UNIVERSITÄT ZU LÜBECK

Aus dem Institut für Technische Informatik
der Universität zu Lübeck
Direktor: Prof. Dr.-Ing. Mladen Berekovic

Beitrag zur Optimierung der statischen Speicherverwaltung in eingebetteten Mehrkernsystemen mit einer harten Echtzeitanforderung

Contribution to the optimization of the static memory management in embedded multicore systems with hard real-time requirements

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck

Aus der Sektion Informatik / Technik

vorgelegt von
Philipp Jungklaß
aus Schwerin

Lübeck, 2025

1. Berichterstatter:
Prof. Dr.-Ing. Mladen Berekovic

2. Berichterstatter:
Prof. Dr.-Ing. Guillermo Payá Vayá

Tag der mündlichen Prüfung:
10.07.2025

Zum Druck genehmigt. Lübeck, den 10.07.2025

Widmung

Ich widme diese Arbeit meinen beiden Töchtern Amell Jungklaß und Benja Jungklaß. Ich werde Euch immer lieben.

Inhaltsverzeichnis

Publikationen

Die folgenden Publikationen sind Teil dieser Arbeit und wurden bereits veröffentlicht:

1. P. Jungklass und M. Berekovic Performance-Oriented Memory Management for Embedded Multicore Microcontrollers. In: *26th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*. 2018. ISBN: 978-3-902457-49-3
2. P. Jungklass und M. Berekovic Effects of concurrent access to embedded multicore microcontrollers with hard real-time demands. In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, S. 1–9. ISBN: 978-1-5386-4155-2/18
3. P. Jungklass und M. Berekovic Intercore-Kommunikation für Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2018*. 2018. ISBN: 978-3-8343-3447-3
4. P. Jungklass und M. Berekovic Performance-orientiertes Speichermanagement bei embedded Multicore-Mikrocontrollern. In: *Tagungsband Embedded Software Engineering Kongress 2018*. 2018. ISBN: 978-3-8343-3447-3
5. P. Jungklass und M. Berekovic Cache-Kohärenz für embedded Multicore-Mikrocontroller mit harter Echtzeitanforderung. In: *Echtzeit 2019*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 129–138. ISBN: 978-3-658-27808-3
6. P. Jungklass und M. Berekovic MemOpt: Automated Memory Distribution for Multicore Microcontrollers with Hard Real-Time Requirements. In: *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. Okt. 2019, S. 1–7. ISBN: 978-1-7281-2769-9/19. DOI: 10.1109/NORCHIP.2019.8906914
7. P. Jungklass, C. Schmidt, T. Jungklass und M. Berekovic Scheduling-Konzepte für echtzeitfähige embedded Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2019*. 2019. ISBN: 978-3-8343-3463-3
8. P. Jungklass, C. Schmidt und M. Berekovic Redundanzkonzepte für embedded Multicore Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2019*. 2019. ISBN: 978-3-8343-3463-3
9. P. Jungklass und M. Berekovic Speicherkonzepte für echtzeitfähige Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2020*. 2020. ISBN: 978-3-8343-2415-3
10. P. Jungklass und M. Berekovic Static Allocation of Basic Blocks Based on Runtime and Memory Requirements in Embedded Real-Time Systems with Hierarchical Memory Layout. In: *Second Workshop on Next Generation Real-Time*

Embedded Systems (NG-RES 2021). Hrsg. von M. Bertogna und F. Terraneo. Bd. 87. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, 3:1–3:14. ISBN: 978-3-95977-178-8. DOI: 10.4230/OASIcs.NG-RES.2021.3. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13479

11. S. Körür, P. Jungklass und M. Berekovic Echtzeitfähige Ethernet-Kommunikation in automobilen Multicore-Systemen mit hierarchischem Speicherlayout. In: *Echtzeit 2021*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 83–92. ISBN: 978-3-658-37751-9. DOI: https://doi.org/10.1007/978-3-658-37751-9
12. C. Böttcher, P. Jungklass und M. Berekovic Hardware-Beschleuniger für automobile Multicore-Mikrocontroller mit einer harten Echtzeitanforderung. In: *Echtzeit 2021*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 63–72. ISBN: 978-3-658-37751-9. DOI: https://doi.org/10.1007/978-3-658-37751-9
13. P. Jungklass, F. Grieger, C. Elvers und M. Berekovic Cache-Konzepte für echtzeitfähige Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2021*. 2021. ISBN: 978-3-8343-6291-9
14. P. Jungklass, F. Grieger und M. Berekovic Predictive Preload at Fixed Preemption Points for Microcontrollers with Hard Real-Time Requirements. In: *Real-time and Autonomous Systems 2022*. Hrsg. von H. Unger und M. Schaible. Cham: Springer Nature Switzerland, 2023, S. 43–51. ISBN: 978-3-031-32700-1

