgänge der Synchronisation.

Werden bei der Synchronisation m Datenworte von oder zu N Prozessorkernen übertragen, ergeben sich, bei Vernachlässigung von Cross-Core-Interferenzen, maximale zusätzliche Laufzeiten von

$$\begin{aligned} t_{ser \rightarrow par} &= m \cdot (t_{schreiben} + t_{lesen}) + t_{sync_isr} \\ t_{par \rightarrow ser} &= m \cdot (t_{schreiben} + t_{lesen}) + N \cdot t_{sync_sem} \end{aligned} \tag{4.4}$$

Bei einem Übergang von serieller zu paralleler Ausführung ist die Interruptlatenz zu berücksichtigen. Nach dem Auslösen der Interruptsignale wird deren Bearbeitung parallel auf den jeweiligen Zielkernen durchgeführt, bis diese letztendlich die jeweilige ISR ausführen. Der Zugriff auf eine Semaphore ist stets ein Versuch, deren Wert zu ändern. Bei konkurrierenden Zugriffen muss dieser Versuch gegebenenfalls wiederholt werden. Wird davon ausgegangen, dass Kerne lediglich innerhalb einer Iteration der Anwendung um eine Semaphore konkurrieren, ist die Anzahl an Versuchen, bis eine Änderung gelingt, durch die Anzahl der um die Semaphore konkurrierenden Kerne begrenzt.

In welchem Maß die Kosten die gesamte Laufzeit der Anwendung beeinflussen, hängt von dem Verhältnis der Laufzeiten der zu synchronisierenden Tasks zueinander

Tabelle 4.1: Gemessene Laufzeiten einer kernübergreifenden Synchronisation und Datenübertragung zwischen zwei Tricore™ TC1.6.1 P Prozessorkernen

Aktion	*CPU-Takte*
Schreiben eines Datenworts (Variable, $t_{schreiben}$)	13
Lesen eines Datenworts (Variable, t_{lesen})	17
Interruptlatenz (t_{sync_isr})	72
Synch. an Semaphore (t_{sync_sem})	73
Inter-Core Message-Passing (t_{msg})	$m \cdot (t_{schreiben} + t_{lesen}) + 1220$

ab. In Abbildung 4.3 ist die Laufzeit von *T2* geringer als die von $T0_b$. Beim Übergang von paralleler zu serieller Ausführung erfolgt die Synchronisation von *T2* während $T0_b$ und *T1* weiterhin Berechnungen ausführen. Die Kosten der Synchronisation von *T2* fallen somit nicht ins Gewicht.

Tabelle 4.1 zeigt die ermittelten Kosten für den MCMC. Die Werte sind als Größenordnung zu verstehen, da sie durch Abweichungen in der Implementierung, beispielsweise durch unterschiedliche Zugriffsmuster auf Daten, variieren können. Die Laufzeiten wurden experimentell mittels kernunabhängiger Hardware-Timer des Mikrocontrollers ermittelt. Die Laufzeiten zum Lesen und Schreiben sind für eine Wortgröße von bis zu 4 Byte gültig. Dies entspricht einer Transaktion zwischen einem Prozessorkern und dem globalen Speichermodul des Infineon Aurix™. Die erhaltenen Messwerte sind spezifische Daten der eingesetzten Hardware-Software-Plattform. Vergleichbare Systemarchitekturen setzen Komponenten ein, die sich im Grunde nicht wesentlich von den hier verwendeten unterscheiden. Da keine spezifischen Funktionen des Mikrocontrollers verwendet wurden, kann davon ausgegangen werden, dass alternative Multi-Core-Mikrocontroller und Betriebssysteme vergleichbare zeitliche Eigenschaften aufweisen.

4.4 Zusammenfassung der Ergebnisse

Zur Realisierung des MCMC wurde eine geeignete Multi-Core-Architektur abgegrenzt und ein Betriebssystem als Softwareplattform gewählt. Als Multi-Core-Mikrocontroller kommen mit Infineon Aurix™ Mikrocontroller Architekturen aus dem Automotive-Bereich zum Einsatz. Entsprechende Systeme bieten ein angemessenes Verhältnis zwischen der Rechenleistung und Leistungsaufnahme, ausreichend leistungsfähige On-Chip-Peripherie zur Signalverarbeitung und Mechanismen zur sicheren Funktionsintegration. Das gewählte Betriebssystem PXROS-HR bietet flexible

Möglichkeiten zur parallelen Integration der fokussierten Anwendungen. Mit dieser Kombination aus Hard- und Software ist eine Plattform gefunden, die ausreichend Leistung und Flexibilität bietet, um die Aufgaben der Parallelisierung und Konsolidierung durchführen zu können. Darüber hinaus können der spezifizierte MCMC und die auf ihm basierenden Forschungsergebnisse durch die Berücksichtigung von Safety-Funktionen praxisnah verwertet werden. Die erhaltenen Ergebnisse sind beispielhaft für diese Plattform. Der Multi-Core-Mikrocontroller und das genutzte Betriebssystem setzen Komponenten ein und bieten Funktionalitäten, die auch alternative Systemarchitekturen nutzen und bieten. Daher können die Ergebnisse auf MCMC bestehend aus alternativen Hard- und Softwarekomponenten übertragen werden.

Durch die Parallelisierung auf Task-Ebene und mittels einer Pipeline wurden zwei Strategien für die Parallelisierung von geberlosen Stromregelungen vorgestellt. Diese basieren auf der Verwendung von Interrupt-Service-Routinen und Semaphoren. Zur Konsolidierung mehrerer Motor-Controller wurde auf Basis von Betriebssystemtasks der Aufbau gekapselter Teilsysteme beschrieben. Das Betriebssystem garantiert hierbei, im Sinne von Safety, eine sichere Kapselung der Teilsysteme beziehungsweise der darin integrierten Motor-Controller-Software.

Eine durchgeführte Evaluation des MCMC beschreibt die grundlegenden Kosten, die im Rahmen einer Parallelisierung entstehen können. Für die Übergänge zwischen serieller und paralleler Ausführung liegen die Kosten für ein zu übertragendes Datenwort bei 102 CPU-Takten. Bei einer Taktrate von 200 MHz entspricht dies 0,51 μs. Für jedes weitere Datenwort steigen die Kosten um zusätzliche 30 Takte (0,15 μs). Werden Regelfrequenzen im Bereich von 40 kHz und mehr betrachtet, sind Kosten im Bereich von 1 μs eine akzeptable und realisierbare Größenordnung. Der MCMC und vergleichbare Architekturen werden daher für die Parallelisierung einer geberlosen Antriebsregelung als prinzipiell geeignet angesehen. Es bleibt jedoch zu untersuchen, ob geberlose Antriebsregelungen ausreichend parallelisiert werden können, um einen effektiven Performance-Gewinn zu erzielen.

Die Konsolidierung von Motor-Controllern erlaubt die Verwendung gemeinsamer Speicher zur Nachbildung von Kommunikationsbeziehungen. Aufgrund der einzuhaltenden Safety-Eigenschaften wird für die Kommunikation zwischen konsolidierten Motor-Controllern beziehungsweise zwischen den entsprechenden Teilsystemen die Message-Passing-Funktion des Betriebssystems verwendet. Im Vergleich zu einer einfachen, jedoch unsicheren Synchronisation der Speicherzugriffe durch Semaphoren sind die Kosten von Message-Passing relativ hoch. Für jede zu sendende Nachricht

entsteht ein Offset von 1220 CPU-Takten (6,1 μs bei 200 MHz). Ersetzt Message-Passing eine Kommunikation über ein externes Bussystem, das separate Motor-Controller miteinander verbindet, so stellt Message-Passing eine deutlich effizientere Alternative dar. Beispielsweise veranschlagt ein standardmäßig verwendeter CAN-Bus eine Übertragungsdauer von mehr als 100 μs pro Datenpaket mit 8 Byte Nutzdaten. Eine Übertragung der gleichen Nutzdatenmenge beträgt über Message-Passing 6,4 μs (zwei Datenwörter zu je 4 Byte und 200 MHz CPU-Takt). Für jedes weitere Datenwort werden lediglich 0,15 μs zusätzlich benötigt. Am Beispiel des CAN-Bus wäre die vollständige Übertragung eines weiteren Datenpakets notwendig. Vor diesem Hintergrund kann der MCMC, neben der potenziellen Möglichkeit Hardware- und Energiekosten einzusparen, deutliche Vorteile gegenüber separaten miteinander interagierenden Motor-Controllern erzielen. Hier bleibt zu evaluieren, in welchem Rahmen Cross-Core-Interferenzen die zeitlichen Eigenschaften der integrierten Teilsysteme und die Datenübertragungsraten von Kommunikationsbeziehungen beeinflussen.

Kapitel 5

Energiebetrachtung des Multi-Core-Motor-Controllers

Bei den betrachteten Multi-Core-Mikrocontrollern sind zugunsten von Safety-Eigenschaften typischerweise keine Funktionen bezüglich einer dynamischen Frequenz und Spannungsskalierung verfügbar. Eine signifikante Reduktion von Energiekosten, wie sie in anderen Anwendungsgebieten erreicht werden kann, ist somit auszuschließen. In den folgenden Betrachtungen wird untersucht, welche Auswirkungen die Erhöhung der Anzahl der Prozessorkerne auf die Leistungsaufnahme und damit den Energiebedarf eines Motor-Controllers hat. Die Untersuchungen basieren auf der experimentellen Ermittlung der Leistungsaufnahme der vorgestellten Infineon Application Kit TC2x7 Systeme mit einem (TC237), zwei (TC267) und drei (TC277) Prozessorkernen [166]. Einflüsse unterschiedlicher Systemarchitekturen und Bauformen können so minimiert werden. Die Betrachtung basiert auf einer verallgemeinerten Beschreibung der eingebetteten Systeme und ist damit nicht auf die konkret verwendeten Architekturen begrenzt [12]. Somit kann das Vorgehen auf alternative Multi-Core-Systeme angewendet werden.

5.1 Bestimmung der Leistungsaufnahme

Die effektive Leistungsaufnahme der Motor-Controller wird über deren Versorgungsspannung und Messungen ihrer Eingangsströme ermittelt. Die Aufnahme kann auf diese Weise präzise für einen gegebenen Betriebszustand des Multi-Core-Mikrocontrollers ermittelt werden. Die erhaltenen Daten summieren die elektrische Leistungsaufnahme der unterschiedlichen Komponenten des Systems. Für eine detaillierte Betrachtung werden die gemessenen Werte nach diesen Komponenten zerlegt. Aus-

gehend von der Leistungsaufnahme des Gesamtsystems können entsprechende Teilleistungen durch systematisches Aktivieren beziehungsweise Deaktivieren einzelner Komponenten ermittelt werden [12].

5.1.1 Systemmodell des Multi-Core-Systems

Abbildung 5.1 beschreibt eine allgemeine Gliederung der Komponenten nach Board-, Chip- und Kern-Level. Der Board-Level umfasst alle elektrischen Komponenten, die auf der Platine des TC2x7 integriert sind. Der Stromverbrauch des Board-Levels entspricht dem gemessenen Verbrauch des gesamten Systems. Der Chip-Level umfasst alle On-Chip-Module innerhalb des Mikrocontrollers. Diese werden von allen Prozessorkernen geteilt. Auf Kern-Level sind alle Komponenten zusammengefasst, die exklusiv einem Prozessorkern zugeordnet sind. Eine Beziehung zwischen zwei Komponenten besteht, wenn sie durch den Austausch von Daten oder Signalen miteinander interagieren. Dies kann innerhalb eines Levels oder über dessen Grenzen hinaus geschehen.

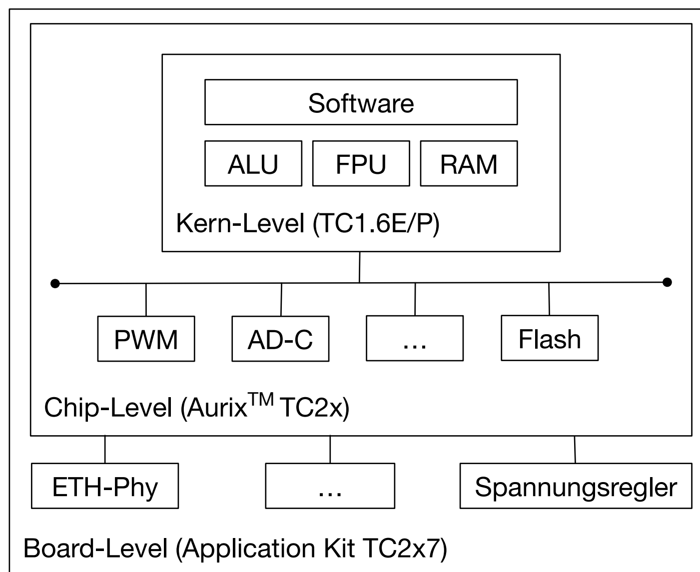


Abbildung 5.1: Systemmodell zur Betrachtung der Leistungsaufnahme

Die Messung der Eingangsströme des Motor-Controllers ergibt entsprechend Gleichung 5.1 die Leistungsaufnahme des Gesamtsystems P_{System}.

$$P_{System} = P_{Idle} + P_{Peripherie} + \sum_{i=0}^{i<N} P_{Kern_i} \tag{5.1}$$

P_{System} summiert den Grundverbrauch des Systems (P_{Idle}) und anwendungsabhängige Anteile, die sich aus der Leistungsaufnahme der genutzten Peripherie ($P_{Peripherie}$)

und die der Prozessorkerne (P_{Kern}) ergeben. Der Grundverbrauch liegt dann an, wenn mit Ausnahme der Prozessorkerne alle Komponenten des Systems deaktiviert sind. Die Kerne befinden sich in einem Wartezustand, der bei dem MCMC über die Prozessorinstruktion *wait* erreicht wird. In diesem Zustand sind die Kerne rechenbereit, führen jedoch keine Instruktionen aus.

5.1.2 Ermittlung der statischen Leistungsaufnahme

P_{Idle} umfasst die temperaturabhängige statische Leistungsaufnahme des Mikrocontrollers, die sich aus der statischen Stromstärke auf dessen 1,3 V Versorgungsleitung ergibt. Für die MCMC beträgt die statische Leistungsaufnahme durchschnittlich 3,6 mW (TC237, 35 °C) [81], 2,6 mW (TC267, 39 °C) [82] und 5,0 mW (TC277, 46 °C) [83]. Die statische Leistungsaufnahme geht mit einem geringen Anteil in P_{Idle} ein, der bei einer Chip-Temperatur zwischen 25 °C und 75 °C im Bereich von 0,1 % bis 0,5 % liegt. Im laufenden Betrieb zeigen die Systeme eine Temperaturvarianz von unter 2 °C. Variierende Temperaturen bezüglich der Ermittlung von P_{Idle} können dadurch vernachlässigt werden. Einen weiteren Anteil von P_{Idle} bildet der frequenzabhängige Verbrauch von Komponenten, die für ein betriebsfähiges System aktiv sein müssen oder nicht vollständig deaktiviert werden können. Dies sind beispielsweise Speicherschnittstellen oder Intra-Chip-Verbindungen. Zur Bestimmung von P_{Idle} werden die Taktraten der Prozessorkerne und der internen Intra-Chip-Verbindungen des Mikrocontrollers konfiguriert und alle Komponenten deaktiviert, die dies zulassen. Die Prozessorkerne werden in den *wait*-Zustand versetzt.

5.1.3 Ermittlung der anwendungsabhängigen Leistungsaufnahme

Bei Peripherie-Modulen und Prozessorkernen kann zwischen aktiver und passiver Leistungsaufnahme unterschieden werden. Ein Modul befindet sich in einem passiven Zustand, wenn es aktiviert wurde, aber nicht aktiv verwendet wird. Die Leistungsaufnahme ist dabei von der Taktfrequenz abhängig, mit der das Modul betrieben wird. Die aktiven Zustände werden durch die Anwendung gesteuert, beispielsweise wenn das ADC-Modul dazu veranlasst wird, eine Signalabtastung durchzuführen. Timer und PWM-Module können als dauerhaft aktiv betrachtet werden, da sie nach einer einmaligen Konfiguration und Aktivierung dauerhaft arbeiten. Ein Prozessorkern arbeitet aktiv, wenn er den *wait*-Zustand verlassen hat und Instruktionen ausführt. Seine aktive Leistungsaufnahme ist somit von seiner Auslastung abhängig, die sich

aus der Laufzeit seiner Tasks und der Häufigkeit ihrer Ausführung zusammensetzt. Die aus der Auslastung resultierende Leistungsaufnahme ist linear von der Taktrate des Kerns abhängig [25].

Bei bekanntem P_{Idle} können $P_{Peripherie}$ und P_{Kern} aus P_{System} berechnet werden, wenn der jeweils andere Wert minimiert wird. Bezüglich der Prozessorkerne genügt es, diese in den *wait*-Zustand zu versetzen, nachdem sie die Peripherie konfiguriert und aktiviert haben. $P_{Peripherie}$ setzt sich aus der Summe der Leistungsaufnahmen aus den verwendeten Board- und Chip-Level-Komponenten zusammen. Diese können per Software deaktiviert werden. Für Board-Level-Komponenten ist ein explizites Deaktivieren oftmals nicht möglich. Eine minimale Leistungsaufnahme kann hier erreicht werden, indem sie von extern angeschlossener Elektronik getrennt und die mit ihnen interagierenden Chip-Level-Komponenten deaktiviert werden. So kann keine Interaktion mit der Board-Level-Komponente ausgelöst werden, die eine Aktivität und eine damit verbundene Leistungsaufnahme verursacht.

5.2 Direkter Vergleich der betrachteten Systeme

Für einen direkten Vergleich wird die Leistungsaufnahme der betrachteten Motor-Controller herangezogen, wenn jeweils ein Motor geregelt wird. Hierzu führen alle Systeme die gleiche Anwendung aus, die eine FOS in Kombination mit DFC zur geberlosen Rotorlagebestimmung realisiert [159]. Hardwareseitig werden die On-Chip-Module Generic Timer Module (GTM) zur Generierung der PWM-Signale, Versatile AD-Converter (VADC) zur Signalerfassung und Capture-Compare Unit (CCU) zur zeitlichen Steuerung der Tasks und des VADC eingesetzt. GTM und VADC sind mit I/O-Pins des Board-Levels verbunden. Softwareseitig liest die GFOS gemessene Signalwerte aus den Registern des VADC und schreibt Vorgaben in die Register des GTM. Komplexe Treiber zur Kommunikation mit den Chip-Level-Modulen werden nicht benötigt. Die Anwendung wird entsprechend Teilsystem 1 aus Abbildung 4.5 auf einem Prozessorkern in Form einer PX-Task integriert. Der Kern wird mit einem CPU-Takt in Höhe von 200 MHz, die I/O-Module mit Takt von 100 MHz betrieben. Nicht genutzte Kerne verbleiben im *wait*-Zustand.

Zur Messung von P_{System} wird die GFOS mit einer Regelfrequenz von 40 kHz ausgeführt. Die verwendete Implementierung erzeugt eine Prozessorlast von 93 %. Diese verteilt sich mit 12 % auf DFC und mit 81 % auf die FOS. Die Messungen der Eingangsstromstärke werden durchgeführt, nachdem die Chip-Temperatur einen nahezu konstanten Wert erreicht hat. Die Strommessung wird bei einer Ver-

sorgungsspannung von 9 V durchgeführt und erstreckt sich bei einer Integrationszeit von 20 ms über 800 Regelzyklen pro Messvorgang.

Tabelle 5.1 zeigt die Mittelwerte aus jeweils 500 Messungen von P_{System} und den daraus abgeleiteten Werten P_{Idle}, $P_{Peripherie}$ und P_{Kern}. Weiterhin ist der prozentuale Unterschied der beiden Multi-Core-Systeme gegenüber dem Single-Core-System gezeigt. Mit steigender Anzahl an Prozessorkernen wurde für P_{System} der einzelnen Systeme 1,60 W, 1,70 W und 1,82 W ermittelt. Mit 89,11 %, 91,33 % und 90,81 % deckt P_{Idle} den größten Anteil der Leistungsaufnahme ab. Der verbleibende anwendungsabhängige Anteil an P_{System} teilt sich mit 3,13 %, 3,54 % und 4,36 % auf die Peripherie und mit 7,76 %, 5,13 % und 4,84 % auf die Prozessorkerne der Systeme TC237, TC267 und TC277 auf. Insgesamt zeigen die Multi-Core-Systeme gegenüber dem Single-Core-System eine Steigerung der Leistungsaufnahme um 6,13 % und 13,76 %.

Tabelle 5.1: Leistungsaufnahme der Motor-Controller während der Regelung eines Antriebs, aufgeschlüsselt nach Komponenten. Für die Multi-Core-Mikrocontroller TC267 und TC277 sind die prozentualen Unterschiede zu dem Single-Core-System TC237 aufgeführt. Die zusätzlichen Prozessorkerne der Multi-Core-Systeme sind nicht belastet.

	TC237 [mW]	*TC267 [%]*	*TC277 [%]*
P_{Idle}	1424	+8,71	+15,94
$P_{Peripherie}$	50	+20,0	+58,0
P_{Kern}	124	-29,84	-29,03
P_{System}	1598	+6,13	+13,76

Im Vergleich zum Single-Core-System steigt P_{Idle} um 8,71 % beziehungsweise 15,94 % und $P_{Peripherie}$ um 20 % beziehungsweise 58 %. Dies ist durch die erhöhte Komplexität der Multi-Core-Mikrocontroller gegenüber dem Single-Core-Derivat begründet. Neben den zusätzlichen zu versorgenden Prozessorkernen ist mit einer erhöhten Leistungsaufnahme der I/O-Peripheriemodule zu rechnen, da deren Komplexität oftmals mit der Rechenleistung des Systems skaliert. Am Beispiel des ADC-Moduls zeigt sich dies durch eine Vervielfältigung paralleler Konvertereinheiten und zusätzlicher Kanäle, um erhöhte Anforderungen seitens der Anwendungen abzudecken. Mit 87 mW und 88 mW sinkt die Leistungsaufnahme der Prozessorkerne der Multi-Core-Systeme gegenüber dem Single-Core-System mit 124 mW um 29,84 % und 29,03 %. Da alle betrachteten Mikrocontroller mit der identischen Prozessorarchitektur, Versorgungsspannung und Taktraten arbeiten, ist der höhere Verbrauch des TC23 auf dessen deutlich größeren Scratchpad-Speicher zurückzuführen, wel-

cher der zweieinhalbfachen Speichermenge des TC267 und der eineinhalbfachen des TC277 entspricht.

5.3 Leistungsaufnahme der Multi-Core Nutzungsszenarien

Werden mehrere dedizierte Motor-Controller auf einem MCMC konsolidiert, so bedeutet dies eine Einsparung von Hardware und eine entsprechende Reduktion der Energiekosten. Bei der Parallelisierung bieten zusätzliche Rechenkerne eine potenzielle Chance zur Steigerung der Regelperformance. Wie in Abschnitt 5.2 beschrieben ist, wird die so erreichte Performance-Steigerung durch einen Anstieg der Leistungsaufnahme erkauft. Soll lediglich die Regelperformance des Single-Core-Systems erreicht werden, so ist eine Reduktion der Leistungsaufnahme möglich, indem nach einer Parallelisierung die Rechenleistung der verwendeten Kerne reduziert wird. Ob und in welchem Rahmen die Leistungsaufnahme mit dieser Strategie reduziert werden kann, ist neben dem Maß der Parallelisierbarkeit der Stromregelung auch von den Fähigkeiten zur Skalierung der Rechenleistung des Mikrocontrollers abhängig.

5.3.1 Leistungsaufnahme nach Konsolidierung

Bei der Konsolidierung wird auf allen Kernen die in Abschnitt 5.2 beschriebene Anwendung ausgeführt und P_{System} gemessen. Für jeden zusätzlichen Antrieb wird die Konfiguration der Peripherie zur Signalverarbeitung erweitert und die notwendige Software auf einem zusätzlichen Kern integriert.

Tabelle 5.2 zeigt die absolute und jeweils zusätzlich aufzubringende elektrische Leistung, wenn die MCMC zwei beziehungsweise drei vollständige Instanzen der GFOS ausführen. Die zusätzliche Leistung bezieht sich auf die Konfiguration mit jeweils einem Antrieb weniger. Insgesamt steigt durch die Integration jedes zusätzlichen Antriebes die durchschnittliche Leistungsaufnahme des Multi-Core-Systems um 6,44 % bis 7,96 %. Die Erhöhung der Leistungsaufnahme der Peripherie liegt zwischen 18,56 % und 33,33 % für jeden zusätzlich integrierten Antrieb. Die Aktivierung zusätzlicher Rechenkerne verursacht einen Leistungsanstieg von P_{Kern}, der um 22,73 % bis 37,5 % über der Leistungsaufnahme liegt, die durch P_{Kern} des ersten Antriebes verursacht wird. Diese erhöhte Leistungsaufnahme der Prozessorkerne ist durch ein Architekturmerkmal der Aurix^{TM} Mikrocontroller begründet, die für den zweiten und dritten Kern eine abgewandelte Mikroarchitektur (TC1.6.1 Performan-

Tabelle 5.2: Leistungsaufnahme der MCMC bei zusätzlichen Antrieben

TC267	*1 Antrieb [mW]*	*2 Antriebe [mW]*
$P_{Peripherie}$	60	+20
P_{Kern}	87	+ 116
P_{System}	1695	+135

TC277	*1 Antrieb [mW]*	*2 Antriebe [mW]*	*3 Antriebe [mW]*
$P_{Peripherie}$	79	+18	+18
P_{Kern}	88	+ 121	+108
P_{System}	1818	+139	+126

ce) einsetzen. Diese ist unter anderem mit einer zusätzlichen Pipeline ausgestattet, um eine höhere Rechenleistung zu erzielen. Die Ausführung der GFOS erzeugt auf diesen Kernen eine Auslastung von 75 %. Der erste Kern (TC1.6.1 Efficiency) ist auf eine erhöhte Effizienz ausgelegt.

Die Konsolidierung ersetzt mehrere separate Motor-Controller durch ein Multi-Core-System. Abbildung 5.2 verdeutlicht den Effekt, den diese Reduktion von Hardware auf die Leistungsaufnahme hat. Im Vergleich zur Verwendung mehrerer Single-Core-Systeme ermöglichen die MCMC die Leistungsaufnahme bei zwei Antrieben um 42,74 % (TC267) und bei drei um 56,55 % (TC277) zu senken.

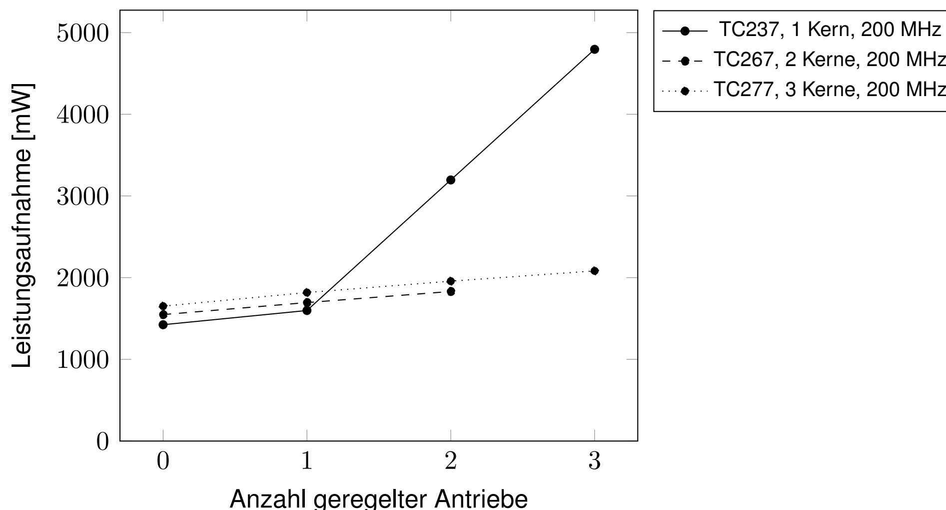


Abbildung 5.2: Gegenüberstellung der Leistungsaufnahme von Single-Core (TC237) und Multi-Core (TC267, TC277) Motor-Controllern

5.3.2 Leistungsskalierung bei Parallelisierung

Multi-Core-Systeme in den Bereichen Desktop und mobile Systeme werden regelmäßig mit Energieeffizienz in Verbindung gebracht. Dies bezieht sich auf die Möglichkeit, mehrere schwächere Rechenkerne einzusetzen, die in Summe eine ähnliche Rechenleistung wie ein Single-Core-System aufbringen, das einen leistungsstarken Prozessorkern besitzt. Die Energieeffizienz ergibt sich dadurch, dass die parallelen Rechenkerne mit geringeren Taktraten und Versorgungsspannungen betrieben werden können [65]. Vor allem die Verringerung der Versorgungsspannung der Prozessorkerne führt durch ihren quadratischen Anteil an der Leistungsaufnahme der Kerne zu einer signifikanten Energieeinsparung [160].

Im hier untersuchten Anwendungsbereich für Antriebsansteuerungen können diese Möglichkeiten zur Skalierung der Rechenleistung nicht im gleichen Maßstab eingesetzt werden. Bei zeit- und sicherheitskritischen Systemen ist eine konstante Versorgungsspannung ein notwendiger Parameter, um die Stabilität und Zuverlässigkeit der Systeme zu gewährleisten [180]. Dementsprechend bieten auch die im Rahmen der Antriebstechnik betrachteten Multi-Core-Architekturen typischerweise keine Möglichkeit, diesen Parameter anzupassen und das entsprechende Potenzial auszunutzen. Somit beschränken sich die Möglichkeiten seitens des MCMC zur Skalierung der Rechenleistung auf eine statische Anpassung der Taktfrequenzen.

Durch die konstante Versorgungsspannung sind die dynamische Leistungsaufnahme eines Prozessorkerns und die Laufzeit seiner Tasks, bei Vernachlässigung ihrer Interaktion mit On-Chip-Komponenten, linear von der Taktfrequenz des Kerns abhängig. Eine Verringerung des Taktes bewirkt in gleichem Maße eine Zunahme der Laufzeiten der Tasks, wodurch insgesamt die Leistungsaufnahme unverändert bleibt. Durch die Verringerung der System- und Peripherietaktraten wird die statische Leistungsaufnahme aller Komponenten des Mikrocontrollers reduziert. Wird die hierdurch entstehende Reduktion der Rechenleistung durch einen zweiten Kern kompensiert, kann gegebenenfalls die Leistungsaufnahme gegenüber einem höher getakteten System verringert werden. Die betrachteten Systeme bieten die Möglichkeit, den globalen Basistakt, auf dem die individuellen Frequenzen aller Komponenten des Mikrocontrollers beruhen, zu halbieren. Prozessorkerne und Speicher werden dann mit einer Taktfrequenz von 100 MHz und Peripheriemodule mit 50 MHz betrieben.

Abbildung 5.3 zeigt die Leistungsaufnahmen der Systeme TC237 und TC267 bei der bekannten Konfiguration mit 200 MHz System- und 100 MHz Peripherietakt. Zusätzlich ist die Leistungsaufnahme des TC267 bei halbierten Taktraten beschrieben. Hierdurch sinkt P_{Idle} um 3,56 % und $P_{Peripherie}$ um 3,17 %. Wird von voll ausgelas-

teten Prozessorkernen ausgegangen, so kann die Leistungsaufnahme von Kern 1 um 47,83 % und von Kern 2 um 52.05 % gesenkt werden. Zur Bestimmung dieser Werte wurden die Kerne durch unterschiedliche Benchmarks [60][52] zu 100 % ausgelastet und die Leistungsaufnahme bestimmt. Die Verwendung verschiedener Benchmarks stellt sicher, dass ein breites Spektrum an ausgeführtem Code und Instruktionen in die Leistungsmessung einfließt. Wird von einer gleichmäßigen Verteilung der Rechenlast auf die beiden Kerne des mit 100 MHz arbeitenden TC267 ausgegangen, so wird P_{System} gegenüber dem mit 200 MHz betriebenen TC267 um 2,82 % gesenkt. Gegenüber dem mit 200 MHz arbeitenden TC237 sinkt die höhere Leistungsaufnahme des TC267 von 6,07 % auf 2,80 %. Für geringer ausgelastete Systeme können sich unterschiedliche Möglichkeiten zur Verteilung ergeben. Durch den linearen Zusammenhang zwischen der Auslastung und P_{Kern}, ist P_{System} prinzipiell unabhängig von der Verteilung.

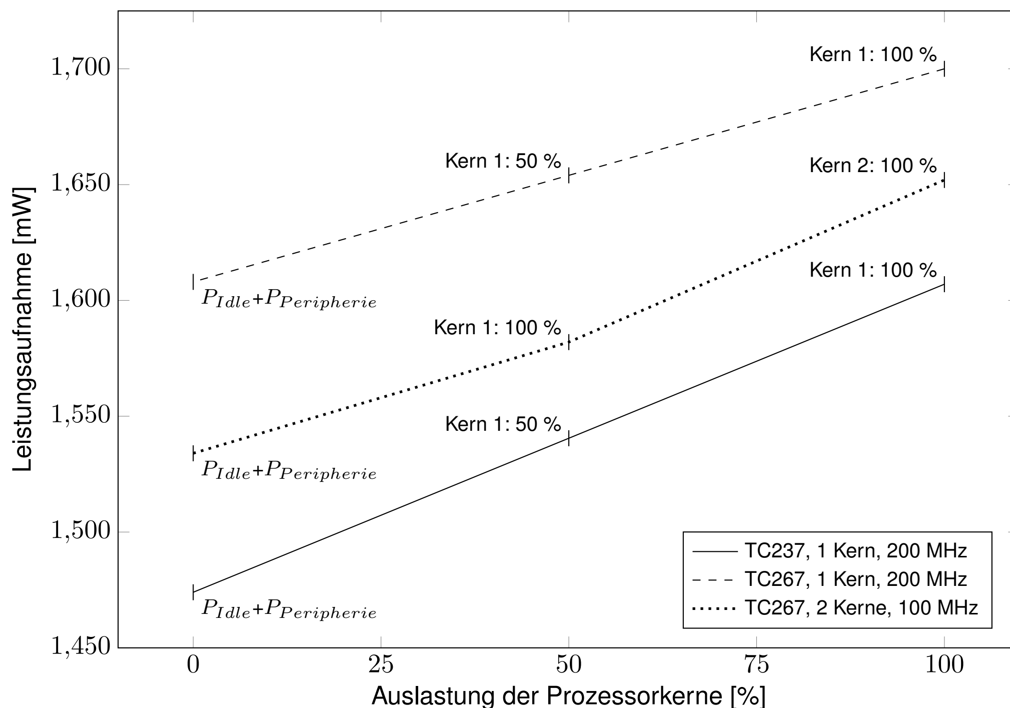


Abbildung 5.3: Leistungsaufnahme eines mit halbem Basistakt betriebenen Multi-Core-Systems: Die Leistungsaufnahme des mit halbem Basistakt betriebenen TC267 ist zu jedem Zeitpunkt höher als die Leistungsaufnahme des mit vollem Takt betriebenen Single-Core-Systems TC237.

5.4 Zusammenfassung der Ergebnisse

Zur Beurteilung der Leistungsaufnahme von MCMC und zum Vergleich gegenüber Single-Core-Systemen wurde ein Vorgehen vorgestellt, um die Leistungsaufnahme einzelner Komponenten der Controller zu bestimmen. Über unterschiedliche Konfigurationen der On-Chip-Peripherie und Prozessorkerne werden die gewünschten Werte über eine Messung der Eingangsströme der Motor-Controller ermittelt. Für vergleichbare Single- und Multi-Core Motor-Controllern, die jeweils das gleiche Maß an Rechenleistung aufbringen, wurde gezeigt, dass die MCMC eine um 6,13 % (TC267) und 13,76 % (TC277) höhere Leistungsaufnahme als das Single-Core-System (TC237) aufweisen. Die Verwendung eines Multi-Core-Systems, ohne dass dessen zusätzliche Kapazitäten genutzt werden, erhöht somit die Energiekosten.

Durch fehlende Funktionen zur dynamischen Frequenz und Spannungsskalierung kann auch eine Verteilung der Rechenlast auf mehrere, jedoch schwächere Prozessorkerne keine nennenswerte Reduktion der Leistungsaufnahme erzielen. Hier kann die Leistungsaufnahme der MCMC lediglich durch das statische Herabsetzen des Basistaktes reduziert werden. Bei halbiertem Basistakt und einer Verteilung der Rechenlast auf zwei Prozessorkerne mit jeweils voller Auslastung liegt die Leistungsaufnahme des MCMC (TC267) weiterhin um 2,82 % über der des Single-Core-Systems (TC237).

Bei einer Erhöhung der Energiekosten im niedrigen einstelligen Prozentbereich ist jedoch abzuwägen, ob die zur Verfügung stehenden Leistungsreserven diese Kosten gegebenenfalls rechtfertigen. Die Rechenleistung eines zweiten Prozessorkerns sowie eine leistungsfähigere On-Chip-Peripherie ermöglichen eine vereinfachte Erweiterung des Motor-Controllers. Dies gilt vor allem dann, wenn für die eingesetzten Single-Core-Systeme keine Derivate mit erhöhtem CPU-Takt verfügbar sind. Wird Rechenleistung über die Kapazitäten eines Single-Core-Systems hinaus genutzt, so zeigen die MCMC gegenüber dem Single-Core-System eine deutliche Verbesserung der Energieeffizienz. Dies zeigt sich am Beispiel der Konsolidierung mehrerer Motor-Controller in ein MCMC. Da hier nicht für jeden Motor-Controller ein vollständiges eingebettetes System betrieben werden muss, kann die Leistungsaufnahme für den Betrieb von zwei und drei PMSM um 42,74 % und 56,55 % gesenkt werden. Eine Parallelisierung der Algorithmen ist dabei nicht notwendig, wodurch eine effiziente Integration ermöglicht wird.

Kapitel 6

Parallelisierung geberloser Antriebsregelungen

Zur Parallelisierung von geberlosen feldorientierten Stromregelungen werden diese Anwendungen in Form eines Taskmodells generalisiert. Auf Basis dieses Modells werden die Möglichkeiten zur Parallelisierung aufgezeigt. Wie in Kapitel 5 gezeigt, ist diese Art der Parallelisierung nicht dazu geeignet, um Energie einzusparen. Daher fokussiert hier die Parallelisierung eine Reduktion der Laufzeiten zur Steigerung der Regelfrequenz.

6.1 Taskmodell der geberlosen Stromregelung

Das Taskmodell gliedert die GFOS in mehrere logische Tasks und beschreibt deren Abläufe auf einem Single-Core-System. Hierzu sowie zur Parallelisierung werden zudem die Datenabhängigkeiten zwischen den Tasks beschrieben.

6.1.1 Allgemeine Form des Taskmodells

Als Grundlage zur Gliederung der Regelung in logische Tasks dient das Blockschaltbild der feldorientierten Stromregelung aus Abbildung 2.2. Der dort aufgeführte Rotorlagegeber wird durch einen Algorithmus zur geberlosen Rotorlageberechnung ersetzt. In einer Top-Down-Betrachtung setzt sich eine GFOS zunächst aus zwei Tasks zusammen: die FOS sowie die geberlose Rotorlageberechnung (GRB). Die FOS kann nach den Komponenten des Blockschaltbildes in die Tasks Signalerfassung (SE), Clark-Transformation (CT), Park-Transformation (PT) , Regelkreise (RK), Inverse Park-Transformation (IPT) und PWM (PWM-Generierung) untergliedert werden.

Zum Ausführungszeitpunkt der Signalerfassung wird davon ausgegangen, dass die

entsprechenden Signale bereits durch die Analog-Digital-Converter des MCMC aufgezeichnet wurden. SE-Tasks können diese Daten aus Registern des ADC-Moduls oder nach ihrer Übertragung in einen definierten Speicherbereich mittels einer DMA-Transaktion lesen. Zur Unterscheidung zwischen dem Erfassen von Spannungen und Strömen können mehrere SE-Tasks eingesetzt und mittels $\mathrm{SE}_{\mathrm{u/i}}$ unterschieden werden.

Die GRB fügt sich in Abhängigkeit des verwendeten Verfahrens in die Gliederung der FOS-Tasks ein. Als GRB wird zunächst die Task bezeichnet, die den Algorithmus zur Berechnung der Rotorlage integriert. Zusätzliche Tasks zur Bereitstellung benötigter Eingangsdaten können eine GRB erweitern.

6.1.2 Integration der GRB in die Stromregelung

Grundsätzlich lassen sich Verfahren zur GRB in zeitdiskrete und zeitkontinuierliche Algorithmen gliedern. Aus softwareseitiger Sicht bestimmt diese Eigenschaft, wie die Tasks der FOS zusammen mit der GRB auf einem Mikrocontroller integriert und ausgeführt werden.

6.1.2.1 Zeitdiskrete Rotorlageberechnung

Zeitdiskrete Methoden, beispielsweise DFC oder Algorithmen basierend auf HFCI, ermitteln die elektrische Rotorlage θ direkt aus einer Aufnahme der Messgrößen des Motors. Somit genügt es, die Task der GRB genau dann auszuführen, wenn θ von der FOS benötigt wird. Die Berechnung der Rotorlage geschieht somit synchron zur FOS. Das Sampling der Messgrößen wird zu Beginn einer Iteration k der GFOS durchgeführt.

Abbildung 6.1 zeigt eine typische Integration zeitdiskreter Methoden. Der GRB-Algorithmus gliedert sich in die Abfolge der FOS-Tasks ein. Die Antwortzeit t_{GFOS} einer Iteration der Regelung ergibt sich aus der Laufzeit aller FOS- und GRB-Tasks. Die Laufzeit t_{GRB} umfasst neben der eigentlichen Berechnung der Rotorlage zusätzlich die Zeit, die zur Vorverarbeitung der Eingangssignale benötigt wird. Alle Tasks werden sequenziell im Takt der Stromregelung ausgeführt. Es entsteht eine synchrone Aufführung der FOS- und GRB-Tasks. Zeitliche Relationen zwischen den Tasks sind dadurch in jeder Iteration identisch. Arbeiten FOS- und GRB-Tasks auf gleichartigen Eingangsdaten, genügt durch die synchrone Ausführung eine einmalige Abtastung und Aufbereitung der Signale. Beide Algorithmen können auf die entsprechenden Werte zugreifen. Weiterhin ist durch die synchrone Ausführung aus Sicht der FOS das Alter der berechneten Rotorlage in jeder Iteration identisch. Winkelfeh-

ler zwischen beiden Algorithmen aufgrund unterschiedlicher Messzeitpunkte werden dadurch ausgeschlossen (vergleiche Abschnitt 2.1.2).

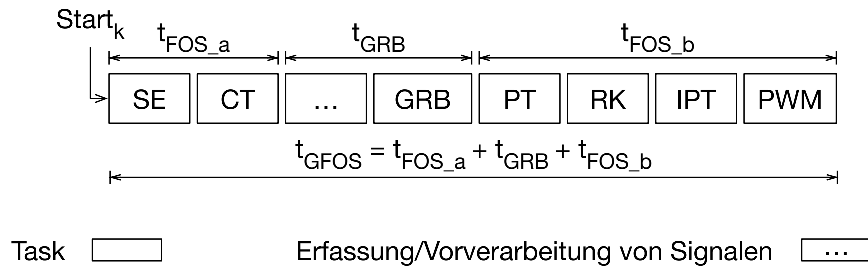


Abbildung 6.1: Zeitdiskrete Ausführung der GRB auf einem Single-Core-System. Die Tasks der geberlosen Rotorlageberechnung reihen sich an geeigneter Stelle zwischen die Tasks der Stromregelung.

6.1.2.2 Zeitkontinuierliche Rotorlageberechnung

Zeitkontinuierliche Methoden, beispielsweise SMO, müssen für eine exakte Berechnung der Rotorlage kontinuierlich ausgeführt werden, um Schätzfehler zu minimieren. In digitalen Systemen ist jedoch nur eine quasi-kontinuierliche Ausführung möglich, wobei die kontinuierliche GRB mit einer festen Frequenz f_{GRB} zu diskreten Zeitpunkten ausgeführt wird. Für jede Iteration wird ein von der FOS unabhängiges Sampling der benötigten Motorsignale und deren Vorverarbeitung durchgeführt.