1

Zusammenfassung

Für die technischen Herausforderungen der nächsten Jahre stellen eingebettete Systeme eine wichtige Schlüsseltechnologie zur Ermittlung von Messdaten, zur Steuerung und Regelung von komplexen Systemen, aber auch zur Absicherung und Überwachung von sicherheitskritischen Anwendungen dar. Um dem steigenden Bedarf an Rechenleistung aufgrund von immer umfassenderen Gesamtsystemen und dem gleichzeitigen Ziel der fortschreitenden Integration von verschiedensten Funktionalitäten auf eingebetteten Systemen gerecht zu werden, erfolgt zunehmend der Einsatz von Mikrocontrollern mit mehreren Prozessorkernen. Jedoch stellen Mehrkernsysteme eine Herausforderung in Umgebungen dar, welche zwingend eine harte Echtzeitfähigkeit voraussetzen. Dies ist darauf zurückzuführen, dass mit der Einführung von Mikrocontrollern mit mehreren Prozessorkernen erstmalig eine gleichzeitige Bearbeitung von verschiedenen Aufgaben möglich ist. Diese Gleichzeitigkeit führt jedoch dazu, dass sich die Prozessorkerne beim Zugriff auf geteilte Ressourcen gegenseitig beeinflussen können, was einen direkten Einfluss auf die Ausführungsgeschwindigkeit hat. Die Folge sind Laufzeitanomalien, welche die harte Echtzeitfähigkeit gefährden können. Ein Nichteinhalten der Anforderungen, welche aus einer harten Echtzeitfähigkeit resultieren, wird als Versagen des Systems gewertet.

Eine besonders kritische, geteilte Ressource in eingebetteten Systemen sind die Speicher, welche einen signifikanten Einfluss auf eine konstante Laufzeit und die Einhaltung von sicherheitsrelevanten Rahmenbedingungen haben. Neben diesen Faktoren haben die Speicher zusätzlich eine direkte Auswirkung auf die Ausführungsgeschwindigkeit der implementierten Software. Dieser Umstand ist darin begründet, dass die Entwicklung der Speichergeschwindigkeit nicht mit der Leistungsentwicklung der Prozessorkerne in den letzten Jahrzehnten mithalten konnte. Zur Kompensation der genannten Effekte integrieren die Hersteller entsprechender Mikrocontroller für sicherheitskritische Anwendungen eine komplexe Speicherhierarchie mit einer Vielzahl von Optimierungen. Die Kombination dieser umfassenden Hardware-Möglichkeiten mit der ebenfalls komplexer werdenden Software stellt eine Herausforderung dar, welche im Rahmen dieser Arbeit untersucht wird. Das vorrangige Ziel ist dabei, einen Ansatz zu entwickeln, welcher das Speichermanagement in seiner Gesamtheit betrachtet und nicht nur ausgewählte Speichertypen optimiert. Zu diesem Zweck erfolgt die Entwicklung eines Ansatzes, welcher die temporale und die

räumliche Isolierung verbessert, die Ausführungsgeschwindigkeit steigert sowie die Einhaltung der Reaktionszeit berücksichtigt.

Für die optimierte Allokation werden im ersten Schritt alle Bestandteile der auszuführenden Software analysiert und priorisiert. Die Priorisierung aller Variablen, Konstanten und Funktionen erfolgt dabei auf Basis verschiedener Faktoren, wie der Kritikalität der Funktionalität, welche diese Bestandteile nutzt, die Aufrufhäufigkeit, die Ausführungszeit sowie der benötigte Speicherbedarf. Dabei ist es essentiell zu detektieren, von welchem Prozessorkern mit welcher Häufigkeit auf die Variablen, Konstanten und Funktionen zugegriffen wird, da dies für die spätere Verteilung relevant ist. Im zweiten Schritt wird die verwendete Zielhardware untersucht, wobei der Fokus auf allen speicherrelevanten Hardware-Komponenten liegt. Dazu gehören die Speicher selbst sowie die Prozessorkerne, die Hardware-Beschleuniger und die Kommunikationssysteme, welche die Komponenten des Mikrocontrollers miteinander verbinden. Im Anschluss erfolgt auf Basis der ermittelten Informationen die Berechnung einer optimierten Allokation, welche in Abhängigkeit der errechneten Priorität die Aufteilung der Software-Bestandteile auf die verschiedenen Speichertypen vornimmt. Zur Verringerung von konkurrierenden Zugriffen werden dafür auch Kopien von bestimmten Funktionen, Variablen und Konstanten erstellt und auf unterschiedliche Speicher verteilt, solange dafür der entsprechende Speicherplatz zur Verfügung steht. Neben der reinen Betrachtung der Laufzeit erfolgt zusätzlich die Berücksichtigung von Speicherschutzmechanismen, indem die entsprechende Granularität bei der Allokation berücksichtigt wird.