Softwareseitige entsteht durch die Integration einer zeitkontinuierlichen GRB in die Stromregelung ein nebenläufiges System, das die zeitlich korrekte und unabhängige Ausführung von zwei Taskmengen mit unterschiedlichen Ausführungsfrequenzen realisieren muss. Abbildung 6.2 zeigt eine solche Integration mit einer mehrfachen Unterbrechung der FOS, wodurch sich deren Antwortzeit entsprechend erhöht. Prinzipiell ist auch eine, die GRB unterbrechende Ausführung der FOS möglich. Da die FOS auf präzise Ergebnisse der GRB angewiesen ist, wird die GRB gegenüber der Regelung vorrangig behandelt.

Die Ausführungsfrequenz der GRB kann mit

$$f_{GRB} = n \cdot f_{FOS} \tag{6.1}$$

als ein Vielfaches der Frequenz der FOS angenommen werden. Für $n > 1$ wird die GRB mehrfach innerhalb eines FOS-Zyklus ausgeführt. Für alle ganzzahlige n wird die Regelung in jeder Iteration zu den gleichen Zeitpunkten unterbrochen. Die genauen Zeitpunkte der Unterbrechung ergeben sich durch die zeitliche Relation ΔStart zwischen den jeweiligen Startpunkten der nebenläufig ausgeführten Taskmengen.

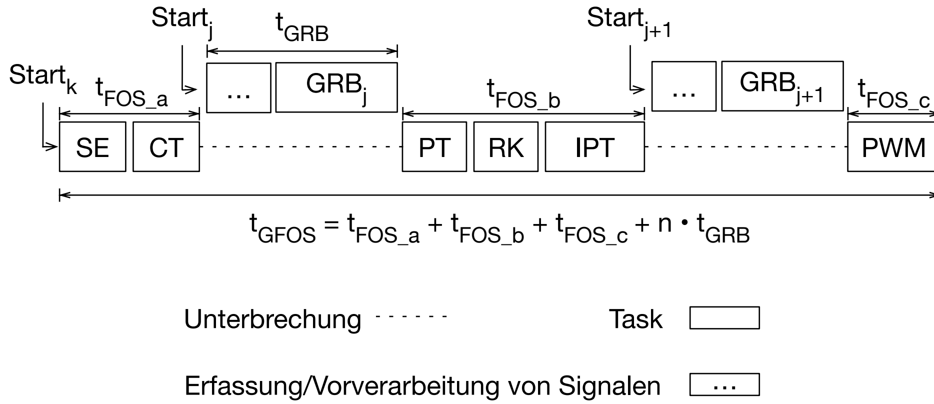


Abbildung 6.2: Nebenläufige Ausführung einer zeitkontinuierlichen GRB auf einem Single-Core-System. Die Tasks der geberlosen Rotorlageberechnung unterbrechen in periodischen Abständen die Stromregelung.

Bei gebrochenen Werten für n wird die FOS in aufeinanderfolgenden Iterationen an verschiedenen Stellen unterbrochen. Es entsteht ein Muster, das sich nach einer bestimmten Anzahl an Iterationen wiederholt. Bezogen auf die FOS ergibt sich die Anzahl dieser Iterationen aus Gleichung 6.2.

$$Iterationen_{FOS} = \frac{kgv(T_{FOS}, T_{GRB})}{T_{FOS}} \tag{6.2}$$

$T_{FOS|GRB} = \frac{1}{f_{FOS|GRB}}$ beschreibt die jeweilige Periodendauer. Abhängigkeit von den Frequenzverhältnissen kann in Relation zu einer gegebenen Task der FOS das Alter der Rotorlage in aufeinanderfolgenden Iterationen der Regelung variieren. Hierdurch kann ein zusätzlicher und variierender Winkelfehler entstehen, der gegebenenfalls berücksichtigt werden muss (vergleiche Abschnitt 2.1.2).

Prinzipiell können auch zeitdiskrete Methoden der Rotorlageberechnung nebenläufig mit einer eigenen Taktfrequenz ausgeführt werden. Dies kann beispielsweise dann notwendig sein, wenn zu Analysezwecken eine höher aufgelöste Rotorlage berechnet werden muss. Dies würde jedoch die Antwortzeit der FOS entsprechend erhöhen. Weiterhin kann die synchrone Ausführung der FOS und GRB aufgelöst werden, wodurch sich das Verhalten der Regelung gegenüber der zeitdiskreten Ausführung ändern kann.

6.1.3 Datenfluss des Taskmodells

Die Reihenfolge, in der die Tasks ausgeführt werden, wird durch den Kontroll- und Datenfluss zwischen den Funktionsblöcken der Regelung definiert (vergleiche Abbildung 2.2). Hier existieren keine bedingten Verzweigungen, sodass eine in jeder

Iteration der Regelung identische Abfolge der Tasks stattfindet. Die konkrete Reihenfolge ist vollständig über den Datenfluss definiert, der somit die Abhängigkeiten zwischen Tasks beschreibt. Hierbei ist eine Task B von Task A abhängig, wenn Task B Daten verarbeitet, die von Task A bereitgestellt werden.

Die Datenflussgraphen in Abbildung 6.3 zeigen die möglichen Abhängigkeiten. Jede eingehende Kante im Datenflussgraph beschreibt die Datenabhängigkeit einer Task von der Vorangehenden. Die GRB kann auf unterschiedlichen Repräsentationen der erfassten Signalwerte beruhen und gliedert sich an entsprechender Stelle in die Abläufe der Stromregelung ein. Abbildung 6.3a zeigt die entstehenden Integrationsmöglichkeiten der GRB in die FOS unter direkter Verwendung der erfassten Messgrößen sowie ihrer Transformation in α-β-Komponenten.

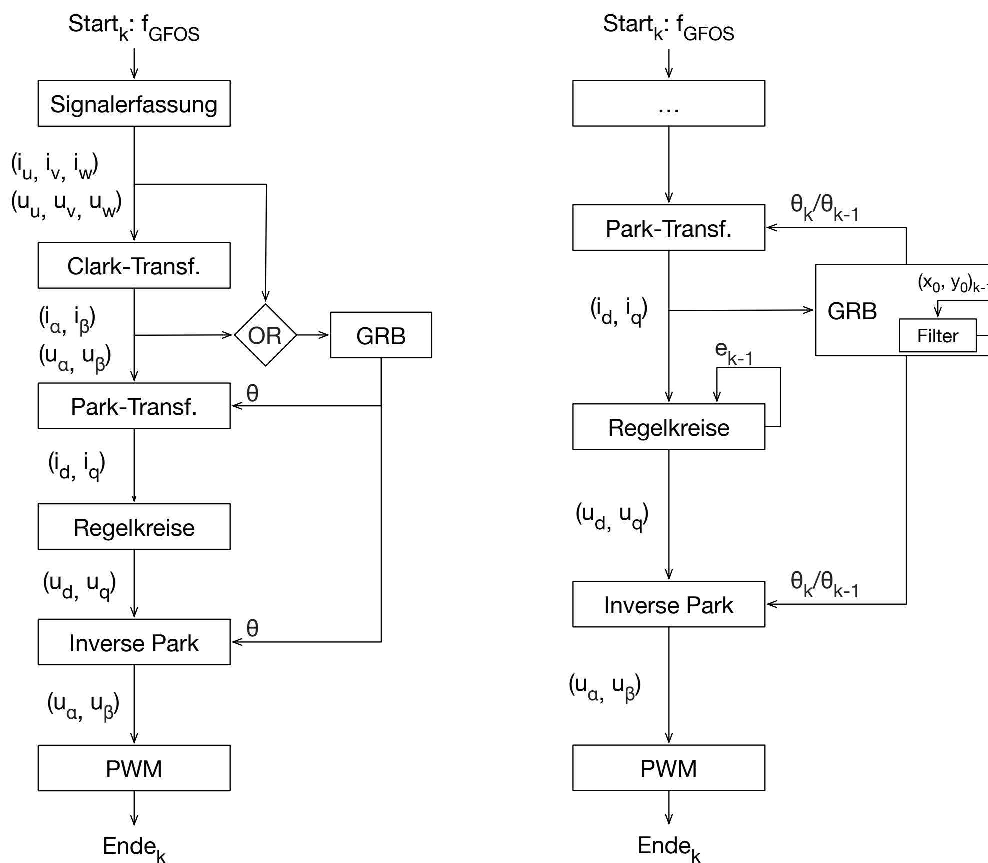


Abbildung 6.3: Datenflussgraphen der geberlosen feldorientierten Stromregelung. Es wird zwischen unterschiedlichen Integrationspunkten der geberlosen Rotorlageberechnung innerhalb eines Regelzyklus sowie zwischen Regelzyklen unterschieden.

Datenabhängigkeiten können ebenfalls zwischen den Iterationen $k-1$ und k be-

stehen, sodass eine Task von Daten aus dem vorangegangenen Zyklus abhängig ist. Abbildung 6.3b verdeutlicht dies durch die Abhängigkeit der Park-Transformation von der GRB-Task. Letztere basiert im Zyklus k auf den im gleichen Zyklus berechneten d-q-Stromkomponenten der Park-Transformation. Die Transformation selbst setzt einen bekannten Wert für die Rotorlage voraus. Zur Durchführung der Transformation im k-ten Regelzyklus wird auf den im vorangegangenen Zyklus $k-1$ berechneten Wert zurückgegriffen. Eine GRB basierend auf d-q-Komponenten findet sich unter anderem bei Methoden basierend auf HFCI. Weitere zyklusübergreifende Abhängigkeiten können sich in den Regelkreisen zur Berechnung der Integral- und Differentialanteile und der Implementierung von Filtern finden. Abbildung 6.3b zeigt Letzteres am Beispiel eines Filters zweiter Ordnung bei der Verwendung von x_1 und y_1, die den Werten x_0 und y_0 aus Iteration $k-1$ entsprechen.

Bei der zeitkontinuierlichen Rotorlageberechnung entsteht durch die nebenläufige Ausführung ein zweiter Datenfluss (siehe Abbildung 6.4). Es besteht kein direkter Zusammenhang zwischen der Ausführung eines Zyklus k der FOS und der eines Zyklus j der GRB. Dementsprechend kann keine direkte Datenabhängigkeit beider Algorithmen bezüglich θ hergestellt werden. Die Regelung muss davon ausgehen, dass zu jedem Zeitpunkt ein Wert θ vorhanden ist. Dieser Wert wird der Regelung über einen persistenten Puffer zur Verfügung gestellt. Es entsteht eine Datenabhängigkeit der FOS zu diesem Puffer. Die Abhängigkeit kann als erfüllt betrachtet werden, sobald der Puffer mit einem gültigen Wert durch die GRB gefüllt wurde. Ein entsprechender Puffer hebt auch eine Datenabhängigkeit zwischen den Regelzyklen $k-1$ und k auf.

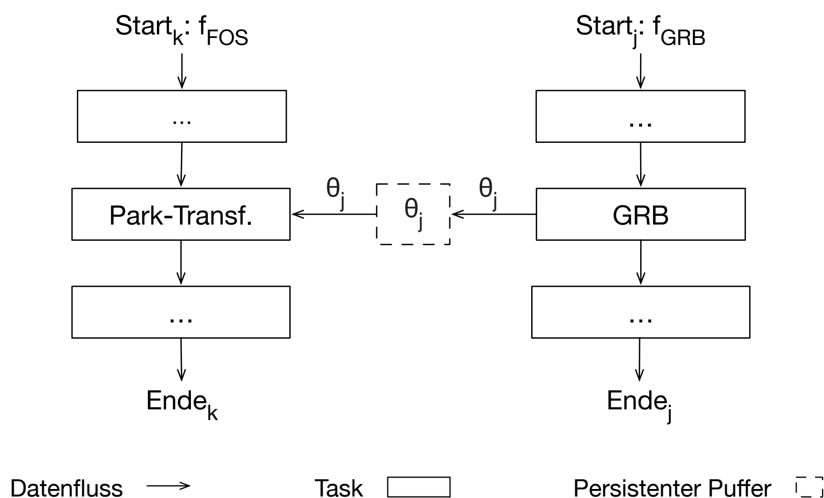


Abbildung 6.4: Datenflussgraphen mit zeitkontinuierlichen Rotorlageberechnung

6.2 Parallelität im Taskmodell

Die Möglichkeit, eine Anwendung zu parallelisieren, setzt das Vorhandensein von parallelisierbaren Strukturen voraus, die in unterschiedlichen Formen in einer Anwendung enthalten sein können. Diese Formen werden nach Kontrollfluss-, Daten- und Pipelineparallelität unterschieden [153]. Taskparallelität liegt dann vor, wenn eine sequenziell ausgeführte Anwendung in mehrere voneinander unabhängig ausführbare Sequenzen aufgeteilt werden kann. Bei Datenparallelität, wie sie auch von GPUs ausgenutzt wird, führen Anwendungen Operationen beziehungsweise einzelne Instruktionen parallel auf unterschiedlichen Datenströmen aus. Pipelineparallelität ist gegeben, wenn aufeinanderfolgende Iterationen einer Anwendung parallel ausgeführt werden können.

Anwendungen zur Regelung mechatronischer Prozesse bestehen überwiegend aus einer Kette aufeinander aufbauender Prozessschritte und geringen Datenmengen, die gleichzeitig verarbeitet werden müssen. Hierdurch ist nur ein moderates Maß an Task- und Datenparallelität zu erwarten, sodass nur ein geringer Anteil der Operationen parallel ausgeführt werden kann. Es besteht die Gefahr, dass die Kosten der Parallelisierung die Gewinne übersteigen. Durch die iterative Ausführung der Algorithmen kann eine Pipelineparallelität angenommen werden, die potenziell zur Erhöhung von Regelfrequenz genutzt werden kann.

Auf Basis des Taskmodells wird allgemein die Parallelisierbarkeit entsprechender Regelungen beschrieben. Dies geschieht auf Basis der in den Abbildungen 6.3 und 6.4 dargestellten Datenströme. Die feldorientierte Stromregelung und die geberlose Rotorlageberechnung werden zunächst als jeweils eine Task betrachtet. Anschließend wird die Regelung in weitere Tasks gegliedert und die Möglichkeiten ihrer parallelen Ausführung in Kombination mit der GRB beschrieben.

6.2.1 Parallele Ausführung der GRB

Grundsätzlich können Tasks dann parallel zueinander ausgeführt werden, wenn ihre Ausführung zu einem gegebenen Zeitpunkt von keinen Kontrollflussentscheidungen abhängig ist und keine unaufgelösten Datenabhängigkeiten bestehen [53]. Wie bereits beschrieben ist die Abfolge der Tasks unabhängig von Kontrollflussentscheidungen, sodass lediglich die Datenabhängigkeiten darüber entscheiden, ob und welche Tasks parallel ausgeführt werden können.

Bei der zeitkontinuierlichen GRB lässt die nebenläufige Ausführung bereits darauf schließen, dass die FOS- und GRB-Task keine Datenabhängigkeiten zueinander

aufweisen und damit auch parallel ausgeführt werden können (siehe Abbildung 6.5). Durch die parallele Ausführung wird die Antwortzeit der FOS nicht weiter durch die Berechnung der Rotorlage verzögert. Somit können beide Taskmengen die volle Rechenleistung des jeweiligen Prozessorkerns in Anspruch nehmen. In diesem Fall können maximale Werte für die Ausführungsfrequenzen f_{FOS} und f_{GRB} auf einem gegebenen System erreicht werden. Die Laufzeiten t_{FOS} und t_{GRB} entsprechen in diesem Fall den Periodendauern T_{FOS} und T_{GRB}. Die Implementierung erfolgt entsprechend Abschnitt 4.2.1.1 als jeweils eine ISR pro Prozessorkern. Jede ISR wird dem entsprechenden Kern statisch zugewiesen und periodisch mit der Frequenz des jeweiligen Algorithmus ausgeführt.

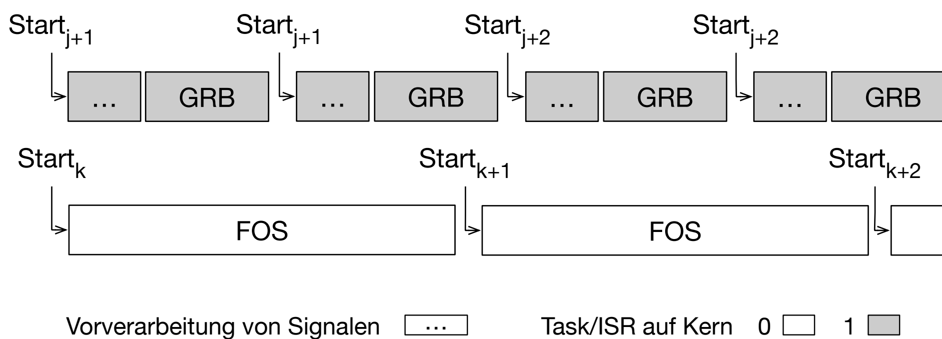


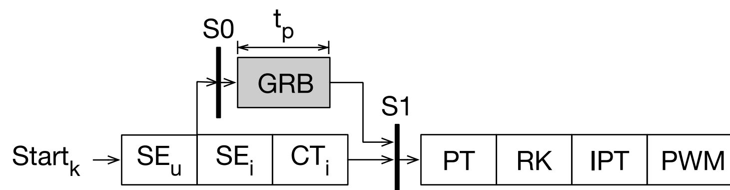
Abbildung 6.5: Parallele Ausführung mit zeitkontinuierlicher GRB. Die Tasks der geberlosen Rotorlageberechnung werden auf einem zweiten Kern parallel zu den Tasks der Stromregelung ausgeführt.

Wird eine zeitdiskrete GRB parallelisiert, muss sichergestellt sein, dass die synchrone Ausführung zwischen Rotorlageberechnung und Stromregelung beibehalten bleibt. Damit entspricht das Verhalten der Anwendung weiterhin dem bei einer seriellen Ausführung. Dies wird unter Beachtung der Datenabhängigkeiten realisiert. Die Rotorlageberechnung kann nach dem Auflösen ihrer Datenabhängigkeiten parallel zu FOS-Tasks ausgeführt werden. Sie muss jedoch beendet sein, bevor das Ergebnis von der Stromregelung verwendet wird. In Abbildung 6.6a wird angenommen, dass die GRB auf Daten der $\mathrm{SE_u}$-Task basiert. Die Datenabhängigkeit von GRB wird durch das Ausführen von $\mathrm{SE_u}$ aufgelöst. Eine Verkürzung der Laufzeit kann erreicht werden, wenn die GRB aus der Taskfolge der FOS herausgenommen und parallel dazu ausgeführt wird. Das Ergebnis muss zum Zeitpunkt der Ausführung der PT-Task bereitstehen.

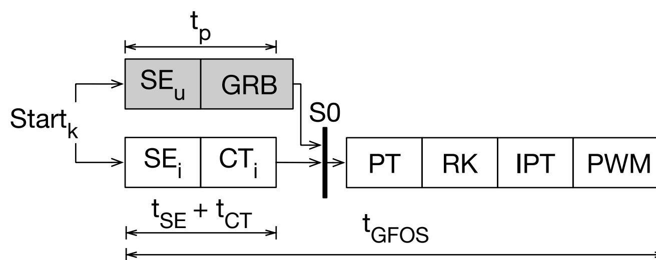
Diese Parallelisierung kann auf unterschiedliche Arten implementiert werden. Sie unterscheiden sich in dem Kern, der die Datenabhängigkeit der GRB auflöst. Abbildung 6.6b zeigt das Auflösen der Datenabhängigkeit der GRB durch Kern 0. Dieser

stellt der GRB auf Kern 1 die Daten durch eine entsprechende Synchronisation bereit. Die Synchronisation löst gleichzeitig das Ausführen der Rotorlageberechnung aus. Alternativ kann eine zusätzliche Instanz von SE_u auf Kern 1 integriert und ausgeführt werden (siehe Abbildung 6.6c). Beide SE-Tasks können auf die durch das ADC-Modul bereitgestellten Daten zugreifen. Ein zusätzliches Sampling der Motorsignale ist nicht notwendig. Eine Synchronisation mit Kern 0 zur Übertragung der Eingangsdaten kann so umgangen werden. Die Ausführung der Tasks auf Kern 1 kann zeitgesteuert und synchron zu denen auf Kern 0 geschehen. Für beide Varianten findet ein synchronisierter Übergang von paralleler zu sequenzieller Ausführung statt. Hier werden die Ergebnisse der GRB zur weiteren Ausführung der FOS-Tasks an Kern 0 übertragen. In Abhängigkeit der Eingangsdaten der GRB kann auch eine Kombination beider Varianten Anwendung finden. Prinzipiell ist eine Integration zu wählen, in der eine möglichst große Laufzeit von Kern 0 auf Kern 1 verschoben wird und die entsprechenden Tasks auf Kern 1 parallel zu denen auf Kern 0 ausgeführt werden.

(a) Serielle Ausführung der GRB und FOS-Tasks



(b) Auflösen der Datenabhängigkeiten von GRB seitens Kern 0



(c) Unabhängige Berechnung der Eingangsdaten von GRB

Abbildung 6.6: Möglichkeiten der parallelen Ausführung einer zeitdiskreten GRB. Die Eingangsdaten der geberlosen Rotorlageberechnung werden von unterschiedlichen Kernen berechnet.

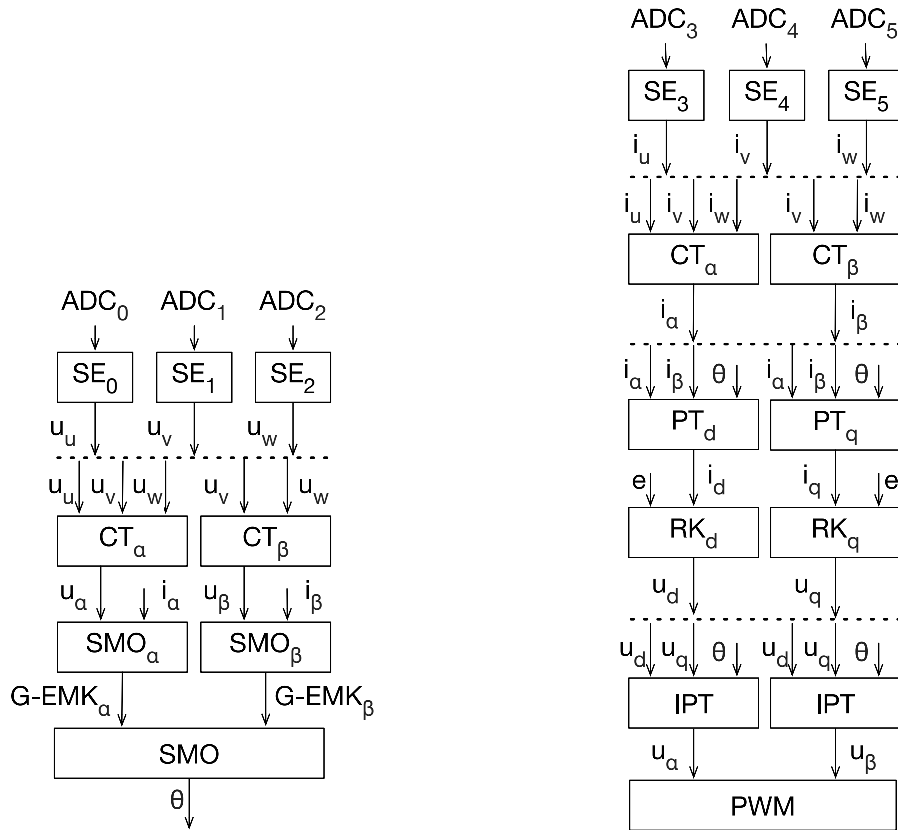
Durch die unterschiedlichen Ansätze können unterschiedliche große Anteile der

GRB parallel zu den Tasks auf Kern 0 ausgeführt werden. In den Abbildungen 6.6b und 6.6c sind die entsprechenden Laufzeiten mit t_p gekennzeichnet. Stellt die GRB die Rotorlage für die PT-Task in der gleichen Iteration bereit, wird die maximale Zeit einer parallelen Ausführung durch die Laufzeiten der Tasks $\mathrm{SE_i}$ und $\mathrm{CT_i}$ mit $t_{SE_i} + t_{CT_i}$ auf Kern 0 beschränkt. Dies entspricht in diesem Fall der maximal erreichbaren Laufzeitersparnis durch eine Parallelisierung. Übersteigt die Laufzeit auf Kern 1 $t_{SE_i} + t_{CT_i}$, so entsteht für Kern 0 eine Wartezeit. Bedingt die GRB-Task die zusätzliche Ausführung von $\mathrm{SE_u}$ und $\mathrm{CT_u}$ (entsprechend Abbildung 6.1), so kann die Ausführung dieser Tasks parallel zu der entsprechenden Verarbeitung der Ströme stattfinden.

Basiert die GRB auf d-q-Komponenten, wird die Rotorlage aus Iteration $k-1$ verwendet, um θ_k in Iteration k zu berechnen. Wird θ_k nicht in Iteration k verwendet, so ist es ausreichend, wenn die Berechnungen von GRB vor dem Ende von k abgeschlossen sind. Für die parallele Berechnung der GRB bedeutet dies, dass sie parallel zur gesamten Laufzeit der FOS-Tasks in Iteration k ausgeführt werden kann. Der maximale Wert für t_p entspricht in diesem Fall t_{GFOS}. Wird θ_k von einer Task in Iteration k benutzt, so muss die Berechnung der Rotorlage vor dem Beginn der jeweiligen Task beendet sein. Dementsprechend wird der mögliche Wert für t_p verringert.

6.2.2 Datenströme innerhalb der Tasks

In Abbildung 6.3 wurden die Datenströme zwischen den einzelnen Tasks beschrieben. Auch innerhalb der Tasks existieren Datenflüsse, die zur Parallelisierung der Tasks herangezogen werden können. Abbildung 6.7 zeigt dies am Beispiel der Integration eines SMO als GRB. Ausgangspunkte sind die erfassten Daten auf mehreren ADC-Kanälen, wodurch Verarbeitungsketten basierend auf gemessenen Spannungen und Strömen unterschieden werden können. Abbildung 6.7a zeigt die Datenflüsse der Spannungen, Abbildung 6.7b die der Ströme. In Abbildung 6.7a fließen die von der Signalerfassung erzeugten Daten bei der Clark-Transformation zusammen. Durch die Berechnung der α-β-Komponenten werden hier zwei neue Datenströme erzeugt. Als orthogonale Größen sind beide Komponenten unabhängig voneinander und können parallel berechnet werden. Gleichermaßen können die d-q-Komponenten der Park-Transformation nach der Zusammenführung von i_α, i_β und θ in PT_d und PT_q parallel berechnet werden. Die Implementierung des SMO führt voneinander unabhängige Berechnungen auf den α- und β-Komponenten aus. Hierdurch kann der Algorithmus in die parallel ausführbaren Tasks SMO_α und SMO_β zerlegt werden. Beide Daten-



(a) Verarbeitungskette der Spannungen

(b) Verarbeitungskette der Ströme

Datenfluss ⟶ Daten zusammenführen - - - - Task ▭

Abbildung 6.7: Datenflussgraphen parallelisierter Verarbeitungsketten am Beispiel eines Sliding-Mode Observer

ströme werden zur endgültigen Ermittlung von θ in SMO_θ zusammengeführt. Bei dem standardmäßigen Einsatz von PI(D)-Reglern zur Stromregelung findet eine separate und damit parallelisierbare Regelung der d-q-Stromkomponenten statt (siehe Abbildung 6.7b). Die Ergebnisse der Regelkreise werden zur parallelen Rücktransformation und anschließenden PWM-Modulation zusammengeführt.

6.2.3 Parallelisierung von Rechenoperationen

In der vorangegangenen Parallelisierung wurden voneinander unabhängige Berechnungsfolgen innerhalb einer Task in mehrere Teiltasks aufgeteilt. Wird der Abstraktionsgrad weiter reduziert, so können einzelne Berechnungen für eine Parallelisierung betrachtet werden. Typische parallelisierbare Berechnungen sind zusammenhängende Multiplikations- und Additionsoperationen der Form

$$z = x_1 \cdot y_1 \pm x_2 \cdot y_2 \pm x_3 \cdot y_3 \pm ... \tag{6.3}$$

Die jeweiligen Multiplikationen sind unabhängig voneinander und können parallel durchgeführt werden. Die Addition der Zwischenergebnisse erfolgt nach Abschluss der Multiplikationen. Regelmäßige Einsatzgebiete dieser Operationen sind unter anderem digitale Filter, beispielsweise Bandpassfilter zur Trennung hochfrequenter Stromkomponenten von Komponenten der Grundschwingung bei HFCI-basierten Verfahren [102] oder Matrixoperationen zur Berechnung von Kalman-Filtern [114].

Die Faktorisierung und parallele Ausführung solcher Rechenoperationen ist bei hardwarebasierten Implementierungen mittels FPGA ein gängiger Ansatz zur Optimierung der Rechenzeit. Bei genügend zur Verfügung stehender Chipfläche ist die Anzahl parallel ausführbarer Operationen quasi unbeschränkt. Bei der softwarebasierten Implementierung können Mikrocontroller, insbesondere DSPs, optimierte Instruktionen zur gleichzeitigen Multiplikation und Addition zur Verfügung stellen. Bei 32-Bit-Architekturen sind dies beispielsweise sogenannte Packed Multiply-Accumulate Instruktionen, die schnell (2-3 CPU-Takte bei TC1.6P [77]) zwei 16 Bit Wertepaare multiplizieren und beide Teilergebnisse addieren [73].

Alternativ ist eine Aufteilung der Berechnungen auf mehrere Prozessorkerne möglich (siehe Abbildung 6.8). Die Anzahl der zwischen den Kernen zu übertragenden Daten ist vom jeweiligen Typ der Berechnung abhängig. Bei der Implementierung digitaler Filter sind die Werte für x typischerweise konstant und können statisch auf den Kernen vorgehalten werden. Weiterhin entsprechen hier die Werte für y einer Zeitreihe, bei der y_2 den Wert von y_1 aus der vorangegangenen Iteration beinhaltet. In diesem Fall müsste bei dem Übergang zur parallelen Berechnung lediglich der y_3 an Kern 1 übertragen werden, da y_4 dem Wert von y_3 aus der vorangegangenen Iteration entspricht und bereits auf Kern 1 vorhanden ist. Wird hingegen eine Matrixmultiplikation parallelisiert, müssen die Werte der entsprechenden Teilmatrizen übertragen werden.

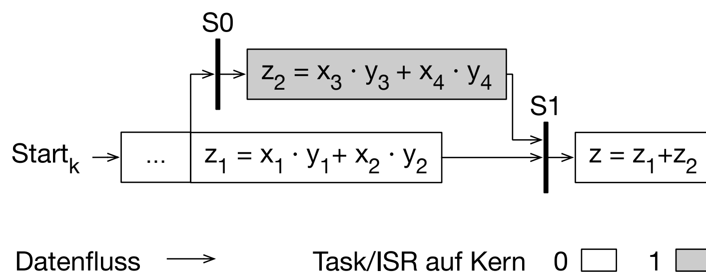


Abbildung 6.8: Parallelisierung von Multiplikationen. Die Prozessorkerne berechnen jeweils ein Teilergebnis des Ausdrucks.

6.2.4 Parallelisierung durch Pipelining

Wie in Abschnitt 4.2.1.2 beschrieben, können durch eine Pipeline aufeinanderfolgende Iterationen eines Algorithmus parallel ausgeführt werden. Hierdurch wird zwar die Antwortzeit der Berechnungen einer Iteration nicht verringert, jedoch können Signale mit einer höheren Frequenz abgetastet und entsprechende Rechenergebnisse mit einer höheren Auflösung bereitgestellt werden.

Für die Stromregelung kommt diese Art der Parallelisierung nur bedingt infrage. Die Parallelisierung der Regelung resultiert in diesem Fall in separaten Regelkreisen, welche auf die zu regelnde Größe wirken. Auch wenn in jeder Stufe identische Regelparameter eingesetzt werden, kann die Parallelisierung das Verhalten der Regelung beeinflussen. Dementsprechend muss die Auslegung der Regelung bei Anwendung einer Pipeline angepasst werden. Ist dies der Fall, kann durch die Pipeline eine höhere Regelfrequenz erreicht werden.

Bei der geberlosen Rotorlageberechnung gilt entsprechendes für Algorithmen, die aktiv Testsignale in den Antrieb einspeisen oder die Generierung der PWM zur Gewinnung benötigter Informationen beeinflussen. Werden lediglich Motorsignale ausgewertet, wie es beispielsweise bei einem SMO der Fall ist, kann eine Pipeline prinzipiell eingesetzt werden. Für den SMO bedeutet dies, dass er mit einer höheren Frequenz ausgeführt und seine Ergebnisse gegebenenfalls verbessert werden können. Allgemein kann eine Pipeline bei ihrer Anwendung auf einen GRB-Algorithmus die Auflösung mit der Lageinformationen bereitgestellt werden erhöhen.

6.3 Evaluation der Parallelisierung von geberlosen Stromregelungen

In den vorangegangenen Abschnitten wurden unterschiedliche Möglichkeiten gezeigt, um die grundlegenden Strukturen einer geberlosen feldorientieren Stromregelungen parallel auf mehreren Kernen auszuführen. Die Laufzeiten der einzelnen Tasks sowie die Kosten der Synchronisation entscheiden, ob und in welchem Maß eine Parallelisierung der gezeigten Strukturen gewinnbringend genutzt werden kann. Um die Parallelisierung zu beurteilen, werden Tasks der GFOS nach den beschriebenen Möglichkeiten verteilt und die Änderungen der Laufzeiten betrachtet. Die Messungen werden entsprechend dem Vorgehen aus Abschnitt 4.3 auf dem MCMC durchgeführt.

Laufzeiten sind von konkreten Anwendungen und deren Implementierung abhän-

gig. Die hier betrachteten Werte sind für eine beispielhafte Implementierung auf dem MCMC gültig. Die gemessenen Daten werden als Größenordnung für die Laufzeiten der jeweiligen Berechnungen betrachtet. Konkrete Laufzeiten anderer Systeme können von den hier gemessenen abweichen. Unter folgenden Gesichtspunkten kann angenommen werden, dass sich die Größenverhältnisse bei vergleichbarer Implementierung nicht signifikant ändern.

In der Signalerfassung werden Rohwerte aus den Registern des ADC-Moduls gelesen und in physikalische Strom- und Spannungsgrößen konvertiert. Laufzeitunterschiede können sich durch optionale Methoden zur weiteren Vorverarbeitung der Daten, beispielsweise durch den Einsatz von Filtern, ergeben. Die Transformationen folgen festgelegten Berechnungen auf Basis trigonometrischer Funktionen. Hier können die Verwendung von Lookup-Tabellen beziehungsweise die Berechnung von Sinus- und Kosinusfunktionen zur Laufzeit die benötigte Rechenzeit beeinflussen. Auch P-, PI- und PID-Regelkreise folgen standardmäßigen Berechnungen, deren Laufzeiten seitens der Implementierung nur geringfügig variieren. Die Erzeugung der PWM-Signale geschieht standardmäßig durch eine Raumzeigermodulation auf Basis der rücktransformierten Spannungen. Hierbei erfolgt die Berechnung der konkreten Schaltmuster der PWM-Signale auf Basis einfacher arithmetischer Operationen. Entsprechend einer Vorverarbeitung in der Signalerfassung können auch hier Verarbeitungsschritte, beispielsweise zur Skalierung oder Limitierung der PWM-Werte, die Laufzeit beeinflussen.

Die Implementierungen der Algorithmen zur geberlosen Rotorlagebestimmung entsprechen für SMO und HFCI den Modellen aus [43], für DFC dem Modell aus [159]. Auch die hier gemessenen Laufzeiten dienen zur Abschätzung der Größenordnung entsprechender Algorithmen und können zwischen unterschiedlichen Ausprägungen und Implementierungen der Verfahren variieren.

6.3.1 Parallele Ausführung mit zeitkontinuierlicher GRB

Die parallele Ausführung einer zeitkontinuierlichen GRB zur Stromregelung entspricht aus Sicht der softwareseitigen Integration einer nebenläufigen Ausführung auf nur einem Prozessorkern. Eine Synchronisation der Abläufe beider Kerne ist nicht notwendig. Am Beispiel einer GFOS unter Verwendung des SMO-Algorithmus fasst Tabelle 6.1 die gemessenen Laufzeiten einer Beispielimplementierung zusammen. In der nebenläufigen Implementierung wird die GRB mit gleicher Frequenz wie die FOS ausgeführt. Die Stromregelung wird somit in jeder Iteration einmal unterbrochen. Insgesamt ergibt sich für die serielle Ausführung eine Lauf- beziehungsweise

Antwortzeit von 3622 CPU-Takten (*1*). Die Rotorlageberechnung basiert hier auf Spannungs- und Stromkomponenten im α-β-Koordinatensystem. Wird die GRB auf Kern 1 ausgelagert, so muss dieser neben der Rotorlageberechnung auch die Bereitstellung der genannten Stromkomponenten durchführen. Dies benötigt eine Laufzeit von 1876 Takten (*2*). Zur Stromregelung benötigt auch Kern 0 transformierte Stromkomponenten und führt zu deren Berechnung die entsprechenden Tasks aus. Durch das Auslagern der GRB erfolgt für Kern 0 eine Verringerung der Antwortzeit um 1666 (*2*-*3*) auf 1956 CPU-Takte (*4*). Die entspricht einer Reduktion der Laufzeit um 46 %.

Tabelle 6.1: Laufzeiten der GFOS-Tasks mit SMO

	Tasks	*∅ Laufzeit [CPU-Takte]*
(1)	*GFOS-SMO seriell*	3622
(2)	SE_i+CT_i+SE_u+CT_u+ *SMO*	1876
(3)	SE_i+CT_i	210
(4)	*GFOS-SMO parallel*	1956

Bei einer CPU-Taktrate von 100 MHz ergibt sich für die Ausführung der GFOS auf einem Prozessorkern, und damit auch für die Ausführung des SMO-Algorithmus, eine maximale Regelfrequenz von 27 kHz. Durch das separate Ausführen von SMO auf Kern 1 kann für die Rotorlageberechnung eine Frequenz von 53 kHz erreicht werden. Für die GRB ist somit eine Steigerung der Frequenz von 96,3 % möglich. Durch das unterbrechungsfreie Ausführen der FOS kann eine maximale Steigerung der Regelfrequenz von 88,9 % auf 51 kHz erreicht werden.

6.3.2 Parallele Ausführung mit zeitdiskreter GRB

In Abschnitt 6.2.1 wird die parallele Ausführung der zeitdiskreten GRB beschrieben. Kern 0 führt hierbei die FOS und Kern 1 die GRB aus (vergleiche Abbildung 6.6c).

6.3.2.1 Laufzeiten unter Anwendung des DFC-Algorithmus

Am Beispiel des DFC-Algorithmus zeigt Tabelle 6.2 die auf dem MCMC gemessenen Laufzeiten. Insgesamt benötigt die GFOS zur Berechnung einer Iteration eine Laufzeit von 2586 CPU-Takten (*1*). Die Ausführung von DFC basiert auf den Daten von SE_u und muss vor der Ausführung der PT-Task beendet sein. Für die Berechnung der Rotorlage wird insgesamt eine Laufzeit von 810 Takten benötigt (*2*). Bis zur Ausführung der PT-Task vergehen insgesamt 1020 CPU-Takte (*2*+*3*).

Tabelle 6.2: Laufzeiten der GFOS-Tasks mit DFC

	Tasks	*∅ Laufzeit [CPU-Takte]*
(1)	*GFOS-DFC seriell*	2586
(2)	SE_u+*DFC*	810
(3)	$SE_i + CT_i$	210
(4)	*GFOS-DFC parallel*	2486

Wird DFC auf Kern 1 ausgelagert, beträgt die gemessene Laufzeit 920 CPU-Takte. Die Differenz zu (*2*) beschreibt die Kosten der Parallelisierung von 110 Takten. Während Kern 1 die Rotorlage berechnet, führt Kern 0 mit einer Laufzeit von 210 Takten die SE_i- und CT_i-Tasks aus (*3*) und wartet nach deren Ende auf die Synchronisation mit Kern 1. Dies geschieht nach 920 Takten, sodass nach dieser Laufzeit die PT-Task ausgeführt werden kann. Somit stehen 1020 CPU-Takte bei serieller Ausführung 920 Takten bei paralleler Ausführung gegenüber. Die Parallelisierung verringert somit die Laufzeit der GFOS um 100 auf 2486 CPU-Takte (*4*). Die Hinzunahme eines zweiten Kerns kann somit die Laufzeit der GFOS um 3,87 % verringern. Bei einer CPU-Taktrate von 100 MHz kann bei serieller Ausführung eine Regelfrequenz der GFOS von 38,67 kHz realisiert werden. Durch die Parallelisierung ist eine Erhöhung der Frequenz um 4,02 % auf 40,23 kHz möglich.

6.3.2.2 Laufzeiten unter Anwendung des HFCI-Algorithmus

Tabelle 6.3 zeigt die gemessenen Laufzeiten bei Verwendung des HFCI-Algorithmus. Insgesamt beträgt die Laufzeit der GFOS 3454 CPU-Takte (*1*). Davon entfallen 1290 Takte auf die Durchführung des HFCI-Algorithmus inklusive der hierzu notwendigen Transformation der hochfrequenten Stromanteile durch die Task PT_{HF} (*2*). Hierzu wird die Rotorlage aus der vorangegangenen Iteration verwendet.

Tabelle 6.3: Laufzeiten der GFOS-Tasks mit HFCI

	Tasks	*∅ Laufzeit [CPU-Takte]*
(1)	*GFOS-HFCI seriell*	3454
(2)	PT_{HF}+*HFCI*	1290
(3)	*FOS-Tasks*	2164
(4)	$SE_i + CT_i + PT_{HF}$+*HFCI*	1526
(5)	*GFOS-HFCI parallel*	2164

Da hier die Ergebnisse der GRB erst zu Beginn der folgenden Iteration bereit-

stehen müssen, kann die Ausführung von HFCI parallel zu allen Tasks der FOS stattfinden. Diese benötigen eine Laufzeit von 2164 Takten (*3*). Die Auslagerung von HFCI auf Kern 1 bedingt die dortige Ausführung der SE_i- und CT_i-Tasks. Die Laufzeit auf Kern 1 beträgt somit 1526 CPU-Takte (*4*) und 1626 Takte inklusive der Synchronisationskosten. Dies ist insgesamt geringer als die von den FOS-Tasks benötigte Laufzeit (*3*). HFCI kann dementsprechend vollständig parallel zu den Berechnungen der Stromregelung auf Kern 0 ausgeführt werden. Die Laufzeit auf Kern 0 sinkt entsprechend um 1290 CPU-Takte auf 2164 Takte (*5*). Dies entspricht einer Verringerung der Laufzeit um 37,34 %. Bei einer CPU-Taktrate von 100 MHz kann bei serieller Ausführung eine Regelfrequenz der GFOS von 29,96 kHz realisiert werden. Durch die Parallelisierung ist eine Erhöhung der Regelfrequenz um 59,57 % auf 46,21 kHz möglich.

6.3.3 Parallelisierung einzelner Tasks nach Datenströmen

In Abschnitt 6.2.2 wurde die Parallelisierung nach Datenströmen beschrieben. Es wurde gezeigt, welche Tasks der FOS prinzipiell aufgeteilt und parallelisiert ausgeführt werden können. Tabelle 6.4 zeigt die gemessenen Laufzeiten und Kosten der Parallelisierung dieser Tasks auf dem MCMC. Die Laufzeiten der seriellen Ausführung sind in die Komponenten aufgeteilt, die parallel ausgeführt werden sollen.

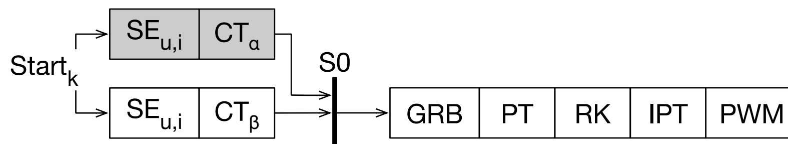
Tabelle 6.4: Parallelisieren der GFOS-Tasks nach Datenströmen auf zwei Kerne. Die Laufzeit bei einer Parallelisierung ergibt sich aus der Summe der unterstrichenen Werte.

		∅ *Laufzeit [CPU-Takte]*			
	Task	*Seriell*	Kosten	*Parallel*	*Daten [m]*
(1)	$SE_u + SE_i$	192+165=357	179	371	3
(2)	$CT_\alpha + CT_\beta$	24+24=48	149	173	1
(3)	$PT_d + PT_q$	42+42=48	151	193	1
(4)	$RK_d + RK_q$	358+358=716	145	503	1
(5)	$SMO_\alpha + SMO_\beta + SMO_\theta$	364+364+662=1390	148	1174	1

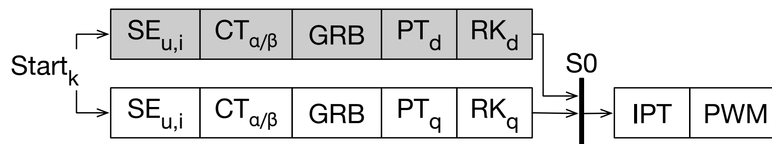
Bei der Signalerfassung wird die Bearbeitung der Spannungen (SE_u-Task) und der Ströme (SE_i-Task) auf separaten Kernen durchgeführt. Die Kerne werden synchron mit dem Takt der FOS gestartet und lesen die benötigten Messwerte aus den Registern des ADC-Moduls. Nach der Durchführung der SE-Tasks wird nach der Übertragung der Daten von Kern 1 nach Kern 0 eine serielle Ausführung der FOS

auf Kern 0 fortgesetzt. Es werden insgesamt drei Datenworte übertragen. Bei serieller Ausführung verteilt sich die Laufzeit mit 192 CPU-Takten auf die Erfassung der Spannungen und mit 165 Takten auf die Ströme (*1*). Durch die parallele Ausführung beider Tasks können maximal 165 Takte eingespart werden. Zuzüglich der gemessenen Synchronisationskosten von 179 CPU-Takten beträgt die Laufzeit bei paralleler Ausführung 371 Takte. Die Laufzeit erhöht sich somit um 14 CPU-Takte.

Die Clark-Transformation erzeugt zwei voneinander unabhängige Komponenten. Bei der parallelen Ausführung der Transformation berechnet jeder Kern eine dieser Komponenten und führt auch die dazu notwendige Signalerfassung durch (siehe Abbildung 6.9a). Die Berechnung der α- und β-Komponenten benötigt auf dem MCMC jeweils eine Laufzeit von 24 CPU-Takten (*2*). Einer maximal erreichbaren Laufzeitreduktion von 24 Takten stehen Kosten in Höhe von 149 CPU-Takten gegenüber (*2*). Die Laufzeit bei paralleler Ausführung der CT-Task steigt somit um 125 Takte. Bei der parallelisierten Park-Transformation führen beide Kerne eine vollständige CT-Task aus (siehe Abbildung 6.9b). Auch hier überwiegen die Kosten und die parallele Laufzeit liegt um 109 Takte über der seriellen Laufzeit (*3*).



(a) Parallele Clark-Transformation



(b) Parallele Park-Transformation und Regelkreise

Datenfluss → Synchronisation | Task/ISR auf Kern 0 ☐ 1 ☐

Abbildung 6.9: Parallelisierung von FOS-Tasks nach Datenströmen. Die Komponenten der Clark-Transformation bzw. die der Park-Transformation und Regelkreise werden parallel berechnet. Zur Senkung der Synchronisationskosten findet eine redundante Berechnung von Eingangsdaten statt.

Werden die Regelkreise der Stromkomponenten parallel ausgeführt, so kann ein direkter Übergang von $\mathrm{PT_d}$ nach $\mathrm{RK_d}$ beziehungsweise von $\mathrm{PT_q}$ nach $\mathrm{RK_q}$ stattfinden (siehe Abbildung 6.9b). Dies ist möglich, da keine Informationen aus dem jeweils parallelen Berechnungszweig notwendig sind. Die gesamte Laufzeit der PT- und RK-

Tasks kann insgesamt um 255 Takte reduziert werden. Hiervon entfallen 213 Takte auf das parallele Ausführen der Regelkreise (*4*). Werden lediglich die FOS-Tasks betrachtet, so entspricht dieses Szenario dem besten Ergebnis, das durch Parallelisieren der Tasks erreicht werden kann. Bei einer durchschnittlich gemessenen Laufzeit von 2164 Takten zur Durchführung einer vollständigen FOS, ohne Berücksichtigung der GRB-Task, kann die Laufzeit maximal um 11,78 % verringert werden. Durch die Parallelisierung ist dabei eine Erhöhung der Regelfrequenz um 15,57 % möglich.

Bei den betrachteten GRB-Verfahren lässt sich der Sliding-Mode Observer nach Datenströmen zerlegen. Die hier durchgeführte Schätzung der Gegen-EMK erfolgt jeweils für die α- und β-Stromkomponente. Beide Berechnungen sind unabhängig voneinander und können parallel ausgeführt werden. Hierzu wird der SMO-Algorithmus in die Tasks SMO_α, SMO_β und SMO_θ aufgeteilt. Die ersten beiden Tasks werden parallel ausgeführt und benötigen jeweils eine Laufzeit von 364 CPU-Takten (*5*). Ihre Ergebnisse werden zur Berechnung der Rotorlage in SMO_θ zusammengeführt. Unter Berücksichtigung der gemessenen Kosten von 148 kann die Laufzeit des SMO um 216 Takte verringert werden.

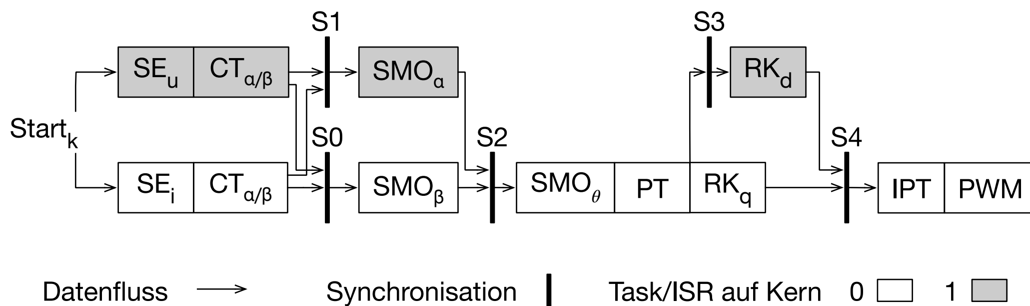


Abbildung 6.10: Parallelisierung der GFOS mit SMO. Parallele Berechnung der SMO-Komponenten und Regelkreise. Zur Senkung der Synchronisationskosten findet eine redundante Berechnung von Eingangsdaten statt.

Wird eine vollständige Anwendung mit SMO betrachtet, so kann entsprechend Abbildung 6.10 parallelisiert werden. Die Erfassung der Ströme und Spannungen gefolgt von den jeweiligen Transformationen werden parallel ausgeführt. An den Semaphoren werden die zur Durchführung von SMO_α beziehungsweise SMO_α fehlenden Eingangsdaten übertragen. An *S0* ist dies die β-Komponente von CT_u und an *S1* die α-Komponente von CT_i. Bei *S2* wird die auf Kern 1 geschätzte α-Komponente der Gegen-EMK an Kern 0 übertragen. Weiterhin werden die Regelkreise auf zwei Kerne verteilt. Hierzu ist ein Übergang von serieller zu paralleler Ausführung an *S3* notwendig. Hier wird die d-Stromkomponente zu Kern 1 übertragen. Die Ergebnis-

se der Regelung werden an *S4* zusammengeführt. Insgesamt werden 5 Datenworte zwischen den Kernen übertragen. Durch die Parallelisierung sinkt die Laufzeit der GFOS auf 3284 CPU-Takte. Im Vergleich zur seriellen Ausführung mit 3622 Takten (siehe Tabelle 6.1 (*1*)) entspricht dies einer Reduktion um 338 Takte beziehungsweise um 9,33 %. Bei einer CPU-Taktrate von 100 MHz kann bei serieller Ausführung eine Regelfrequenz der GFOS von 27,60 kHz realisiert werden. Durch die Parallelisierung ist eine Erhöhung der Regelfrequenz um 10,29 % auf 30,45 kHz möglich.

6.3.4 Parallele Ausführung mittels einer Pipeline

Als Beobachter hat SMO keinen Einfluss auf die Signale des elektrischen Antriebes. Entsprechende Algorithmen können somit prinzipiell innerhalb einer Pipeline ausgeführt werden. Abbildung 6.11 zeigt dies am Beispiel von zwei Prozessorkernen. Eine entsprechende Anpassung der Abtastung der Signale durch das ADC-Modul vorausgesetzt, führt jeder Kern den vollständigen SMO-Algorithmus mit einer Periodendauer von T_{GRB} aus. Der Takt der Pipeline entspricht $\frac{T_{GRB}}{2}$, wodurch die Ausführungsfrequenz des SMO verdoppelt wird.

Der Algorithmus verwendet in der Task SMO_θ eine Phasenregelschleife, um die Rotorlage aus den geschätzten Werten der Gegen-EMK zu generieren [43]. Hierbei wird in Iteration k auf θ aus der vorangegangenen Iteration $k-1$ zurückgegriffen. Somit besteht eine Abhängigkeit von SMO_θ in k zu SMO_θ in $k-1$. Der maximale Abstand der voneinander abhängigen Operationen j_θ und i_θ innerhalb von SMO_θ entspricht D_θ.

Zwei aufeinanderfolgende Iterationen müssen nach Gleichung 4.3 mindestens mit einem zeitlichen Versatz $\Delta Start = \frac{t_{SMO}}{2} = \frac{1876}{2}$ ausgeführt werden (t_{SMO} entsprechend Tabelle 6.1 (*2*)). Mit $D_\theta = 662$ CPU-Takte (entsprechend Tabelle 6.4 (*5*)) genügt dieser Versatz, um die Abhängigkeit von SMO_θ aufzulösen.

6.3.5 Parallelisierung von Rechenoperationen

Lassen sich Algorithmen nicht anhand von Datenströmen parallelisieren, wie auch DFC und HFCI, so können gegebenenfalls einzelne Berechnungen zerlegt und parallel ausgeführt werden. Die Beispiele der Transformationen zeigen jedoch, dass Berechnungen entsprechend umfangreich sein müssen, um in Anbetracht der Kosten einen Laufzeitgewinn zu erzielen. Tabelle 6.5 zeigt an den Beispielen der Matrixmultiplikation und der Implementierung eines digitalen Filters, ab welcher Größenordnung eine Verteilung von Rechenoperationen in Betracht gezogen werden kann. Die Bei-

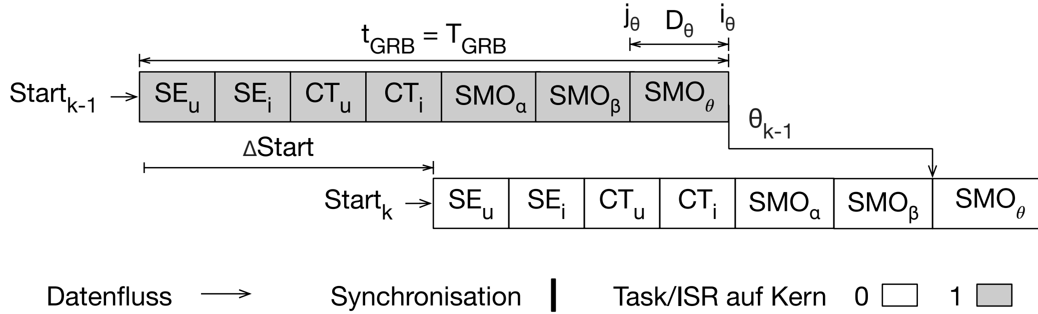


Abbildung 6.11: Ausführung von SMO innerhalb einer zweistufigen Pipeline. Zwei aufeinanderfolgende Iterationen der Regelung werden parallel mit einem zeitlichen Versatz ausgeführt.

Tabelle 6.5: Parallele Ausführung von Rechenoperationen

	∅ Laufzeit [CPU-Takte]		
Berechnung	1 Kern	2 Kerne	3 Kerne
Filter Mult/Add			
5/4	92	272 (+74,36 %)	384 (+146,15 %)
15/14	390	400 (+2,56 %)	568 (+45,64 %)
20/19	510	448 (-12,16 %)	622 (+21,96 %)
Matrizen A·B			
4x4	3970	3312 (-16,57 %)	2734 (-31,13 %)
6x6	15146	11214 (-25,96 %)	9136 (-39,68 %)
12x12	120326	81936 (-31,90 %)	62942 (-47,69 %)

spiele basieren auf zusammenhängenden Multiplikations- und Additionsoperationen, die gleichmäßig auf zwei und drei Kerne verteilt werden.

Es wird von der Implementierung eines Tiefpassfilters 2-ter Ordnung der Form

$$y(n) = b_0 \cdot x(n) + b_1 \cdot x(n-1) + b_2 \cdot x(n-2) - a_1 \cdot y(n-1) - a_2 \cdot y(n-2) \quad (6.4)$$

mit 5 Multiplikationen und 4 Additionen ausgegangen. Durch eine hypothetische Aneinanderreihung mehrere Filter werden die Rechenoperationen vervielfältigt. Die Parallelisierung erfolgt entsprechend Abbildung 6.8, indem die Operationen zur Bildung von Teilsummen verteilt werden. Bei dem Übergang von sequenzieller zu paralleler Ausführung wird mit $x(n)$ ein Datenwort übertragen. Am Übergang zur seriellen Ausführung wird das Teilergebnis an Kern 1 übermittelt. Aufgrund der relativ geringen Anzahl an Operationen erzielt der Einsatz eines zweiten Kerns erst dann eine Verringerung der Laufzeit, wenn mehr als 15/14-Rechenoperationen verteilt werden sollen.

Die Implementierung von Matrizen erfolgt typischerweise in Form zweidimensionaler Felder, welche die Zeilen und Spalten abbilden. Zugriffe auf einzelne Elemente der Felder bedingen das Berechnen beziehungsweise Laden von Indizes. Hierdurch benötigen Datenzugriffe mehr Laufzeit als Zugriffe auf einfache Variablen. Weiterhin sind durch die erhöhte Anzahl an Multiplikationen und Additionen die Laufzeiten der Matrixmultiplikation deutlich höher als die zur Berechnung der Filter. Entsprechend lässt sich mehr Rechenaufwand parallel ausführen, wodurch bereits bei der parallelen Multiplikation von zwei $4x4$ Matrizen auf drei Kernen 31,13 % der Laufzeit eingespart werden kann. Bei der Parallelisierung wurde davon ausgegangen, dass Matrix B konstante Koeffizienten beinhaltet und auf jedem Kern verfügbar ist. Matrix A wird zu gleichen Anteilen auf die parallelen Kerne verteilt.

6.4 Interferenzen bei Parallelisierung

Bei einer Parallelisierung, beispielsweise entsprechend den Abbildungen 6.6 und 6.10, können konkurrierende Zugriffe auf globale Module entstehen. Diese Konkurrenz kann in einer Erhöhung der für die Zugriffe benötigten Laufzeit resultieren und damit die Kosten der Parallelisierung steigern [165].

Bezüglich lesender Zugriffe auf I/O-Module können bei der Ausführung von SE-Tasks konkurrierende Zugriffe entstehen. Nach Abschnitt 7.5.1 Tabelle 7.1 kann sich die Laufzeit lesender Zugriffe auf I/O-Module um 4 bis 8 Takte erhöhen, wenn zwei beziehungsweise drei Prozessorkerne parallel auf solche Module zugreifen. An den Synchronisationspunkten werden Daten über gemeinsamen Speicher ausgetauscht. Auch hier kann eine Konkurrenz entstehen, typischerweise dann, wenn Daten nach der Synchronisation von mehreren Kernen parallel gelesen werden (vergleiche Abbildung 6.10 $\mathrm{RK_d}$ und $\mathrm{RK_q}$ nach *S3*). Konkurrenz bei schreibenden Zugriffen kann nach einem Datenaustausch (vergleiche Abbildung 6.10 Tasks bei *S0* und *S1*) oder bei dem Übergang zu einer seriellen Ausführung auftreten (vergleiche Abbildung 6.10 SMO_α und SMO_β bei *S2*). Die Laufzeiten schreibender Zugriffe können sich hier um 8 beziehungsweise 16 Takte erhöhen. Für lesende Zugriffe entsteht erst bei drei konkurrierenden Kernen eine Laufzeiterhöhung. Diese liegt bei bis zu 3 CPU-Takten für jeden Zugriff. Die zusätzlichen Laufzeiten werden auf die regulären Laufzeiten der Zugriffe addiert. Bezüglich Gleichung 4.4 ist hierdurch eine steilere Zunahme der Synchronisationskosten zu erwarten.

Bei der in Abbildung 6.10 beschriebenen Parallelisierung werden durch die Tasks $\mathrm{SE_u}$ und $\mathrm{SE_i}$ jeweils drei lesende Zugriffe durchgeführt, die um das I/O-Modul kon-

kurrieren. Die CT-Tasks konkurrieren mit der Synchronisation von u_β an *S0* und i_α an *S1* um den schreibenden Zugriff für jeweils ein Datenwort. SMO_α und SMO_β führen je einen konkurrierenden lesenden Zugriff auf den Speicher durch, um die Daten der CT-Tasks entgegenzunehmen. An *S2* finden keine konkurrierenden Zugriffe auf den Speicher statt, da SMO_β und SMO_θ auf dem gleichen Kern arbeiten und daher keine Daten über das globale Speichermodul übertragen müssen. Es findet lediglich ein Zugriff seitens SMO_θ statt, um den von SMO_α übertragenen Wert der Gegen-EMK zu erhalten. Auch an *S3* und *S4* finden keine parallelen Zugriffe statt, da RK_q und IPT auf dem gleichen Kern ausgeführt werden. Insgesamt werden drei lesende konkurrierende Zugriffe auf ein I/O-Modul und jeweils ein lesender und schreibender konkurrierender Zugriff auf den globalen Speicher durchgeführt. Wird für jeden Zugriff eine maximale Laufzeiterhöhung entsprechend Abschnitt 7.5.1 Tabelle 7.1 angenommen, kann eine maximale Erhöhung der Synchronisationskosten von 25 CPU-Takten entstehen. Bei der parallelen Laufzeit von 3284 CPU-Takten entspricht dies einer Laufzeiterhöhung von 0,76 %.

6.5 Zusammenfassung der Ergebnisse

Zur Untersuchung der Parallelisierbarkeit von geberlosen feldorientieren Stromregelungen wurden die zu betrachtenden Anwendungen in Form eines Taskmodells generalisiert. Hierbei wird zwischen der Integration von zeitkontinuierlichen und zeitdiskreten GRB unterschieden. Auf Basis dieser Unterscheidung wurde das Ausführen der geberlose Rotorlageberechnung parallel zu den Tasks der FOS beschrieben.

Durch die inhärente Nebenläufigkeit bei zeitkontinuierlichen GRB können der GRB-Algorithmus sowie notwendige Tasks zur Vorbereitung der Eingangssignale vorteilhaft auf einen zweiten Kern ausgelagert werden. Eine Synchronisation zwischen beiden Kernen ist nicht notwendig. Abhängig von den Laufzeiten der ausgelagerten Tasks erreicht hier die Parallelisierung eine entsprechend hohe Reduktion der Rechenzeit. Am Beispiel einer GFOS unter Verwendung des SMO-Algorithmus kann eine Reduktion der Laufzeit um 46 % erreicht werden. Dies ermöglicht die Frequenz des SMO um 96,3 % und die der FOS um 88,9 % zu steigern.

Bei der Verwendung zeitdiskreter GRB wird die Ausführung der GRB typischerweise mit der Ausführung der FOS-Tasks synchronisiert. Auch hier wurde gezeigt, dass durch das Ausführen der GRB-Tasks parallel zu denen der FOS Laufzeit eingespart werden kann. Hierbei wurde die maximal erreichbare Laufzeitreduktion bestimmt. Diese ist durch die Summe der Laufzeiten der SE- und CT-Tasks begrenzt,

wenn die Rotorlage in der gleichen Iteration verwendet werden muss, in der sie berechnet wurde. Für die MCMC liegt dieser Wert in einem Bereich von 210 CPU-Takten ($t_{SE_i} - t_{CT_i}$ siehe Abschnitt 6.2.1). Abzüglich der notwendigen Synchronisationskosten für mindestens ein Datenwort in der Größenordnung von 100 Takten, liegt die hier erreichbare Laufzeitreduktion bei 110 CPU-Takten. Folglich hat die Parallelisierung den größten Effekt, wenn die Laufzeit der GRB maximal 210 CPU-Takte beträgt. Bei einer Laufzeit der FOS von 2164 CPU-Takten kann die Laufzeit der resultierenden GFOS (2374 CPU-Takte) um maximal 4,63 % verringert werden. Am Beispiel einer GFOS unter Verwendung von DFC kann auf diese Weise eine Laufzeitreduktion von 3,87 % erreicht werden. Dies entspricht einer Steigerung der Regelfrequenz von 4,02 %.

Kann oder muss die Rotorlage aus einer vorangegangenen Iteration verwendet werden, so beschreibt die gesamte Laufzeit der FOS-Tasks die maximal erreichbare Laufzeitreduktion (durchschnittlich 2164 CPU-Takte für den MCMC). Der HFCI-Algorithmus benötigt die Rotorlage aus einer vorangegangenen Iteration. Bei der Parallelisierung einer entsprechenden GFOS kann HFCI vollständig auf einen parallelen Kern ausgelagert werden, wodurch die Laufzeit der GFOS um 37,34 % verringert wird. Für die Regelfrequenz bedeutet dies ein Plus von 59,57 %.

Weiterhin wurden die Datenströme des Taskmodells und die Möglichkeiten, Tasks auf dieser Basis zu parallelisieren, beschrieben. Durch die vergleichsweise kurzen Laufzeiten einzelner FOS-Tasks übersteigen hier die Kosten größtenteils den erreichbaren Laufzeitgewinn. Am Beispiel einer GFOS mit SMO konnte dennoch eine Reduktion der Laufzeit um 9,33 % durch eine Parallelisierung des SMO-Algorithmus und die der Regelkreise erreicht werden. Dies ist durch die Struktur von SMO begründet, da hier ein Großteil der Berechnungen auf den voneinander unabhängigen α- und β-Komponenten der Spannungen und Ströme durchgeführt werden.

Insgesamt können die Laufzeiten der betrachteten Anwendungen durch eine Parallelisierung in einem Bereich zwischen 3 % und 46 % reduziert werden. Dies ist jedoch der Tatsache geschuldet, dass die volle Rechenleistung des zweiten verwendeten Prozessorkerns zur Verfügung steht. So konnten zum Teil identische Tasks von beiden Kernen ausgeführt werden, um Synchronisationskosten einzusparen. Vor allem für Anwendungen mit zeitkontinuierlicher GRB kann durch paralleles Ausführen der GRB-Tasks eine deutliche Steigerung der Ausführungsfrequenzen erzielt werden. Entsprechendes gilt für Anwendungen, die auf ältere Werte der Rotorlage zurückgreifen können. In diesen Fällen sind für Anwendungen, die auf das Taskmodell abgebildet werden können, Ergebnisse im zweistelligen Bereich möglich.

Kapitel 7

Motor-Controller Konsolidierung und Cross-Core-Interferenzen

Cross-Core-Interferenzen können immer dann auftreten, wenn parallele Prozessorkerne auf geteilte Peripherie- oder Speichermodule zugreifen. Solche Zugriffe können entstehen, wenn mehrere dedizierte Motor-Controller in einem Multi-Core-System konsolidiert werden. Die Verwendung dedizierter eingebetteter Systeme als Motor-Controller realisiert eine räumliche als auch zeitliche Isolation zwischen diesen. Eine räumliche Isolation verhindert kernübergreifende Zugriffe auf Speicher und Peripheriemodule. Daten, Code sowie die Konfiguration der genutzten On-Chip-Peripherie sind dadurch vor interferierenden Zugriffen geschützt. Die zeitliche Isolation verhindert Konkurrenz um den Zugriff auf geteilte Ressourcen. Dadurch, dass eine Ressource nicht durch ein anderes System belegt sein kann, wird ein konstantes und damit vorhersagbares zeitliches Verhalten für den Zugriff bewahrt. Die Konsolidierung hebt beide Formen der Isolation auf, sodass sich funktionale und zeitliche Eigenschaften der Systeme nach der Konsolidierung ändern können. Die betrachtete Multi-Core-Plattform stellt die räumliche Isolation durch die Kapselung von Software beziehungsweise Tasks wieder her. Werden mehrere Prozessorkerne genutzt, so wirkt sich das Fehlen der zeitlichen Isolation auf das zeitliche Verhalten der konsolidierten Systeme aus.

In den folgenden Betrachtungen wird das Auftreten potenziell miteinander konkurrierender Zugriffe nach einer Konsolidierung beschrieben. Hierdurch entstehende Laufzeiterhöhungen und Möglichkeiten zu deren Beeinflussung werden experimentell untersucht.

7.1 Zugriffe auf globale Module

Die im Anwendungsgebiet betrachteten Multi-Core-Systeme setzen unterschiedliche Intra-Chip-Verbindungen ein: Einen Peripheriebus zur Anbindung von I/O-Peripheriemodulen sowie eine Crossbar zur Anbindung von Speichermodulen an die Prozessorkerne (vergleiche Abschnitt 4.1). Bezüglich des Speichers entsteht eine Konkurrenz um das entsprechende Modul, wenn parallel versucht wird, auf dieses zuzugreifen. Bei Zugriffen auf Module an einem Peripheriebus entsteht Konkurrenz um den Zugriff auf den Bus selbst. Somit ist es irrelevant, ob mehrere Prozessorkerne versuchen, zeitgleich auf ein I/O-Modul oder auf unterschiedliche zuzugreifen. Zwischen parallelen Zugriffsversuchen auf Speicher und I/O-Peripherie entsteht aufgrund der getrennten Intra-Chip-Verbindungen keine Konkurrenz.

Zugriffe auf I/O-Peripherie sind durch das Taskmodell beschrieben. Diese sind das Lesen mehrerer ADC-Kanäle zur Erfassung von Spannungen und Strömen (SE-Task) sowie die Konfiguration der PWM-Kanäle zur Erzeugung der Ausgabesignale (PWM-Task). Es wird davon ausgegangen, dass Messergebnisse der ADC-Kanäle pro Iteration einmal aus den Registern des Moduls gelesen und zur Verarbeitung im lokalen Arbeitsspeicher abgelegt werden. Entsprechendes gilt für die Konfiguration der PWM-Kanäle und das Schreiben der entsprechenden Werte in die Register des Moduls. ADC- und PWM-Kanäle sind typischerweise innerhalb entsprechender ADC- beziehungsweise PWM-Module gebündelt, sodass Konkurrenz bei Zugriffen auf einzelne Kanäle entstehen kann. Abhängig von der Implementierung der Tasks können zusätzliche Timer zu deren Koordination eingesetzt werden. Sie aktivieren beispielsweise das Sampling der ADC-Kanäle zu definierten Zeitpunkten oder starten die automatische Übertragung einer neuen PWM-Konfiguration aus temporären Konfigurationsregistern in die aktuell von der nächsten PWM-Phase verwendeten Register. Letzteres kann der konsistenten Konfiguration aller PWM-Kanäle dienen. Nach einmaliger Konfiguration, typischerweise zu Systemstart, arbeiten die Timer selbstständig. Zugriffe zur Laufzeit der Anwendung auf das Timer-Modul sind daher nicht notwendig, wodurch sie bei der Betrachtung konkurrierender Zugriffe vernachlässigt werden können.

Zugriffe auf gemeinsamen Speicher finden statt, wenn konsolidierte Systeme Daten austauschen. Es wird realistischerweise angenommen, dass bei einem Kommunikationsvorgang die Übertragung mehrerer Datenworte stattfindet. Dies kann in mehreren unmittelbar aufeinanderfolgenden Zugriffen auf das globale Speichermodul resultieren. Entsprechend Abschnitt 4.2.3 wird für eine kernübergreifende Kommunikation auf den Message-Passing-Dienst des verwendeten Betriebssystems zurück-

gegriffen. Dieser ermöglicht eine sichere Datenübertragung (im Sinne von Safety) zwischen konsolidierten Systemen. Die Nachrichtenübertragung benötigt eine vergleichsweise lange Laufzeit. Daher wird sie typischerweise nebenläufig zu den im Taskmodell definierten Aufgaben der GFOS durchgeführt. Zeitliche Eigenschaften dieser höher priorisierten Tasks werden so nicht verletzt. Zugriffe auf globalen Speicher im Rahmen einer Inter-Core-Kommunikation begrenzen sich somit auf die Zeit, in der keine Tasks der GFOS ausgeführt werden.

Die im MCMC verwendeten Prozessorkerne haben vergleichsweise große lokale Arbeitsspeicher, welche für die betrachteten Anwendungen ausreichende Speicherkapazitäten bieten. Dies ist dem ursprünglichen Anwendungsgebiet der Systeme geschuldet und gilt nicht für alle eingebetteten Multi-Core-Architekturen. Somit sind Systemkonfigurationen denkbar, die eine Kombination aus lokalem und globalem Arbeitsspeicher erfordern. Wird ein solches System zur Ausführung der GFOS-Tasks verwendet, können Zugriffe auf globalen Speicher über die gesamte Laufzeit der GFOS verteilt sein.

Nach einer Konsolidierung können prinzipiell alle Zugriffe auf globale Module um den Modulzugriff miteinander konkurrieren. Für eine Beschreibung, wann eine Konkurrenz entstehen kann, wird zwischen der Konsolidierung identischer und unterschiedlicher Motor-Controller unterschieden. Im ersten Fall werden mehrere Instanzen einer GFOS-Implementierung auf dem MCMC integriert. Bei einer solchen Konsolidierung kann angenommen werden, dass alle Prozessorkerne synchron den gleichen Code ausführen und entsprechend gleichzeitig auf I/O-Peripherie zugreifen. Somit ist bei jedem Modulzugriff mit Interferenzen zu rechnen. Eine Synchronisation der Kerne beziehungsweise der Zeitpunkte, an denen eine neue Iteration der FOS oder der GRB ausgeführt wird, kann implizit durch die Verwendung von hardwarebasierten Timern geschehen. Sie realisieren letztendlich die PWM-Signale aller Motor-Controller. Die PWM wiederum beschreibt die Zeitpunkte, an denen Messwerte erfasst werden und ihre Verarbeitung stattfindet. Die den Motor-Controllern zugeordneten Timer sind ähnlich den ADC- und PWM-Kanälen in einem globalen Timer-Modul gebündelt. Ein übliches Vorgehen bei der Programmierung solcher Module ist die einmalige Konfiguration der genutzten Timer zu Systemstart und ihre gemeinsame Aktivierung im Anschluss daran. Dies kann beispielsweise durch den Start des gesamten Moduls geschehen. Als Folge kann von einem nahezu synchronen Start der einzelnen Timer und der daraus folgenden Synchronisation der PWM-Perioden ausgegangen werden. Dementsprechend wird auch die Software der Motor-Controller synchron ausgeführt.

7.2 Konkurrierende Zugriffe auf gemeinsame Module

Bei Zugriffen auf globale I/O- und Speichermodule werden regelmäßig mehrere Zugriffe nacheinander durchgeführt. Eine Iteration der Stromreglung führt beispielsweise mehrere Spannungs- und Strommessungen auf den Phasen des Motors durch. Seitens des ADC-Moduls wird jeder Messwert in einem separaten Register abgelegt. Im Rahmen der Signalerfassung werden diese nacheinander ausgelesen. Entsprechendes gilt für das Versenden von Daten über das Message-Passing-System. Eine Nachricht enthält typischerweise mehrere Datenwerte, beispielsweise die berechnete Rotorlage sowie die gemessenen Spannungen und Ströme der Phasen. Sind diese Daten erfasst, werden sie nacheinander in den globalen Speicher einer Nachricht kopiert, bevor diese gesendet werden kann.

7.2.1 Abläufe konkurrierender Zugriffe

Abbildung 7.1 zeigt die zeitlichen Abläufe, wenn drei Prozessorkerne jeweils zwei Zugriffe auf ein gemeinsames Modul durchführen. Es wird davon ausgegangen, dass die ersten Zugriffsanfragen b_0 nahezu zeitgleich zum Zeitpunkt 0 gestellt werden.

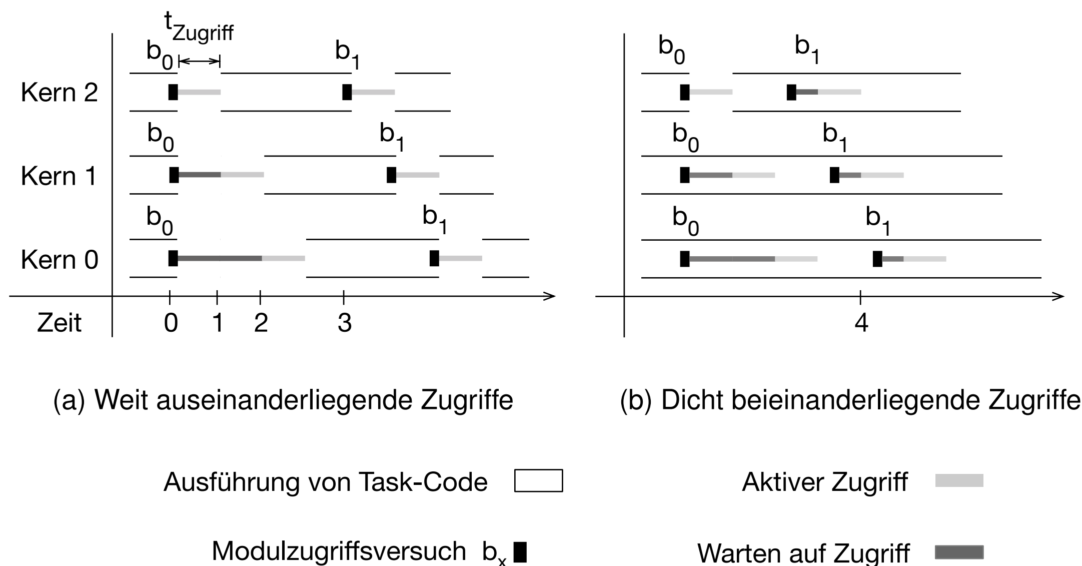


Abbildung 7.1: Konkurrierende Zugriffe auf ein gemeinsames Hardwaremodul. Durch Konkurrenz unterliegen Prozessorkerne bei parallelen Modulzugriffsversuchen unterschiedlich großen Wartezeiten, bis sie aktiven Modulzugriff erhalten.

In Abbildung 7.1a liegen die Zugriffsversuche zeitlich relativ weit auseinander. Ausgehend von einer auf Prioritäten basierenden Arbitrierung der Zugriffe erhält

Kern 2 die höchste Priorität und somit vorrangigen Zugriff auf das gemeinsame Modul. Die Ausführung von Task-Code blockiert für die Zugriffsdauer $t_{Zugriff}$. Sie wird nach dem Ende des Zugriffs durch die Freigabe des Moduls zum Zeitpunkt 1 fortgesetzt. Für die Kerne 0 und 1 entsteht eine Wartezeit. Durch die gleichzeitigen Zugriffsversuche ist diese maximal und entspricht der Zugriffszeit von Kern 2. Nach der Freigabe des Moduls erhält Kern 1 vorrangig Zugriff und die Wartezeit für Kern 0 steigt um die Zugriffszeit von Kern 1. Der Zugriff von Kern 1 endet zum Zeitpunkt 2. Durch $t_1 < t_2$ entsteht ein zeitlicher Versatz zwischen den Kernen 2 und 1, wodurch Kern 2 in der Taskausführung Kern 1 vorauseilt. Der zeitliche Versatz entsteht auch zwischen Kern 1 und 0, wodurch Kern 1 in der Taskausführung Kern 0 vorauseilt. Das Vorauseilen zeigt sich an den nun versetzten Zeitpunkten der jeweils zweiten Zugriffsversuche b_1. Gelangt Kern 2 zum Zeitpunkt 3 zu b_1, so befindet sich Kern 1 um $t_{Zugriff}$ hinter b_1. Während des aktiven Zugriffs von Kern 2 nähert sich Kern 1 b_1 an und erreicht b_1, wenn Kern 2 den Modulzugriff beendet hat. Kern 1 kann nun ohne eine zusätzliche Wartezeit auf das Modul zugreifen. Entsprechendes gilt für Kern 0. Hat sich der zeitliche Versatz für alle Prozessorkerne eingestellt, kann bei N Prozessorkernen die Taskausführung ohne weitere Wartezeiten fortgesetzt werden, wenn

- die Zugriffsversuche auf jedem Kern mindestens den zeitlichen Abstand $(N-1) \cdot t_{Zugriff}$ haben und
- auf jedem Kern identischer Code ausgeführt wird.

In Abbildung 7.1b wird der zeitliche Abstand zwischen b_0 und b_1 verringert. In diesem Fall erreicht Kern 2 b_1, bevor Kern 0 den Zugriff von b_0 beendet hat. Für Kern 2 entsteht eine Wartezeit, bis Kern 0 den aktiven Zugriff zum Zeitpunkt 4 beendet hat. Entsprechendes gilt für die Kerne 0 und 1 bei b_1 bezüglich des jeweils vorangehenden aktiven Zugriffs. Dadurch, dass die Kerne 1 und 2 aktiv Task-Code ausführen, während Kern 0 auf das Modul zugreift, sind die an b_1 entstehenden Wartezeiten geringer als die bei b_0.

Ist der zeitliche Abstand zwischen b_0 und b_1 geringer als $t_{Zugriff}$, erreicht Kern 2 b_1, während Kern 1 aktiven Zugriff bei b_0 hat. Zusätzlich entsteht eine Wartezeit, die sich aus der restlichen Dauer des Zugriffs von Kern 1 und der Dauer des vollständigen Zugriffs von Kern 0 an b_0 zusammensetzt.

7.2.2 Einflussfaktoren auf die Abstände aufeinanderfolgender Modulzugriffe

Dicht aufeinanderfolgende Modulzugriffe beziehungsweise Zugriffsversuche entstehen, wenn im Quellcode Zugriffe auf globalen Speicher oder I/O-Module jeweils blockweise angeordnet sind. Dies kann beispielsweise durch das gleichzeitige Auslesen mehrerer ADC-Kanäle oder durch Zugriffe auf gemeinsamen Speicher zur Synchronisation von Daten geschehen. Letztendlich resultieren Modulzugriffsversuche aus Load- und Store-Instruktionen, die auf den Adressbereich eines global zugreifbaren Moduls zielen. Diese Instruktionen entstehen durch die Übersetzung des Quellcodes in ausführbaren Maschinencode. Auf Ebene des Quellcodes gibt es für den Zugriff auf Datenspeicher beziehungsweise Register unterschiedliche Implementierungsmuster. Dies sind beispielsweise die Verwendung einfacher Variablen, Pointern oder Bitfelder. Oftmals entscheiden persönliche Präferenzen des Entwicklers über die Realisierung von Zugriffen oder die verwendeten Bibliotheken und Codegeneratoren. Die Art und Anzahl der aus den unterschiedlichen Mustern resultierenden Instruktionen können bis zum Erreichen der Load- und Store-Instruktionen, die den Modulzugriffsversuch auslösen, variieren. Somit können unterschiedliche Laufzeiten und damit zeitliche Abstände zwischen aufeinanderfolgenden Modulzugriffsversuchen entstehen. Folglich können dadurch Wartezeiten verringert werden.

Zusätzliche Abstände können auch durch eine Umstrukturierung von Code realisiert werden. Berechnungen, die auf Daten aus globalen Modulen beruhen, können zwischen die einzelnen Zugriffe verschoben werden. Eine signifikante Änderung der Anzahl oder Art von Instruktionen entsteht hierbei nicht, wodurch auch keine signifikante Veränderung der regulären Laufzeit zu erwarten ist.

Durch die Verwendung einer Compiler-Optimierung kann eine Optimierung der aus dem Quellcode erzeugten Instruktionen stattfinden, um beispielsweise die Laufzeit einer Anwendung zu verringern [88]. Dies bedeutet auch, dass die zeitlichen Abstände zwischen Modulzugriffen signifikant verringert und vermehrt Wartezeiten durch konkurrierende Zugriffe entstehen können. Die Verwendung einer Optimierung verringert zwar die absolute Laufzeit der Anwendungen, allerdings kann die Differenz zwischen der Laufzeit in Isolation und unter Verwendung paralleler Kerne zunehmen.

7.3 Parallele Zugriffe auf globale I/O-Module

Durch das Taskmodell ist bekannt, welche Tasks auf I/O-Module beziehungsweise auf die entsprechende Intra-Chip-Verbindung zugreifen. Weiterhin werden die Startpunkte der Anwendungen auf den Kernen und die durchschnittlichen Laufzeiten der Tasks als gegeben angenommen. Somit ist bekannt, wann Tasks, in Relation zum Start der Anwendung auf dem jeweiligen Kern, ausgeführt werden.

7.3.1 Konsolidierung gleichartiger Motor-Controller

Abbildung 7.2 zeigt die Konsolidierung von zwei identischen Motor-Controllern auf den Kernen 0 und 1. Es sind jeweils zwei Iterationen einer GFOS mit zeitdiskreter GRB dargestellt. Beide Teilsysteme arbeiten mit derselben Regelfrequenz.

7.3.1.1 Synchrone Ausführung der Teilsysteme

In jeder Iteration findet ein synchroner Start der Teilsysteme statt. Da auf allen Prozessorkernen derselbe Code ausgeführt wird, kann angenommen werden, dass die Prozessorkerne nahezu zeitgleich identische Tasks ausführen. Somit kann weiter angenommen werden, dass auch Modulzugriffe innerhalb der Tasks parallel ausgeführt werden und entsprechende Zugriffsversuche miteinander konkurrieren. Konkret betroffen sind hier die SE- und PWM-Tasks. In diesem Fall kann eine vollständig parallele Ausführung der jeweiligen Tasks angenommen werden.

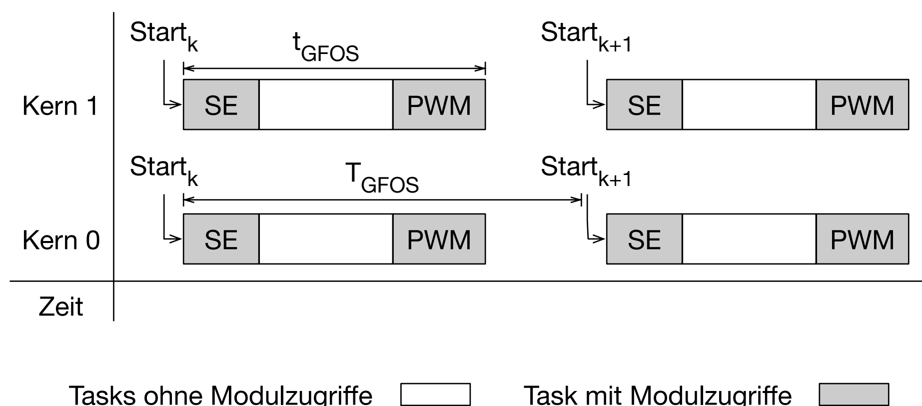


Abbildung 7.2: Synchrone Ausführung gleicher Anwendungen/Implementierungen

Bei einer Arbitrierung nach Prioritäten bedeuten konkurrierende Zugriffsversuche, dass ein Prozessorkern so lange blockieren kann, bis alle Kerne mit einer höheren Priorität ihre Zugriffe beendet haben. Bei N Kernen und jeweils m Modulzugriffen innerhalb der SE-Task ergibt sich die maximale Wartezeit für einen Kern mit der

Priorität i aus $((N-1)\cdot i)\cdot(m\cdot t_{Zugriff})$. Der sich durch das Warten einstellende zeitliche Versatz kann Interferenzen bei der parallelen Ausführung der PWM-Tasks verringern und gegebenenfalls vollständig verhindern. Dies ist davon abhängig, wie viel des Versatzes bis zum Erreichen der PWM-Tasks erhalten bleibt sowie von der Anzahl der dort durchzuführenden Modulzugriffe.