Für den Nachweis der korrekten Funktionsweise wird der erarbeitete Ansatz auf einem selbst entwickelten Testsystem evaluiert, welcher sich an der Software-Architektur von automobilen Steuergeräten für echtzeitfähige Anwendungen orientiert. Zu diesem Zweck erfolgt die Portierung einer Vielzahl von Benchmarks für eingebettete Systeme als repräsentative Lastszenarien. Als Hardware-Plattform kommt ein Mehrkernmikrocontroller von der Firma Infineon vom Typ AUtomotive Realtime Integrated NeXt Generation Architecture (AURIX) TC39x zum Einsatz, welcher vorwiegend in sicherheitskritischen Systemen mit einer harten Echtzeitanforderung Verwendung findet.

Wie die durchgeführten Messungen aufzeigen, kann mit Hilfe des hier vorgestellten Konzepts eine signifikante Verbesserung in allen Zielstellungen erreicht werden. Durch die konsequente Nutzung der verfügbaren Hardware-Komponenten kann sowohl die temporale als auch die räumliche Isolation verbessert werden. Zusätzlich kann durch die optimierte Nutzung der schnellen Zwischenspeicher die Ausführungsgeschwindigkeit gesteigert und die Reaktionszeit verringert werden.

Summary

For the technical challenges of the coming years, embedded systems represent an important key technology for determining measurement data, controlling and monitoring complex systems, but also for safeguarding and observing safety-critical applications. Microcontrollers with multiple processor cores are increasingly being used to meet the growing demand for computing power due to the need for more comprehensive overall systems and the goal of progressively integrating a wide range of functionalities in embedded systems. However, multi-core systems are a challenge in environments that require hard real-time capability. This is due to the fact that the introduction of microcontrollers with multiple processor cores makes it possible for the first time to process different tasks simultaneously. However, this concurrency means that the processor cores can influence each other when accessing shared resources, which has a direct impact on the execution speed. This results in runtime anomalies, which can jeopardize hard real-time capability. Fail to comply with one requirement, resulting from hard real-time capability will be considered as fail of the complete system.

A particularly critical shared resource in embedded systems is the memory, which has a significant influence on a constant runtime and compliance with safety-relevant conditions. In addition to these factors, the memory also has a direct impact on the execution speed of the implemented software. This is due to the fact that the development of memory speed has not been able to keep pace with the performance development of processor cores in recent decades. To compensate for these effects, manufacturers of microcontrollers for safety-critical applications integrate a complex memory hierarchy with a large number of optimizations. The combination of these comprehensive hardware options with the increasingly complex software represents a challenge that is being investigated as part of this work. The primary goal is to develop an approach that considers memory management in its entirety and not just optimizes selected memory types. To this end, an approach is developed that improves temporal and spatial isolation, increases execution speed and takes reaction time into account.

For optimized allocation, the first step is to analyse and prioritize all components of the software to be executed. The prioritization of all variables, constants and functions is based on various factors, such as the criticality of the functionality that uses these components, the call frequency, the execution time and the memory requirements. It is essential to detect from which processor core the variables, constants and functions are accessed and with what frequency, as this is relevant for the subsequent distribution. In the second step, the target hardware used is examined, with focus on all memory-relevant hardware components. This includes

the memory itself as well as the processor cores, the hardware accelerators and the communication systems that connect the components of the microcontroller. The information obtained is used to calculate an optimized allocation, which allocates the software components to the various memory types depending on the calculated priority. To reduce concurrent accesses, copies of certain functions, variables and constants are also created and distributed to different memories as long as the corresponding memory space is available. In addition to the pure consideration of the runtime, memory protection mechanisms are taken into account by considering the corresponding granularity in the allocation.

To prove the correct functionality, the developed approach is evaluated on a self-developed test system, which is based on the software architecture of automotive control units for real-time capable applications. For this purpose, a large number of benchmarks for embedded systems are ported as representative load scenarios. The hardware platform used is a multi-core microcontroller from Infineon of the type AURIX TC39x, which is primarily used in safety-critical systems.

As the measurements carried out show, a significant improvement in all objectives can be achieved with the help of the concept presented here. By consistently using the available hardware components, both temporal and spatial isolation can be improved. In addition, the optimized use of fast memories can increase the execution speed and reduce the response time.

2

Einleitung

Viele der großen Herausforderungen der nächsten Jahre, wie der Umstieg auf regenerative Energien, das autonome Fahren, eine effektive Verkehrs- und Logistikregelung, aber auch die Automatisierung von umfangreichen Produktionsketten, erfordern eine immer größere Zahl an Sensorik und Steuerung [15] [16] [17]. Nur mit Hilfe ausreichend vieler Messdaten und einer intelligenten Regelung können die entsprechenden Ressourcen effizient genutzt werden. Beispielsweise zeigen Forschungsprojekte im Bereich der erneuerbaren Energien wie moderne Energiemanagementsysteme einen Beitrag zur nachhaltigen Energiewirtschaft leisten können [18]. Wichtig ist dabei zu beachten, dass diese Systeme nur dann zuverlässig funktionieren, wenn eine entsprechende Menge an Messwerten in ausreichender Güte sowie eine leistungsstarke Steuerung zur Verfügung stehen. Um diesen Ansprüchen gerecht zu werden, erfolgt in den letzten Jahren verstärkt der Einsatz von eingebetteten Systemen, welche die Schnittstelle zwischen der physikalischen und der digitalen Welt darstellen [15].

Eine besondere Gruppe von eingebetteten Systemen stellen die sogenannten Echtzeitsysteme dar, welche in der Regel dort eingesetzt werden, wo eine garantierte Reaktionszeit wichtig für die korrekte Funktionsweise des Gesamtsystems ist. Zwei typische Beispiele für solch ein System sind das Elektronisches Stabilitätsprogramm (ESP) in einem Fahrzeug oder auch die Leistungselektronik in einem Windkraftwerk. Bei beiden Anwendungsfällen ist es essentiell, dass das Echtzeitsystem mit der Erfassung des physikalischen Zustands sowie der entsprechenden Reaktion darauf fristgerecht fertig ist. Sollte das Echtzeitsystem die zeitlichen Anforderungen nicht erfüllen können, können die Auswirkungen ein instabiles Fahrverhalten im Falle des Fahrzeugs oder ein Kurzschluss im Falle der Leistungselektronik sein [19]. Aus diesen Folgen können Gefahr für Mensch und Maschine entstehen, weswegen Echtzeitsysteme vorwiegend in sicherheitskritischen Anwendungen zu finden sind.

Eine Schwierigkeit bei der Beurteilung von Echtzeitsystemen hinsichtlich ihrer Pünktlichkeit ist die sogenannte Vorhersagbarkeit. Nur, wenn ein System ein vorhersagbares Verhalten aufweist, können Aussagen über die maximale Reaktionszeit getroffen werden. Dies stellt einen signifikanten Unterschied zu anderen digitalen Systemen dar, welche häufig auf eine minimale durchschnittliche Ausführungszeit hin optimiert sind. Dieses Vorgehen führt in den meisten Fällen zu einer gesteigerten Performance, jedoch gibt es Situationen, bei welchen diese Art der Optimierun-

gen nicht greifen, was mit einer drastischen Erhöhung der Reaktionszeit einhergeht. Solche Laufzeitspitzen können dann dazu führen, dass Zeitziele nicht eingehalten werden. Aus diesem Grund sollten in Echtzeitsystemen nur Leistungssteigerungen eingeführt werden, welche die maximale Ausführungszeit reduzieren [20] [21].

Wie in anderen digitalen Systemen auch steigt der Bedarf an Rechenleistung in Echtzeitsystemen stetig an. Bisherige Ansätze zur deterministischen Leistungssteigerung, wie beispielsweise die Erhöhung der Taktfrequenz, stoßen zunehmend an physikalische Grenzen [22]. Um trotzdem die benötigte Rechenleistung zur Verfügung zu stellen, erfolgt, analog zu den Entwicklungen im Bereich der Computertechnik, zunehmend der Einsatz von Mikrocontrollern mit mehreren Prozessorkernen [23] [24] [25] [26]. Diese Entwicklung erfolgt jedoch mit einer deutlichen Verzögerung im Vergleich zu anderen digitalen Systemen. Der Grund für dieses Vorgehen besteht darin, dass die Vorhersagbarkeit von Mehrkernsystemen deutlich komplexer ist als die von Mikrocontrollern mit nur einem Prozessorkern [27] [28]. Dieser Umstand resultiert daraus, dass Mehrkernsysteme erstmalig eine echte parallele Verarbeitung zulassen, da mehrere Prozessorkerne unterschiedliche Aufgaben zur selben Zeit ausführen können. Im Gegensatz dazu wird in bisherigen Einkernsystemen die Gleichzeitigkeit nur simuliert, indem die Ausführung in sich unterbrechenden Zeitscheiben erfolgt. Der Vorteil dieser simulierten Parallelität besteht darin, dass jede Zeitscheibe während ihrer Ausführung exklusiven Zugriff auf alle Ressourcen im System, wie beispielsweise interne Kommunikationssysteme, Speicher oder Peripherien, hat [29] [30]. Dahingegen teilen sich in Mehrkernsystemen mehrere autarke Prozessorkerne die verfügbaren Ressourcen, was zu Wartezeiten bei gleichzeitigen Zugriffen führen kann [2].