7.3.1.2 Zeitlich versetzte Ausführung der Teilsysteme

Eine synchrone Ausführung der Teilsysteme kann unterbunden werden, indem die Timer zur Erzeugung der PWM-Signale zeitlich versetzt gestartet werden. Abbildung 7.3 zeigt eine um ΔStart verzögerte Ausführung auf Kern 1 bezüglich Kern 2. In diesem Fall eilt Kern 1 der Ausführung von Kern 2 nach. Iteration k auf Kern 1 wird dementsprechend zu einem späteren Zeitpunkt gestartet. Der gezeigte Versatz ist ausreichend groß, damit auf Kern 1 die Ausführung der Tasks mit Modulzugriffen in einen Zeitraum geschoben wird, in dem Kern 2 keine Zugriffe auf das gemeinsame Modul durchführt. Der Betrag von ΔStart muss somit größer sein als die Laufzeiten von SE und PWM. Mit $t_{SE} < t_{PWM}$ gilt im Beispiel $\Delta Start \geq t_{PWM}$.

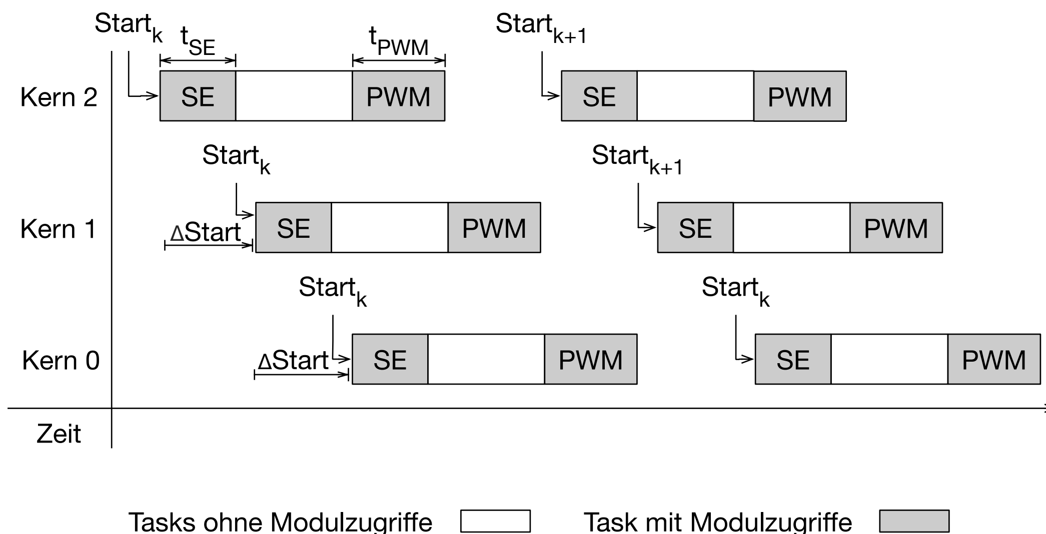


Abbildung 7.3: Zeitlich verschobene Ausführung gleicher Anwendungen

Um eine ausreichend große Verschiebung realisieren zu können, muss nach jeder Task mit Modulzugriff ein zeitlicher Abschnitt existieren, in dem keine Zugriffe durchgeführt werden. Im gezeigten Beispiel sind dies die Abschnitte zwischen der SE- und PWM-Task und nach der PWM-Task bis zum Start der SE-Task einer neuen Iteration.

Wird ein drittes Teilsystem auf Kern 0 ausgeführt, so ist die zugriffsfreie Zeit zwischen der SE- und PWM-Task auf Kern 2 nicht ausreichend groß, damit wäh-

rend dieser Zeit zusätzlich die SE-Task von Kern 0 ausgeführt werden kann. Ihre Ausführung findet teilweise zeitgleich mit der PWM-Task von Kern 2 statt, sodass hier Konkurrenz entstehen kann.

ΔStart muss die zeitgleiche Ausführung der Tasks nicht notwendigerweise vollständig verhindern. Es genügt, wenn ΔStart groß genug ist, um lediglich die parallele Ausführung der Task-Bereiche, welche die Modulzugriffe implementieren, entsprechend zu verschieben. Abbildung 7.4 zeigt dies am Beispiel von drei Modulzugriffen pro Teilsystem. Die Zeit, in der alle Modulzugriffe durchgeführt werden, entspricht $t_{Zugriff}$. Während des Zeitraums t_{Code} wird Task-Code ohne Zugriffe auf ein gemeinsames Medium ausgeführt.

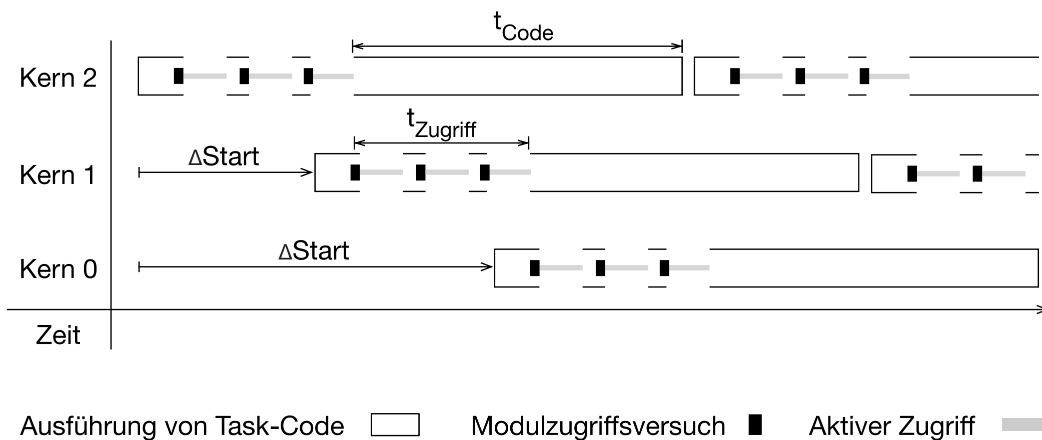


Abbildung 7.4: Modulzugriffe innerhalb einer Task bei zeitlich versetzter Ausführung. Werden gleiche Tasks mit einem zeitlichen Versatz parallel ausgeführt, können Wartezeiten bei aufeinanderfolgenden Modulzugriffsversuchen innerhalb der Tasks verhindert werden.

ΔStart ist so gewählt, dass der durch $t_{Zugriff}$ beschriebene Abschnitt auf Kern 0 zeitgleich zu t_{Code} auf Kern 1 ausgeführt wird. Durch die periodische Ausführung gilt dies auch für $t_{Zugriff}$ auf Kern 2 bezüglich t_{Code} auf Kern 0. Mit $t_{Zugriff}$ wird hier das exakte zeitliche Intervall der Modulzugriffe betrachtet. Dies kann deutlich kleiner angenommen werden als die in Abbildung 7.3 betrachteten Laufzeiten t_{SE} oder t_{PWM} der vollständigen Tasks. Entsprechend kann auch für ΔStart ein vergleichsweise geringerer Betrag angenommen werden. In diesem Fall kann der Abschnitt ohne Modulzugriff mit t_{Code} genügen, um auch für größere $t_{Zugriff}$ oder zusätzliche Teilsysteme konkurrierende Modulzugriffe durch einen zeitlichen Versatz zu serialisieren. Analog zu dem zeitlichen Versatz, der sich durch eine Wartezeit bei einem belegten Modul einstellen würde, muss $\Delta Start = (N-1) \cdot t_{Zugriff}$ realisiert sein, wenn eine interferenzfreie Ausführung von N Teilsystemen erreicht werden soll.

7.3.2 Konsolidierung unterschiedlicher Motor-Controller

Abbildung 7.5 beschreibt die Integration von zwei Motor-Controllern, die unterschiedliche Anwendungen zur Regelung der Antriebe integrieren. Unterschiede können in verschiedenen Regelfrequenzen, Laufzeiten und Implementierungen liegen. Im Beispiel ist auf Kern 0 eine GFOS mit zeitdiskreter GRB (GFOS 0) und auf Kern 1 eine mit zeitkontinuierlicher GRB (GFOS 1) integriert. Die Periodendauer von GFOS 1 entspricht $1,5 \cdot GFOS\,0$.

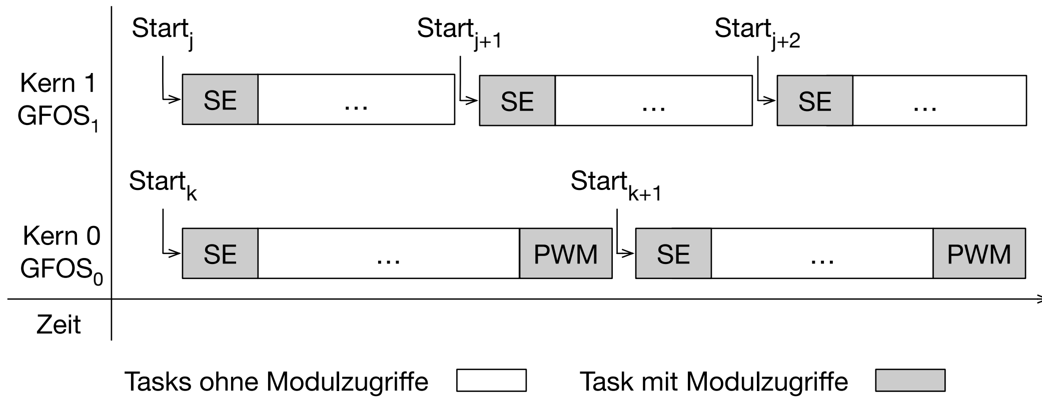


Abbildung 7.5: Synchrone Ausführung unterschiedlicher Anwendungen

Durch die unterschiedlichen Implementierungen und Regelfrequenzen variieren in aufeinanderfolgenden Iterationen die parallel zueinander ausgeführten Tasks. Auf Kern 0 konkurrieren die Tasks SE und PWM in Iteration k mit der Task SE in den Iterationen j und $j+1$ auf Kern 1. Modulzugriffe in Iteration $k+1$ werden ohne Konkurrenz ausgeführt. Dementsprechend unterscheiden sich auch die Interferenzen in aufeinanderfolgenden Iterationen. Für eine Betrachtung der Interferenzen entsprechend Abschnitt 7.3.1 müssen alle Iterationen erfasst werden, in denen unterschiedliche Kombinationen der parallelen Ausführung von Tasks mit Modulzugriffen entstehen. Dies ist mit dem in Abschnitt 6.1.2.2 beschriebenen Muster der Unterbrechung der FOS durch die nebenläufig ausgeführte GRB vergleichbar. Während sich bei der nebenläufigen Ausführung die Reihenfolge der Unterbrechungen in aufeinanderfolgenden Iterationen ändert, so ändern sich bei der parallelen Ausführung die zeitgleich ausgeführten Tasks in aufeinanderfolgenden Iterationen. Die Anzahl der Iterationen, in denen unterschiedliche Kombinationen parallel ausgeführter Tasks auftreten, kann dementsprechend Gleichung 6.2 berechnet werden. In Abbildung 7.5 sind dies zwei Iterationen auf Kern 0 und drei auf Kern 1.

Ist für jeden Kern die entsprechende Anzahl an Iterationen bekannt, kann auch hier ein zeitlicher Versatz ΔStart betrachtet werden, um gleichzeitige Modulzugriffe

zu reduzieren. Bei der Betrachtung gleichartiger Anwendungen (ohne zeitlichen Versatz) traten Modulzugriffe stets parallel auf. Bei unterschiedlichen Anwendungen ist dieses Verhalten nicht gegeben. Parallele Zugriffe können zu beliebigen Zeitpunkten stattfinden. Dementsprechend müssen über alle der festgelegten Iterationen die parallel ausgeführten Tasks ermittelt werden, um einen geeigneten Wert für ΔStart zu finden.

7.4 Parallele Zugriffe auf globalen Arbeitsspeicher

Zugriffe auf globalen Arbeitsspeicher werden durch zwei Szenarien abgedeckt: Zugriffe, die im Rahmen einer Inter-Core-Kommunikation stattfinden und die Erweiterung von lokalem durch globalen Speicher. Der MCMC stellt für beide Szenarien allen Prozessorkernen ein globales Speichermodul zur Verfügung. Alle parallelen Zugriffsversuche auf das Modul können miteinander konkurrieren.

7.4.1 Kommunikation zwischen Motor-Controllern

Entsprechend Abschnitt 4.2.3 wird für eine kernübergreifende Kommunikation auf den Message-Passing-Dienst des verwendeten Betriebssystems zurückgegriffen. Jede Nachricht beinhaltet einen Nutzdatenbereich, der im Adressraum des globalen Speichermoduls liegt. Vor dem Senden der Nachricht müssen die zu übertragenden Daten in diesen Datenbereich geschrieben werden. Entsprechendes gilt bezüglich des Lesens der Daten nach dem Empfang einer Nachricht.

Abbildung 7.6 zeigt den Versand und Empfang von zwei Nachrichten unter Verwendung von drei Prozessorkernen. Es wird davon ausgegangen, dass auf den sendenden Kernen 0 und 1 identische Motor-Controller integriert sind. Motor-Controller 2 führt keine GFOS aus. Hier findet lediglich die Bearbeitung der erhaltenen Daten, beispielsweise zur Realisierung eines Monitorings, statt. Der Nachrichtenaustausch geschieht nebenläufig zu den Tasks der GFOS. Letztere sind höher priorisiert als der Nachrichtenversand, sodass dieser nur in der Zeit zwischen zwei Iterationen stattfinden kann. Entsprechend muss $t_{GFOS} < T_{GFOS}$ gelten. Durch die identischen Motor-Controller ist davon auszugehen, dass das Kopieren der zu sendenden Daten in den globalen Nutzdatenspeicher der Nachrichten parallel verläuft. Ebenso verlaufen die auf das Kopieren folgenden Sendevorgänge parallel. Das Empfangen der Nachrichten und die lesenden Zugriffe auf ihre Nutzdaten seitens Motor-Controller 2 finden seriell statt.

Auch hier ergibt sich entsprechend Abschnitt 7.3.1 die Möglichkeit, den Start von

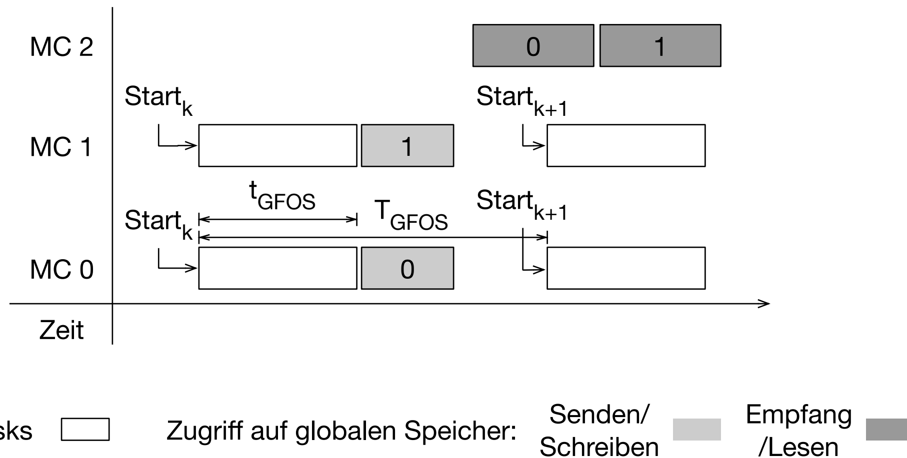


Abbildung 7.6: Paralleler kernübergreifender Nachrichtenversand. Zwei Motor-Controller senden parallel eine Nachricht an einen gemeinsamen Empfänger. Das Senden der Nachrichten verursacht konkurrierende Zugriffe auf gemeinsamen Speicher.

Iteration k auf einem der Motor-Controller um ΔStart zu verschieben. Weiterhin kann der Nachrichtenversand beziehungsweise der Zugriff auf den Nutzdatenspeicher zwischen dem Beenden der GFOS-Tasks in Iteration k und dem Beginn von $k + 1$ verschoben werden. Soll in jeder Iteration ein Datenversand stattfinden, so kann dieser zeitliche Versatz höchstens $T_{GFOS} - t_{GFOS}$ betragen. Unter Umständen kann hierbei der Nachrichtenversand durch die GFOS-Tasks aus $k+1$ unterbrochen werden. Das Verschieben kann konkurrierende Zugriffe zwischen schreibenden und lesenden Zugriffen hervorrufen. Wird beispielsweise das Senden von Nachricht 1 verschoben, kann das Schreiben der entsprechenden Daten parallel zu dem Lesen der Daten von Nachricht 0 stattfinden. Dies kann eine Konkurrenz zwischen Motor-Controller 1 und Motor-Controller 2 verursachen.

Werden unterschiedliche Motor-Controller eingesetzt, gelten auch hier die Annahmen aus Abschnitt 7.3.2. Komplexere Kommunikationsmuster bedingen jeweils eine entsprechende Betrachtung. Durch die nebenläufige Ausführung kann eine Kommunikation lediglich zwischen zwei Iterationen der GFOS stattfinden, sodass Zugriffe auf gemeinsamen Speicher für alle Kommunikationsmuster auf diesen Zeitraum eingeschränkt werden können.

Durch die Nebenläufigkeit wirken entstehende Wartezeiten bei Zugriffen auf das globale Speichermodul nicht auf die zeitlichen Eigenschaften der GFOS-Tasks. Lediglich die Laufzeiten des Nachrichtenversands sind betroffen. Dies wirkt sich auf die Latenz einer Übertragung und somit auf die maximal erreichbare Datenübertragungsrate aus.

7.4.2 Erweiterung des lokalen Speichers

Die Erweiterung des lokalen Speichers dient typischerweise dem Auslagern von vergleichsweise größeren Datenstrukturen. Die Zugriffszeiten sind durch die Anbindung der Speichermodule über eine Intra-Chip-Verbindung als auch aufgrund ihrer Architektur, regelmäßig in Form von SRAM, größer als die von lokalen Scratchpad-Speichern. Entsprechend werden eher solche Daten ausgelagert, auf die nur begrenzt oft zugegriffen wird. Wann Zugriffe geschehen, ist von der jeweiligen Anwendung abhängig und kann nicht vorausgesagt werden. Finden Zugriffe parallel statt, gelten die Beschreibungen bei parallelen Zugriffen auf I/O-Module sowie bei der Kommunikation von Motor-Controllern.

Im Gegensatz zur Kommunikation zwischen Motor-Controllern finden hier die Zugriffe auf globalen Speicher innerhalb der Ausführung von GFOS-Tasks statt. Wartezeiten durch konkurrierende Zugriffe können somit die zeitlichen Eigenschaften der Regelung beeinflussen.

7.4.3 Einflüsse auf den zeitlichen Versatz

Bei den vorangegangenen Betrachtungen wurde jeweils davon ausgegangen, dass in jeder Iteration der GFOS der gleiche Code ausgeführt wird und somit alle Iterationen eines Kerns eine identische Laufzeit aufweisen. Durch bedingte Verzweigungen innerhalb des Quellcodes kann sich der ausgeführte Code in aufeinanderfolgenden Iterationen unterscheiden. Welche Codepfade genommen werden, ist typischerweise von den Eingangsdaten der Algorithmen abhängig. Daraus folgende Laufzeitunterschiede können auf einen zeitlichen Versatz einwirken und diesen vergrößern oder verkleinern. Somit kann sich der Versatz in aufeinanderfolgenden Iterationen unterschiedlich auf konkurrenzbedingte Wartezeiten und damit auf Laufzeitunterschiede zwischen der Ausführung in Isolation und unter Parallelität auswirken.

7.5 Messung von Cross-Core-Interferenzen

In den vorangegangenen Abschnitten wurden konkurrierende Zugriffe auf gemeinsam genutzte Module des Multi-Core-Mikrocontrollers und ihr mögliches Auftreten bei der parallelen Integration mehrere Motor-Controller beschrieben. Die entstehende Konkurrenz kann die Laufzeiten der auf den Kernen ausgeführten Algorithmen beeinflussen. Das Maß dieser Beeinflussung ist von den zeitlichen Parametern der Intra-Chip-Verbindung und der Dichte aufeinanderfolgender Zugriffe abhängig. In

den folgenden Untersuchungen werden gezielt konkurrierende Zugriffe erzeugt und resultierende Laufzeiterhöhungen gemessen. Als Referenz zur Beurteilung der Laufzeiterhöhung dient die Laufzeit der zu testenden Anwendungen in Isolation. Diese wird den Messungen bei paralleler Ausführung mit zwei und drei Kernen gegenübergestellt. Die Messungen der Laufzeiten entsprechend dem Vorgehen aus Abschnitt 4.3. Für jeden Prozessorkern wird ein separater Timer verwendet. Die Untersuchungen werden auf einem MCMC mit einem Aurix^{TM} TC29 Multi-Core-Mikrocontroller durchgeführt. Diese integrieren drei Prozessorkerne des Typs TC1.6.1 Performance. Somit muss nicht auf Performance-Unterschiede zwischen den unterschiedlichen Typen der Tricore^{TM} Prozessorkerne geachtet werden.

In den folgenden Betrachtungen werden Laufzeiterhöhungen bei verschiedenen Konstellationen parallel ausgeführter Load- (LD) und Store-Instruktion (ST) untersucht. Die Tabellen 7.1 bis 7.4 zeigen die gemessenen Laufzeiten. Eine Zeile gibt mit $\underline{LD}$ beziehungsweise $\underline{ST}$ die Instruktion an, für die die Laufzeit bestimmt wurde. Die Spalten zeigen die dazu parallel ausgeführten Instruktionen. Die Laufzeiten für $\underline{ST}$ beziehen sich auf einen vollständig gefüllten Store-Buffer.

7.5.1 Maximale Laufzeiterhöhung durch Interferenzen

Eine maximale Laufzeiterhöhung entsteht, wenn mehrere Kerne zeitgleich versuchen auf ein Modul zuzugreifen. Zeitgleiche Zugriffe können prinzipiell bei jedem Zugriffsversuch geschehen. Die maximale Wartezeit entspricht hierbei der Laufzeit der Datenphase auf der jeweiligen Intra-Chip-Verbindung (vergleiche Abbildung 7.1a t_{Modul}). Mehrere Datenphasen können typischerweise nicht parallel durchgeführt werden, sodass eine Wartezeit entsteht. Diese ist maximal, wenn die betroffenen Prozessorkerne zeitgleich versuchen, eine Datenphase zu starten (vergleiche Abbildung 7.1a Zugriff b_0). Die Werte können experimentell ermittelt werden, indem mehrere Prozessorkerne synchron gestartet werden und eine Load- oder Store-Instruktion ausführen. Bei schreibenden Zugriffen ist darauf zu achten, dass vor dem Ende der Zeitmessung eine Synchronisation zwischen den Prozessorkernen und ihren Store-Buffern ausgeführt wird. Andernfalls werden die Zugriffe im Intervall der Messung gegebenenfalls nicht berücksichtigt. Da kein taktgenauer synchroner Start mehrere Kerne vorausgesetzt werden kann, können die Startzeitpunkte um wenige Takte gegeneinander verschoben werden, um eine Differenz auszugleichen.

Tabelle 7.1 zeigt die entsprechenden Werte für den MCMC, wenn eine maximale Erhöhung eines Zugriffs stattfindet. Bezüglich Zugriffen auf I/O-Module kann jeder Prozessorkern, der parallel Load-Instruktionen ausführt, die Laufzeit von $\underline{LD}$

und *ST* jeweils um maximal 4 CPU-Takte erhöhen. Entsprechend ergibt sich für (*LD // LD // LD*) und (*ST // LD // LD*) eine maximale Laufzeiterhöhung von jeweils 8 Takten. Aus Tabelle 7.1 ergeben sich diese 8 Takte aus den Vergleichen zwischen (*LD* // -) mit (*LD // LD // LD*) und zwischen (*LD* // -) mit (*ST // LD // LD*). Konkurrierende Store-Instruktionen können Laufzeiterhöhungen von 2 (*ST // ST*) sowie 4 CPU-Takten (*LD // ST*) verursachen. Für die Einträge (*ST // ST // ST*) und (*LD // ST // ST*) ergeben sich maximal 4 beziehungsweise 8 zusätzliche CPU-Takte. Bei Zugriffen auf globalen Speicher können parallele Load-Instruktionen die Laufzeiten um maximal 5, parallele Store-Instruktionen um maximal 8 Takte erhöhen. So können sich für (*LD // LD // LD*) und (*ST // ST // ST*) Laufzeiten von maximal 26 beziehungsweise 29 CPU-Takten ergeben.

Tabelle 7.1: Maximale Laufzeiten konkurrierender Load- und Store-Instruktionen

	Laufzeit [CPU-Takte]					
	-	*LD*	*ST*	*LD // LD*	*ST // ST*	*LD // ST*
I/O-Module						
LD //	10	14	14	18	18	18
ST //	8	12	10	16	12	14
Speicher						
LD //	12	17	20	22	26	25
ST //	13	18	21	23	29	26

In den SE- und PWM-Tasks sind mehrere aufeinanderfolgende Modulzugriffe anzunehmen. Für die synchrone Ausführung drei gleichartiger Motor-Controller zeigt Tabelle 7.2, unter Verwendung unterschiedlicher GRB, die maximalen Laufzeiterhöhungen. Die Anzahl der Modulzugriffe setzt sich aus den zu lesenden ADC-Messwerten und zu setzenden PWM-Vorgaben zusammen. Zusätzlich sind für eine korrekte Nutzung der Module weitere Zugriffe notwendig. Beispielsweise für die Bedienung eines Mechanismus, der zum Erhalt der Konsistenz alle neu geschrieben PWM-Werte zeitgleich in das Modul übernimmt (Shadow-Transfer [77]). Mit einer Laufzeiterweiterung von 1,77 % und 1,16 % wirken sich konkurrierende Zugriffe auf I/O-Module geringfügig auf die Laufzeiten der GFOS mit SMO und HFCI aus. DFC muss für jede Phase insgesamt vier Spannungswerte ermitteln. Weiterhin benötigt das PWM-Modul zur Erzeugung der hier benötigten Muster eine aufwendige Konfiguration. Insgesamt sind in jeder Iteration 23 Modulzugriffe notwendig. In Kombination mit der vergleichsweise kurzen Laufzeit drücken sich diese Zugriffe in einer Laufzeiterhöhung um bis zu 5,88 % aus. Werden lediglich die FOS-Tasks ohne den

Einsatz einer GRB betrachtet, so entsteht eine Laufzeiterhöhung um bis 1,82 %.

Tabelle 7.2: Maximale Laufzeiterhöhung durch konkurrierende SE- und PWM-Tasks

GFOS	*LD*	*ST*	*Laufzeit [CPU-Takte]*	*Erhöhung [CPU-Takte] / [%]*
SMO	6	4	3622	64 / 1,77
HFCI	3	4	3453	40 / 1,16
DFC	15	8	2587	152 / 5,88
FOS	3	4	2192	40 / 1,82

7.5.2 Laufzeiterhöhung bei aufeinanderfolgenden Zugriffen

Entsprechend Abschnitt 7.2.1 stellt sich nach dem ersten konkurrierenden Zugriff ein zeitlicher Versatz ein. Hierdurch, insbesondere in Kombination mit entsprechend großen zeitlichen Abständen zwischen Zugriffsversuchen, können sich Wartezeiten verringern. Eine solche Verringerung kann für die Betrachtung der Laufzeiterhöhung für das parallele Ausführen von identischem Code herangezogen werden. Hier ist die zeitliche Relation zwischen den Zugriffen der parallelen Kerne bekannt (vergleiche Abschnitt 7.3.1). Bei nicht identischem Code können parallele Zugriffe zu jedem Zeitpunkt geschehen, sodass in diesem Fall eine maximale Laufzeiterhöhung entsprechend Abschnitt 7.5.1 angenommen werden muss.

Bei aufeinanderfolgenden Zugriffen werden die Laufzeiterhöhungen betrachtet, die sich nach dem Einstellen eines minimalen zeitlichen Versatzes ergeben. Dieser Versatz entspricht dem geringsten Abstand, den aufeinanderfolgende Modulzugriffe aufgrund der Laufzeiten von Load-/Store-Instruktionen und den Zugriffszeiten der Intra-Chip-Verbindung aufweisen können. Diese Abstände werden erzeugt, indem n Load- oder Store-Instruktionen nacheinander ausgeführt werden. Um die Laufzeiterhöhung zu messen, wird die Laufzeit bestimmt, welche die Instruktion $n + 1$ benötigt.

7.5.2.1 Zugriffe auf I/O-Module

Tabelle 7.3 zeigt die entstehenden Laufzeiten für unterschiedliche Kombinationen parallel ausgeführter Instruktionen beziehungsweise Modulzugriffe auf I/O-Peripherie des MCMC. Jeder zusätzliche Prozessorkern, der Load-Instruktionen ausführt, verursacht für $\underline{LD}$ und $\underline{ST}$ eine Laufzeiterhöhung von jeweils 2 CPU-Takten. Aus Tabelle 7.3 ergeben sich diese zusätzlichen CPU-Takte aus dem Vergleich zwischen ($\underline{LD}$ || -) mit ($\underline{LD}$ || *LD*) beziehungsweise zwischen ($\underline{ST}$ || -) mit ($\underline{ST}$ || *LD*). Die

Vergleiche von ($\underline{LD}$ || -) mit ($\underline{LD}$ || LD || LD) und ($\underline{ST}$ || -) mit ($\underline{ST}$ || LD || LD) zeigen jeweils 4 zusätzliche CPU-Takte. Konkurrierende Store-Instruktionen zeigen hier keinen Einfluss auf die Laufzeiten von $\underline{LD}$ und $\underline{ST}$.

Tabelle 7.3: Laufzeiten aufeinanderfolgender Zugriffe auf I/O-Module

	Laufzeit [CPU-Takte]					
	-	LD	ST	$LD \parallel LD$	$ST \parallel ST$	$LD \parallel ST$
$\underline{LD} \parallel$	10	12	10	14	10	12
$\underline{ST} \parallel$	8	10	8	12	8	10

Bei parallelen Store-Instruktionen findet keine zusätzliche Laufzeiterhöhung statt. Dies kann dadurch begründet werden, dass zwischen den Datenphasen auch Phasen stattfinden, die parallel ausgeführt werden können. Bezüglich der Intra-Chip-Verbindung sind dies Adressierungsphasen. Diese erhöhen die Abstände aufeinanderfolgender Datenphasen. In Kombination mit dem sich einstellenden zeitlichen Versatz entstehen hierdurch keine zeitgleichen Datenphasen und somit keine Wartezeiten.

Werden die für aufeinanderfolgende Zugriffe angepassten Laufzeiten für das in Tabelle 7.2 gezeigte Beispiel verwendet, können die Wartezeiten präzisiert werden. Durch den synchronen Start der Motor-Controller kann für die Zugriffsfolge innerhalb der SE-Tasks angenommen werden, dass auch sie synchron starten. Somit wird für den jeweils ersten Zugriff eine maximale Laufzeiterhöhung angenommen. Für die darauffolgenden Zugriffe werden Wartezeiten entsprechend Tabelle 7.3 verwendet. Durch die Konkurrenz in der SE-Task entsteht ein zeitlicher Versatz zwischen den Motor-Controllern. Es wird angenommen, dass dieser bis zur PWM-Task erhalten bleibt. Als Folge findet keine vollständige synchrone Aufführung der Zugriffe auf das PWM-Modul statt. Dementsprechend werden in der PWM-Task für alle Zugriffe die Wartezeiten aus Tabelle 7.3 verwendet. Für die GFOS mit SMO, HFCI und DFC ergeben sich Laufzeiterhöhungen von 0,77 %, 0,46 % und 2,47 %. Es ist jedoch zu beachten, dass sich die zeitliche Relation der parallelen Anwendungen über die Laufzeit hin ändern kann, beispielsweise durch das Ausführen unterschiedlicher Codepfade. So kann zum Zeitpunkt der Ausführung der PWM-Task ein vergrößerter zeitlicher Versatz existieren, durch den konkurrierende Zugriffe reduziert werden können. Entsprechend können der Versatz verkleinert und Wartezeiten erhöht werden.

7.5.2.2 Zugriffe auf gemeinsamen Speicher

Tabelle 7.4 zeigt die Laufzeiterhöhungen bei Zugriffen auf ein globales Speichermodul über die Crossbar des MCMC. Parallel ausgeführte Load-Instruktionen verursachen erst bei drei Kernen eine Laufzeiterhöhung *(LD // LD //LD)*. Diese liegt bei 3 zusätzlichen CPU-Takten. Parallele Store-Instruktionen erhöhen die Laufzeit lesender Zugriffe um 1 Takt bei *(LD // ST)* und um 9 Takte bei *(LD // ST // ST)*. Nachdem der Store-Buffer gefüllt ist, erhöhen konkurrierende schreibende Zugriffe die Laufzeit jeder zusätzlichen Store-Instruktion um 3 *(ST // ST)* und 11 Takte *(ST // ST // ST)*. Durch diese Laufzeiterhöhung wird auch das Abarbeiten des Buffer verlangsamt, sodass er bei zwei aktiven Kernen bereits nach 5, bei drei Kernen nach 4 Store-Instruktionen vollständig gefüllt ist. Die Laufzeit der Store-Instruktionen steigt somit früher. Konkurrierende Load-Instruktionen beeinflussen die Laufzeit von Store-Anweisungen um 3 *(ST // LD)* und 6 *(ST // LD // LD)* CPU-Takte.

Tabelle 7.4: Laufzeiten aufeinanderfolgender Zugriffe auf globalen Speicher

	Laufzeit [CPU-Takte]					
	-	*LD*	*ST*	*LD // LD*	*ST // ST*	*LD // ST*
LD //	12	12	13	15	21	19
ST //	13	16	16	19	24	21

Wird gemeinsamer Speicher für den Austausch von Nachrichten zwischen konsolidierten Systemen eingesetzt, so können sich Interferenzen auf die maximale Datenübertragungsrate und die Latenz der Nachrichten auswirken. Für die Übertragung eines 4 Byte großen Datenwortes inklusive dem Schreiben und Lesen der Daten in und aus dem globalen Speicher 1250 CPU-Takte benötigt (siehe Tabelle 4.1). Dies entspricht einer Datenübertragungsrate von 625 KiB/s. Die Dauer der reinen Nachrichtenübertragung ist unabhängig von der Anzahl der zu übertragenden Nutzdaten, wodurch die Datenübertragungsrate mit zunehmender Nutzdatengröße wächst. Beispielsweise beträgt die Übertragungsrate für 8 Datenworte (32 Byte, 1460 CPU-Takte) 4280 KiB/s. Interferenzen vergrößern die Zeit, die zum Schreiben und Lesen der Daten in und aus dem Nutzdatenspeicher benötigt wird. Bei drei parallelen Kernen und einer maximalen Wartezeit für den jeweils ersten schreibenden und lesenden Zugriff kann die Übertragungsdauer eines Wortes um 1,28 % und von 8 Wörtern um 7,24 % zunehmen. Folglich sinkt die maximal erreichbare Datenübertragungsrate.

7.5.3 Veränderung zeitlicher Abstände aufeinanderfolgender Modulzugriffsversuche

Wie in Abschnitt 7.2.2 beschrieben, können unterschiedliche Faktoren die zeitlichen Abstände zwischen aufeinanderfolgenden Modulzugriffsversuchen und damit entstehende Wartezeiten beeinflussen [163].

7.5.3.1 Unterschiedliche Zugriffsmuster

Auf Ebene des Quellcodes werden Modulzugriffe durch Zugriffsmuster der jeweils verwendeten Programmiersprache formuliert. Im Vergleich zu einem direkten Zugriff über eine Load-Instruktion erzeugen diese Muster mehrere Instruktionen. Bei wiederholenden Zugriffen vergrößern diese den Abstand zwischen aufeinanderfolgenden Load-Instruktionen. Werden einfache Variablen zum Zugriff auf globalen Speicher verwendet, wird, im Vergleich zur direkten Verwendung von Load-Instruktionen, die Laufzeit für das Lesen aus globalem Speicher um 5 auf 17 Takte pro Zugriff erhöht (siehe Tabelle 7.4 (*LD* || -) zuzüglich 5 Takte). Hierdurch entsteht zwischen aufeinanderfolgenden Modulzugriffen ein zeitlicher Abstand, der ausreichend groß ist, sodass auch bei drei parallel ausgeführten lesenden Speicherzugriffen (*LD* // *LD* // *LD*) keine Konkurrenz um den Modulzugriff entsteht. Somit bleibt die Laufzeit pro Zugriff unverändert bei einem Wert von 17 CPU-Takten.

Ähnliches gilt für lesende Zugriffe auf I/O-Module. Durch die erhöhten zeitlichen Abstände der Modulzugriffe wird die konkurrenzbedingte Laufzeiterhöhung reduziert. Für zwei konkurrierende Zugriffe (*LD* // *LD*) entfällt dadurch die Laufzeiterhöhung vollständig. Für drei konkurrierende Zugriffe wird die Laufzeiterhöhung von 4 auf 2 CPU-Takte reduziert.

Die Laufzeit von schreibenden Zugriffen erhöht sich durch eine Zuweisung von Daten an eine Variable um 3 Takte. Konkurrierende schreibende Zugriffe profitieren an dieser Stelle jedoch nicht davon. Die Laufzeit zur Ausführung der zusätzlichen Instruktionen ersetzt einen Teil des Wartens auf einen freien Platz im Store-Buffer. Somit bleiben insgesamt die Laufzeiten bei vollständig gefülltem Buffer und damit die Laufzeiterhöhungen unverändert. Dies gilt für Zugriffe auf Speicher als auch auf I/O-Module.

Für die in Abschnitt 7.5.2 ermittelten Laufzeiterhöhungen der unterschiedlichen GFOS bedeutet dies, dass sich die Laufzeiterhöhung der lesenden Zugriffe auf I/O-Module um 2 Takte pro Zugriff verringert. Eine Ausnahme ist der erste Zugriff, da dieser auf allen Motor-Controller synchron stattfindet. Dies konnte mittels ei-

ner Laufzeitmessung bestätigt werden. Für den Prozessorkern mit der niedrigsten Priorität bezüglich der Arbitrieung am Peripheriebus wurde eine Laufzeiterhöhung von 0,50 % bei SMO, 0,35 % bei HFCI und von 1,39 % bei Verwendung des DFC-Algorithmus als GRB gemessen. Für den Kern mit der höchsten Priorität entfällt jeweils die Laufzeiterhöhung für den ersten Zugriff, sodass sich hier Werte von 0,28 %, 0,12 % und von 1,08 % ergeben.

7.5.3.2 Umstrukturierung von Quellcode

Ähnliche Effekte kann eine Umstrukturierung von Quellcode hervorrufen. Am Beispiel der PWM-Task kann eine solche Umstrukturierung für das Schreiben neuer PWM-Vorgaben in das entsprechende I/O-Modul gezeigt werden. Diese Konfiguration des Moduls wird typischerweise am Ende der Task durchgeführt, wenn alle Daten berechnet wurden. Eine Synchronisation zwischen Store-Buffer und Prozessorkern beendet den Vorgang. Entsprechend kann eine Konkurrenz zwischen den abzuarbeitenden Store-Buffer-Jobs zur Konfiguration der PWM-Kanäle mehrere Motor-Controller entstehen. Hier kann eine Umstrukturierung stattfinden, indem das I/O-Modul schrittweise konfiguriert wird. Einzelne Daten können geschrieben und damit in den Buffer eingefügt werden, sobald sie berechnet wurden. Das Abarbeiten der Jobs kann auf diese Weise bereits während der Ausführung des verbleibenden Codes durchgeführt werden. Folglich können die abschließende Synchronisation verkürzt und Wartezeiten signifikant reduziert werden [163].

7.5.3.3 Compiler-Optimierung

Bei der Übersetzung von Quell- in Maschinencode stellen Compiler Optimierungsmethoden zur Verfügung, um die Laufzeit zu verringern. Die Optimierung beeinflusst die vom Compiler erzeugten Instruktionen. Die hierbei erreichte Verringerung der Laufzeit bedeutet jedoch, dass Zugriffe auf gemeinsame Speicher und I/O-Module zeitlich dichter beisammen liegen. Dementsprechend kann sich eine Optimierung auch auf Cross-Core-Interferenzen auswirken. Dies wird in Anlehnung an die parallele Matrixmultiplikation aus Abschnitt 6.3.5 gezeigt.

Es werden zwei beziehungsweise drei konsolidierte gleichartige Motor-Controller betrachtet, die alle eine vollständige Matrixmultiplikation durchführen. Die zu multiplizierende Matrix A sowie die Ergebnismatrix liegen in einem gemeinsamen Speicherbereich. Alle Kerne kopieren zunächst Matrix A in ihren lokalen Arbeitsspeicher. Nach den Berechnungen werden die Ergebnisse in den globalen Speicher zurückgeschrieben. Durch das lokale Zwischenspeichern der Werte werden die Zugriffe auf

das globale Speichermodul minimiert. Während den Berechnungen der Matrixmultiplikation finden keine Zugriffe auf gemeinsamen Speicher statt. Konkurrenz und das Entstehen einer Laufzeiterhöhung beschränken sich somit auf die Kopiervorgänge.

Der verwendete Tricore™ GCC-Compiler stellt die für C/C++ Compiler typischen Optimierungsoptionen [47] zur Verfügung. Zur Durchführung der Evaluation werden die Optionen *O0* für eine nicht optimierte Übersetzung und *Os* für eine Optimierung der Laufzeit verwendet. Tabelle 7.5 zeigt die Ergebnisse für unterschiedlich große Matrizen. Es sind jeweils die Laufzeiten der vollständigen Multiplikation und die Laufzeiterhöhung durch Konkurrenz angegeben.

Tabelle 7.5: Effekt einer Compiler-Optimierung auf konkurrierende Zugriffe. Unter Verwendung der Optimierung GCC Os ist ein deutlicher Anstieg von Cross-Core-Interferenzen zu beobachten.

	Laufzeit [CPU-Takte]		Laufzeiterhöhung [%]			
	1 Kern		2 Kerne		3 Kerne	
	O0	Os	O0	Os	O0	Os
2x2	912	138	0,44	4,35	0,55	18,12
4x4	5002	1031	0,08	3,98	0,10	15,42
6x6	17854	3192	0,02	3,20	0,03	11,72
12x12	129332	19526	0,0	2,19	0,0	8,05

Die Ergebnisse zeigen, dass konkurrierende Zugriffe einen deutlich höheren Einfluss auf die jeweilige Referenzlaufzeit nehmen, wenn eine Compiler-Optimierung verwendet wird. Ohne die Verwendung einer Optimierung konkurrieren bei zwei als auch drei aktiven Kernen lediglich die ersten Zugriffe miteinander. Durch den darauf folgenden zeitlichen Versatz entsteht keine weitere Konkurrenz. Mit Optimierung kann bei zwei aktiven Kernen eine Laufzeiterhöhung von bis zu 4,35 %, bei drei aktiven Kernen von bis zu 18,12 % gemessen werden. Mit zunehmender Größe der Matrizen nimmt die für die Berechnungen der Multiplikation benötigte Laufzeit stärker zu, als die für das Kopieren der Daten benötigte Zeit. Da Laufzeiterhöhungen lediglich durch die Kopiervorgänge hervorgerufen werden, sinkt mit zunehmender Größe der Matrizen ihr Anteil an der Gesamtlaufzeit.

7.5.4 Verschieben von Startzeitpunkten

Abschnitt 7.3.1 zeigt den Einsatz eines zeitlichen Versatzes ΔStart für den parallelen Start gleichartiger konsolidierter Motor-Controller. Abbildung 7.4 verdeutlicht hier-

bei das Serialisieren paralleler Modulzugriffe durch ΔStart. In dem vergleichsweise einfachen Beispiel mit eng zusammen liegenden Modulzugriffen kann die Größe von ΔStart über t_{Modul} aus der Summe aller benötigten Zugriffszeiten gewonnen werden. Bei vielen und über die gesamte Laufzeit der Anwendungen verteilten Zugriffen ist ihre vollständige Serialisierung und einfache Bestimmung von ΔStart unwahrscheinlich.

In der folgenden Untersuchung wird der Einfluss von ΔStart für zwei Szenarien betrachtet: bei nahezu vollständig ausgelasteten Prozessorkernen und bei freien Rechenkapazitäten. Die hierzu verwendeten Motor-Controller greifen in ihren SE- und PWM-Tasks auf globale I/O-Module zu. Zugriffe auf globalen Speicher werden über alle Tasks verteilt, indem die zwischen den Tasks fließenden Daten in globalem Speicher gehalten werden. Dies dient einer späteren Übertragung der Daten zu Kern 2, der diese weiterverarbeitet. Als GRB wird der DFC-Algorithmus eingesetzt. Bei der Implementierung der Motor-Controller wurde eine Softwarestruktur genutzt, die eine vereinfachte Integration und Synchronisation der GRB- und FOS-Tasks sowie einer Task zur Datenübertragung an Kern 2 ermöglicht [164]. Hierbei werden Kommunikations- und Synchronisationsmechanismen des Betriebssystems eingesetzt. Im Vergleich zu den in Abschnitt 6.3 ermittelten Zeiten vergrößern diese Mechanismen die Laufzeiten der Tasks. Für die GRB liegt hier die Laufzeit bei 1385, für die vollständige GFOS bei 6923 CPU-Takten. Die GRB-Task wird nebenläufig zu den FOS-Tasks mit $f_{GRB} = 2 \cdot f_{FOS} = 40kHz$ ausgeführt. Bei einer Taktrate von 200 MHz lasten die Motor-Controller die Prozessorkerne nahezu vollständig aus.

Abbildung 7.7 beschreibt den Verlauf der durchschnittlichen Laufzeiterhöhung, wenn ΔStart mit einer Auflösung von 50 CPU-Takten variiert wird. Die gestrichelte Linie beschreibt die Laufzeiterhöhung für die Ausführung der vollständigen GFOS. Die zeitliche Verschiebung der Anwendung entspricht hierbei Abbildung 7.4. Kern 0 wird in Bezug auf Kern 1 verzögert gestartet. Eine Iteration von Kern 0 kann somit zwei Iterationen von Kern 1 tangieren und umgekehrt. Mit der verwendeten Auflösung von ΔStart wurde eine maximale Laufzeiterhöhung von 9,59 % bei einem Versatz von 400 Takten und eine minimale Erhöhung von 3,82 % bei 1300 Takten gefunden. Im weiteren Verlauf bewegt sich die Laufzeitverzögerung in einem Bereich zwischen 4 % und 6 % [164]. Für größer werdende ΔStart bleibt die Verzögerung in diesem Bereich. Nähert sich ΔStart T_{GRB}, kann ein erneutes Ansteigen der Laufzeiterhöhung beobachtet werden, da sich die Startzeitpunkte der GRB erneut annähern.

Für die Betrachtung der Laufzeiterhöhung der GRB-Task wird diese ohne Regelung durch eine FOS in Form einer offenen Regelschleife ausgeführt [159]. Entspre-

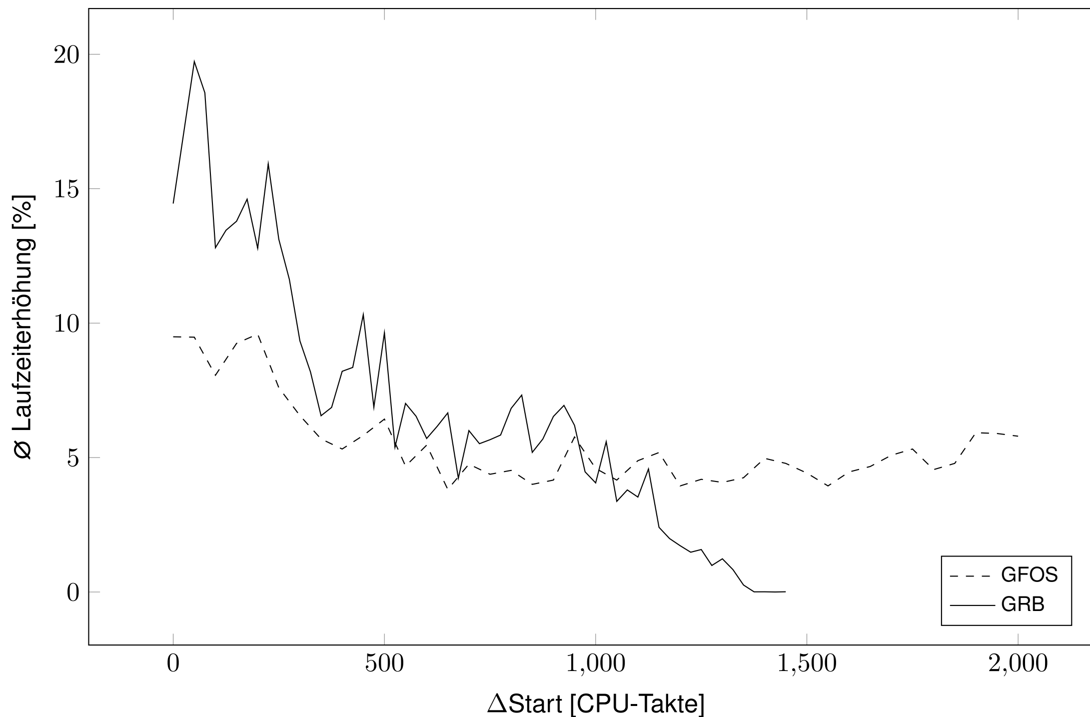


Abbildung 7.7: Einfluss von ΔStart auf die Laufzeiterhöhung. Mit zunehmendem ΔStart ist eine deutliche Abnahme der Laufzeiterhöhung zu beobachten.

chend werden lediglich die Laufzeiterhöhungen gemessen, die durch das parallele Ausführen der GRB-Task entstehen. Bei den gegebenen Werten für f_{GRB} und t_{GRB} werden die Prozessorkerne zu 55,4 % ausgelastet, sodass zwischen zwei Iterationen eine Idle-Zeit entsteht. Durch ΔStart wird die Ausführung der GRB-Task sukzessive in Richtung der Idle-Zeit des benachbarten Kerns verschoben (vergleiche Abbildung 7.3).

Auch hier zeigt sich das Maximum der gemessenen Laufzeiterhöhung (19,7 %) nahe der synchronen Ausführung bei einem Versatz von 100 Takten. Durch die über die Laufzeit verteilten Modulzugriffe zeigen die Laufzeiterhöhungen auch hier eine deutliche Varianz bei wachsendem ΔStart. Gleichzeitig nehmen, aufgrund der relativ großen Idle-Zeit der Prozessorkerne, die parallele Ausführung und damit die konkurrierenden Zugriffe ab. Spätestens bei $\Delta Start = t_{GRB}$ entsteht somit keine Laufzeiterhöhung mehr.

7.6 Zusammenfassung der Ergebnisse

Die Konsolidierung von Motor-Controllern beschreibt eine zentrale Anwendung von MCMC. Ein Nachteil sind hier Cross-Core-Interferenzen, die das zeitliche Verhalten

der integrierten Motor-Controller beeinflussen können. Allgemein sind Interferenzen schwer zu bestimmen, da oftmals nicht bekannt ist, wann konkurrierende Zugriffe auf gemeinsame Module stattfinden. Im betrachteten Anwendungsgebiet kann diese Einschränkung durch unterschiedliche Faktoren aufgehoben werden. Die betrachteten Motor-Controller beziehungsweise deren Implementierung kann auf das Taskmodell abgebildet werden. Hierdurch ist bekannt, welche Tasks auf globale Module zugreifen können und wann sie dies regelmäßig tun. Es findet eine statische Integration der Motor-Controller statt, das heißt, es existiert kein dynamischer Austausch von Tasks auf den Prozessorkernen. Weiterhin ist die Ausführung der Motor-Controller, basierend auf den die jeweiligen Antriebe ansteuernden PWM-Signalen, zeitgesteuert. Die zeitliche Relation der Signale unterschiedlicher Antriebe kann konfiguriert werden, sodass auch die zeitliche Relation zwischen den parallel ausgeführten Motor-Controllern zueinander beziehungsweise derer Tasks bekannt ist.

Auf dieser Basis wurde ausgehend von der Konsolidierung identischer Motor-Controller beschrieben, wann parallele Zugriffe auf I/O-Module und auf gemeinsamen Speicher stattfinden können. Hierbei wurden die zeitlichen Abstände zwischen den Modulzugriffsversuchen und der Einfluss von konkurrenzbedingten Wartezeiten auf die parallele Ausführung der Zugriffe einbezogen. Beide Faktoren ermöglichen eine Abschätzung von entstehenden Wartezeiten, die bei blockweisen Zugriffen auf gemeinsame Module entstehen können. Auf I/O-Peripherie entstehen diese Zugriffe durch die Tasks zur Signalerfassung und PWM-Generierung. Für die betrachteten Motor-Controller kann durch konkurrierende Zugriffe dieser Tasks eine Laufzeiterhöhung zwischen 0,5 % und 6 % entstehen. Für alternative Anwendungen, die auf dem Taskmodell basieren, sind Werte in der gleichen Größenordnung anzunehmen. Da bei diesen Betrachtungen minimale Abstände zwischen aufeinanderfolgenden Zugriffen angenommen wurden, zeigen Laufzeitmessungen deutlich geringere Werte. Entsprechendes wurde für Speicherzugriffe im Rahmen einer Kommunikation zwischen Motor-Controllern durchgeführt. Bei einer Datenübertragung von bis zu 32 Byte können Interferenzen die erreichbare Datenübertragungsrate in einer Größenordnung von 1 % bis 8 % mindern.

Diese Abschätzungen konnten durchgeführt werden, da die jeweiligen Zugriffe eng im Quellcode der Anwendungen zusammenliegen und dadurch die Laufzeiten zwischen ihnen bekannt sind. Weiterhin wurde die zeitliche Relation zwischen den einzelnen Systemen als bekannt angenommen. Wird jedoch globaler Arbeitsspeicher genutzt, so treten Modulzugriffe über die gesamte Laufzeit der Anwendung auf. Laufzeiten zwischen einzelnen Zugriffen und damit auch die zeitliche Relati-

on zwischen den Systemen sind in diesem Fall nicht bekannt und müssen für alle Zugriffsversuche ermittelt werden. Ist dies möglich, können zudem die Einflüsse variierender Implementierungen und Parameter der parallelen Ausführung auf Laufzeiterhöhungen untersucht werden. Es wurde gezeigt, dass dies für Anwendungen mit vielen Zugriffen auf globalen Speicher sinnvoll ist. Hier treten Laufzeiterhöhungen im zweistelligen Bereich auf, die durch Variationen der genannten Faktoren positiv beeinflusst werden können.

Um den Laufzeitunterschied zwischen der Single- und Multi-Core-Ausführung zu reduzieren, ist es sinnvoll, eine Implementierung zu wählen, in der eine möglichst große Laufzeit zwischen einzelnen Modulzugriffen realisiert wird. Solche Maßnahmen können eine Erhöhung der absoluten Laufzeit verursachen. Hier ist abzuwägen, ob die Verringerung der Gefahr von Interferenzen nach einer Integration in ein Multi-Core-System diese Kosten aufwiegt. An dieser Stelle ist die Verwendung einer Compiler-Optimierung hervorzuheben. Die Abstände zwischen Modulzugriffen können im optimierten Code signifikant abnehmen. Diesbezüglich konnte für optimierten Code eine Zunahme von Interferenzen im zweistelligen Prozentbereich gemessen werden.

Weiterhin ist eine definierte zeitliche Relation zwischen den Startzeitpunkten der integrierten Motor-Controller eine effektive Möglichkeit, um Interferenzen ohne eine Beeinflussung der Implementierung zu verringern. Die Beschreibungen der Tasks mit Hardwarezugriffen sowie die unterschiedlichen Integrationsmöglichkeiten der Motor-Controller zeigen, wie diese Relation gewählt werden kann, um einen positiven Effekt zu erzeugen.

Kapitel 8

Analytische Betrachtung von Cross-Core-Interferenzen

Im Rahmen der Konsolidierung dedizierter Motor-Controller wurde im vorherigen Kapitel beschrieben, wann Cross-Core-Interferenzen bei Zugriffen auf gemeinsame I/O-Peripherie- und globale Speichermodule auftreten können. Die Größenordnung grundlegender Wartezeiten wurde experimentell für unterschiedliche Szenarien auf dem MCMC ermittelt. Bei der experimentellen Ermittlung der Wartezeiten besteht prinzipiell die Gefahr, dass Interferenzen das Echtzeitverhalten der Anwendungen negativ beeinflussen. Ein Übertreten der zeitlichen Schranken kann die korrekte Funktion eines Antriebes verhindern und diesen beschädigen. Weiterhin ist die Ermittlung zeitaufwendig, vor allem dann, wenn unterschiedliche Parameter getestet werden sollen. Durch die Umstellung von Quellcode oder das Einbringen eines zeitlichen Versatzes entstehen zahlreiche Varianten, die getestet werden können beziehungsweise müssen.

Auf Basis des in Kapitel 7 beschriebenen Verhaltens konkurrierender Zugriffe können für Motor-Controller, die auf dem Taskmodell basieren, die entstehenden Wartezeiten berechnet werden [162]. Ausgehend von der Anwendungsausführung der Controller auf einem Single-Core-System können unterschiedliche Konfigurationen einer parallelen Ausführung betrachtet werden, ohne die Controller auf der Multi-Core-Plattform ausführen zu müssen. Dies ermöglicht eine effiziente Analyse einer Vielzahl von Integrationsparametern der zu konsolidierenden Systeme.

Abbildung 8.1 fasst das Vorgehen zum Berechnen von Cross-Core-Interferenzen beziehungsweise der daraus resultierenden Laufzeiterhöhungen am Beispiel von zwei Motor-Controllern zusammen. Diese sind auf den Prozessorkernen c_0 und c_1 integriert. Als Eingabedaten der Interferenzanalyse dienen Instruction-Traces, die für

jede zu integrierende Anwendung aufgenommen werden. Jeder Trace beinhaltet alle Prozessorinstruktionen, die ein Prozessorkern bei der Ausführung einer gegebenen Anwendung abarbeitet. Ein Trace wird aufgenommen, wenn die Anwendung auf dem MCMC in Isolation oder einem Single-Core-System mit identischer Prozessor- und Peripheriearchitektur ausgeführt wird. Zeitliche Informationen innerhalb der Traces werden somit nicht durch Interferenzen beeinflusst.

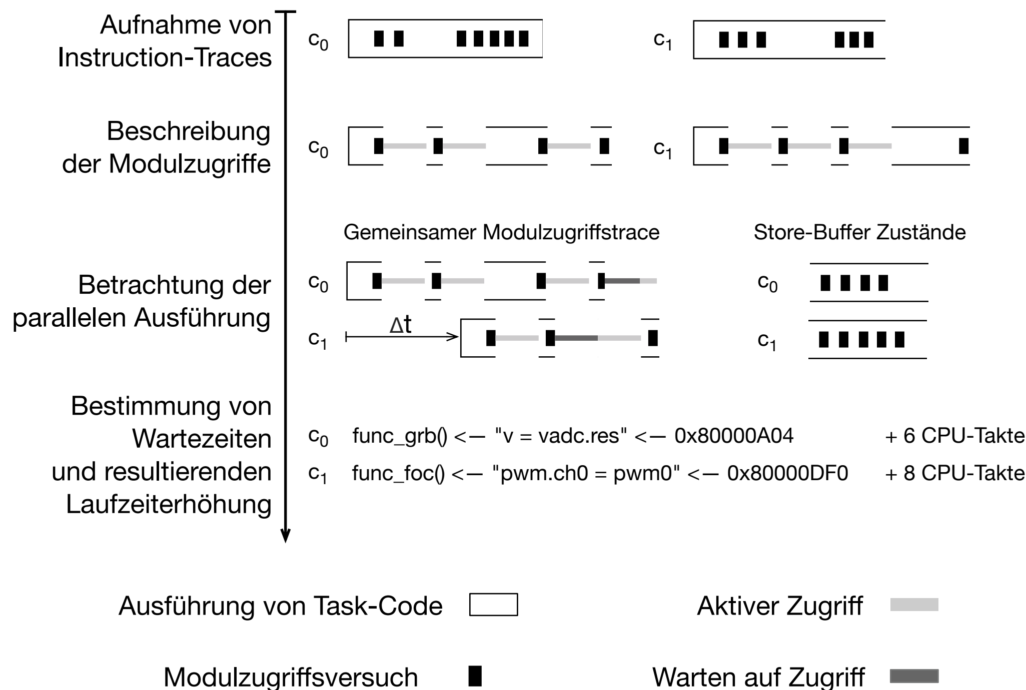


Abbildung 8.1: Ablauf der analytischen Interferenzbestimmung. Das Verfahren beginnt mit der Aufzeichnung von Instruction-Traces und endet mit der Berechnung von Wartezeiten und der daraus resultierenden Laufzeiterhöhung.

Ein Instruction-Trace ist eine Liste von Instruktionen, die von einem Prozessorkern innerhalb eines definierten zeitlichen Intervalls ausgeführt wurden. Load-Instruktionen zum Lesen und Store-Instruktionen zum Schreiben von Daten können Zugriffe auf gemeinsame Module verursachen. Die Zeiten dieser Zugriffe werden in Form eines Modulzugriffstrace beschrieben. Auf Basis der Modulzugriffstraces von mehreren Teilsystemen werden konkurrierende Zugriffe auf gemeinsame Module ermittelt und daraus resultierende Wartezeiten für die Prozessorkerne abgeleitet. Es wird angenommen, dass die Teilsysteme periodisch ausgeführt werden und der zeitliche Versatz $\Delta Start$ zwischen den Startzeitpunkten der jeweiligen Perioden bekannt beziehungsweise konfigurierbar ist.

Die Wartezeiten können die Laufzeit der mit den Zugriffen korrespondierenden Instruktionen und dementsprechend die Laufzeit der gesamten Anwendung erhöhen.

Um eine Laufzeiterhöhung zu bestimmen, müssen die Instruktionen ermittelt werden, auf die sich Wartezeiten auswirken. Ist ein gemeinsames Modul zum Zeitpunkt der Ausführung einer Load-Instruktion auf dieses Modul belegt, blockiert der Prozessorkern solange, bis das Modul freigegeben wurde und der Zugriff durchgeführt werden kann. Dieses Blockieren drückt sich in einer Erhöhung der Laufzeit dieser Instruktion aus. Store-Instruktionen greifen nicht direkt auf ein Modul zu. Sie fügen Jobs zum Schreiben von Daten in den Store-Buffer des Prozessorkerns ein. Der Buffer arbeitet diese Jobs parallel zur Instruktionsausführung des Kerns ab. Wartezeiten beeinflussen die Zeit, bis ein Job aus dem Store-Buffer abgearbeitet wurde. Das Leeren des Store-Buffer wird dadurch verlangsamt. Dies wirkt sich dann auf die Laufzeit von Store-Instruktionen aus, wenn der Buffer zum Zeitpunkt des Ausführens einer Store-Instruktion vollständig gefüllt ist. In diesem Fall muss vor dem erfolgreichen Einfügen gewartet werden, bis ein Job aus dem Buffer entfernt wurde. Auch Store-Instruktionen, die Daten in lokale Speicher schreiben, fügen Jobs in den Store-Buffer ein. Somit kann auch die Laufzeit dieser Instruktionen durch Interferenzen beeinflusst werden. Um das Wirken von Wartezeiten auf konkrete Instruktionen zurückführen zu können, müssen, neben der Bestimmung eines für alle Teilsysteme gemeinsamen Modulzugriffstrace, die Zustände der lokalen Store-Buffer aller Prozessorkerne zu jedem Zeitpunkt der Anwendungsausführung betrachtet werden.

8.1 Aufbau des Instruction-Trace

Ein Instruction-Trace $o_i = [\iota_0, \iota_1, ..., \iota_{n-1}]$ ist eine Liste von n Instruktionen, die auf einem Prozessorkern c_i ausgeführt wurden. Die Instruktionen sind nach dem Zeitpunkt ihrer Ausführung sortiert. Eine Anweisung hat die Form $\iota = (x, t, y, \Delta)$ und ist eindeutig über ihre Adresse $\iota.x$ im Arbeitsspeicher des Mikrocontrollers identifizierbar. Sie kann mehrfach ausgeführt werden, wodurch sie an unterschiedlichen Stellen im Instruction-Trace erscheint. Mehrfache Ausführungen unterscheiden sich durch den Zeitpunkt der Ausführung $\iota.t$. Der Typ einer Anweisung wird durch $\iota.y$ beschrieben. Die Typen *r[m]* (read) und *w[m]* (write) stehen für eine lesende beziehungsweise schreibende Anweisung. Hierbei bezeichnet m die Zieladresse eines Zugriffs. Über den Adressraum, in dem m liegt, können Zugriffe auf lokalen Scratchpad-Arbeitsspeicher (m_l) sowie auf globale Speicher- und I/O-Module (m_g) unterschieden werden. Befehle des Typs e (execute) arbeiten ausschließlich mit Prozessorregistern ohne direkte Zugriffe auf Speicher- oder I/O-Module. Sie umfassen beispielsweise arithmetische Berechnungen oder Befehle zur Steuerung des Kontrollflusses.

Die Dauer der Ausführung einer Instruktion ist durch $\iota.\Delta$ beschrieben. Diese Laufzeiten können typischerweise dem Datenblatt des Mikrocontrollers beziehungsweise der Beschreibung von dessen Mikroarchitektur entnommen werden. Entsprechende Zeiten beschreiben im Allgemeinen die maximalen Laufzeiten der Befehle. Durch parallele Pipelines kann die real benötigte Zeit davon abweichen. Die tatsächliche Laufzeit kann für jede Instruktion direkt aus dem Instruction-Trace abgelesen werden. Wurde eine Anweisung ι_k zum Zeitpunkt $\iota_k.t$ aufgezeichnet, kann deren Laufzeit über $\iota_{(k+1)}.t - \iota_k.t$ bestimmt werden. Bei Befehlen des Typs *r[m]* umfasst der berechnete Wert die Zeit, die zur Bereitstellung der Daten durch die Schnittstelle des Zielmediums benötigt wird. Durch die Verwendung von Store-Buffern zur Realisierung der out-of-order Ausführung werden Daten bei der Abarbeitung einer Instruktion des Typs *w[m]* nicht direkt in Speicher oder Register des Zielmediums geschrieben. Stattdessen werden die Daten im Store-Buffer abgelegt. Ist der Buffer nicht vollständig gefüllt, geschieht dies mit einer konstanten Laufzeit. Die aus dem Instruction-Trace berechnete Laufzeit einer Anweisung des Typs *r[w]* umfasst die Zeit für das Schreiben in den Store-Buffer.

8.2 Aufbau des Modulzugriffstrace

Die Berechnung der Wartezeiten basiert auf den Sequenzen von Modulzugriffen, welche die parallel auszuführenden Anwendungen durchführen, wenn sie jeweils in Isolation ausgeführt werden. Diese Zugriffe werden für jeden Prozessorkern in Form eines Modulzugriffstrace $mzt_i = [b_0, ..., b_{k-1}]$ betrachtet. Der Trace ist eine geordnete Liste, die alle Zugriffe des Kerns c_i auf globale Module beinhaltet. Ein Modulzugriff wird durch $b = (y, t_{req}, t_{acc}, t_{rel}, \gamma)$ beschrieben. Das Attribut $b.y$ beschreibt, analog zum Typ einer Instruktion, den Typ des Modulzugriffs. Ein Modulzugriff beginnt mit einer Zugriffsanfrage zum Zeitpunkt $b.t_{req}$ (access request). Bei lesenden Zugriffen wird die Anfrage durch den Prozessorkern und bei schreibenden durch den Store-Buffer gestellt. Ist die Intra-Chip-Verbindung nicht durch eine andere Transaktion belegt, so erhält die anfragende Instanz zum Zeitpunkt $b.t_{acc}$ (access) Zugriff und kann die Datenübertragung durchführen. Ist die Verbindung belegt, so wird aktiv und unterbrechungsfrei auf deren Freigabe gewartet. Aus der Sicht eines Prozessorkerns kann die Intra-Chip-Verbindung durch schreibende Transaktionen ausgehend vom eigenen und von benachbarten Store-Buffern als auch von lesenden Transaktionen anderer Kerne belegt sein. Entsprechendes gilt aus der Sicht von Store-Buffern. Eine Transaktion wird mit der Freigabe der Verbindung zum Zeitpunkt $b.t_{rel}$ (access

release) beendet. Das Attribut $b.\gamma$ beschreibt den Betrag der Wartezeit, die für b durch konkurrierende Modulzugriffe entstehen kann.

Viele Mikrocontroller bieten neben der Aufzeichnung der ausgeführten Anweisungen auch die Möglichkeit, die Zeitpunkte der Modulzugriffe aufzuzeichnen. Liegen solche Informationen vor, müssen die einzelnen Zugriffe den Instruktionen zugeordnet werden, die sie ausgelöst haben. Konkrete Vorgehensweisen hierzu sind abhängig von den Daten, die in den Instruction-Traces und Modulzugriffstraces vorhanden sind. Können die Modulzugriffe nicht direkt aufgezeichnet werden, müssen sie aus dem Instruction-Trace abgeleitet werden. Die Sequenz für lesende Modulzugriffe ergibt sich aus der Reihenfolge der korrespondierenden Load-Instruktionen. Durch ihre out-of-order Ausführung gilt dies nicht für Store-Anweisungen. In Relation zur entsprechenden Store-Instruktion geschieht hier der Modulzugriff zu einem späteren Zeitpunkt. Dieser ist abhängig von der Anzahl der sich im Store-Buffer befindenden Jobs und der Menge von lesenden Anweisungen, die auf dem entsprechenden Kern nach der betrachteten Store-Instruktion ausgeführt werden. Lesende Zugriffe des Prozessorkerns werden standardmäßig vorrangig behandelt. Dementsprechend können lesende Modulzugriffe vor schreibenden ausgeführt werden, auch wenn die Load-Instruktion zeitlich nach der entsprechenden Store-Instruktion ausgeführt wird. Daher müssen bei der Bestimmung der Modulzugriffe die Store-Buffer-Zustände zu den Zeitpunkten der Ausführung von Load- und Store-Instruktionen bestimmt werden, um die korrekte Reihenfolge der Modulzugriffe abzuleiten.

8.2.1 Zugriffe auf Speicher und Caches

Jeder Kern verfügt über lokale Scratchpad-Speichermodule und Cache-Speicher, auf die er über eine entsprechende Schnittstelle exklusiven Zugriff hat. Über eine Intra-Chip-Verbindung (Crossbar) sind alle Kerne an globale Arbeits- und Flash-Speicher angebunden. Jedem Speichermodul ist ein eindeutiger Adressbereich zugewiesen, über den auf das Modul zugegriffen werden kann. Zugriffe auf Cache-Speicher erfolgen implizit, indem auf den Adressbereich eines Speichers zugegriffen wird, dem ein Cache vorgeschaltet ist.

Zugriffe auf globalen Arbeitsspeicher werden ohne die Verwendung von Cache-Speichern ausgeführt. Load- und Store-Instruktionen führen zu einem direkten Zugriff auf das adressierte Speichermodul und somit zu einer Transaktion über die Crossbar. Zugriffe auf lokalen Arbeitsspeicher werden ebenfalls ohne Caches durchgeführt, da lokale Scratchpad-Arbeitsspeicher Zugriffszeiten im Bereich von Caches realisieren. Allen Zugriffen auf Flash-Speicher werden lokale Instruction-Caches vor-

geschaltet. Es wird davon ausgegangen, dass die Kapazität des Cache ausreichend groß ist, um alle Tasks der GFOS und verwendeten Funktionen des Betriebssystems aufzunehmen. Ein Zugriff auf den Adressbereich des Flash-Speichers resultiert nach dem erstmaligen Befüllen des Cache nicht in einer Transaktion auf der Intra-Chip-Verbindung. Stattdessen wird über eine lokale Speicherschnittstelle auf den Cache zugegriffen. Konkurrierende Zugriffe auf die Flash-Module sind somit ausgeschlossen. Kann dies nicht angenommen werden, so müssen Informationen bezüglich der Cache-Inhalte ermittelt und zum Aufbau der Modulzugriffstraces hinzugezogen werden. Hierbei müssen Instruktionen identifiziert werden, durch die ein Cache-Miss verursacht und ein Zugriff des Cache auf das globale Speichermodul ausgelöst wird.

8.2.2 Transaktionen einer Intra-Chip-Verbindung

Transaktionen auf einer Intra-Chip-Verbindung teilen sich typischerweise in eine Adressierungs- und eine anschließende Datenübertragungsphase auf. Die Adressierungsphase entspricht der verallgemeinerten Bezeichnung der Modulzugriffsanfrage. Typischerweise verwendet jeder Prozessorkern eigene Adressleitungen, um Zugriffsanfragen an ein Modul zu stellen. Hierdurch können mehrerer Kerne zeitgleich Adressierungsphasen durchführen, ohne dass Wartezeiten entstehen.

Die Laufzeiten der Phasen einer Transaktion sind für gewöhnlich in den Datenblättern des Mikrocontrollers dokumentiert. Alternativ lassen sich diese Daten über Laufzeitmessungen annähern. Die Laufzeit einer Load-Instruktion auf ein globales Modul umfasst eine vollständige Transaktion. Werden Load-Instruktionen aufeinanderfolgend ausgeführt und die Laufzeiten t_n und t_{n+1} für die Ausführungen von n beziehungsweise $n + 1$ Anweisungen gemessen, so entspricht ihre Differenz der Laufzeit einer vollständigen Transaktion. Dies entspricht $\iota.\Delta$ einer Load-Instruktion auf das entsprechende Modul. Der Wert ist für jede zusätzliche Anweisung konstant. Durch die Bildung der Differenz werden zeitliche Aufwände zur Durchführung der Laufzeitmessungen ausgeblendet.

Unter Zuhilfenahme eines zweiten Prozessorkerns kann die Transaktionszeit durch gezieltes Erzeugen von Interferenzen in die Anteile der Adressierungs- und Datenübertragungsphase aufgeteilt werden. Führen beide Kerne synchron eine Load-Instruktion auf ein globales Modul aus, so enthält der ermittelte Wert für $\iota.\Delta$ auf dem Kern mit der niedrigeren Priorität die durch die Konkurrenz entstehende Wartezeit. Diese entspricht der Laufzeit einer Datenphase. Die Differenz aus $\iota.\Delta$ bei nicht verzögerter Ausführung und der Datanphase entspricht der Laufzeit der Adressierungsphase. Eine entsprechende Bestimmung der Laufzeit der Datenphase kann auch

für das Schreiben von Daten durchgeführt werden. Hierzu muss der Store-Buffer nach dem Einfügen des ersten Jobs explizit geleert werden. Dies kann durch eine spezielle Instruktion zum Synchronisieren des Kerns mit dem Store-Buffer erreicht werden. Eine solche Anweisung löst das sofortige Verarbeiten der Jobs des Buffer aus und blockiert den Prozessorkern, bis alle Jobs abgearbeitet wurden. Eine solche Anweisung ist im Befehlssatz des MCMC unter der Bezeichnung *dsync* (data synchronization) vorhanden. Geschieht dies synchron auf zwei Prozessorkernen, ergibt sich die Wartezeit bei *dsync* aus der Laufzeit der Datenphase für eine schreibende Transaktion.

8.2.3 Beschreibung der Store-Buffer-Zustände

Jeder Prozessorkern setzt für schreibende Zugriffe auf lokale Speicher und globale Module einen Store-Buffer ein. Jedes Element im Buffer beschreibt einen Job j zum Schreiben von Daten. Mit $sb = [tail, head]$ repräsentiert jeder Buffer einen nach dem FIFO-Prinzip arbeitenden Ringspeicher mit begrenzter Kapazität. Die Ausführung einer Instruktion des Typs $w[m_{l|g}]$ resultiert stets in einem neuen Job $j = (y, t_{req}, t_{acc}, t_{rel}, prog)$. Ein Job wird bei $sb.tail$ in den Ringspeicher eingefügt. Abschließend rückt $sb.tail$ auf den nächsten Platz des Rings. Der nächste zu bearbeitende Job wird durch $sb.head$ beschrieben. Analog zu Instruktionen und Modulzugriffen bezeichnen die Attribute y, t_{req}, t_{acc} und t_{rel} den Typ des Zugriffs mit dessen Zieladresse sowie die entsprechenden Timings.

Mit $sb.tail \neq sb.head$ hat der Buffer freie Kapazitäten und der Prozessorkern kann einen neuen Job einfügen. Bei freien Kapazitäten resultiert dies in einer konstanten Ausführungszeit von Store-Instruktionen. Andernfalls wird die Laufzeit der Store-Anweisungen durch aktives Warten des Kerns unterbrechungsfrei verzögert, bis ein bestehender Job abgearbeitet und aus dem Buffer entfernt wurde. Die Ausführung eines Jobs wird dann begonnen, sobald dieser von $sb.head$ erreicht wurde. Dies entspricht dem Zeitpunkt $j.t_{req}$. Interne Abläufe des Store-Buffer zum Management des Ringspeichers können Laufzeiten verursachen, welche das Setzen von $sb.head$ verzögern. Diese Zeiten müssen bei der Bestimmung von t_{req} berücksichtigt werden. Die Dauer der Ausführung ist mit $j.prog$ (progress) beschrieben. Sie entspricht der Laufzeit eines schreibenden Zugriffs auf lokalen Speicher oder der Dauer einer Transaktion (Adress- + Datenphase) auf der Intra-Chip-Verbindung. Für Zugriffe auf lokalen Speicher gilt $t_{req} = t_{acc}$.

Für den MCMC muss unterschieden werden, ob $sb.head$ auf den ersten Job gesetzt wird, der in einen leeren Store-Buffer eingefügt wurde oder von einem abgeschlos-

senen Job auf den nächsten im Buffer verschoben wird. Im ersten Fall entsteht eine zusätzliche Laufzeit, die in der folgenden Betrachtung als t_{init} bezeichnet wird. Im zweiten Fall entsteht eine Laufzeit bei dem Wechsel auf den jeweils folgenden Job. Diese wird als t_{next} bezeichnet. Zur Bestimmung von t_{init} wird ein Job mittels einer Store-Instruktion dem Store-Buffer hinzugefügt und dessen umgehende Abarbeitung durch *dsync* erzwungen (siehe Abbildung 8.2a). Die Zeit t_{init} verstreicht in diesem Fall nicht vollständig. Der Prozessorkern blockiert bei *dsync* für die vollständige Transaktionszeit. Durch das Einfügen von nop-Befehlen (no operation) zwischen der Store-Instruktion und *dsync* wird die Laufzeit der Anwendung schrittweise um einen Prozessorzyklus verlängert. Während dieser Zeit verstreicht t_{init} (siehe Abbildung 8.2b). Entspricht t_{init} der Laufzeit von n nop-Befehlen, so wird nach n nop-Anweisungen die Abarbeitung des Jobs gestartet. Die Laufzeit t_b entspricht $t_a + t_{init} = t_a + n \cdot t_{nop}$. Sind weitere nop-Befehle vorhanden, vergleiche Abbildung 8.2c, werden diese parallel zur Abarbeitung des Jobs ausgeführt. Als Folge verringert sich die Blockierzeit von *dsync* um die Laufzeit der zusätzlichen Operationen. Insgesamt bleibt die Laufzeit unverändert ($t_b = t_c$).

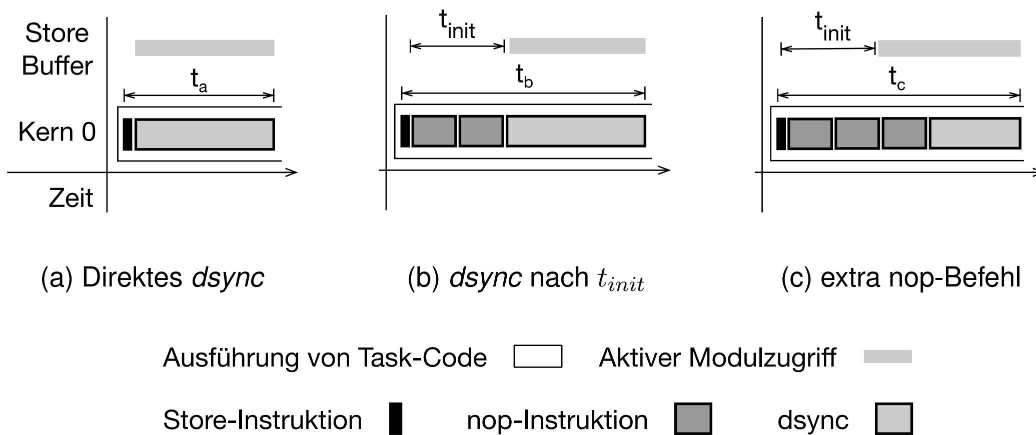


Abbildung 8.2: Verwendung von nop-Befehlen zur Bestimmung von t_{init}. nop-Befehle mit konstanter Laufzeit werden nach dem Ausführen einer Store-Instruktion und vor dem expliziten Leeren des Store-Buffer eingefügt.

Werden aufeinanderfolgend mehrere Store-Instruktionen ausgeführt, verlängert sich die Blockierzeit von *dsync* mit jeder zusätzlichen Instruktion um die Transaktionszeit eines Zugriffs zuzüglich t_{next}. Bei bekannten Laufzeiten für die Adress- und Datenphase kann der Wert für t_{next} ermittelt werden.

8.3 Erzeugung des Modulzugriffstrace

Zur Betrachtung der Cross-Core-Interferenzen wird für jeden Kern ein Modulzugriffstrace benötigt. Dieser muss alle Intra-Chip-Verbindungen berücksichtigen, auf denen Interferenzen auftreten können. In der folgenden Beschreibung wird der Aufbau des Trace allgemein beschrieben. Es werden die entsprechenden Stellen aufgezeigt, um weitere Verbindungen in das Verfahren zu integrieren. Ausgangspunkt zur Beschreibung der Modulzugriffe ist der Instruktion-Trace des jeweils zu betrachtenden Prozessorkerns. Weiterhin fließen hardwarespezifische Timings in die Berechnung ein, die zeitliche Eigenschaften der betrachteten Hardware beschreiben. Dies sind beispielsweise die Zeiten t_{init} und t_{next} des Store-Buffer oder die Laufzeiten der Adress- und Datenphasen der betrachteten Intra-Chip-Verbindung. Solche Zeiten werden durch die Addition von α berücksichtigt. Algorithmus 1 beschreibt das Vorgehen zur Ableitung aller Modulzugriffe aus einem gegebenen Instruction-Trace.

Der Instruction-Trace wird nach allen Load- und Store-Anweisungen, unabhängig ihrer Zieladresse, durchsucht. Anweisungen mit der Zieladresse m_g werden zur Identifikation von Modulzugriffen genutzt. Da der Store-Buffer die schreibenden Zugriffe durchführt, müssen zudem alle Instruktionen betrachtet werden, die dessen Zustand beeinflussen können. Neben Store-Instruktionen mit dem Ziel m_g sind dies auch Store-Anweisungen auf lokalen Speicher mit m_l. Auch diese Zugriffe resultieren in Store-Buffer-Jobs und beeinflussen den Buffer entsprechend. Load-Instruktionen auf sowohl globale als auch lokale Ziele können durch ihre vorrangige Behandlung die Abarbeitung von Jobs mit entsprechendem Ziel verzögern. Store-Instruktionen auf lokalem Speicher und Load-Instruktionen können somit die Durchlaufzeit eines Store-Buffer-Jobs mit Zieladresse m_g und damit auch die Zeiten des daraus resultierenden Modulzugriffs beeinflussen.

Abhängig vom Typ der gefundenen Anweisung und des Zielmediums wird eine entsprechende Verarbeitung der Instruktion durchgeführt. Für jede gefundene Anweisung ι_k wird zunächst die Laufzeit δ berechnet, die bis zum Erreichen von ι_k benötigt wird (Zeile 5). Sie beschreibt das zeitliche Intervall, das nach der Ausführung von $\iota_{(k-1)}$ beginnt und bei dem Erreichen von ι_k endet. Da der Prozessorkern lediglich mittels Store- und Load-Instruktionen den Store-Buffer beeinflussen kann und alle diese Instruktionen betrachtet werden, wird der Buffer während δ nicht durch den Kern beeinflusst. Der Buffer hat seinen Zustand lediglich dahingehend geändert, indem er bestehende Jobs abgearbeitet hat. Ausgehend vom zuletzt bekannten Zustand des Store-Buffer zum Zeitpunkt $\iota_{k-1}.t$, wird die Zustandsänderung während δ berechnet (Zeile 6). Werden während δ Store-Jobs abgeschlossen, werden daraus

resultierende Modulzugriffe im Modulzugriffstrace *mzt* festgehalten. Anschließend wird die Ausführung von ι_k verarbeitet (Zeilen 9, 12, 15 und 18).

Algorithmus 1 Vorgehen zur Erzeugung des Modulzugriffstrace

```
1: function ModulzugriffsTrace(InstructionTrace o, StoreBuffer sb, HardwareTimings α)
2:     Modulzugriffstrace mzt ← ∅
3:     Job j ← 0, Zeit δ ← 0
4:     for all ι_k ∈ o do
5:         δ ← δ + ι_k.t − (ι_(k−1).t + ι_(k−1).Δ)
6:         mzt ← mzt + ProcStoreBuffer(ι, sb, δ, α)
7:         switch ι_k.y do
8:             case r[m_l]
9:                 ProcessLocalRead(ι_k, sb, δ)
10:            end case
11:            case w[m_l]
12:                ProcessWrite(mzt, ι_k, sb)
13:            end case
14:            case r[m_g]
15:                mzt ← mzt + ProcessGlobalRead(mzt, ι_k, sb, δ)
16:            end case
17:            case w[m_g]
18:                ProcessWrite(mzt, ι_k, sb)
19:            end case
20:        end switch
21:    end for
22:    return mzt
23: end function
```

8.3.1 Behandlung von Load-Anweisungen

Die Behandlung von Load-Instruktionen unterscheidet zwischen Instruktionen mit Zugriff auf lokalen Speicher (Algorithmus 1 Zeile 9) und Instruktionen mit Zugriff auf globale Speicher- oder I/O-Module (Algorithmus 1 Zeile 15).

8.3.1.1 Globale Load-Anweisungen

In Algorithmus 2 ist die Verarbeitung lesender Zugriffe auf einen globalen Adressbereich m_g beschrieben. Zunächst wird geprüft, ob sich bereits ein Job für die zu lesende Zieladresse im Store-Buffer befindet (Zeile 3). Ist dies der Fall, hat der Prozessorkern die Daten aus dem Store-Buffer gelesen und es finden keine weiteren Speicherzugriffe statt. Die von ι benötigte Ausführungszeit wird δ hinzugefügt und bei der nächsten zu betrachten Anweisung berücksichtigt. Existiert kein entsprechender Job im Store-Buffer, resultiert aus ι ein neuer Modulzugriff b. Dieser startet mit dem Zugriffsversuch auf das Zielmedium beziehungsweise der Adressierungsphase zum Zeitpunkt $\iota.t$ zuzüglich der hardwarespezifischen Zeit, die für die Execution-Phase von ι vom Prozessorkern benötigt wird (Zeile 6). Da Load-Instruktionen für die Dauer der Transaktion auf der Intra-Chip-Verbindung blockieren, wird die Verbindung nach Abschluss der Instruktion freigegeben. Entsprechend kann t_{rel} mittels der aus dem Instruktion-Trace enthaltenen Information $\iota.\Delta$ bestimmt werden (Zeile 7). Der Start der Datenphase bei $b.t_{acc}$ kann über $b.t_{rel}$ und die Dauer der Datenphase berechnet werden (Zeile 8).

Hatte zum Zeitpunkt des Modulzugriffs bei $b.t_{acc}$ der Store-Buffer bereits aktiven Zugriff auf den gleichen Adressbereich m_g mit einem Ende nach $b.t_{acc}$ (Zeile 9), so entstand für den Prozessorkern zwischen $b.t_{req}$ und $b.t_{acc}$ eine Wartezeit, bis der Job abgeschlossen wurde. Entsprechend wird für die verbleibende Zugriffszeit des Buffer dessen Status aktualisiert und der daraus resultierende Modulzugriff in b_{sb} festgehalten (Zeile 10). Nach Abschluss des Jobs j_0 zeigt $sb.head$ auf den nächsten Job im Ringspeicher und j wird entsprechend aktualisiert (Zeile 11).

Weiterhin wird der zum Zeitpunkt $b.t_{acc}$ aktuelle Job j bei $sb.head$ betrachtet (Zeile 13). Liegt das Ziel dieses Jobs im gleichen Speicherbereich wie das von b und überschneiden sich die Modulzugriffszeiten von j und b, so müssen die Zugriffszeiten von j angepasst werden (Zeile 14). Der Store-Buffer erhält erst nach dem Beenden der lesenden Transaktion des Prozessorkerns Zugriff auf das Modul. Die angepasste Modulzugriffszeit enthält mit α(konsekutiv) eine spezifische Latenz des MCMC, die jeweils zwischen zwei direkt aufeinanderfolgenden Modulzugriffen eines Prozessorkerns veranschlagt wird. In diesem Fall unterscheidet die Intra-Chip-Verbindung des MCMC nicht zwischen dem Kern und dessen Store-Buffer. Zielt j auf einen anderen Speicherbereich als b, so gilt die gesamte Ausführungszeit $\iota.\Delta$ als Verarbeitungszeit δ des Jobs. Diese Zeit wird festgehalten (Zeile 16) und in der nächsten Iteration von Algorithmus 1 verarbeitet.