Zur Lösung der Probleme der konkurrierenden Zugriffe auf geteilte Ressourcen haben sich in anderen Bereichen der digitalen Systeme verschiedene Ansätze entwickelt. Ein entsprechendes Konzept stellt die Synchronisierung der Prozessorkerne dar, wodurch die Zugriffe auf geteilte Ressourcen mittels fest definierter Zeitschlitze geregelt werden [31]. Dieser Ansatz ist jedoch nur bedingt für sicherheitskritische Systeme mit einer harten Echtzeitanforderungen geeignet, da die Prozessorkerne in der Regel möglichst autark voneinander arbeiten sollen. Der Hintergrund für dieses Vorgehen besteht darin, dass bei einem möglichen Fehlverhalten eines Prozessorkerns nicht das gesamte System versagt [32]. Ein alternativer Lösungsansatz stellt die separate Bereitstellung aller Ressourcen im System für jeden Prozessorkern dar, was jedoch aufgrund der damit verbundenen Kosten keine wirtschaftliche Option darstellt [29]. Aus diesem Grund bietet die gezielte und effiziente Nutzung der verfügbaren Ressourcen derzeit das vielversprechendste Konzept.

Im Gegensatz zu den anderen geteilten Ressourcen im System hat die Speicherhierarchie einen besonders großen Einfluss auf die Vorhersagbarkeit und die Ausführungsgeschwindigkeit in Echtzeitsystemen [33] [34]. Dieser Umstand ist darin begründet, dass in Mikrocontrollern für sicherheitskritische Anwendungen häufig eine große Anzahl verschiedener Speichertechnologien zum Einsatz kommen, welche sich hinsichtlich ihrer Zugriffsgeschwindigkeit und Speicherkapazität signifikant unterscheiden [4]. Eine Möglichkeit zur effektiven Nutzung dieser unterschiedlichen Eigenschaften stellt die dynamische Speicherverwaltung dar. Mittels dieses Kon-

zepts können gezielt die Informationen durch die schnellen Speicher bereitgestellt werden, welche aufgrund der aktuellen Berechnung benötigt werden. Jedoch stellen dynamische Ansätze in sicherheitskritischen Systemen ein Problem bei der Vorhersagbarkeit dar, weswegen ihr Einsatz in der Regel nur eingeschränkt möglich ist [35]. Stattdessen wird häufig auf ein statisches Verfahren zurückgegriffen, bei welchem die Nutzung der Speicher vor dem eigentlichen Systemstart bereits definiert ist. Bedingt durch dieses Vorgehen kann jedoch nicht zur Laufzeit auf unterschiedliche Ausführungspfade in der aktuellen Berechnung dynamisch reagiert werden. Um die damit verbundenen Leistungseinbuße möglichst gering zu halten, ist ein optimiertes, statisches Speichermanagement in Systemen mit einer harten Echtzeitanforderung essentiell für die maximale Rechenleistung sowie Vorhersagbarkeit [6].

2.1 Zielstellung

Bedingt durch die gestiegene Leistungsfähigkeit von Mikrocontrollern für sicherheitskritische Anwendungen ist seit einiger Zeit ein Trend zur Zentralisierung von Steuergeräten in automobilen Entwicklungen zu beobachten [36]. Durch dieses Vorgehen soll die Anzahl von Electronic Control Units (ECUs), welche in Oberklassemodellen zwischenzeitlich mehr als 100 Stück betrug, wieder deutlich reduziert werden [37]. Das Hauptziel für dieses Vorgehen ist dabei die Verringerung der Produktionskosten. Weiterhin ergeben sich Vorteile wie das Freiwerden von Bauraum, eine Reduzierung der benötigten Verkabelung, eine damit verbundene Gewichtsreduktion zukünftiger Automobile sowie die Möglichkeit zum Absenken des Energiebedarfs. All diese Faktoren zeigen, wie wichtig eine effiziente Nutzung der Leistungsfähigkeit von modernen eingebetteten Mehrkernprozessoren ist. Ein weiterer Punkt, welcher bei der fortschreitenden Verbreitung von sogenannten Integrationssteuergeräten zu beachten ist, ist die Beibehaltung der hohen Sicherheitsanforderungen, welche in der Automobilindustrie durch die International Organization for Standardization (ISO) 26262 vorgegeben sind. Neben vielen weiteren Aspekten fordert die ISO 26262 die Interferenzfreiheit zwischen Software-Komponenten unterschiedlicher Kritikalität in Bezug auf Laufzeit und der Integrität des Speichers [38]. Diese Forderung stellt in Verbindung mit der fortschreitenden Zentralisierung jedoch ein Problem dar, da immer mehr Funktionalitäten unterschiedlicher Kritikalität auf die wenigen Integrationssteuergeräte allokiert werden müssen. Hinzu kommt, dass die Software-Entwicklung in automobilen Anwendungen parallel sowie verteilt über diverse Zulieferer erfolgt, wodurch ein Zusammenführen aller Software-Komponenten erst spät im Entwicklungszyklus möglich ist [39]. Durch diesen dynamischen Ablauf ergibt sich das Problem, dass Abschätzungen in Bezug auf Laufzeit, Speicherverbrauch, Reaktionszeit und Echtzeitfähigkeit ebenfalls erst gegen Ende der Entwicklungsarbeiten vorgenommen werden können [40]. Ein weiterer, nicht zu unterschätzender Aspekt stellt die mehrfache Verwendung von bestimmten Software-Komponenten in diversen Integrationssteuergeräten in unterschiedlichen automobilen Plattformen dar. Dieses Vorgehen wird durch den Einsatz von AUTomotive Open System ARchitecture (AUTOSAR) ermöglicht, welche standardisierte Schnittstellen für ent-

sprechende Komponenten vorsieht. Dabei gilt jedoch zu beachten, dass sich dieselbe Software-Komponente auf anderen Steuergeräteplattformen hinsichtlich ihrer Laufzeit, Reaktionszeit und Echtzeitfähigkeit unterschiedlich verhalten kann. Diese Situation ist darauf zurückzuführen, dass sich die Plattformen unter anderem in Bezug auf den verwendeten Mehrkernprozessor, das Speichermanagement und die weiteren integrierten Software-Komponenten unterscheiden. Gerade der eingesetzte Mehrkernprozessor kann in Abhängigkeit der genutzten Prozessorkerne, der Anzahl der integrierten Hardware-Beschleuniger, der bereitgestellten Sicherheitsmechanismen sowie der Speicherhierarchie einen massiven Einfluss auf die Laufzeit, die Echtzeitfähigkeit sowie das erreichbare Sicherheitslevel einer Software-Komponente haben. Aufgrund all dieser Faktoren ist es essentiell, dass in Abhängigkeit der genutzten Steuergeräteplattform sowie der Anforderungen, welche sich durch die verwendete Software-Komponente ergeben, ein angepasstes Speichermanagement zum Einsatz kommt.

Aus diesem Grund wird in dieser Arbeit ein statisches Verfahren zur optimierten Speicherverwaltung in eingebetteten Mehrkernsystemen mit einer harten Echtzeitanforderung vorgestellt. Das Ziel dieses Ansatzes besteht darin, mittels einer gezielten Priorisierung von Funktionen und Daten in Abhängigkeit ihrer Kritikalität, ihres Speicherverbrauchs sowie ihres Einflusses auf die Laufzeit, eine angepasste Speicherallokation vorzunehmen, welche speziell auf den verwendeten Mehrkernprozessor zugeschnitten ist. Der Fokus dieser Arbeit liegt dabei auf eingebetteten automobilen Mehrkernprozessoren, welche bis zum höchsten Sicherheitslevel der ISO 26262 (ASIL D) spezifiziert sind. Neben diesen Anforderungen ist weiterhin keine Anpassung der genutzten Hardware-Plattformen vorgesehen, wodurch eine leichtere Überführung in bestehende Entwicklungsprozesse realisiert werden soll.

Der Beitrag dieser Arbeit gliedert sich dabei in die vier Zielstellungen, Temporale Isolierung, Räumliche Isolierung, Steigerung der Ausführungsgeschwindigkeit und Einhaltung der Reaktionszeit, welche im Folgenden detailliert erläutert werden.