Abschließend werden der Modulzugriff b und gegebenenfalls abgeschlossene Zu-

griffe b_{sb} an Algorithmus 1 übergeben (Zeile 19) und dem Modulzugriffstrace hinzugefügt.

Algorithmus 2 Verarbeitung globaler Lesezugriffe

1: **function** ProcessGlobalRead(Instruktion ι, StoreBuffer sb, Zeit δ, HardwareTimings α)
2: $\quad$ $\mathrm{Job\ j \leftarrow sb.head, Modulzugriff\ b \leftarrow 0, b_{sb} \leftarrow 0}$
3: $\quad$ **if** $\exists_{\mathrm{j \in sb}} : \mathrm{j.y} = \iota.\mathrm{y}$ **then**
4: $\quad\quad$ $\delta \leftarrow \delta + \iota.\Delta$
5: $\quad$ **else**
6: $\quad\quad$ $\mathrm{b.t_{req}} \leftarrow \iota.\mathrm{t} + \alpha(\mathrm{execution\text{-}phase}\ \iota)$
7: $\quad\quad$ $\mathrm{b.t_{rel}} \leftarrow \iota.\mathrm{t} + \iota.\Delta$
8: $\quad\quad$ $\mathrm{b.t_{acc}} \leftarrow \mathrm{b.t_{rel}} - \alpha(\mathrm{datenphase})$
9: $\quad\quad$ **if** $\mathrm{j.y = w[m_g] \wedge j.t_{acc} < b.t_{acc} < j.t_{rel}}$ **then**
10: $\quad\quad\quad$ $\mathrm{b_{sb}} \leftarrow \textsc{ProcStoreBuffer}(\iota, \mathrm{sb}, \mathrm{j.prog}, \alpha)$
11: $\quad\quad\quad$ $\mathrm{j \leftarrow sb.head}$
12: $\quad\quad$ **end if**
13: $\quad\quad$ **if** $\mathrm{j.y = w[m_g] \wedge b.t_{acc} \leq j.t_{acc} < b.t_{rel}}$ **then**
14: $\quad\quad\quad$ $\mathrm{j.t_{acc}} \leftarrow \mathrm{b.t_{rel}} + \alpha(\mathrm{konsekutiv})$
15: $\quad\quad$ **else if** $\mathrm{sb} \neq \emptyset$ **then**
16: $\quad\quad\quad$ $\delta \leftarrow \delta + \iota.\Delta$
17: $\quad\quad$ **end if**
18: $\quad$ **end if**
19: $\quad$ **return** $\mathrm{b_{sb} + b}$
20: **end function**

8.3.1.2 Lokale Load-Anweisungen

Load-Anweisungen auf lokalen Speicher erzeugen keine Modulzugriffe, die dem Modulzugriffstrace hinzugefügt werden. Durch die vorrangige Behandlung lesender Zugriffe gegenüber schreibenden können Lesende die Abarbeitung von Store-Buffer-Jobs verzögern. Wird durch eine lokale Load-Anweisung das Abarbeiten eines Jobs mit einer Zieladresse auf lokalen Speicher verzögert, wirkt sich dies auf die nachfolgenden Jobs im Store-Buffer aus. Entsprechend kann diese Verzögerung auf Jobs wirken, die auf einen globalen Adressbereich zielen. Die Behandlung lokaler Load-Anweisungen erfolgt entsprechend Algorithmus 2, jedoch muss in den Zeilen 9 und 13 auf Jobs mit einer Zieladresse aus dem Bereich m_l geprüft werden. Weiterhin

entsteht kein neuer Modulzugriff b, der an mzt zurückgegeben würde.

8.3.2 Behandlung von Store-Anweisungen

Algorithmus 3 zeigt die Verarbeitung von Store-Anweisungen. Es werden alle ι unabhängig des Adressbereichs ihrer Zieladresse betrachtet. Aus allen Anweisungen resultiert ein neuer Job j. Wird davon ausgegangen, dass der Job in einen leeren Buffer eingefügt wurde, beginnt dessen Abarbeitung nach t_{init} bei $j.t_{req}$ (Zeile 5). War der Store-Buffer zum Zeitpunkt des Einfügens nicht leer, wird $j.t_{req}$ durch die Aktualisierung der Store-Buffer-Zustände bei der Bearbeitung der folgenden Load-/Store-Anweisungen angepasst.

Algorithmus 3 Verarbeitung von Schreibzugriffen

1: **function** ProcessWrite(Instruktion ι, StoreBuffer sb, HardwareTimings α)
2:     Job $\mathrm{j} \leftarrow 0, \mathrm{Modulzugriff\ b} \leftarrow 0$
3:     $\mathrm{j.y} \leftarrow \iota.\mathrm{y}$
4:     **if** $\mathrm{sb} = \emptyset$ **then**
5:         $\mathrm{j.t_{req}} \leftarrow \iota.\mathrm{t} + \iota.\Delta + \alpha(\mathrm{init})$
6:     **else if** $\mathrm{sb.head} = \mathrm{sb.tail}$ **then**
7:         $\iota.\Delta \leftarrow \iota.\Delta + \mathrm{j.prog}$
8:         $\mathrm{b} \leftarrow \textsc{ProcStoreBuf}(\iota, \mathrm{sb}, \mathrm{sb.head.prog}, \alpha)$
9:     **else**
10:         $\delta \leftarrow \delta + \iota.\Delta$
11:     **end if**
12:     $\mathrm{j.prog} \leftarrow \alpha(\mathrm{transaktionszeit}_{\mathrm{m_l|m_g}})$
13:     sb.tail ENQUEUE j
14:     **if** $\mathrm{b.y} = \mathrm{m_l}$ **then** $\mathrm{b} \leftarrow 0$ **end if**
15:     **return** b
16: **end function**

Ist der Buffer zum Zeitpunkt $\iota.t$ vollständig gefüllt, blockiert der Prozessorkern bei ι so lange, bis ein Job aus dem Buffer entfernt wurde. In diesem Fall wird für die verbleibende Bearbeitungszeit $j.prog$ des Jobs bei $sb.head$ der Zustand des Store-Buffer aktualisiert (Zeile 8), wodurch j aus dem Buffer entfernt wird. Im aufgezeichneten Instruction-Trace ist die entstandene Wartezeit ($j.prog$) des Kerns bereits in $\iota.\Delta$ enthalten. Diese Information kann für einen Abgleich der berechneten Store-Buffer-Timings mit den aufgezeichneten Werten des Instruction-Trace verwendet werden.

Bei der in Abschnitt 8.4.3 durchgeführten Integration von durch Cross-Core-Interferenzen hervorgerufenen Wartezeiten in Instruction- und Modultimings, werden Laufzeiten von Instruktionen neu berechnet. Hierzu wird auch Algorithmus 3 eingesetzt. Wird der Algorithmus im Kontext von Abschnitt 8.4.3 verwendet, müssen Laufzeiten von Store-Instruktionen um entstandene Wartezeiten erweitert werden (Zeile 7).

Ist der Store-Buffer nicht leer und hat freie Kapazitäten, wird der Job eingefügt. Weiterhin hat der Store-Buffer während der Ausführung von ι den aktuellen Job weiter bearbeitet. Dies fließt in der nächsten Iteration von Algorithmus 1 bei der Aktualisierung des Store-Buffer-Status ein (Zeile 10). Die regulär benötigte Zeit zum Ausführen des Jobs ist von der Zugriffszeit des Zielmediums abhängig und wird entsprechend gesetzt (Zeile 12). Abschließend wird j dem Buffer hinzugefügt (Zeile 13) und ein vom Store-Buffer abgeschlossener Modulzugriff auf einen globalen Adressbereich an den Modulzugriffstrace übergeben (Zeilen 14 und 15).

8.3.3 Verarbeitung des Store-Buffer

Die Aktualisierung des Store-Buffer durch die Prozedur *ProcStoreBuffer* ist in Algorithmus 4 beschrieben. Die Fortschritte des Buffer werden für die jeweils übergebene Laufzeit δ berechnet. Es werden alle Jobs bei $sb.head$ abgearbeitet, deren Zugriffsanfragen vor dem betrachteten Zeitpunkt $\iota.t$ gestellt werden und deren Fortschritt abzüglich δ den Wert 0 erreicht. Somit resultiert aus dem Job j bei $sb.head$ ein Modulzugriff b. Die Timings von b basieren auf $j.t_{head}$ und $j.t_{start}$. Existiert ein Job mit $sb.head.prog > \delta$, so wird der verbleibende Anteil δ von dessen Fortschritt $sb.head.prog$ subtrahiert (Zeile 13).

8.3.4 Integration zusätzlicher Adressbereiche

Die vorangegangenen Beschreibungen haben Zugriffe auf lokalen Speicher und einen globalen Adressbereich behandelt. Letztere werden im Modulzugriffstrace festgehalten. Modulzugriffe, die über zusätzliche Intra-Chip-Verbindungen ausgeführt werden, müssen ebenfalls in den Trace aufgenommen werden. Zu ihrer Behandlung wird Algorithmus 1 um weitere Fälle, analog zur Verarbeitung der Zugriffe auf m_g (Zeilen 14 bis 19), erweitert. Dies gilt für jeden zusätzlich zu betrachtenden Adressbereich einer Intra-Chip-Verbindung. Weiterhin ist eine Anpassung von Algorithmus 2 (Zeilen 9 und 13) erforderlich, um lesende Zugriffe auf diese Adressbereiche zu berücksichtigen.

Algorithmus 4 Verarbeitung des Store-Buffer

```
1: function ProcStoreBuffer(Instruktion ι, StoreBuffer sb, Fortschritt δ,
   HardwareTimings α)
2:     Modulzugriffstrace mzt ← ∅, Modulzugriff b ← 0
3:     Job j ← sb.head
4:     while sb ≠ ∅ ∧ j.t_req ≤ ι.t ∧ j.prog ≤ δ do
5:         δ ← δ − sb.head.prog,  j ← sb.head
6:         b.t_req ← j.t_req
7:         b.t_acc ← b.t_req + α(adressierungsphase)
8:         b.t_rel ← b.t_acc + α(datenphase)
9:         mzt ← mzt + b
10:        DEQUEUE   sb.head,  j ← sb.head
11:        j.t_req ← b.t_rel + α(next)
12:    end while
13:    j.prog ← j.prog − δ
14:    return mzt
15: end function
```

8.4 Betrachtung der parallelen Ausführung

Die Instruction-Traces und Modulzugriffstraces wurden jeweils für die isolierte Ausführung der Teilsysteme erstellt. Zur Bestimmung der Cross-Core-Interferenzen wird ihre parallele Ausführung angenommen. Die auf den Prozessorkernen ausgeführten Instruktionen ändern sich hierbei nicht. Lediglich die Timings der Instruktionen und die der entstehenden Zugriffe auf globale Module werden beeinflusst. Zur Betrachtung einer angenommenen parallelen Ausführung werden die Instruction-Traces und Modulzugriffstraces der in die Untersuchung einbezogenen Teilsysteme auf eine gemeinsame Zeitbasis abgebildet. Bezogen auf die globale Zeit können zeitgleiche und damit konkurrierende Modulzugriffe erkannt werden. In einer realen parallelen Ausführung löst der Multi-Core-Mikrocontroller diese entstehende Konkurrenz durch eine Serialisierung der Zugriffe auf. Diese Serialisierung, die daraus entstehenden Wartezeiten, deren Auswirkungen auf die Timings der Instruktionen und damit die Einflüsse auf das zeitliche Verhalten der Teilsysteme werden ermittelt.

8.4.1 Abbildung der Traces auf eine gemeinsame Zeitbasis

Die Abbildung der Traces auf eine gemeinsame Zeitbasis bedingt, dass die Timings jedes Teilsystems in Relation zu einem gemeinsamen Bezugspunkt gesetzt werden können. Entsprechend Abschnitt 7.3 können Teilsysteme über eine hardwareseitige Konfiguration ihrer Startsignale synchronisiert und mittels ΔStart kann eine zeitliche Relation zwischen den Systemen realisiert werden. Abbildung 8.3 zeigt dies für die Kerne 0 und 1. Der gemeinsame Bezugspunkt entspricht dem Start der k-ten Iteration von Kern 0. Dies entspricht dem Nullpunkt der gemeinsamen Zeitbasis. Die Timings des Instruction-Trace beziehungsweise die der Modulzugriffe von Kern 0 entsprechen somit direkt der Betrachtung innerhalb der gemeinsamen Zeitbasis. Für alle weiteren Kerne c_i kann $\Delta Start_i \geq 0$ gewählt werden. Die jeweiligen Werte sind abhängig von der zu betrachteten Konfiguration der parallelen Ausführung. Im Beispiel wird die j-te Iteration von Kern 1 bezüglich der k-ten Iteration von Kern 0 um $\Delta Start_1 = 2$ verzögert gestartet. $\Delta Start_1$ gilt als Offset, über den die Timings von Kern 1 auf die gemeinsame Zeitbasis abgebildet werden.

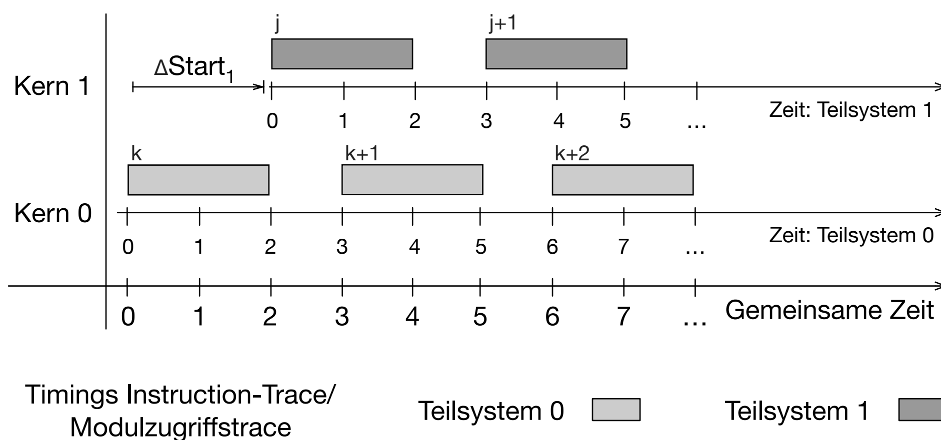


Abbildung 8.3: Abbildung zweier Multi-Core Teilsysteme auf eine gemeinsame Zeitbasis. Die Timings jeder aufgezeichneten Iteration von Kern 1 werden mittels $\Delta Start_1$ auf die gemeinsame Zeitbasis übertragen.

8.4.2 Berechnung von Wartezeiten bei konkurrierenden Modulzugriffen

Auf Basis der gemeinsamen Zeit können konkurrierende Modulzugriffe erkannt und entstehende Wartezeiten berechnet werden. Abbildung 8.4 zeigt die Modulzugriffe b_i und b_j der Kerne c_i und c_j. Der Kern c_i beginnt bei $b_j.t_{req}$ die Adressierungsphase und erhält bei $b_i.t_{acc}$ aktiven Modulzugriff. Parallel hierzu beginnt Kern c_j bei $b_j.t_{req}$

die Adressierung. Regulär würde c_j bei $b_j.t_{acc}$ aktiven Modulzugriff erhalten. Durch die bereits aktive Datenphase von c_i entsteht für c_1 die Wartezeit γ_1.

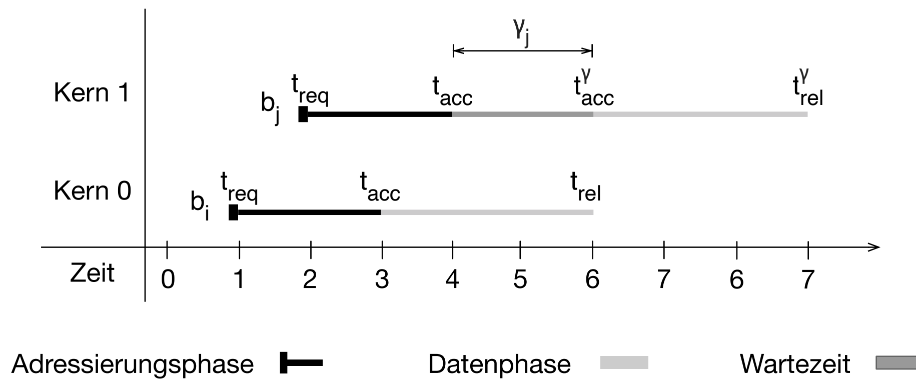


Abbildung 8.4: Einfluss der Wartezeit auf die Timings eines Modulzugriffs. Durch die Wartezeit γ_j ergibt sich der Zeitpunkt des aktiven Modulzugriffs bei t^{γ}_{acc} und der Zeitpunkt der Freigabe des Moduls bei t^{γ}_{rel}.

Eine Wartezeit γ tritt allgemein dann auf, wenn konkurrierende Phasen der Modulzugriffe innerhalb eines gemeinsamen Zeitintervalls liegen. Auf dem MCMC besteht Konkurrenz zwischen parallelen Datenphasen, sodass hier die Timings der Datenphasen $b.t_{acc}$ und $b.t_{rel}$ betrachtet werden. Für c_j entsteht eine Wartezeit, da $b_j.t_{acc}$ innerhalb des zeitlichen Intervalls $[b_i.t_{acc}, b_i.t_{rel}]$ liegt. Das Maß der Wartezeit ergibt sich aus der verbleibenden Dauer der Datenphase von c_i zum Zeitpunkt $b_j.t_{acc}$. Für c_j ergibt sich die Wartezeit $\gamma_j = 2$.

Wird der Modulzugriff b_i von c_i höher priorisiert als b_j von c_j oder hat c_i bereits aktiven Zugriff, so ergibt sich für b_j die Wartezeit γ_j entsprechend Gleichung 8.1:

$$\gamma_j(b_i, b_j) = (b_i.t_{rel} - b_i.t_{acc}) - (b_j.t_{acc} - b_i.t_{acc}) \tag{8.1}$$

Bei einem positiven Wert für γ_j ergibt sich der aus der Wartezeit resultierende Zeitpunkt der Datenphase von c_j aus $b_j.t_{acc} + \gamma_j$. Ein negativer Wert für γ_j bedeutet, dass der reguläre Modulzugriff von b_j mit $(b_j.t_{acc} > b_i.t_{rel})$ zeitlich nach der Freigabe des Moduls durch c_i erfolgt.

Algorithmus 5 zeigt das verwendete Vorgehen zur Berechnung aller Wartezeiten bei beliebig vielen parallelen Teilsystemen. Nach der Übertragung der Timings auf die gemeinsame Zeitbasis werden sukzessive alle Modulzugriffe betrachtet. Die Fortschrittszeit t beschreibt die Zeitpunkte, zu denen die Wartezeiten und anschließend die Timings der parallelen Ausführung berechnet werden. Die Zeit richtet sich nach den Startzeiten der Datenphasen, da hier ein exklusiver Zugriff auf ein Modul begonnen wird. Aufgrund dieses Modulzugriffs können andere Kerne bei entsprechenden

Zugriffsanfragen blockiert werden. Initial ergibt sich für t der Start der frühesten Datenphase (Zeile 4).

Algorithmus 5 Berechnung von Wartezeiten

```
 1: procedure BerechneWartezeiten(Instruction Trace o[], Modulzugriffstrace
    mzt[], HardwareTimings α)
 2:     ∀_{i:0≤i<N} : kerne[] + c_i, Modulzugriff b_i ← mzt_i[0]
 3:     while kerne[] ≠ ∅ do
 4:         Zeit t ← min(b.t_acc)
 5:         ∀_{i:0≤i<N} : δ_i = δ(t, b_i.t_acc)
 6:         Sortiere(kerne, δ)
 7:         Arbitriere(kerne, b_i.y, δ)
 8:         for all c_j ∈ kerne ∧ 0 < j < N do
 9:             b_j.γ ← b_j.γ + γ(b_0, b_j)
10:             mzt[b_j] ← mzt[b_j] + AktualisiereTimings(b_j, o_j, α)
11:         end for
12:         b_0 ← mzt_0.next
13:     end while
14: end procedure
```

Alle aktiven Prozessorkerne werden in einer Liste verwaltet. Diese wird aufsteigend nach den zum Zeitpunkt t geltenden zeitlichen Abständen δ_i (Zeile 4) aller Kerne c_i zu ihrem jeweils nächsten Modulzugriff b_i sortiert (Zeile 6). Zielen mehrere Zugriffsversuche auf Module des gleichen Adressbereichs, können konkurrierende Zugriffsanfragen an das Modul entstehen. Dies geschieht, wenn zum Zeitpunkt t die Entfernung δ eines oder mehrerer Kerne den Wert 0 erreicht hat. Die Arbitrierungsstrategie der jeweiligen Intra-Chip-Verbindung entschiedet über die Zugriffsreihenfolge und passt die Reihenfolge der Kerne mit $\delta = 0$ an (Zeile 7). Der Kern am Kopf der Liste wird als c_0 bezeichnet und erhält Modulzugriff. Ist ein Kern am Kopf der Liste angelangt, so sind für diesen Kern alle Wartezeiten für den Zugriff b_0 berechnet.

Der Zugriff von c_0 kann Wartezeiten für die in der Liste folgenden Kerne c_j mit $0 < j < N$ verursachen. Diese Wartezeiten werden für alle c_j berechnet. Bis ein c_j den Kopf der Liste erreicht hat, können mehrere durch die Arbitrierung höher priorisierte Kerne verarbeitet werden. Für ein b_j können somit mehrere Wartezeiten entstehen (vergleiche Abschnitt 7.2). Diese werden aufsummiert (Zeile 9) und zur Anpassung der Timings von c_j herangezogen (Zeile 10). Wurden die von c_0 verursachten Wartezeiten für alle c_j berechnet, wird der nächste Modulzugriff von c_0

erfasst (Zeile 12) und die Behandlung von c_0 ist abgeschlossen. Der nächste zu betrachtende Zeitpunkt ergibt sich erneut aus der nun frühesten geplanten Datenphase (Zeile 4).

8.4.3 Integration der Wartezeit in Instruktion- und Modul-Timings

Wartezeiten wirken sich auf die Timings des betroffenen Modulzugriffes und gegebenenfalls auf folgende Zugriffe und Instruktionen aus. Algorithmus 6 beschreibt das Vorgehen zur Anpassung der Timings des Kerns c_i, wenn für einen Zugriff b dieses Kerns eine Wartezeit γ berechnet wurde. Durch die Wartezeit startet und endet die Datenphase dieses Zugriffs verspätet. Entsprechend werden diese Zeiten angepasst (Zeile 2 und 3). Timings, bei denen γ berücksichtigt wurde, werden durch t^γ gekennzeichnet.

Algorithmus 6 Berechnung und Anpassung der Timings von Instruktionen

1: **function** AKTUALISIERETIMINGS(Modulzugriff b, Instruction-Trace o, HardwareTimings α)
2: $\quad \mathrm{b.t}^\gamma_{\mathrm{acc}} \leftarrow \mathrm{b.t}_{\mathrm{acc}} + \mathrm{b.}\gamma$
3: $\quad \mathrm{b.t}^\gamma_{\mathrm{rel}} \leftarrow \mathrm{b.t}_{\mathrm{rel}} + \mathrm{b.}\gamma$
4: $\quad$ **if** $\mathrm{b.y} = \mathrm{r[m_g]}$ **then**
5: $\qquad$ Instruktion $\iota \leftarrow$ Instruktion(b)
6: $\qquad \iota.\Delta \leftarrow \iota.\Delta + \mathrm{b.}\gamma$
7: $\qquad$ **for all** $\iota_\mathrm{i} \in \mathrm{o} \wedge \iota_\mathrm{i}.\mathrm{t} > \iota.\mathrm{t}$ **do**
8: $\qquad\quad \iota.\mathrm{t}^\gamma \leftarrow \iota.\mathrm{t} + \gamma$
9: $\qquad$ **end for**
10: $\quad$ **else if** $\mathrm{b.y} = \mathrm{w[m_g]}$ **then**
11: $\qquad$ StoreBuffer sb $\leftarrow$ StoreBuffer($\mathrm{b.t_{acc}}$)
12: $\qquad$ sb.head.prog $\leftarrow$ sb.head.prog + b.γ
13: $\quad$ **end if**
14: $\quad$ **return** MODULZUGRIFFSTRACE($\mathrm{o[b.t_{acc}]}$, StoreBuffer($\mathrm{b.t_{acc}}$), α)
15: **end function**

Lesende Modulzugriffe resultieren aus der Ausführung von Instruktionen des Typs $\iota.y = r[m_g]$. Diese Instruktion wird ermittelt (Zeile 5). Als blockierende Instruktionen ist die Dauer des Modulzugriffs Teil der Ausführungszeit $\iota.\Delta$, wodurch auch eine Wartezeit in diese einfließt. Folglich werden auch die auf ι folgenden Instruktionen um γ verspätet ausgeführt und deren Timings im Instruction-Trace angepasst

(Zeilen 7 bis 9).

Durch die out-of-order Ausführung von schreibenden Modulzugriffen wirken Wartezeiten nicht direkt auf die Laufzeit der entsprechenden Store-Instruktion. Wartezeiten haben erst dann einen Einfluss auf Instruktionen des Typs $\iota.y = w[m_g]$, wenn die Instruktionen einen Job in einen vollständig gefüllten Store-Buffer einfügen will und dessen Abarbeitung durch Wartezeiten verlangsamt wird. In diesem Fall erhöht sich analog zu lesenden Zugriffen $\iota.\Delta$ die Laufzeit der Store-Anweisung und die Ausführungszeitpunkte der auf ι folgenden Instruktionen. Der betrachtete Modulzugriff ist der aktuell vom Store-Buffer auszuführende Job. Entsprechend verlängert γ dessen Fortschrittszeit (Zeile 12).

Durch die angepassten Timings können sich die Zeitpunkte und die Reihenfolge der auf b folgenden Zugriffe ändern, beispielsweise dann, wenn während eines wartenden Store-Buffer-Jobs eine Load-Instruktion ausgeführt wird. Da der lesende Modulzugriff vorrangig ausgeführt wird, kann sich so die Zugriffsreihenfolge im Vergleich zur Single-Core-Ausführung ändern. Durch den erneuten Aufbau des Modulzugriffstrace für alle Instruktionen ab dem Zeitpunkt des regulären Modulzugriffs von b werden die durch γ verursachten Einflüsse berücksichtigt (Zeile 14).

Durch die Berechnung der Multi-Core-Timings werden neue Laufzeiten für die in den Instruction-Traces aufgezeichneten und entsprechend ausgeführten Instruktionen berechnet. Die Differenz zwischen dem neu berechneten Wert für $\iota.\Delta$ und der aufgezeichneten Laufzeit ergibt die Laufzeiterhöhung einer konkreten Anweisungsausführung. Die Summe aller Laufzeiterhöhungen ergibt die Erhöhung der Laufzeit der gesamten Anwendung auf dem betrachteten Prozessorkern.

Jede Instruktion ist über ihre Adresse $\iota.x$ eindeutig bestimmbar und kann dem Quellcode zugewiesen werden, aus dem sie im Zuge seiner Übersetzung in Maschinencode erzeugt wurde. Ist eine Instruktion Teil einer Funktion, kann sie mehrfach ausgeführt werden. Über den Instruction-Trace und $\iota.t$ ist festgelegt, zu welchen Zeitpunkten der Anwendungsausführung und in welchem Teil der Anwendung die Ausführung einer Instruktion stattgefunden hat. Weiterhin ist durch die gemeinsame Zeitbasis bekannt, welche Modulzugriffe miteinander konkurrieren. Hieraus kann auf die konkurrierenden Instruktionen, auf den entsprechenden Quellcode und letztendlich auf die miteinander konkurrierenden Anwendungsteile geschlossen werden.

Diese Informationen können unter anderem dazu genutzt werden, um Codeabschnitte zu identifizieren, die bei paralleler Ausführung ein hohes Maß an Interferenzen hervorrufen können. Die Lokalisation der Laufzeiterhöhungen bietet Optimierungsmöglichkeiten, durch die das Maß der Interferenzen oder deren Varianz bei

unterschiedlichen Anwendungskonfigurationen verringert werden kann. Dies kann beispielsweise durch eine entsprechende Gestaltung des Quellcodes oder durch eine Anpassung des zeitlichen Versatzes ΔStart erreicht werden.

8.5 Evaluation der Interferenzanalyse

Zur Evaluation der Interferenzanalyse wird nicht auf die bisher verwendete Motor-Controller-Software zurückgegriffen. Stattdessen werden Anwendungen genutzt, die unterschiedlich komplexe Zugriffsszenarien auf globalen Speicher realisieren. Durch die so mögliche Vielfalt können aussagekräftigere Ergebnisse als durch die Verwendung der Motor-Controller erhalten werden. Es werden mehrere Anwendungen aus den Mälardalen WCET Benchmarks [60] sowie EEMBC CoreMark® verwendet. Als Multi-Core-System wird ein Infineon Application Kit TC297 verwendet.

8.5.1 Bestimmung der Laufzeiterweiterung

Für jede Benchmark-Anwendung wird die Laufzeiterweiterung sowohl per Laufzeitmessung als auch rechnerisch bestimmt. Hierzu wird eine zu betrachtende Anwendung zunächst mehrfach in Isolation, das heißt auf einem Kern des Multi-Core-Mikrocontrollers ausgeführt. Hierbei entstehen keine konkurrierenden Zugriffe. Für jede Ausführung werden neue Eingabedaten generiert. Abhängig vom jeweiligen Benchmark bestehen die Eingabedaten aus ein- oder mehrdimensionalen Feldern ganzer Zahlen, die per Zufallsgenerator initialisiert werden. Durch unterschiedliche Eingabedaten sollen ein möglichst großer Teil des Anwendungscodes abgedeckt und eine Varianz in den ausgeführten Codepfaden erzeugt werden. Die Datenstrukturen der Anwendung werden in einem globalen Speichermodul abgelegt, auf das bei der Anwendungsausführung mehrfach lesend und schreibend zugegriffen wird. Für jede Ausführung werden die Laufzeit und der Instruction-Trace aufgezeichnet.

Zur Messung der Laufzeiten unter konkurrierenden Zugriffen wird eine zu testende Anwendung mehrfach parallel auf drei Prozessorkernen ausgeführt. Hierzu wird die zeitliche Relation $\Delta Start = 0$ verwendet. Es wird derselbe Satz an Eingabedaten wie bei der Ausführung in Isolation verwendet. Die Eingabedaten sind für alle Kerne stets gleich. Dieses Vorgehen bildet das in Abschnitt 7.3.1 beschriebene Szenario der Konsolidierung gleicher Anwendungen ab, um möglichst viele konkurrierende Modulzugriffe zu provozieren. Zur Berechnung der Laufzeiten werden die aufgezeichneten Instruction-Traces der Ausführung in Isolation herangezogen. Die Berechnung findet für $\Delta Start = 0$ statt.

Die gemessenen Laufzeiten der einzelnen Traces zeigen aufgrund der unterschiedlichen Eingabedaten nur geringe Unterschiede. Diese liegen unter einem Prozent. Die hierauf zurückzuführende Varianz der Laufzeiterhöhung beträgt nur wenige CPU-Takte. Tabelle 8.1 zeigt die Mittelwerte der berechneten und gemessenen Laufzeiterhöhungen. Die Crossbar, an der das Speichermodul angebunden ist, nutzt eine faire Round-Robin-Arbitrierung. Dadurch zeigen alle Prozessorkerne ein nahezu identisches Maß an Interferenzen. Die berechneten Werte der Laufzeiterhöhungen liegen zwischen 5 % und 24 % der Laufzeit ohne Interferenzen. Im Vergleich zu den gemessenen Werten entspricht dies einer Genauigkeit von 93,83 % bis 99,93 %. Wird die daraus resultierende Laufzeit betrachtet, so wirkt sich die Fehlerabweichung nur geringfügig aus. Die vollständige Laufzeit auf den Prozessorkernen wurde mit Abweichungen zwischen 0,01 % und 0,72 % gegenüber den gemessenen Zeiten berechnet (siehe Tabelle 8.2).

Die Ergebnisse der Interferenzanalyse sind von der Genauigkeit abhängig, mit der die Modulzugriffe aus dem Instruction-Trace und die Zustände des Store-Buffer bestimmt werden können. Neben der Präzision der Hardware-Timings beeinflussen zwei Eigenschaften bezüglich der Arbeitsweise des Mikroprozessors die Genauigkeit der berechneten Daten. Dies sind die Sprungvorhersage bei verzweigten Anweisungen und der Übergang zwischen den Iterationen von Schleifen. Die Sprungvorhersage unterstützt im Rahmen der Performance-Optimierung die Pipeline-Mechanismen des Prozessors. Dies führt zu dem Effekt, dass Modulzugriffe zeitlich vor der aufgezeichneten Ausführung der dazugehörigen Anweisung stattfinden. Dementsprechend entsteht eine Differenz zwischen dem tatsächlichen und berechneten Zeitpunkt der Ausführung des Zugriffs.

Weiterhin werden die Ergebnisse durch Load-Anweisungen auf Adressen beeinflusst, die sich bereits im Store-Buffer befinden. Die vom Instruction-Trace zur Verfügung gestellten Informationen bezüglich der ausgeführten Anweisung haben die Form $LD.W[a15], d15$. Durch $ST.W$ wird beschrieben, dass es sich um einen schreibenden Zugriff handelt. Die zu schreibenden Daten befinden sich im Datenregister *d15*. Die Daten werden an die Adresse geschrieben, die sich im Adressregister *a15* befindet. Die Inhalte der Register werden nicht durch den Trace zur Verfügung gestellt. Dementsprechend besteht nicht die Möglichkeit, die konkreten Inhalte des Store-Buffer bezüglich der Zieladressen zu verfolgen und für die Interferenzanalyse zu nutzen. Somit kann eine unpräzise Auswertung von lesenden Zugriffen die Analyse beeinflussen.

Tabelle 8.1: Berechnete und gemessenen Laufzeiterhöhungen

	Laufzeit [CPU-Takte]	Laufzeiterweiterung [%]		
	Single-Core	Gemessen	Berechnet	Fehlerabweichung
bsort100	5324	18.71	18.78	0.40
cnt	14654	19.53	19.54	0.07
coremark	164636	5.06	4.82	4.71
crc	67016	4.53	4.86	6.13
fdct	8304	8.16	8.41	2.95
insertsort	5508	23.35	23.53	0.78
select	5662	13.46	14.27	6.04
st	51760	17.42	17.44	0.11

Tabelle 8.2: Berechnete und gemessenen Multi-Core Laufzeiten

	Laufzeit [CPU-Takte]		
	Gemessen	Berechnet	Fehlerabweichung [%]
bsort100	6320	6324	0,06
cnt	17516	17518	0,01
coremark	172962	172570	-0,23
crc	70082	70270	0,27
fdct	8982	9002	0,22
insertsort	6794	6804	0,15
select	6424	6470	0,72
st	60778	60788	0,02

8.5.2 Optimierung der parallelen Ausführung

Im Rahmen der Parallelisierung oder dem Datenaustausch zwischen Teilsystemen ist es sinnvoll, Wartezeiten zu reduzieren. So können die Kosten der Parallelisierung gesenkt oder der Durchsatz von Nachrichten erhöht werden. Weiterhin wird eine Konsolidierung vereinfacht, wenn bereits bei der Implementierung der entsprechenden Motor-Controller eine Verringerung von Interferenzen berücksichtigt werden kann.

Für eine Reduktion von Interferenzen müssen parallele Modulzugriffe, die eine Laufzeiterhöhung von Instruktionen verursachen, erkannt werden. Durch die Analyse werden die Zeitpunkte der Modulzugriffe, Wartezeiten als auch alle Instruktionen ermittelt, deren Laufzeit durch die Wartezeit erhöht wird. Abbildung 8.5a zeigt eine Visualisierung dieser Daten für Kern 0 am Beispiel eines Ausschnitts aus der par-

allelen Ausführung der CoreMark®-Anwendung. Es werden alle Instruktionen zu dem Zeitpunkt ihrer Ausführung dargestellt, die innerhalb des gegebenen Intervalls Modulzugriffe durchführen. Für jede Instruktion wird die berechnete Wartezeit angegeben. Zugriffe auf den gemeinsamen Speicher werden fair arbitriert. Hierdurch ergibt sich für die konkurrierenden Kerne 1 und 2 ein nahezu identisches Diagramm. Im Bereich zwischen 82 μs und 86 μs werden keine Modulzugriffe durchgeführt. Diese Zeit kann genutzt werden, um Konkurrenz zu senken. Wird die Ausführung der Kerne 1 und 2 um 2 beziehungsweise 4 μs mittels ΔStart verzögert, so führen diese ihre Modulzugriffe zwischen 82 μs und 84 μs beziehungsweise zwischen 84 μs und 86 μs aus. Die Zugriffe werden nicht mehr zeitgleich ausgeführt, wodurch Wartezeiten vermieden werden.

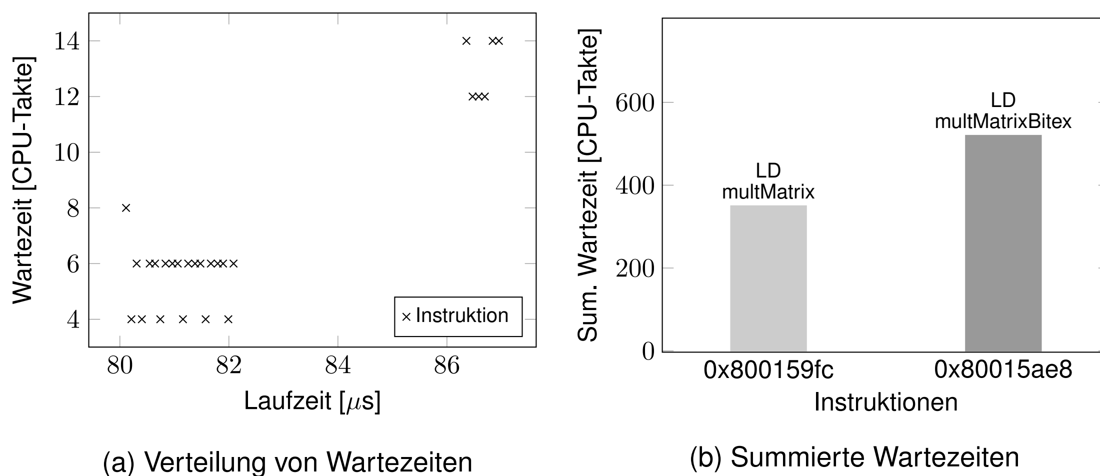


Abbildung 8.5: Beispielhafte Visualisierung von Analysedaten

Bei vielfältigen und breit über eine Anwendung verteilten konkurrierenden Instruktionen ist nicht davon auszugehen, dass ausreichend große und passend verteilte Intervalle existieren, um ΔStart ablesen zu können. Alternativ kann die Berechnung der Laufzeiterhöhung mit beliebig vielen Werten für ΔStart durchgeführt werden, um Konstellationen mit höheren als auch geringen Wartezeiten zu identifizieren (vergleiche Abschnitt 7.5.4).

Die Analyse liefert zu jeder der gezeigten Instruktionen auch deren Adresse. Da innerhalb des beschriebenen Intervalls mehrere Instruktionen eine identische Adresse aufweisen, kann auf eine wiederholte Ausführung eines Quellcodeabschnittes, beispielsweise durch eine Schleife, geschlossen werden. Abbildung 8.5b verdeutlicht dies, indem mehrfache Wartezeiten für eine Instruktion summiert werden. Weiterhin ist der Quellcode angegeben, aus dessen Übersetzung die Instruktionen erzeugt wurden. Auf diese Weise kann Quellcode identifiziert werden, der bei gegebener Integration

zu konkurrierenden Modulzugriffen führt. An diesen Stellen kann gegebenenfalls eine Optimierung, beispielsweise durch eine Umstrukturierung des Codes erfolgen.

8.6 Zusammenfassung der Ergebnisse

Die vorgestellte Analyse dient der Berechnung von Laufzeiterhöhungen, die infolge von Cross-Core-Interferenzen auf dem MCMC auftreten können. Bei der Parallelisierung eines Motor-Controllers können hierdurch die Anteile der Cross-Core-Interferenzen an den Kosten der Parallelisierung ermittelt werden. Im Rahmen der Konsolidierung mehrerer Motor-Controller ermöglicht die Ermittlung von Laufzeiterhöhungen eine Beurteilung, ob die zeitlichen Rahmenbedingungen der einzelnen Systeme auch nach einer Konsolidierung weiterhin eingehalten werden können. Die Berechnungen erfolgen auf Basis von Instruction-Traces. Diese werden für alle Systeme, die auf dem Multi-Core-System parallel ausgeführt werden sollen, aufgezeichnet. Dies geschieht bei ihrer Ausführung in Isolation, sodass eine reale Ausführung nicht notwendig ist. Gemessen an der gesamten Laufzeit des Instruction-Traces, wird die Multi-Core-Laufzeit mit einer Fehlerabweichung von unter 1 % ermittelt. Zusätzlich zu den berechneten Wartezeiten gehen aus der Analyse Daten hervor, die für eine Reduktion von Interferenzen genutzt werden können.

Eine berechnete Laufzeiterhöhung stellt für eine betrachtete Anwendung keine garantierte obere Grenze dar. Dies wird durch zwei Faktoren bedingt: durch die Präzision der berechneten Wartezeiten zur Ermittlung der Laufzeiterhöhung sowie durch die Vollständigkeit der in der Analyse betrachteten parallelen Codeausführung. Die Präzision der berechneten Wartezeiten ist von der Genauigkeit und Verfügbarkeit von hardwarespezifischen Informationen abhängig. Diese umfassen die Abläufe und dazugehörigen Timings der Intra-Chip-Verbindungen, die der Store-Buffer und die der Ausführung von Load- und Store-Instruktionen. Diese werden zur Berechnung paralleler Modulzugriffe und entstehender Wartezeiten verwendet. Ein Großteil dieser Informationen wird durch entsprechende Datenblätter bereitgestellt. Insofern die Verhaltensweisen der Hardware bekannt sind, können unbekannte Werte gegebenenfalls experimentell ermittelt werden. Dennoch bleibt hier eine Unschärfe, wenn das komplexe Verhalten der Hardware nicht vollständig bekannt ist.

Bei einer gegebenen zeitlichen Relation zwischen den zu betrachtenden Motor-Controllern ist das Ergebnis einer berechneten Laufzeiterhöhung nur für diejenigen Codesequenzen gültig, die in den Instruction-Traces aufgenommen wurden. Eine reale Ausführung kann jedoch weitere Sequenzen und damit parallele Zugriffe erzeu-

gen, die nicht durch die Traces abgedeckt und somit auch nicht in der Berechnung enthalten sind. Die Entstehung solcher Codesequenzen kann über das Taskmodell eingegrenzt werden. Das Taskmodell beschreibt, welche Tasks während einer Iteration ausgeführt werden können. Eine sporadische Ausführung von Tasks zu nicht bekannten Zeitpunkten innerhalb einer Iteration ist ausgeschlossen. Somit ist die Reihenfolge der Tasks festgelegt und für alle Iterationen identisch. Eine Ausnahme bildet die nebenläufige Ausführung. Aufgrund von Unterbrechungen durch höher priorisierte Tasks können sich die in verschiedenen Iterationen ausgeführten Tasks unterscheiden. Wann Tasks ausgeführt werden, ist durch die Ausführungsfrequenzen der jeweiligen Algorithmen bestimmt. Somit kann entsprechend Abschnitt 7.3.2 festgelegt werden, wie viele Iterationen betrachtet werden müssen, um alle Taskabfolgen und daraus entstehenden Codesequenzen in dem Instruction-Trace aufzuzeichnen.