2.1.1 Temporale Isolierung

Wie bereits erwähnt, erfolgt auf Integrationssteuergeräten für sicherheitskritische Anwendungen die Ausführung von Funktionalitäten unterschiedlicher Kritikalität. Bedingt durch dieses Vorgehen teilen sich verschiedene Software-Komponenten die gleichen Ressourcen, wodurch es in Systemen mit mehreren Prozessorkernen vermehrt zu Konflikten kommen kann. Die Gründe für diese Ressourcenkonflikte sind dabei vielfältig und erstrecken sich über alle Bereiche eines Mehrkernprozessors, wie beispielsweise Rechenzeit, Speicherkapazität oder Peripherien. Ein gängiges Problem, welches bereits von Steuergeräten mit nur einem Prozessorkern bekannt ist, ist die Nebenläufigkeit von Zeitscheiben in einem Echtzeitsystem. In automobilen Anwendungen erfolgt in Echtzeitsystemen die Verwendung eines präemptiven Scheduling-Algorithmus, welcher den Zeitscheiben anhand ihrer Periode eine feste Priorität zuweist. Dieser als Rate Monotonic Scheduling (RMS) bekannte Ansatz ermöglicht die Unterbrechung der Ausführung einer niederpriorisierten durch eine höherpriorisierte Task. Dadurch kann es bei der niederpriorisierten Zeitscheibe zu

Verzögerungen kommen, welche die rechtzeitige Fertigstellung der Berechnung negativ beeinflussen kann. Neben der reinen Rechenzeit entsteht zusätzlich noch weiterer Verzug durch die Speicher im System. Dieser Faktor ist darauf zurückzuführen, dass moderne Mehrkernprozessoren Caches zur Beschleunigung von sich wiederholenden Speicherzugriffen nutzen. Durch die Unterbrechung einer niederpriorisierten durch eine höherpriorisierte Zeitscheibe kann ein Umladen der Caches erfolgen, was bei der fortgesetzten Ausführung der niederpriorisierten Task zu weiteren Verzögerungen führt. Bei Mehrkernprozessoren kommt neben diesen Cache-Effekten zusätzlich noch das Problem der konkurrierenden Zugriffe auf geteilte Ressourcen hinzu. Dieser Umstand entsteht daraus, dass mehrere Prozessorkerne zur gleichen Zeit auf dieselben Speicher zugreifen, wodurch es zu Verzögerungen kommt, welche nur schwer in einer Laufzeitanalyse zu berücksichtigen sind. Gerade in Systemen mit einer harten Echtzeitanforderung ist es essentiell, dass sicherheitsrelevante Funktionalitäten zu einem definierten Zeitpunkt vollständig ausgeführt sind. Durch wiederholte konkurrierende Zugriffe können sich jedoch die daraus resultierenden Verzögerungen aufsummieren, was die Echtzeitfähigkeit gefährden kann. Durch diesen Effekt können sich sogar Zeitscheiben auf unterschiedlichen Prozessorkernen negativ beeinflussen, welche auf der rein funktionalen Ebene keinen Interaktionen miteinander haben. Neben den Prozessorkernen gibt es in aktuellen eingebetteten Prozessoren zusätzlich noch eine Vielzahl von integrierten Co-Prozessoren und Beschleunigern für unterschiedliche Anwendungsfälle. Beispielsweise können Hardware Security Modules (HSMs) eine sichere Enklave für vertrauliche Informationen darstellen oder Direct Memory Access (DMA)-Controller die Kopierzeiten von großen Datenmengen signifikant reduzieren [41] [12]. Was all diese Module gemeinsam haben, ist der Zugriff auf die geteilten Speicher im System, wodurch es ebenfalls zu konkurrierenden Zugriffen kommen kann.

Aus diesen Gründen wird in der ersten Zielstellung dieser Arbeit die temporale Isolierung deutlich verbessert. Dabei werden folgende Punkte im Detail optimiert:

1. Durch ein optimiertes Speichermanagement werden die vorhandenen Speicher im System den Prozessorkernen so zugewiesen, dass diese möglich exklusiv von einem Kern genutzt werden können. Zu diesem Zweck erfolgt eine Priorisierung von Funktionen und Daten anhand ihrer Kritikalität, ihrer Zugriffshäufigkeit durch die verschiedenen Prozessorkerne innerhalb eines definierten Beobachtungsfensters sowie ihres Speicherbedarfs. Anhand dieser Priorisierung wird anschließend eine Verteilung auf die vorhandenen Speicher vorgenommen, so dass zeitliche Verzögerungen aufgrund konkurrierender Zugriffe signifikant reduziert werden und Software-Komponenten unterschiedlicher Kritikalität sich nicht gegenseitig beeinflussen.
2. In Systemen mit einer harten Echtzeitanforderung ist es erforderlich, dass mittels einer Laufzeitanalyse die Einhaltung der Zeitziele nachgewiesen werden kann. Nur, wenn dieser Faktor gegeben ist, ist eine Zulassung für den Straßenverkehr möglich. Wichtig ist dabei zu beachten, dass bei jeder Ausführung immer die Worst-Case Execution Time (WCET) als Basis angenommen wird. Nur, wenn dieser Umstand gegeben ist, kann sichergestellt werden, dass alle Zeitscheiben