Eine Task führt jedoch nicht notwendigerweise in jeder Iteration den gleichen Code aus. Dieser kann aufgrund bedingter Verzweigungen in der Implementierung variieren. Seitens der FOS-Tasks sind keine systembedingten Verzweigungen notwendig. Jedoch kann die Implementierung einzelner Bearbeitungsschritte, beispielsweise zur Behandlung unterschiedlicher Vorzeichen, Verzweigungen und damit unterschiedliche Codepfade mit variierenden Laufzeiten hervorrufen. In den betrachteten GRB-Algorithmen sind ebenfalls keine Verzweigungen vorhanden, welche auf die Modelle, die den Berechnungen zugrunde liegen, zurückzuführen sind. Prinzipiell können diese für GRB-Tasks jedoch auftreten. Existieren dennoch unterschiedliche Codepfade, kann diesbezüglich von einer geringen Varianz ausgegangen werden, da diese Pfade regelmäßig bis auf wenige Operationen identisch sind. Gegebenenfalls können Verzweigungen durch eine Verbesserung des Quellcodes entfernt werden. Existieren nur wenige Pfade, kann die Anzahl der aufgezeichneten Iterationen eines Motor-Controllers erweitert werden, bis alle parallel ausgeführten Kombinationen in dem Trace enthalten sind. Anschließend können Laufzeiterhöhungen für unterschiedliche Kombinationen der Traces, die jeweils verschiedene parallel ausgeführte Codepfade abdecken, berechnet werden.

Unter Beachtung der genannten Punkte können sowohl die Präzision der Analyse als auch die Vertrauenswürdigkeit der erhaltenen Ergebnisse verbessert werden. Im Vergleich zu anderen Anwendungsgebieten ist die Komplexität der betrachteten Tasks und die ihrer Implementierung relativ gering. Werden Instruction-Traces genügend groß bemessen und ausreichend Kombinationen ihrer parallelen Ausführung betrachtet, können Warte- und somit Multi-Core-Laufzeiten nahe oberer Grenzen berechnet werden.

Kapitel 9

Diskussion und Übertragbarkeit der Ergebnisse

In den Kapiteln 5 bis 8 wurde untersucht, ob sich Multi-Core-Mikrocontroller als zentrale Recheneinheit von Motor-Controllern eignen. Die Untersuchungen wurden bezüglich der Leistungsaufnahme von Multi-Core-Mikrocontrollern, der Parallelisierbarkeit geberloser Stromregelungen sowie der gemeinsamen Integration mehrerer Motor-Controller in einen Multi-Core Motor-Controllern durchgeführt.

9.1 Leistungsaufnahme von Multi-Core Motor-Controllern

Gegenstand der Untersuchung war der Einfluss zusätzlicher Prozessorkerne auf die Leistungsaufnahme von Motor-Controllern. Bei General-Purpose-Systemen und insbesondere bei mobilen Systemen werden Funktionen zur dynamischen Skalierung der Versorgungsspannung der Prozessorkerne genutzt, um eine hohe Energieeffizienz zu erreichen. Zugunsten von Safety-Eigenschaften unterstützen die eingesetzten Multi-Core-Mikrocontroller diese Funktionen nicht. Daher führt hier eine Parallelisierung der Anwendungen bei gleichzeitiger Skalierung der Rechenleistung durch Herabsetzen der Taktraten der Prozessorkerne zu keiner Reduktion der Leistungsaufnahme. Somit wurde die Leistungsaufnahme stets bei konstanter Versorgungsspannung und Taktfrequenz der Prozessorkerne betrachtet.

Die Ergebnisse zeigen, dass der Übergang von einem Single- zu einem Multi-Core-Mikrocontroller die Leistungsaufnahme eines Motor-Controllers um durchschnittlich 10 % erhöht. Dies gilt, wenn die zusätzlichen Kerne nicht genutzt werden und als Leistungsreserve dienen. Werden die Kerne hingegen genutzt, um mehr Rechenleis-

tung zu verwenden, als ein Single-Core-System bereitstellt, so bedingt eine Verdopplung der Rechenleistung einen Anstieg der Leistungsaufnahme um durchschnittlich 15 %. Bei einer Verdreifachung der Rechenleistung steigt der Leistungsbedarf durch das Multi-Core-System um durchschnittlich 30 %. Die hierdurch erreichbare Energieeffizienz eines MCMC wird insbesondere bei der Konsolidierung von Motor-Controllern deutlich. Jeder Prozessorkern eines MCMC ersetzt dabei einen vollständigen Single-Core Motor-Controller. Durch das Einsparen mehrerer Single-Core-Systeme werden durch das Multi-Core-System durchschnittlich bis zu 50 % der Leistungsaufnahme eingespart, welche die Single-Core Motor-Controller für das Ansteuern einer entsprechenden Anzahl an PMSM benötigt hätten. Hier ist jedoch folgender Gesichtspunkt zu berücksichtigen: zum Zeitpunkt der Untersuchungen existierten keine für einen Vergleich geeigneten Single-Core Mikrocontroller, welche die Rechenleistung mehrere Prozessorkerne zur Verfügung stellen können. Daher wurde der Einsatz eines MCMC, dem von zwei beziehungsweise drei Single-Core Motor-Controllern gegenübergestellt, welche jeweils die Rechenleistung eines Prozessorkerns des Multi-Core-Systems bereitstellen. Stehen vergleichbare Single-Core-Mikrocontroller zur Verfügung, ist eine Neubewertung der Leistungsaufnahme notwendig.

9.2 Parallelisierung von Motor-Controllern

Die Rechenleistung von Multi-Core-Systemen kann effektiv genutzt werden, wenn eine Anwendung ausreichend parallelisiert werden kann. Diesbezüglich wurde untersucht, inwiefern Motor-Controller zur geberlosen Regelung von PMSM parallelisiert werden können, um eine effektive Erhöhung der Regelfrequenz zu erreichen. Zur Beschreibung der Parallelisierbarkeit wurden unterschiedliche Algorithmen zur geberlosen Rotorlageberechnung ausgewählt, um ein möglichst breites Anwendungsspektrum abzudecken. Zur Auswahl wurden Faktoren wie die von den Algorithmen verwendeten Datenströme, die Art ihrer Integration in die Regelung als auch ihre Eignung für unterschiedliche Drehzahlbereiche betrachtet. Ausgehend von ihrer Integration in die Stromregelung wird zwischen zeitkontinuierlichen und zeitdiskreten Algorithmen unterschieden. Für beide Varianten wurden die Vorgehensweisen und Bedingungen der Integration untersucht und daraus die Möglichkeiten einer Parallelisierung abgeleitet.

9.2.1 Zeitkontinuierliche geberlose Rotorlageberechnung

Für Algorithmen, die eine zeitkontinuierliche geberlose Rotorlageberechnung durchführen, wurde gezeigt, dass diese vollständig auf einen parallelen Kern ausgelagert werden können. Dies ist möglich, da hier die Berechnung der Rotorlage zeitlich unabhängig von der Stromregelung erfolgt, sodass eine Synchronisation beider Algorithmen nicht notwendig ist. Durch das parallele Ausführen der Rotorlageberechnung wird die Laufzeit der GFOS um die Laufzeit der Rotorlageberechnung, gegebenenfalls um ein Vielfaches davon, reduziert. Eine Obergrenze einer Laufzeitreduktion kann hier nicht gegeben werden, da die Laufzeit der Rotorlageberechnung prinzipiell beliebig lang sein kann. Durch die bekannten Größen der Regelfrequenz, der Frequenz der Rotorlageberechnung sowie der Laufzeit einer Iteration der Berechnung lässt sich die Laufzeitreduktion für eine gegebene Regelung berechnen. Am Beispiel einer GFOS unter Verwendung des SMO-Algorithmus wurde eine Reduktion der FOS-Laufzeit um 46 % nachgewiesen. Dies ermöglicht eine Steigerung der Regelfrequenz um 88,9 %. Gleichzeitig kann die Frequenz des SMO um 96,3 % erhöht werden.

Ergänzend wurde eine zweite Möglichkeit der Parallelisierung beschrieben, um die Frequenz einer zeitkontinuierlichen Rotorlageberechnung zu steigern: die Parallelisierung mittels einer Pipeline. Eine Pipeline kann zusätzlich zu der bereits beschriebenen Parallelisierung angewendet werden. Der maximale Faktor, um den die Frequenz der Rotorlageberechnung gesteigert werden kann, entspricht hierbei der Anzahl an Prozessorkernen, welche für die Pipeline genutzt werden. Am Beispiel von SMO kann dessen Ausführungsfrequenz durch eine zweistufige Pipeline (zwei Prozessorkerne für SMO, einen für FOS) nochmals verdoppelt werden.

9.2.2 Zeitdiskrete geberlose Rotorlageberechnung

Für Regelungen mit zeitdiskreter geberloser Rotorlageberechnung wurden zwei Typen identifiziert, die zur Betrachtung der Parallelisierung unterschieden werden müssen. Diese unterscheiden sich im Zeitpunkt, zu dem die berechnete Rotorlage verwendet wird.

Verfahren zur Rotorlageberechnung, die selbst auf die Rotorlage zurückgreifen müssen (beispielsweise HFCI) nutzen den in der vorangegangenen Iteration berechneten Wert der Rotorlage. Idealerweise nutzt in diesem Fall auch die FOS diesen Wert, um Phasenfehler zu reduzieren. Verwendet die FOS die in der vorangegangenen Iteration berechnete Rotorlage, so ermöglicht dies das Auslagern des gesamten

Algorithmus zur Rotorlageberechnung auf einen zweiten Kern. Nach dieser Parallelisierung entspricht die Laufzeit der GFOS dem Maximum der Laufzeiten von FOS und GRB zuzüglich der Synchronisationskosten. Dieses Vorgehen ist insbesondere dann effektiv, wenn die Rotorlageberechnung und Stromregelung eine ähnliche lange Laufzeit aufweisen. Bei Vernachlässigung der Synchronisationskosten und identischen Laufzeiten von GRB und FOS kann theoretisch eine Halbierung der GFOS-Laufzeit erreicht werden. Dementsprechend kann eine Steigerung der Regelfrequenz um bis zu 100 % erzielt werden. Am Beispiel des HFCI-Algorithmus zur Rotorlagebestimmung kann die Laufzeit der GFOS durch das parallele Ausführen von HFCI um 37,34 % verringert werden. Für die Regelfrequenz bedeutet dies ein Plus von 59,57 %. Um eine effektive Parallelisierung zu erzielen, kann theoretisch jede GFOS mit zeitdiskreter GRB so implementiert werden, dass die FOS die Rotorlage aus der vorangegangenen Iteration verwendet. Dies stellt jedoch einen Eingriff in die Abläufe der Regelung dar und bedingt eine Neubewertung der Regelung bezüglich ihrer Stabilität und Performance.

Die Effektivität der Parallelisierung nimmt deutlich ab, wenn die Rotorlage in demselben Zyklus der Stromregelung berechnet wird, in dem sie von der Regelung verwendet wird. Eine solche Berechnung wird typischerweise dann durchgeführt, wenn das Verfahren zur Rotorlageberechnung selbst nicht auf der Rotorlage basiert. Die maximal erreichbare Laufzeitreduktion ist in diesem Fall durch die Summe der Laufzeiten der Signalerfassung und der Clark-Transformation begrenzt. Unabhängig vom eingesetzten Verfahren zur Rotorlageberechnung liegt dieser Wert für den MCMC in einem Bereich von 110 CPU-Takten. Wird zu einer durchschnittlichen FOS-Laufzeit von 2164 CPU-Takten die Laufzeit einer GRB addiert, so kann durch Parallelisierung die Laufzeit der resultierenden GFOS um maximal 4,63 % verringert werden. Am Beispiel von DFC wurde auf diese Weise eine Laufzeitverringerung von 3,87 % gezeigt. Dies entspricht einer Steigerung der Regelfrequenz um 4,02 %.

Sowohl die feldorientierte Stromregelung als auch Verfahren zur geberlosen Rotorlageberechnung sind keine typischen Algorithmen, die eine hohe Datenparallelität innehaben. Prinzipiell schränkt dies die Effektivität einer Parallelisierung deutlich ein. Dies geschieht insbesondere dann, wenn häufig Daten zwischen den Prozessorkernen synchronisiert werden müssen. Um die genannten Werte zu erreichen, ist es notwendig, die Synchronisationskosten zu minimieren. Dies wurde in den Untersuchungen mittels redundanter Berechnungen erreicht. Datenwerte, die in Berechnungen auf unterschiedlichen Kernen einfließen müssen, werden hierbei von den jeweiligen Kernen berechnet. Eine Übertragung dieser Werte zwischen den Kernen und die damit

verbundenen Synchronisationskosten können somit eingespart werden. Redundante Berechnungen sind unter der Voraussetzung möglich, dass die Rechenleistung der parallel genutzten Prozessorkerne vollständig für die geberlose feldorientierte Stromregelung genutzt werden kann. Dies kann hier als gegeben betrachtet werden, da die betrachtete Stromregelung alle standardmäßigen Funktionen eines eingebetteten Motor-Controllers umfasst und somit keine weiteren Funktionalitäten berücksichtigt und integriert werden müssen. Darüber hinaus ist eine effektive Parallelisierung der Regelung auf zwei Prozessorkerne begrenzt. Mit maximal drei zur Verfügung stehenden Prozessorkernen des betrachteten MCMC sind somit noch freie Kapazitäten verfügbar, um gegebenenfalls zusätzliche Funktionen zu integrieren.

9.2.3 Parallelisierung einzelner Tasks

Die bereits beschriebenen Möglichkeiten der Parallelisierung haben die Algorithmen zur Rotorlageberechnung und zur Stromregelung im Ganzen betrachtet. Ergänzend hierzu wurde eine Parallelisierung der Tasks der Stromregelung untersucht. Die Parallelisierung dieser Tasks ist aufgrund ihrer kurzen Laufzeiten und vergleichsweise hohen Synchronisationskosten nur bedingt effektiv.

Die Effektivität der Parallelisierung steigt hier, wenn ausreichend viele Berechnungen auf unterschiedlichen Komponenten der transformierten Ströme oder Spannungen stattfinden. Dies ist beispielsweise bei SMO der Fall. Durch das parallele Ausführen dieser Berechnungen konnte eine Verringerung der Laufzeit der gesamten GFOS um 9,33 % gezeigt werden. Dies entspricht einer Erhöhung der Regelfrequenz um 10,29 %.

Wird lediglich die FOS betrachtet, so können prinzipiell die Task der Park-Transformation in Kombination mit der Task zur Regelung der Ströme parallelisiert werden. Die Transformation kann beide Stromkomponenten unabhängig voneinander berechnen. Bei der Regelung der Stromkomponenten bestehen ebenfalls keine Abhängigkeiten zwischen den jeweiligen Berechnungen. Durch die Aufteilung der Transformation und Stromregelung in entsprechende Tasks zur Behandlung der jeweiligen Komponenten und durch deren Verteilung auf zwei Prozessorkerne kann die Laufzeit der Stromreglung um 255 CPU-Takte reduziert werden. Bei einer durchschnittlichen Laufzeit von 2164 CPU-Takten entspricht dies einer erreichbaren Verkürzung der FOS-Laufzeit von bis zu 11,78 %. Dies entspricht einer Erhöhung der Regelfrequenz um bis zu 15,57 %. Hier ist zu beachten, dass diese Parallelisierung nur dann effektiv genutzt werden kann, wenn keine GRB integriert ist. Die Integration einer zeitdiskreten GRB beeinflusst die Datenströme zwischen den Transformationen

und der Regelung der Stromkomponenten, sodass zusätzliche Synchronisationskosten entstehen können. Daher muss in Abhängigkeit des jeweiligen Verfahrens zur Rotorlageberechnung eine Parallelisierung beider Tasks neu betrachtet werden.

9.3 Analyse von Cross-Core-Interferenzen

Die Integration mehrerer Motor-Controller auf einem MCMC ermöglicht die Reduktion von Hardware- und Energiekosten und eine Steigerung der Datenrate zwischen miteinander kommunizierenden Motor-Controllern. Eine Konsolidierung bedingt Seiteneffekte, die sich negativ auf die Laufzeiten der Motor-Controller auswirken. Diese Effekte wurden anhand der Softwarestruktur der geberlosen Stromregelung als auch analytisch untersucht, um ihre Auswirkungen auf eine gemeinsame Integration mehrerer Motor-Controller beurteilen zu können.

9.3.1 Strukturelle Untersuchung von Cross-Core-Interferenzen

Zur Bemessung von Interferenzen wurde die maximale Laufzeiterhöhung ermittelt, die auf eine geberlose Stromregelung wirken kann. Hierzu wurden Zugriffsmuster auf gemeinsame Speicher- und I/O-Module untersucht. Diese Muster entstehen aufgrund der verwendeten Algorithmen und unterschiedlichen Varianten der Integration mehrerer Motor-Controller auf einem Multi-Core-System. Es wurde gezeigt, dass Interferenzen aufgrund konkurrierender Zugriffe auf I/O-Module zum Lesen von ADC-Messwerten und Schreiben von PWM-Parametern die Laufzeit der Stromregelung um bis 1,82 % erhöhen. Wird zusätzlich eine geberlose Rotorlagebestimmung integriert, erhöhen sich die Laufzeiten der resultierenden GFOS um bis zu 1,16 % für HFCI, 1,77 % für SMO und 5,88 % für DFC. Bei der Verwendung von gemeinsamen Speichermodulen zum Datenaustausch zwischen Motor-Controllern erhöhen Interferenzen die Dauer einer Datenübertragung um durchschnittlich 1 % für jedes zu übertragende 4 Byte große Datenwort. Dementsprechend wird die maximal erreichbare Datenübertragungsrate reduziert.

Es wurde gezeigt, dass die zeitliche Relation zwischen den integrierten Motor-Controllern ein effektiver Parameter ist, um Interferenzen zu reduzieren. Die Eingrenzung von Hardwarezugriffen innerhalb der Motor-Controller sowie die Beschreibung ihrer parallelen Ausführung zeigen, wie die zeitliche Relation zwischen den Startzeitpunkten der Stromregelungen synchronisiert werden muss, damit Hardwarezugriffe nicht zeitgleich oder zumindest vermindert zeitgleich stattfinden. Hierdurch

können Interferenzen insbesondere dann verringert und gegebenenfalls vollständig vermieden werden, wenn die Algorithmen eine ausreichend große Rechenzeit aufweisen, in der keine Hardwarezugriffe geschehen. Diese Synchronisation gleichzeitig ausgeführter Motor-Controller hat keinen Einfluss auf die Laufzeiten der einzelnen Anwendungen.

Als implementierungsabhängiger Parameter für das Entstehen von Interferenzen wurde der zeitliche Abstand zwischen Hardwarezugriffen innerhalb eines Motor-Controllers identifiziert. Es wurde gezeigt, dass eine Vergrößerung des Abstandes durch eine Anpassung des Quellcodes Interferenzen deutlich verringern und gegebenenfalls vermeiden kann. Insbesondere das Abschalten von Compiler-Optimierungen kann Interferenzen signifikant reduzieren. Entsprechende Änderungen am Quellcode können jedoch zu einer Erhöhung der absoluten Laufzeit des Motor-Controllers führen. Diese fällt gegebenenfalls deutlich höher aus als der Gewinn durch die Reduktion von Interferenzen. Anstelle des Gewinns in Form einer Laufzeitreduktion muss an dieser Stelle generell das Auftreten von Interferenzen betrachtet werden. Durch die Beachtung implementierungsabhängiger Parameter kann das Maß, indem Interferenzen nach einer Integration der Motor-Controller in ein Multi-Core-System auftreten, verringert werden. Entsprechend sinken auch die negativen Effekte, die nach einer Integration auftreten können. Eine angepasste Implementierung kann Interferenzen verhindern, sodass im Idealfall die Laufzeit eines Motor-Controllers auf einem Single-Core-System, der auf einem Multi-Core-System entspricht. Dies vereinfacht den Integrationsprozess von Motor-Controllern in einen MCMC deutlich. Es ist somit anwendungsabhängig abzuwägen, ob diese Vereinfachung höhere Laufzeiten rechtfertigt. Eine entsprechende Anpassung des Quellcodes ist jedoch nicht immer möglich oder sinnvoll, insbesondere dann, wenn Code ausgehend von mathematischen Modellen der Regelung automatisch generiert wird. Die Anpassung von generiertem Code kann zusätzliche Fehler in die Anwendung einbringen. Zudem entsteht eine Inkonsistenz mit dem Modell, wodurch nachträgliche Anpassungen deutlich erschwert werden können.

9.3.2 Analytische Untersuchung von Cross-Core-Interferenzen

Zur analytischen Ermittlung der Effekte von Cross-Core-Interferenzen wurde ein neues Verfahren vorgestellt, das resultierende Laufzeiterhöhungen auf Basis der Single-Core-Ausführungen der konsolidierenden Motor-Controller berechnet. Auf Basis von Instruction-Traces der Single-Core-Ausführung wird eine parallele Aus-

führung der Controller simuliert, dabei entstehende konkurrierende Modulzugriffe extrahiert und die resultierenden Laufzeiterhöhungen berechnet. Hierbei können unterschiedliche Integrationsparameter und Implementierungsvarianten berücksichtigt werden, um Konfigurationen mit maximalen und minimalen Laufzeiterhöhungen zu finden. Die Berechnung Laufzeiterhöhung basiert auf Aufzeichnungen der Motor-Controller-Ausführung auf Single-Core-Systemen. Um vollständige Ergebnisse zu erhalten, muss bei diesen Aufnahmen sichergestellt sein, dass sie den vollständigen Code der Motor-Controller beinhalten. Für die betrachteten Anwendungen der Antriebstechnik kann dieses Kriterium jedoch erfüllt werden, da die Anwendungen bezüglich bedingter Anweisungen und Schleifen eine vergleichsweise geringe Komplexität aufweisen. Bei ausreichend vielen Regelzyklen kann hierdurch eine vollständige Abdeckung des Codes erreicht und über den Instruction-Trace verifiziert werden.

Neben den Aufzeichnungen der Motor-Controller-Ausführung beruhen die ermittelten Laufzeitverlängerungen auf hardwarespezifischen Informationen über die eingesetzten Multi-Core-Mikrocontroller. Die Verfügbarkeit dieser Informationen ist eine Voraussetzung, um präzise Laufzeiten zu berechnen. Ein Großteil dieser Informationen wird durch entsprechende Datenblätter bereitgestellt oder kann experimentell ermittelt werden. Für den Multi-Core-Mikrocontroller des MCMC liegt die Fehlerabweichung für die berechneten Laufzeiten im Durchschnitt unter 1 %. Abweichungen sind durch die verwendeten Timings und hardwareseitigen Optimierungen bezüglich der Instruktionsausführung begründet. Insbesondere Sprungvorhersagen des Mikroprozessors stören ein exaktes Berechnen der Timings von Hardwarezugriffen, die aus diesen Vorhersagen entstehenden. Diese Problematik tritt insbesondere bei Anwendungen mit verschachtelten Schleifen und bedingten Verzweigungen auf. Die betrachteten Anwendungen der Antriebstechnik nutzen diese Elemente nur in einem geringen Maße und typischerweise ohne Hardwarezugriffe darüber zu steuern. Dementsprechend kann allgemein für die geberlose Stromregelung von einer deutlich geringeren Fehlerabweichung ausgegangen werden.

9.4 Übertragbarkeit der Ergebnisse

Die Untersuchungen zur Leistungsaufnahme, zur Parallelisierbarkeit und die Betrachtung und Berechnung von Cross-Core-Interferenzen wurden anhand konkreter Anwendungsalgorithmen sowie einer definierten Multi-Core-Plattform durchgeführt. Trotz dieser Tatsachen sind die Ergebnisse nicht auf die verwendeten Algorithmen und die Plattform beschränkt. Die gezeigten Vorgehensweisen und Ergebnisse kön-

nen auf andere Algorithmen des Anwendungsgebiets sowie alternative Multi-Core-Plattformen übertragen werden.

9.4.1 Übertragbarkeit auf andere Multi-Core-Plattformen

Die Multi-Core-Plattform setzt sich aus einem Echtzeitbetriebssystem und einem Multi-Core-Mikrocontroller zusammen. Das parallele Ausführen der Anwendungen nutzt keine spezifischen Eigenschaften des eingesetzten Betriebssystems oder des Multi-Core-Mikrocontrollers. Das Definieren und Ausführen von Funktionen in Form von Tasks und Interrupt-Service-Routinen sowie atomare Operationen zum Synchronisieren dieser Funktionen sind Standardfunktionen moderner Betriebssysteme und Multi-Core-Mikrocontroller.

Für eine Konsolidierung von Motor-Controllern werden Safety-Funktionen des Betriebssystems und des Multi-Core-Mikrocontrollers genutzt. Solche Funktionen sind keine standardmäßigen Eigenschaften von Multi-Core-Mikrocontrollern und entsprechenden Betriebssystemen. Mit Ausnahme des Datenaustauschs zwischen konsolidierten Systemen haben diese Funktionen keinen Einfluss auf die gezeigten Vorgehensweisen und Ergebnisse. Bezüglich des Datenaustauschs beeinflussen sie lediglich die Laufzeiten des Nachrichtenaustauschs zwischen Motor-Controllern. Die beschriebenen parallelen Abläufe und konkurrierenden Zugriffe konsolidierter Motor-Controller werden hiervon nicht beeinflusst. Dementsprechend können die Ergebnisse auf Systeme ohne Safety-Funktionen übertragen werden.

9.4.2 Anwendbarkeit der Parallelisierung auf alternative Algorithmen

Allgemein kann die feldorientierte Stromregelung durch Erweiterungen und Optimierungen des Algorithmus von der in dieser Arbeit genutzten Implementierung abweichen. Die Parallelisierung erfolgt auf Basis eines Taskmodells, dessen Tasks Erweiterungen und Optimierungen der Regelung abbilden können. In Abhängigkeit vom jeweiligen Verfahren zur geberlosen Rotorlagebestimmung werden entsprechende Algorithmen softwareseitig mittels unterschiedlicher Strategien in eine feldorientierte Stromregelung integriert. Diese Strategien lassen sich auf zwei Varianten begrenzen, die im Rahmen der Analyse betrachtet wurden. Somit lassen sich die gezeigten Vorgehensweisen und Ergebnisse der Parallelisierung auf alternative Anwendungen beziehungsweise Implementierungen zur geberlosen Stromregelung übertragen, wenn sich die feldorientierte Stromregelung durch das Taskmodell abbilden

und die geberlose Rotorlagebestimmung entsprechend den verwendeten Strategien integrieren lässt.

9.4.3 Anwendbarkeit der Berechnung von Interferenzen auf alternative Multi-Core-Mikrocontroller

Das Modell zur Berechnung von Cross-Core-Interferenzen umfasst Architekturmerkmale des Multi-Core-Mikrocontrollers und der Prozessorkerne. Konkret sind dies mehrere separate Bussysteme für die Anbindung der Prozessorkerne an Speicher- und I/O-Module sowie eine out-of-order Ausführung von Schreibbefehlen unter Verwendung von Store-Buffern. Dies sind jedoch typische Architekturmerkmale und finden sich auch in alternativen Multi-Core-Mikrocontrollern wieder. Variierende Ausprägungen dieser Architekturmerkmale werden durch das Modell zur Interferenzberechnung berücksichtigt, sodass es auf andere Multi-Core-Mikrocontroller angepasst und angewendet werden kann.

9.4.4 Anwendbarkeit der Energiebetrachtung auf alternative Multi-Core-Mikrocontroller

Das Modell nutzt einen allgemeinen Aufbau von Mikrocontrollern. Es kann auf andere Systeme angewendet werden, insofern diese das Deaktivieren einzelner On-Chip-Module unterstützen.

9.4.5 Übertragbarkeit der gemessenen Laufzeiten

Die experimentell ermittelten Laufzeiten sind spezifisch für den eingesetzten Multi-Core-Mikrocontroller. Es werden jedoch keine speziellen Instruktionen der Prozessorkerne verwendet. Zudem weisen die Kerne eine Architektur auf, die für Systeme aus dem Automotive-Bereich typisch ist. Entsprechende Merkmale sind insbesondere Scratchpad-Arbeitsspeicher, parallele Pipelines für Load-/Store- und Arithmetische-Instruktionen und eine out-of-order Ausführung schreibender Zugriffe auf Speicher- und I/O-Module. Aufgrund dieser Merkmale ist zu erwarten, dass alternative Multi-Core-Mikrocontroller Laufzeiten aufweisen, die in der gleichen Größenordnung wie die gemessenen Daten liegen. Dementsprechend können Aussagen bezüglich der Parallelisierbarkeit und dem Maß von Interferenzen in ihren Grundzügen auf andere Multi-Core-Mikrocontroller übertragen werden.

Die gezeigten Verfahren zur Gewinnung der entsprechenden Daten können uneingeschränkt übertragen werden, insofern der verwendete Multi-Core-Mikrocontroller Werkzeuge zur Messung von Laufzeiten bereitstellt. Diese Werkzeuge sind Standardfunktionen moderner Mikrocontroller.

Kapitel 10

Fazit und Ausblick

Die vorliegende Arbeit befasst sich mit der Fragestellung, ob sich Multi-Core-Mikrocontroller als Basis für neue Implementierungsmethoden zur Realisierung von Motor-Controllern zur geberlosen Stromregelung von PMSM eignen. Die Motivation für den Einsatz von Multi-Core-Mikrocontrollern ist eine effiziente Steigerung der Rechenleistung von Motor-Controllern sowie Möglichkeiten zur Reduktion von Hardware- und Energiekosten.

Ergänzend zu den beiden zentralen Forschungsfragen wurden die Einflüsse von Multi-Core-Mikrocontrollern auf den Energiebedarf von Motor-Controllern untersucht. Der Wechsel von einer standardmäßig verwendeten Single-Core-Architektur zu einem Multi-Core-Mikrocontroller erhöht die Leistungsaufnahme eines Motor-Controllers in Abhängigkeit der verwendeten Rechenleistung in einer Größenordnung zwischen 10 % und 30 %. Bezogen auf diesen Gesichtspunkt können MCMC nur dann als gewinnbringende Implementierungsmethode betrachtet werden, wenn insgesamt mehr Rechenleistung der parallelen Kerne genutzt wird, als ein Single-Core-System bereitstellt. Die Möglichkeiten, die Algorithmen von Motor-Controllern zu parallelisieren, sind vergleichsweise gering, wodurch keine hohe Auslastung mehrerer Prozessorkerne zu erwarten ist. Ein Multi-Core-Mikrocontroller ist jedoch höchst effizient, wenn er als Ersatz für mehrere Single-Core Motor-Controller eingesetzt wird. Der Energiebedarf seitens der Motor-Controller kann hierdurch um bis zu 50 % reduziert werden.

Untersuchung der Parallelisierbarkeit von geberlosen Stromregelungen

Die erste betrachtete Forschungsfrage umfasst die Untersuchung der Parallelisierbarkeit geberloser Motor-Controllern für PMSM mittels Multi-Core-Mikrocontroller. Aufgrund einer vergleichsweise geringen Datenparallelität gehören Algorithmen zur geberlosen Stromregelung nicht zu solchen, die sich naturgemäß effektiv paralle-

lisieren lassen. Da die feldorientierte Stromregelung und die geberlose Rotorlageberechnung größtenteils voneinander unabhängige Berechnungen durchführen, ist eine effektive Parallelisierung jedoch nicht ausgeschlossen.

Zur Untersuchung der Parallelisierbarkeit wurde ein Modell zur Eingrenzung der zu betrachtenden Algorithmen und zur Beschreibung ihrer Integration auf einem Single-Core-Mikrocontroller aufgebaut. Anhand dieses Modells wurden die Möglichkeiten zur Parallelisierung und Integration der Algorithmen auf einem Multi-Core-Mikrocontroller abgeleitet und evaluiert. Die Ergebnisse zeigen, dass eine effektive Parallelisierung möglich ist, wenn die Stromregelung mit einem Phasenversatz von mindestens einer Iteration zur Rotorlageberechnung ausgeführt werden kann. Hier konnte eine Steigerung der Regelfrequenz erzielt werden, die anwendungsabhängig in einem Bereich zwischen 50 % und 90 % liegt. Diese Ergebnisse werden insbesondere für Motor-Controller erzielt, welche die Verfahren SMO und HFCI zur Rotorlageberechnung einsetzen. Dies erhöht die Relevanz dieser sehr guten Ergebnisse nochmals, da die Verbreitung von SMO und HFCI in praxisnahen Anwendungen kontinuierlich zunimmt. Zur Realisierung der Parallelisierung beschreibt das Modell die Aufteilung geberloser Stromregelungen in Tasks sowie deren Verteilung und Synchronisation auf parallele Prozessorkerne. In Abhängigkeit der jeweiligen Anwendungen werden dadurch Vorgehensweisen beschrieben, um die jeweils erreichbare Laufzeitreduktion abzuschätzen.

Diese Ergebnisse erweitern die Design- und Implementierungsprozesse von Motor-Controllern, indem Multi-Core-Mikrocontroller in diese Prozesse einbezogen werden können. Dies beginnt bei der Entscheidungsfindung zur Designzeit des Motor-Controllers, ob ein effektiver und effizienter Einsatz eines Multi-Core-Mikrocontrollers in dem Motor-Controller möglich ist. Hierdurch kann beurteilt werden, ob eine Parallelisierung beziehungsweise ein MCMC aus funktionaler Sicht einen Mehrgewinn für das jeweilige Antriebssystem bietet. Die in dieser Arbeit vorgestellten Strategien zur Verteilung und Synchronisation von Tasks unterstützen hierbei die Entwicklung paralleler Software zur Realisierung eines MCMC. Aus nicht-funktionaler Sicht kann die zusätzliche Betrachtung der Leistungsaufnahme des entstehenden MCMC eine Beurteilung ergänzen.

Untersuchung von Cross-Core-Interferenzen bei konsolidierten Motor-Controllern

Die zweite betrachtete Forschungsfrage behandelt die Untersuchung der Einflüsse von Cross-Core-Interferenzen auf das zeitliche Verhalten von Motor-Controllern. Interferenzen entstehen durch konkurrierende Hardwarezugriffe. Sie wirken in Form

von Laufzeiterhöhungen und treten regelmäßig nach einer gemeinsamen Integration mehrerer dedizierter Single-Core-Systeme in ein Multi-Core-System auf. Diese Laufzeiterhöhungen beeinträchtigen das Echtzeitverhalten der Controller und damit ihre korrekte Funktionsweise.

Anhand des erstellten Modells für geberlose Stromregelungen wurden die Auswirkungen von Interferenzen auf die Laufzeiten von konsolidierten Motor-Controllern untersucht. Aus der Untersuchung geht hervor, dass die zur Signalanalyse und Signalerzeugung notwendigen Hardwarezugriffe Interferenzen erzeugen, welche die Laufzeiten der Motor-Controller in einem Bereich zwischen 1 % und 6 % erhöhen können. Die konkurrierenden Zugriffe werden durch ihre Abhängigkeiten von den Ausführungsfrequenzen der Algorithmen und von der Synchronisation der Prozessorkerne charakterisiert. Anhand dieser Charakterisierung wurden Strategien abgeleitet, um Konkurrenz und damit das Auftreten von Laufzeitverlängerungen zu reduzieren und gegebenenfalls zu verhindern. Bezüglich der Implementierung sind dies Strategien, welche die Umsetzung und Platzierung von Hardwarezugriffen auf der Ebene des Quellcodes optimieren. Seitens der Integration wird eine Optimierung bezüglich Synchronisation der gleichzeitig ausgeführten Motor-Controller vorgestellt. Ergänzend wurde eine analytische Untersuchung von Cross-Core-Interferenzen durchgeführt. Hieraus ist ein neues Verfahren hervorgegangen, um konkurrierende Hardwarezugriffe zu identifizieren und die entstehenden Laufzeitverlängerungen zu berechnen. An dieser Stelle ist hervorzuheben, dass bei den Berechnungen keine Ausführung der Motor-Controller auf dem Multi-Core-Mikrocontroller notwendig ist. So werden kritische Situationen während des Entwicklungsprozesses eines MCMC verhindert, die durch ein fehlerhaftes Verhalten der Antriebe hervorgerufen werden können. Die aus den Berechnungen entstehenden Daten unterstützen die genannten Möglichkeiten zur Reduktion von Cross-Core-Interferenzen, indem sie Daten bezüglich der Optimierungen der Motor-Controller-Implementierung und deren Integration bereitstellen.

Ergänzend zur Parallelisierung leisten die Ergebnisse der Untersuchung von Cross-Core-Interferenzen einen weiteren Beitrag zur Erweiterung des Entwicklungsprozesses von Motor-Controllern. Der Prozess wird dahingehend erweitert, dass Multi-Core-Mikrocontroller zur gleichzeitigen Integration mehrerer Motor-Controller eingesetzt werden. Methoden zur Evaluation der Effekte von Cross-Core-Interferenzen und zu deren Reduktion ergänzen bisherige Methoden zur Implementierung und Integration von Motor-Controllern. Wie anfangs erläutert, bieten die so entstehenden Multi-Core Motor-Controller signifikante Möglichkeiten zum Einsparen von

Hardware- und Energiekosten für das Anwendungsgebiet.

Ausblick
Diese Arbeit leistet einen grundlegenden Beitrag, um Multi-Core-Mikrocontroller in digitalen Motor-Controllern zu nutzen. Aufbauend auf diesen Ergebnissen können Anschlussarbeiten den praxisnahen Einsatz von Multi-Core Motor-Controllern nachhaltig festigen.

Hierzu sind die Formalisierung der Konzepte zur Parallelisierung der Algorithmen der Antriebstechnik und die Integration dieser Parallelisierung in die modellgetriebene Entwicklung antriebstechnischer Anwendungen zu realisieren. Dies versetzt entsprechende Codegeneratoren in der Lage, parallele Tasks zu erzeugen und mit Synchronisationsmethoden auszustatten. Auf diese Weise wird fehlerfreier Quellcode erzeugt und die Konsistenz zwischen dem Quellcode und den ihm zugrunde liegenden Modellen bleibt erhalten.

Im Rahmen der Konsolidierung gilt es automatisch Code zu erzeugen, der eine verminderte Anfälligkeit für Interferenzen aufweist. Hierzu ist die modellgetriebene Entwicklung antriebstechnischer Anwendungen durch die Abstraktion des Multi-Core Motor-Controllers zu erweitern. Formalisierte Regeln zur Erzeugung von interferenzsicherem Code sowie hardwarespezifische Informationen über Zugriffe auf geteilte Module können ausgehend von dieser Ebene in den gesamten Modellierungsprozess bis hin zur Codegenerierung eingehen. Die Codegenerierung erhält auf diese Weise Informationen über das Gesamtsystem, um interferenzsicheren Code aus den Modellen der Antriebsregelung zu erzeugen.

Die vorgestellte Berechnung von Laufzeitverlängerungen berechnet keine oberen Schranken der Laufzeiten. Es ist sinnvoll, die Ergebnisse dieses Verfahrens mit denen aus WCET-Analysen zu vergleichen, um bezüglich oberer Schranken die Zuverlässigkeit der Ergebnisse zu evaluieren. Um den Entwicklungsprozess hinsichtlich der Integration von Motor-Controllern in einen MCMC zu vervollständigen, ist eine automatische Verifikation und Optimierung des MCMC notwendig. Dies umfasst eine Verknüpfung der modellierten und zu integrierenden Motor-Controller mit Systemen zur Gewinnung der notwendigen Daten zur Berechnung von Laufzeitverlängerungen. Letztere können beispielsweise durch Emulatoren der Mikroprozessoren oder Hardware-in-the-Loop-Systeme gewonnen werden. Die so gewonnenen Ergebnisse können direkt in den Entwicklungsprozess zurückfließen, um beispielsweise die Synchronisation der Motor-Controller zur Reduktion von Interferenzen anzupassen.

Literaturverzeichnis

[1] Agarwal, V. ; Hrishikesh, M S. ; Keckler, S W. ; Burger, D: Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In: *Proceedings of 27th International Symposium on Computer Architecture*, 2000, S. 248–259

[2] Akhter, Shameem ; Roberts, Jason: *Multi-Core Programming*. Intel Press, 2006

[3] Akin, Özkan ; Alan, Irfan: The Use of FPGA in Field-Oriented Control of an Induction Machine. In: *Turkish Journal of Electrical Engineering and Computer Science* (2010), S. 934–962

[4] Al-Ayasrah, O ; Alukaidey, T ; Pissanidis, G: DSP Based N-Motor Speed Control of Brushless DC Motors Using External FPGA Design. In: *IEEE International Conference on Industrial Technology*, 2006, S. 627–631

[5] Andraka, Ray: A Survey of CORDIC Algorithms for FPGA Based Computers. In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. New York, NY, USA : ACM, 1998, S. 191–200

[6] ARM Limited (Hrsg.): *Cortex-R4 and Cortex-R4F Technical Reference Manual*. Revision: r1p3. CB1 9NJ Cambridge, UK: ARM Limited, 2011

[7] Attia, Khaled ; El-Hosseini, Mostafa ; Ali, Hesham: Dynamic power management techniques in multi-core architectures: A survey study. In: *Ain Shams Engineering Journal* 8 (2017), Nr. 3, S. 445–456

[8] Awan, Muhammad ; Petters, Stefan: Energy-Aware Partitioning of Rasks onto a Heterogeneous Multi-Core Platform. In: *19th Real-Time and Embedded Technology and Applications Symposium*, 2013, S. 205–214

[9] Bae, Bon-Ho ; Sul, Seung-Ki: A Compensation Method for Time Delay of Full-Digital Synchronous Frame Current Regulator of PWM AC Drives. In: *IEEE Transactions on Industry Applications* 39 (2003), Nr. 3, S. 802–810

[10] Bafumba-Lokilo, David ; Savaria, Yvon ; David, David: Generic Crossbar Network on Chip for FPGA MPSoCs. In: *Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, 2008, S. 269–272

[11] Bambagini, Mario ; Marinoni, Mauro ; Aydin, Hakan ; Buttazzo, Giorgio: Energy-Aware Scheduling for Real-Time Systems: A Survey. In: *ACM Transactions on Embedded Computing Systems (TECS)* 15 (2016), Nr. 1

[12] Basmadjian, Robert ; Meer, Hermann de: Evaluating and Modeling Power Consumption of Multi-Core Processors. In: *Third International Conference on Future Systems: Where Energy, Computing and Communication Meet*, IEEE, 2012, S. 1–10

[13] Behnam, Moris ; Inam, Rafia ; Nolte, Thomas ; din, Mikael Sj o.: Multicore Composability in the Face of Memory-bus Contention. In: *SIGBED Rev.* 10 (2013), Nr. 3, S. 35–42

[14] Ben Othman, S ; Ben Salem, A K. ; Saoud, Slim: MPSoC Design of RT control Applications Based on FPGA SoftCore Processors. In: *15th IEEE International Conference on Electronics, Circuits and Systems*, 2008, S. 404–409

[15] Benini, Luca ; De Micheli, Giovanni: Networks on Chips: A New SoC Paradigm. 35 (2002), Nr. 1, S. 70–78

[16] Berkel, C van: Multi-Core for Mobile Phones. In: *Design, Automation and Test in Europe*, IEEE, 2009, S. 1260–1265

[17] Bishop, Robert: Motion-Control. In: *Mechatronic System Control, Logic, and Data Acquisition*, CRC Press, 2007, S. 15.1 – 15.25

[18] Blake, Geoffrey ; Dreslinski, Ronald ; Mudge, Trevor: A Survey of Multicore Processors. In: *IEEE Signal Processing Magazine* 26 (2009), Nr. 6, S. 26–37

[19] Blaschke, Felix: *Das verfahren der feldorientierung zur regelung der asynchronmaschine*, Technische Universität Braunschweig, Dissertation, 1973

[20] Bode, Arndt: Multicore-Architekturen. In: *Informatik-Spektrum* 29 (2006), Nr. 5, S. 349–352

[21] Bohr, Mark: The New Era of Scaling in an SoC World. In: *IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2009, S. 23–28

[22] Bregenzer, Jürgen: *Effizienter Einsatz von Multicore-Architekturen in der Steuerungstechnik*, Julius-Maximilians-Universität Würzburg, Dissertation, 2014

[23] Buccella, Concettina ; Cecati, Carlo ; Latafat, Hamed: Digital Control of Power Converters - A Survey. In: *IEEE Trans. Industrial Informatics* 8 (2012), Nr. 3, S. 437–447

[24] Bueno, Emilio ; Hernandez, Álvaro ; Rodriguez, Francisco ; Girón, Carlos ; Mateos, Raúl ; Cobreces, Santiago: A DSP- and FPGA-Based Industrial Control With High-Speed Communication Interfaces for Grid Converters Applied to Distributed Power Generation Systems. In: *IEEE Transactions on Industrial Electronics* 56 (2009), Nr. 3, S. 654–669

[25] Chandrakasan, Anantha ; Brodersen, Robert: Minimizing Power Consumption in Digital CMOS Circuits. In: *Proceedings of the IEEE* 83 (1995), Nr. 4, S. 498–523

[26] Chang, Yen-Chuan ; Tzou, Ying-Yu: Single-chip FPGA Implementation of a Sensorless Speed Control IC for Permanent Magnet Synchronous Motors. In: *IEEE Power Electronics Specialists Conference*, 2007, S. 593–598

[27] Codrescu, Lucian ; Anderson, Willie ; Venkumanhanti, Suresh ; Zeng, Mao ; Plondke, Erich ; Koob, Chris ; Ingle, Ajay ; Tabony, Charles ; Maule, Rick: Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. In: *IEEE Micro* 34 (2014), Nr. 2, S. 34–43

[28] Corley, Matthew ; Lorenz, Robert: Rotor Position and Velocity Estimation for a Salient-Pole Permanent Magnet Synchronous Machine at Standstill and High Speeds. In: *IEEE Transactions on Industry Applications* 34 (1998), Nr. 4, S. 784–789

[29] Cullmann, Christoph ; Ferdinand, Christian ; Gebhard, Gernot ; Grund, Daniel ; Maiza, Claire ; Reineke, Jan ; Triquet, Benoît ; Wegener, Simon ; Wilhelm, Reinhard: Predictability Considerations in the

Design of Multi-Core Embedded Systems. In: *Ingénieurs de l'Automobile* 807 (2010), S. 36–42

[30] Cumming, Peter: The TI OMAP^{TM} Platform Approach to SOC. In: Martin, Grant (Hrsg.) ; Chang, Henry (Hrsg.): *Winning the SoC Revolution: Experiences in Real Design*. Boston, MA : Springer US, 2003, S. 97–118

[31] Czechowski, Kent ; Battaglino, Casey ; McClanahan, Chris ; Chandramowlishwaran, Aparna ; Vuduc, Richard: Balance Principles for Algorithm-architecture Co-design. In: *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, USENIX Association, 2011, S. 9–9

[32] Dasari, Dakshina ; Nelis, Vincent ; Akesson, Benny: A Framework for Memory Contention Analysis in Multi-Core Platforms. In: *Real-Time Systems* 52 (2016), Nr. 3, S. 272–322

[33] Dayu Wang ; Kaiping Yu ; Hong Guo: Functional Design of FPGA in a Brushless DC Motor System Based on FPGA and DSP. In: *IEEE Vehicle Power and Propulsion Conference*, 2008, S. 1–4

[34] Dhaouadi, Rached ; Mohan, Ned ; Norum, Lars: Design and Implementation of an Extended Kalman Filter for the State Estimation of a Permanent Magnet Synchronous Motor. In: *IEEE Transactions on Power Electronics* 6 (1991), Nr. 3, S. 491–497

[35] Diao, Lijun ; Tang, Jing ; Loh, Poh ; Yin, Shaobo ; Wang, Lei ; Liu, Zhigang: An Efficient DSP–FPGA-Based Implementation of Hybrid PWM for Electric Rail Traction Induction Motor Control. In: *IEEE Transactions on Power Electronics* 33 (2018), Nr. 4, S. 3276–3288

[36] Dijkstra, Edsger: Cooperating Sequential Processes. In: *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, Springer New York, 2002, S. 65–138

[37] Ditty, Michael ; Montrym, John ; Wittenbrink, Craig: NVIDIA'S Tegra K1 System-on-Chip. In: *IEEE Hot Chips 26 Symposium (HCS)*, 2014, S. 1–26

[38] Dogan, Ibrahim: *Microcontroller Based Applied Digital Control*. John Wiley & Sons Ltd., 2006

[39] EL SALLOUM, Christian ; ELSHUBER, Martin ; HÖFTBERGER, Oliver ; ISAKOVIC, Haris ; WASICEK, Armin: The ACROSS MPSoC - A new generation of multi-core processors designed for safety-critical embedded systems. In: *Microprocessors and Microsystems* 37 (2013), Nr. PC, S. 1020–1032

[40] EMBEDDED OFFICE GMBH & CO. KG: Flexible Safety RTOS. In: *https://www.embedded-office.com/de/safety-rtos.html* (2019). – Version: 02.06.2019

[41] EMMA, Philip ; DAVIDSON, Edward: Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance. In: *IEEE Trans. Computers* C-36 (1987), Nr. 7, S. 859–875

[42] EVIDENCE EMBEDDED TECHNOLOGY: Erika Enterprise RTOS v3. In: *http://www.erika-enterprise.com* (2019). – Version: 02.06.2019

[43] FABBRI, Stefano ; SCHUHMACHER, Klaus ; NIENHAUS, Matthias ; GRASSO, Emanuele: Combination of two different sensorless techniques for complete speed range sensorless drive and control of small sized PMSMs. In: *Innovative Small Drives and Micro-Motor Systems*, 2019, S. 1–6

[44] FERNANDEZ, Gabriel ; JALLE, Javier ; ABELLA, Jaume ; QUIÑONES, Eduardo ; VARDANEGA, Tullio ; CAZORLA, Francisco J.: Increasing Confidence on Measurement-Based Contention Bounds for Real-Time Round-Robin Buses. In: *52nd Annual Design Automation Conference*. New York, New York, USA : ACM Press, 2015, S. 1–6

[45] FERNÁNDEZ, Mikel ; GIOIOSA, Roberto ; QUIÑONES, Eduardo: Assessing the Suitability of the NGMP Multi-Core Processor in the Space Domain. In: *Proceedings of the tenth ACM international conference on Embedded software*, 2012, S. 175–184

[46] FLYNN, Michael J. ; RUDD, Kevin W.: Parallel architectures. In: *ACM Computer Surveys* 28 (1996), Nr. 1, S. 67–70

[47] FREE SOFTWARE FOUNDATION, INC.: *GCC online documentation - Options That Control Optimization*. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, 23. Juli 2021. – Zugegriffen: 24.03.2021

[48] FREESCALE SEMICONDUCTOR INC. (Hrsg.): *e200z4 Power Architecture Core - Reference ManuaL*. e200z4RM Rev. 0. Tempe Arizona 85284, USA: Freescale Semiconductor Inc., 2019

[49] Fuchsen, Rudolf: How to address certification for multi-core based IMA platforms: Current status and potential solutions. In: *29th Digital Avionics Systems Conference*, 2010, S. 5.E.3–1–5.E.3–11

[50] Fujii, Yusuke ; Azumi, Takuya ; Nishio, Nobuhiko ; Kato, Shinpei ; Edahiro, M: Data Transfer Matters for GPU Computing. In: *2013 International Conference on Parallel and Distributed Systems*, 2013, S. 275–282

[51] Furber, Steve: *ARM System-on-Chip Architecture*. 2nd. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000

[52] Gal-On, Shay ; Levy, Markus: *Exploring CoreMark$_{TM}$ - A Benchmark Maximizing Simplicity and Efficacy*. The Embedded Microprocessor Consortium, 2012

[53] Girkar, Milind ; Polychronopoulos, Constantine: Automatic Extraction of Functional Parallelism from Ordinary Programs. In: *IEEE Transactions on Parallel and Distributed Systems* 3 (1992), Nr. 2, S. 166–178

[54] Golyanik, Vladislav ; Nasri, Mitra ; Stricker, Didier: Towards Scheduling Hard Real-Time Image Processing Tasks on a Single GPU. In: *IEEE International Conference on Image Processing*, 2017, S. 4382–4386

[55] Grasso, Emanuele ; Merl, Daniel ; Nienhaus, Matthias: A Direct Flux Observer for implementation of PMSMs sensorless control in embedded systems. In: *42nd Annual Conference of the IEEE Industrial Electronics Society*, 2016, S. 6657–6662

[56] Grecu, Cristian ; Pande, Partha P. ; Ivanov, André ; Saleh, Res: Structured Interconnect Architecture - A Solution for the Non-Scalability of Bus-Based SoCs. In: *ACM Great Lakes Symposium on VLSI* (2004), S. 192

[57] Guan, Nan ; Stigge, Martin ; Yi, Wang ; Yu, Ge: Cache-aware Scheduling and Analysis for Multicores. In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. New York, USA : ACM, 2009, S. 245–254

[58] Guangzhen, Zhang ; Feng, Zhao ; Yongxing, Wang ; Xuhui, Wen ; Wei, Cong: Analysis and Optimization of Current Regulator Time Delay in Permanent Magnet Synchronous Motor Drive System. In: *International Conference on Electrical Machines and Systems*, 2013, S. 2286–2290

[59] Guerrier, Pierre ; Greiner, Alain: Architecturally Homogeneous Power-Performance Heterogeneous Multicore Systems. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21 (2013), Nr. 4, S. 670–679

[60] Gustafsson, Jan ; Betts, Adam ; Ermedahl, Andreaas ; Lisper, Björn: The Mälardalen WCET Benchmarks: Past, Present And Future. In: *10th International Workshop on Worst-Case Execution Time Analysis* 15 (2010), S. 136–146

[61] Hammarlund, P ; Martinez, A J. ; Bajwa, A A. ; Hill, D L. ; Hallnor, E ; Jiang, H ; Dixon, M ; Derr, M ; Hunsaker, M ; Kumar, R. ; Osborne, R B. ; Rajwar, R ; Singhal, R ; D'Sa, R ; Chappell, R ; Kaushik, S ; Chennupaty, S ; Jourdan, S ; Gunther, S ; Piazza, T ; Burton, T: Haswell: The Fourth-Generation Intel Core Processor. In: *IEEE Micro* 34 (2014), Nr. 2, S. 6–20

[62] Hamouda, Mahmoud ; Blanchette, Handy ; Al-Haddad, Kamal ; Fnaiech, Farhat: An Efficient DSP-FPGA-Based Real-Time Implementation Method of SVM Algorithms for an Indirect Matrix Converter. In: *IEEE Transactions on Industrial Electronics* 58 (2011), Nr. 11, S. 5024–5031

[63] Hattendorf, Anton ; Raabe, Andreas ; Knoll, Alois: Shared Memory Protection for Spatial Separation in Multicore Architectures. In: *7th IEEE International Symposium on Industrial Embedded Systems*, 2012, S. 299–302

[64] Henrey, Michael ; Edmond, Sean ; Shannon, Lesley ; Menon, Carlo: Bio-Inspired Walking: A FPGA Multicore System for a Legged Robot. In: *22nd International Conference on Field Programmable Logic and Applications*, 2012, S. 105–111

[65] Herbert, Sebastian ; Marculescu, Diana: Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In: *Proceedings of the IEEE* (2007), S. 38–43

[66] HighTec EDV Systeme GmbH: PXROS - Real-time OS for TriCore and AURIX. In: *https://hightec-rt.com/en/products/real-time-os.html* (2019). – Version: 02.06.2019

[67] HighTec EDV Systeme GmbH: Tricore Development Platform. In: *https://hightec-rt.com/en/products/development-platform.html* (2019). – Version: 02.06.2019

[68] Hirata, Kazuyuki ; Goodacre, John: ARM MPCore; The Streamlined and Scalable ARM11 Processor Core. In: *19th Asia and South Pacific Design Automation Conference*, 2007, S. 747–748

[69] Hovland, Rune: *Latency and Bandwidth Impact on GPU-Systems*, Norwegian University of Science and Technology, Technical Report, 2008

[70] Hwang, Seon-Hwan ; Liu, Xiaohu ; Kim, Jang-Mok ; Li, Hui: Distributed Digital Control of Modular-Based Solid-State Transformer Using DSP+FPGA. In: *IEEE Transactions on Industrial Electronics* 60 (2013), Nr. 2, S. 670–680

[71] Idkhajine, L ; Monmasson, E ; Maalouf, A: Extended Kalman Filter for AC Drive Sensorless Speed Controller - FPGA-Based Solution or DSP-Based Solution. In: *IEEE International Symposium on Industrial Electronics*, 2010, S. 2759–2764

[72] Idkhajine, Lahoucine ; Monmasson, Eric ; Maalouf, Amira: Fully FPGA-Based Sensorless Control for Synchronous AC Drive Using an Extended Kalman Filter. In: *IEEE Transactions on Industrial Electronics* 59 (2012), Nr. 10, S. 3908–3918

[73] Infineon Technologies AG (Hrsg.): *TriCore® TC1.6P & TC1.6E Instruction Set 32-bit Unified Processor Core*. User Manual Volume 2. 81726 Munich, Germany: Infineon Technologies AG, 2011

[74] Infineon Technologies AG (Hrsg.): *TriCore V1.6 Core Architecture*. V1.0. 81726 Munich, Germany: Infineon Technologies AG, 2012

[75] Infineon Technologies AG (Hrsg.): *AURIX™ TC23x User's Manual*. V1.1. 81726 Munich, Germany: Infineon Technologies AG, 2014

[76] Infineon Technologies AG (Hrsg.): *AURIX™ TC26x-B User's Manual*. V1.3. 81726 Munich, Germany: Infineon Technologies AG, 2014

[77] Infineon Technologies AG (Hrsg.): *AURIX™ TC27x-D User's Manual*. V2.2. 81726 Munich, Germany: Infineon Technologies AG, 2014

[78] Infineon Technologies AG (Hrsg.): *AURIX™ TC27x-D User's Manual - Capture Compare Unit 6 (Kapitel 26)*. V2.2. 81726 Munich, Germany: Infineon Technologies AG, 2014

[79] Infineon Technologies AG (Hrsg.): *AURIX^{TM} TC29x-B User's Manual.* V1.3. 81726 Munich, Germany: Infineon Technologies AG, 2014

[80] Infineon Technologies AG (Hrsg.): *Application Kit TC2X7 - User's Manual.* V1.0. 81726 Munich, Germany: Infineon Technologies AG, 2015

[81] Infineon Technologies AG (Hrsg.): *TC237 Data Sheet.* V1.1. 81726 Munich, Germany: Infineon Technologies AG, 2015

[82] Infineon Technologies AG (Hrsg.): *TC267 Data Sheet.* V1.0. 81726 Munich, Germany: Infineon Technologies AG, 2017

[83] Infineon Technologies AG (Hrsg.): *TC277 Data Sheet.* V1.0. 81726 Munich, Germany: Infineon Technologies AG, 2017

[84] Intel Corporation: Achieving Real-Time Performance on a Virtualized Industrial Control Platform. (2014)

[85] Jansen, Patrick ; Lorenz, Robert: Transducerless Position and Velocity Estimation in Induction and Salient AC Machines. In: *IEEE Transactions on Industry Applications* 31 (1995), Nr. 2, S. 240–247

[86] Jia, Qingzhong ; Wang, Xingdou: Research on Communication Technology of Complex System Based on FPGA Dual-Port RAMs. In: *22nd International Conference on Automation and Computing*, 2016, S. 32–36

[87] Jia, Qingzhong ; Wang, Xingdou: Research on High-Speed Communication Technology Between DSP and FPGA. In: *7th IEEE Control and System Graduate Research Colloquium*, 2016, S. 62–66

[88] Jones, M. T. ; 2005: Optimization in GCC. In: *Linux Journal* (2005), Nr. 131

[89] Kaiping Yu ; Hong Guo ; Dayu Wang ; Lanfeng Li: Design of Multi-Redundancy Electro-Mechanical Actuator Controller with DSP and FPGA. In: *International Conference on Electrical Machines and Systems*, 2007, S. 584–587

[90] Karam, Lina ; Alkamal, Ismail ; Gatherer, Alan ; Frantz, Gene ; Anderson, David ; Evans, Brian: Trends in multicore DSP platforms. In: *IEEE Signal Processing Magazine* 26 (2009), Nr. 6, S. 38–49

[91] KATO, Shinpei ; LAKSHMANAN, Karthik ; RAJKUMAR, Ragunathan ; ISHIKAWA, Yutaka: TimeGraph: GPU Scheduling for Real-Time Multi-tasking Environments. In: *Proceedings of the 2011 USENIX Conference*. Berkeley, CA, USA : USENIX Association, 2011, S. 17–30

[92] KIM, Hyoseung ; NIZ, Dionisio de ; ANDERSSON, Björn ; KLEIN, Mark ; MUTLU, Onur ; RAJKUMAR, Ragunathan: Bounding Memory Interference Delay in COTS-based Multi-Core Systems. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014, S. 145–154

[93] KLARENBACH, Hans-Christoph: *Hochdynamische Servoantriebe mit paralleler Algorithmenverarbeitung*, Technische Universität München, Dissertation, 2013

[94] KOLLIG, Peter ; OSBORNE, Coline ; HENRIKSSON, Thomas: Heterogeneous Multi-Core Platform for Consumer Multimedia Applications. In: *2009 Design, Automation & Test in Europe Conference & Exhibition*, S. 1254–1259

[95] KOLPE, Tejaswini ; ZHAI, Antonia ; SAPATNEKAR, Sachin: Enabling Improved Power Management in Multicore Processors Through Clustered DVFS. In: *Design, Automation Test in Europe*, 2011, S. 1–6

[96] KUMAR, Rakesh ; ZYUBAN, Victor ; TULLSEN, Dean M.: Interconnections in Multi-Core Architectures - Understanding Mechanisms, Overheads and Scaling. In: *32nd International Symposium on Computer Architecture* (2005), S. 408–419

[97] KUNG, Ying-Shieh ; TSAI, Ming-Hung: FPGA-Based Speed Control IC for PMSM Drive With Adaptive Fuzzy Control. In: *IEEE Transactions on Power Electronics* 22 (2007), Nr. 6, S. 2476–2486

[98] LE-HUY, Hoang: Microprocessors and digital ICs for motion control. In: *Proceedings of the IEEE* 82 (1994), Nr. 8, S. 1140–1163

[99] LEONHARD, Werner: Microcomputer Control of High Dynamic Performance AC-Drives - A Survey. In: *Automatica* 22 (1986), Nr. 1, S. 1–19

[100] LIANG, Wenyi ; WANG, Jianfeng ; LUK, Patrick ; FANG, Weizhong ; FEI, Weizhong: Analytical Modeling of Current Harmonic Components in PMSM Drive With Voltage-Source Inverter by SVPWM Technique. In: *IEEE Transactions on Energy Conversion* 29 (2014), Nr. 3, S. 673–680

[101] Lindholm, E ; Nickolls, J ; Oberman, S ; Montrym, J: NVIDIA Tesla: A Unified Graphics and Computing Architecture. In: *IEEE Micro* 28 (2008), März, Nr. 2, S. 39–55

[102] Linke, Marco ; Kennel, Ralph ; Holtz, Joachim: Sensorless Position Control of Permanent Magnet Synchronous Machines Without Limitation at Zero Speed. In: *IEEE 2002 28th Annual Conference of the Industrial Electronics Society*, 2002, S. 674–679

[103] Littlefield-Lawwill, Justin ; Kinnan, Larry: System considerations for robust time and space partitioning in Integrated Modular Avionics. In: *27th Digital Avionics Systems Conference*, 2008, S. 1.B.1–1–1.B.1–11

[104] Lu, Qiu-xia ; Dong, Xue-ren: Application of DSP in Sensorless PMSM Control System. In: *International Conference on Measuring Technology and Mechatronics Automation*, IEEE, 2009, S. 357–360

[105] Luenberger, David: An Introduction to Observers. In: *IEEE Transactions on Automatic Control* 16 (1971), Nr. 6, S. 596–602

[106] Lysecky, Roman ; Vahid, Frank: A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores Using Dynamic Hardware/Software Partitioning. In: *Design, Automation & Test in Europe Conference & Exhibition*, 2005, S. 18–23

[107] Ma, Zhixun ; Gao, Jianbo ; Kennel, Ralph: FPGA Implementation of a Hybrid Sensorless Control of SMPMSM in the Whole Speed Range. In: *IEEE Transactions on Industrial Informatics* 9 (2013), August, Nr. 3, S. 1253–1261

[108] Macher, Georg ; Höller, Andrea ; Armengaud, Eric ; Kreiner, Christian: Automotive embedded software: Migration challenges to multi-core computing platforms. In: *13th International Conference on Industrial Informatics*, 2015, S. 1386–1393

[109] Mars, Jason ; Tang, Lingjia ; Soffa, Mary L.: Directly Characterizing Cross Core Interference Through Contention Synthesis. In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. New York, NY, USA : ACM, 2011, S. 167–176

[110] Matzke, Doug: Will Physical Scalability Sabotage Performance Gains? In: *Computer* 30 (1997), Nr. 9, S. 37–39

[111] Miñambres-Marcos, V ; Guerrero-Martínez, M A. ; Romero-Cadaval, E ; Gutiérrez, J: Issues and Improvements of Hardware/software Co-Design Sensorless Implementation in a Permanent Magnet Synchronous Motor Using Veristand. In: *3rd IEEE International Symposium on Sensorless Control for Electrical Drives*, 2013, S. 1–7

[112] Misra, Sanjay ; Alfa, Abraham A. ; Olaniyi, Mikail O. ; Adewale, Sunday O.: Exploratory Study of Techniques for Exploiting Instruction-Level Parallelism. In: *2014 Global Summit on Computer & Information Technology*, IEEE, 2014, S. 1–6

[113] Monmasson, Eric ; Cirstea, Marcian: FPGA Design Methodology for Industrial Control Systems - A Review. In: *IEEE Transactions on Industrial Electronics* 54 (2007), Nr. 4, S. 1824–1842

[114] Monmasson, Eric ; Idkhajine, Lahoucine ; IEEE, I Bahri 2010 5. ; 2010: Design methodology and FPGA-based controllers for power electronics and drive applications. In: *5th IEEE Conference on Industrial Electronics and Applications*, S. 2328–2338

[115] Monmasson, Eric ; Idkhajine, Lahoucine ; Naouar, Mohamed: FPGA-Based Controllers. In: *IEEE Industrial Electronics Magazine* 5 (2011), Nr. 1, S. 14–26

[116] Munshi, Aaftab: The OpenCL Specification. In: *2009 IEEE Hot Chips 21 Symposium*, 2009, S. 1–314

[117] Mutlu, Onur ; Moscibroda, Thomas: Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, S. 146–160

[118] Mutlu, Onur ; Stark, Jared ; Wilkerson, Chris ; Patt, Yale N.: Runahead Execution - An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In: *The Ninth International Symposium on High-Performance Computer Architecture*, 2003, S. 129–140

[119] Nesbit, Kyle ; Laudon, James ; Smith, James: Virtual Private Caches. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2007, S. 57–68

[120] Nickolls, John ; Buck, Ian ; Garland, Michael ; Skadron, Kevin: Scalable Parallel Programming with CUDA. In: *Queue - GPU Computing* 6 (2008), Nr. 2, S. 40–53

[121] Nickolls, John ; Dally, William: The GPU Computing Era. In: *IEEE Micro* 30 (2010), Nr. 2, S. 56–69

[122] Nowotsch, Jan ; Paulitsch, Michael: Leveraging Multi-core Computing Architectures in Avionics. In: *Ninth European Dependable Computing Conference*, 2012, S. 132–143

[123] Ogasawara, Satoshi ; Akagi, hirofumi: An approach to position sensorless drive for brushless DC motors. In: *IEEE Transactions on Industry Applications* 27 (1991), Nr. 5, S. 928–933

[124] Owens, John ; Houston, Mike ; Luebke, David ; Green, Simon ; Stone, John ; Phillips, James: GPU Computing. In: *Proceedings of the IEEE* 96 (2008), Nr. 5, S. 879–899

[125] Paolieri, Marco ; Quiñones, Eduardo ; Cazorla, Francisco ; Valero, Mateo: An Analyzable Memory Controller for Hard Real-Time CMPs. In: *IEEE Embedded Systems Letters* 1 (2009), Nr. 4, S. 86–90

[126] Pellizzoni, Rodolfo ; Caccamo, Marco: Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. In: *IEEE Transactions on Computers* 59 (2010), Nr. 3, S. 400–415

[127] Pellizzoni, Rodolfo ; Schranzhofer, Andreas ; Jian-Jia Chen ; Caccamo, marco ; Thiele, Lothar: Worst case delay analysis for memory interference in multicore systems. In: *Design, Automation Test in Europe*, 2010, S. 741–746

[128] Qureshi, Moinuddin ; Patt, Yale: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, S. 423–432

[129] Radojkovi c, Petar ; Girbal, Sylvain ; Grasset, Arnaud ; ones, Eduardo Qui n. ; Yehia, Sami ; Cazorla, Francisco J.: On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical

Environments. In: *ACM Transactions on Architecture and Code Optimization* 8 (2012), Nr. 4, S. 34:1–34:25

[130] Reichenbach, Frank ; Wold, Alexander: Multi-core Technology - Next Evolution Step in Safety Critical Systems for Industrial Applications? In: *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, S. 339–346

[131] Reid, Alastair ; Lin, Yuan: SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Dystems on Chip. In: *Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems*, ACM, 2008, S. 95–104

[132] Reineke, Jan ; Liu, Isaac ; Patel, Hiren ; Kim, Sungjun ; Lee, Edward A.: PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In: *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, ACM, 2011, S. 99–108

[133] Robert Bosch GmbH (Hrsg.): *GTM-IP Specification.* Revision: 3.1.5.1. 70839 Gerlingen-Schillerhöhe, Germany: Robert Bosch GmbH, 2016

[134] Rodríguez-Andina, Juan ; Moure, María ; Valdés, María: Features, Design Tools, and Application Domains of FPGAs. In: *IEEE Trans. Industrial Electronics* 54 (2007), Nr. 4, S. 1810–1823

[135] Rosen, Jakob ; Andrei, Alexandru ; Eles, Petru ; Peng, Zebo: Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In: *30th IEEE Real-Time Systems Symposium*, 2007, S. 49–60

[136] Ross, Philip: Why CPU Frequency Stalled. In: *IEEE Spectrum* 45 (2008), Nr. 4, S. 72–72

[137] Rupprecht, Gabriel ; Leonhard, Werner ; Nordby, Craig: Field-Oriented Control of a Standard AC Motor Using Microprocessors. In: *IEEE Transactions on Industry Applications* IA-16 (1980), Nr. 2, S. 186–192

[138] Saidi, Selma ; Ernst, Rolf ; Uhrig, Sascha ; Theiling, Henrik ; Dinechin, Benoît D.: The Shift to Multicores in Real-Time and Safety-Critical Systems. In: *International Conference on Hardware/Software Codesign and System Synthesis* (2015), S. 220–229

[139] Samar, Asri ; Saedin, Pais ; Tajudin, A ; Adni, Nor: The Implementation of Field Oriented Control for PMSM Drive Based on TMS320F2808 DSP Bontroller. In: *IEEE International Conference on Control Systems, Computing and Engineering*, 2012, S. 612–616

[140] Schauder, Colin: Adaptive Speed Identification for Vector Control of Induction Motors Without Rotational Transducers. In: *IEEE Transactions on Industry Applications* 28 (1992), Nr. 5, S. 1054–1061

[141] Schranzhofer, Andreaas ; Chen, Jian-Jia ; Thiele, Lothar: Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, S. 215–224

[142] Schranzhofer, Andreaas ; Pellizzoni, Rodolfo ; Chen, Jian-Jia ; Thiele, Lothar ; Caccamo, marco: Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, S. 213–222

[143] Schrödl, Manfred: Detection of the Rotor Position of a Permanent Magnet Synchronous Machine at Standstill. In: *International Conference on Electrical Machines*, 1988, S. 195–197

[144] Schrodl, Manfred: Sensorless Control of AC Machines. In: *VDI Fortschrittsberichte* 117 (1992)

[145] Schuster, Tobias: *AURIX 32-bit Microcontroller Family - Performance Meets Safety*. 1. 81726 Munich, Germany, 2018

[146] Shah, Hardik ; Coombes, Andrew ; Raabe, Andreas ; Huang, Kai ; Knoll, Alois: Measurement Based WCET Analysis for Multi-core Architectures. In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. New York, NY, USA : ACM, 2014, S. 257:257–257:266

[147] Shirabe, Kohei ; Swamy, Mahesh ; Kang, Jun-Koo ; Hisatsune, Masaki ; Wu, Yifeng ; Kebort, Don ; Honea, Jim: Advantages of High Frequency PWM in AC Motor Drive Applications. In: *2012 IEEE Energy Conversion Congress and Exposition*, 2012, S. 2977–2984

[148] Šmídl, Václav ; Nedvěd, Robert ; Košan, Tomáš ; Peroutka, Zdeneˇk: FPGA Implementation of Marginalized Particle Filter for Sensorless Control of

PMSM Drives. In: *39th Annual Conference of the IEEE Industrial Electronics Society*, 2013, S. 8227–8232

[149] SMITH, James ; SOHI, Gurindar: The Microarchitecture of Superscalar Processors. In: *Proceedings of the IEEE* 83 (1995), Nr. 12, S. 1609–1624

[150] SORENSEN, Tyler ; EVRARD, Hugues ; DONALDSON, Alastair: Cooperative Kernels: GPU Multitasking for Blocking Algorithms (Extended Version). In: *11th Joint Meeting of the European Software Engineering Conference*, 2017, S. 431–441

[151] STMICROELECTRONICS NV (Hrsg.): *32-bit Power Architecture Microcontroller for Automotive SIL3/ASILD Chassis and Safety Applications* . DocID023953 Rev 5. Genf, Schweiz: STMicroelectronics NV, 2015

[152] STROTHMANN, Rolf: *Separately Excited Electric Machine.* EP Patent 1005716B1, 2001

[153] SUBHLOK, Jaspal ; STICHNOTH, James M. ; O'HALLARON, David R. ; GROSS, Thomas: Exploiting Task and Data Parallelism on a Multicomputer. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* New York, NY, USA : ACM, 1993, S. 13–22

[154] SUHENDRA, Vivy ; MITRA, Tulika: Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In: *Proceedings of the 45th Annual Design Automation Conference.* New York, NY, USA : ACM, 2008, S. 300–303

[155] SULEMAN, M A. ; MUTLU, Onur ; QURESHI, Moinuddin K. ; PATT, Yale N.: Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In: *Proceedings of the 4th international conference on Architectural support for programming languages and operating systems* (2009), S. 253

[156] TAM, David ; AZIMI, Reza ; SOARES, Livio ; STUMM, Michael: Managing Shared L2 Caches on Multicore Systems in Software. In: *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007, S. 1–8

[157] TEXAS INSTRUMENTS INCORPORATED (Hrsg.): *TMS570LS Series 16/32-BIT RISC Flash Microcontroller.* SPNS141G. Dallas Texas 75265, USA: Texas Instruments Incorporated, 2018

[158] Thiemann, P ; Mantala, C ; Mueller, T ; Strothmann, R ; Zhou, E: Direct Flux Control (DFC): A New Sensorless Control Method for PMSM. In: *46th International Universities' Power Engineering Conference*, 2011, S. 1–6

[159] Thiemann, P ; Mantala, C ; Mueller, T ; Strothmann, R ; Zhou, E: PMSM Sensorless Control with Direct Flux Control for all Speeds. In: *3rd IEEE International Symposium on Sensorless Control for Electrical Drives*, 2012, S. 1–6

[160] Tiwari, Vivek ; Singh, Deo ; Rajgopal, Suresh ; Mehta, Gaurav ; Patel, Rakesh ; Baez, Franklin: Reducing Power in High-Performance Microprocessors. In: *35th Annual Design Automation Conference* (1998), S. 732–737

[161] van der Broeck, Heinz ; Skudelny, Hans-Christoph ; Stanke, Georg: Analysis and Realization of a Pulsewidth Modulator Based on Voltage Space Vectors. In: *IEEE Transactions on Industry Applications* 24 (1988), Nr. 1, S. 142–150

[162] Wagner, Eric ; Karls, Christoph ; Lehser, Martina: Quantification and Localization of Cross-Core Interference for Embedded Multi-Core Control Applications. In: *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, Association for Computing Machinery, 2018, S. 189–193

[163] Wagner, Eric ; Lehser, Martina ; Baeuml, Maximilian: Manipulating multi-core interferences for sensorless electrical drive applications on implementation level. In: *42th IEEE Conference on Industrial Electronics and Applications*, IEEE, 2016, S. 6681–6686

[164] Wagner, Eric ; Lehser, Martina ; Nienhaus, Matthias: Embedded Multi-Core Systems for the Integration of Multi-Axis Motor-Controllers for Sensorless Electrical Drives. In: *10. ETG/GMM-Symposium on Innovative small Drives and Micro-Motor Systems*, VDE, 2015, S. 1–6

[165] Wagner, Eric ; Lehser, Martina ; Nienhaus, Matthias: Evaluation of a Safety Multi-Core Platform for the Sensorless Drive of a PMSM. In: *13th IFAC and IEEE Conference on Programmable Devices and Embedded Systems* Bd. 48, IFAC/IEEE, 2015, S. 284–289

[166] Wagner, Eric ; Lehser, Martina ; Nienhaus, Matthias: Power Analysis of Embedded Multi-Core Systems for the Sensorless Operation of Multiple

Electrical Drives. In: *11. ETG/GMM-Symposium on Innovative small Drives and Micro-Motor Systems*, VDE, 2017, S. 1–6

[167] WANG, Guohui ; WENMING LI ; XIALI HEI: Energy-Aware Real-time Scheduling on Heterogeneous Multi-Processor. In: *49th Annual Conference on Information Sciences and Systems*, 2015, S. 1–7

[168] WANG, Guohui ; XIONG, Yingen ; YUN, Jay ; CAVALLARO, Joseph: Accelerating computer Vision Algorithms Using OpenCL Framework on the Mobile GPU - A case study. In: *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2013, S. 2629–2633

[169] WANG, Zhenning ; YANG, Jun ; MELHEM, Rami ; CHILDERS, Bruce ; ZHANG, Youtao ; GUO, Minyi: Simultaneous Multikernel GPU: Multi-Tasking Throughput Processors via Fine-Grained Sharing. In: *2016 IEEE International Symposium on High Performance Computer Architecture*, 2016, S. 358–369

[170] WEIYI, Shuai ; DI, Yan ; XIAOYU, Li ; KE, Sun: Parallelization Method of Digital Signal Processing Based on Multi-Core Pipeline. In: *IEEE 17th International Conference on Communication Technology (ICCT)*, 2017, S. 350–354

[171] WILHELM, Reinhard ; ABEL, Andreas ; BENZ, Florian: Impact of Resource Sharing on Performance and Performance Prediction: A Survey. In: D'ARGENIO, Pedro R. (Hrsg.) ; MELGRATTI, Hernán (Hrsg.): *Concurrency Theory*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, S. 25–43

[172] WILHELM, Reinhard ; GRUND, Daniel ; REINEKE, Jan ; SCHLICKLING, Marc ; PISTER, Markus ; FERDINAND, Christian: Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (2009), Nr. 7, S. 966–978

[173] WOLF, Wayne: Multiprocessor System-on-Chip Technology. In: *IEEE Signal Processing Magazine* 26 (2009), Nr. 6, S. 50–54

[174] YAN, Zhang ; UTKIN, Vadim: Sliding Mode Observers for Electric Machines - An Overview. In: *28th Annual Conference of the Industrial Electronics Society* Bd. 3, 2002, S. 1842–1847

[175] YIM, Jung-Sik ; SUL, Seung-Ki ; BAE, Bon-Ho ; PATEL, Nitin ; HITI, Silvia: Modified Current Control Schemes for High-Performance Permanent-Magnet

AC Drives With Low Sampling to Operating Frequency Ratio. In: *IEEE Transactions on Industry Applications* 45 (2009), Nr. 2, S. 763–771

[176] Yongdong, Li ; Hao, Zhu: Sensorless Control of Permanent Magnet Synchronous Motor – A Survey. In: *IEEE Vehicle Power and Propulsion Conference*, 2008, S. 1–8

[177] Youness, Hassan ; Moness, Mohamed ; Khaled, Mahmoud: MPSoCs and Multicore Microcontrollers for Embedded PID Control: A Detailed Study. In: *IEEE Transactions on Industrial Informatics* 10 (2014), Nr. 4, S. 2122–2134

[178] Zhang, Xiao ; Dwarkadas, Sandhya ; Shen, Kai: Towards Practical Page Coloring-based Multicore Cache Management. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. New York, NY, USA : ACM, 2009, S. 89–102

[179] Zhao, L ; Iyer, R ; Illikkal, R ; Moses, J ; Makineni, S ; Newell, D: CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, 2007, S. 339–352

[180] Zhu, Dakai ; Aydin, Hakan: Energy Management for Real-time Embedded Systems with Reliability Requirements. In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*. New York, NY, USA : ACM, 2006, S. 528–534

[181] Zhuravlev, Sergey ; Saez, Juan ; Blagodurov, Sergey ; Fedorova, Alexandra ; Prieto, Manuel: Survey of Energy-Cognizant Scheduling Techniques. In: *IEEE Transactions on Parallel and Distributed Systems* 24 (2013), Nr. 7, S. 1447–1464

[182] Ziakas, Dimitrios ; Baum, Allen ; Maddox, Robert ; Safranek, Robert: Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In: *18th IEEE Symposium on High Performance Interconnects*, 2010, S. 1–6

Abbildungsverzeichnis

2.1 Abgrenzung von hard- und softwarebasierten Integrationsmethoden . 6
2.2 Aufbau der feldorientierten Stromregelung 7
2.3 Abläufe der Regelung auf einem Mikroprozessor 8
2.4 PWM-Muster zur Messung der Spannungspotenziale zwischen realem und künstlichem Sternpunkt 11

4.1 Erweiterung der hard- und softwarebasierten Integrationsmethoden durch Multi-Core-Systeme 33
4.2 Allgemeiner Aufbau der betrachteten Multi-Core-Architekturen aus dem Automotive-Bereich 35
4.3 Parallelisierung durch Tasks 38
4.4 Parallelisierung durch eine zweistufige Pipeline 39
4.5 Softwarestruktur zur Konsolidierung 41

5.1 Systemmodell zur Betrachtung der Leistungsaufnahme 47
5.2 Gegenüberstellung der Leistungsaufnahme von Single- und Multi-Core Motor-Controllern 52
5.3 Leistungsaufnahme eines mit halbem Basistakt betriebenen Multi-Core-Systems 54

6.1 Zeitdiskrete Ausführung der GRB auf einem Single-Core-System . . . 58
6.2 Nebenläufige Ausführung einer zeitkontinuierlichen GRB auf einem Single-Core-System 59
6.3 Datenflussgraphen der geberlosen feldorientierten Stromregelung . . . 60
6.4 Datenflussgraphen mit zeitkontinuierlichen Rotorlageberechnung . . . 61
6.5 Parallele Ausführung mit zeitkontinuierlicher GRB 63
6.6 Möglichkeiten der parallelen Ausführung einer zeitdiskreten GRB . . 64
6.7 Datenflussgraphen parallelisierter Verarbeitungsketten am Beispiel eines Sliding-Mode Observer 66
6.8 Parallelisierung von Multiplikationen 67

6.9 Parallelisierung von FOS-Tasks nach Datenströmen 73
6.10 Parallelisierung der GFOS mit SMO 74
6.11 Ausführung des SMO innerhalb einer zweistufigen Pipeline 76

7.1 Konkurrierende Zugriffe auf ein gemeinsames Hardwaremodul 83
7.2 Synchrone Ausführung gleicher Anwendungen/Implementierungen . . 86
7.3 Zeitlich verschobene Ausführung gleicher Anwendungen 87
7.4 Modulzugriffe innerhalb einer Task bei zeitlich versetzter Ausführung 88
7.5 Synchrone Ausführung unterschiedlicher Anwendungen 89
7.6 Paralleler kernübergreifender Nachrichtenversand 91
7.7 Einfluss von ΔStart auf die Laufzeiterhöhung 102

8.1 Ablauf der analytischen Interferenzbestimmung 106
8.2 Verwendung von nop-Befehlen zur Bestimmung von t_{init} 112
8.3 Abbildung zweier Multi-Core Teilsysteme auf eine gemeinsame Zeitbasis 120
8.4 Einfluss der Wartezeit auf die Timings eines Modulzugriffs 121
8.5 Beispielhafte Visualisierung von Analysedaten 128

Tabellenverzeichnis

4.1 Gemessene Laufzeiten einer kernübergreifenden Synchronisation und Datenübertragung . . . 43

5.1 Leistungsaufnahme der Motor-Controller während der Regelung eines Antriebs . . . 50
5.2 Leistungsaufnahme der MCMC bei zusätzlichen Antrieben . . . 52

6.1 Laufzeiten der GFOS-Tasks mit SMO . . . 70
6.2 Laufzeiten der GFOS-Tasks mit DFC . . . 71
6.3 Laufzeiten der GFOS-Tasks mit HFCI . . . 71
6.4 Parallelisieren der GFOS-Tasks nach Datenströmen auf zwei Kerne . 72
6.5 Parallele Ausführung von Rechenoperationen . . . 76

7.1 Maximale Laufzeiten konkurrierender Load- und Store-Instruktionen . 94
7.2 Maximale Laufzeiterhöhung durch konkurrierende SE- und PWM-Tasks 95
7.3 Laufzeiten aufeinanderfolgender Zugriffe auf I/O-Module . . . 96
7.4 Laufzeiten aufeinanderfolgender Zugriffe auf globalen Speicher . . . 97
7.5 Effekt einer Compiler-Optimierung auf konkurrierende Zugriffe . . . 100

8.1 Berechnete und gemessenen Laufzeiterhöhungen . . . 127
8.2 Berechnete und gemessenen Multi-Core Laufzeiten . . . 127

Algorithmenverzeichnis

1 Vorgehen zur Erzeugung des Modulzugriffstrace 114
2 Verarbeitung globaler Lesezugriffe 116
3 Verarbeitung von Schreibzugriffen 117
4 Verarbeitung des Store-Buffer 119
5 Berechnung von Wartezeiten 122
6 Berechnung und Anpassung der Timings von Instruktionen 123

Abkürzungen und Symbole

Abkürzungen

ADC Analog-Digital-Converter

BLDC Brushless DC

CCU Capture-Compare Unit

CPU Central Processing Unit

CT Clark-Transformation

DFC Direct Flux Control

DSP Digitaler Signalprozessor

EMK Elektromotorische Kraft

FIFO First In - First Out

FOS Feldorientierte Stromregelung

FPGA Field Programmable Gate Array

GCC GNU Compiler Collection

GFOS Geberlose feldorientierte Stromregelung

GP General-Purpose

GPU Graphics Processing Unit

GRB Geberlose Rotorlageberechnung

GTM Generic Timer Module

HFCI High Frequency Current Injection

INFORM Indirect Flux-detection by Online Reactance Measurement

IPT Inverse Park-Transformation

ISR Interrupt-Service-Routine

MCMC Multi-Core Motor-Controller

MPU Memory Protection Units

PMSM Permanenterregter Synchronmotor

PT Park-Transformation

PWM Pulsweitenmodulation

RK Regelkreis

SE Signalerfassung

SMO Sliding-Mode Observer

SRAM Static random-access Memory

VADC Versatile AD-Converter

WCET Worst Case Execution Time

Symbole

$\mathbf{Start_k}$ Startzeitpunkt der k-ten Iteration einer Task oder Regelung

$\mathbf{Ende_k}$ Ende der k-ten Iteration einer Task oder Regelung

$\mathbf{TX_a}$ Task X Abschnitt a

$\mathbf{\Delta Start}$ Zeitlicher Versatz zwischen Tasks oder Multi-Core-Teilsystemen

$\mathbf{T}$ Periodendauer

$\mathbf{t_x}$ Laufzeit einer oder mehrerer Tasks oder eines Zugriffs auf ein Hardwaremodul

$\mathbf{P_x}$ Leistungsaufnahme der Komponente x

$\mathbf{f_x}$ Frequenz der Regelung, Task oder Taskmenge x

$\mathbf{i}$	Gemessener Strom auf einem Motorstrang
$\mathbf{i}$	Gemessene Spannung an einem Motorstrang
$\mathbf{\Theta_k}$	Rotorposition in der k-ten Iteration
$\mathbf{S_x}$	Synchronisationspunkt x von zwei oder mehreren Tasks
j, i	Zwei voneinander abhängige Instruktionen: j ist abhängig von i
D	Zeitliche Distanz zwischen zwei voneinander abhängigen Instruktionen
b	Modulzugriffsversuch