innerhalb ihrer Periode vollständig bearbeitet werden können. Durch Cache-Effekte können jedoch potentielle Verzögerungen entstehen, welche bei einer WCET-Analyse immer mit der maximal möglichen Latenz berücksichtigt werden müssen. Die Folge ist eine pessimistische WCET, welche die nutzbare Rechenzeit auf einem Integrationssteuergerät drastisch reduziert. Aus diesem Grund wird im Rahmen dieser Zielstellung eine optimierte Verwendung der Caches vorgenommen, wodurch eine einfachere Berücksichtigung in Laufzeitanalysen ermöglicht wird.
3. Aufgrund der bereits beschriebenen Limitierungen bei der Bereitstellung von zusätzlicher Rechenleistung in Systemen mit einer harten Echtzeitanforderung erfolgt in modernen Mehrkernprozessoren zunehmend der Einsatz von spezialisierten Beschleunigern für unterschiedliche Aufgaben, welche ebenfalls einen signifikanten Einfluss auf die Speicherauslastung haben. Daher wird in dieser Zielstellung ebenfalls der Einfluss von Beschleunigern auf die zeitliche Isolierung berücksichtigt und das Speicherlayout entsprechend optimiert.

2.1.2 Räumliche Isolierung

Neben der temporalen Isolierung ist der Schutz der Speicherintegrität ebenfalls ein relevanter Aspekt, welcher bei der Zulassung beachtet werden muss. Nur, wenn Funktionalitäten unterschiedlicher Kritikalität in voneinander getrennten Speicherbereichen allokiert sind und zusätzliche Schutzmechanismen aktiviert sind, kann eine Manipulation der Speicher effektiv verhindert werden. Wichtig ist dabei zu beachten, dass nicht nur die unerlaubte Veränderung von Daten und Programm-Code verhindert, sondern auch das Auslesen von geschützten Informationen unterbunden wird [42]. Grundlegend gibt es bei der Realisierung des Speicherschutzes zwei wichtige Faktoren zu berücksichtigen. Der erste Punkt ist die Granularität der eingesetzten Memory Protection Unit (MPU), welche bestimmt, wie groß die Sektionen im Speicher sind, welche minimal geschützt werden können. Je nach Mehrkernprozessor und Derivat kann sich die Granularität signifikant unterscheiden, was wiederum einen Einfluss auf die Speicherverwaltung hat. Der zweite wichtige Faktor ist die potentielle Überlagerung von erforderlichen Informationen. Beispielsweise könnten zwei Software-Komponenten unterschiedlicher Kritikalität dieselbe Funktion zur Ausführung benötigen. Durch die strikte Trennung dieser beiden Komponenten im Speicher ist dies nicht ohne Weiteres möglich, was bei der Speicherverwaltung ebenfalls berücksichtigt werden muss.

Zur Lösung dieser Probleme wird in der zweiten Zielstellung ein Ansatz entwickelt, welcher die räumliche Isolation optimiert. Zur Realisierung dieser Anforderungen werden die folgenden zwei Punkte umgesetzt:

1. Der Programm-Code und die Daten der Software-Komponenten werden auf Basis der Granularität der MPU so in den Speicher allokiert, dass Funktionalitäten unterschiedlicher Kritikalität voneinander isoliert werden können. Bei der Auswahl der entsprechenden Speicherbereiche werden die Anforderungen, welche sich aus der Zielstellung der temporalen Isolation ergeben, ebenfalls berücksich-

tigt. Durch dieses Vorgehen werden sowohl die Anforderungen an die Vorhersagbarkeit als auch die Anforderungen an die Speicherintegrität berücksichtigt.
2. Um die Anforderungen der räumlichen Isolation erfüllen zu können, werden Ansätze berücksichtigt, welche durch gezielte Überlagerung von Speicherbereichen oder die Duplikation von ausgewählten Teilen des Programm-Codes eine Trennung der Speicherbereiche ermöglichen. Das Ziel ist dabei, einen Kompromiss aus Speicherverbrauch und möglichen Ressourcenkonflikten zu erzielen, bei welchem sowohl die Vorhersagbarkeit als auch die Speicherintegrität garantiert werden können.

2.1.3 Steigerung der Ausführungsgeschwindigkeit

In modernen Mehrkernprozessoren für sicherheitskritische Anwendungen gibt es eine Vielzahl an integrierten Speichern, welche sich hinsichtlich Speicherkapazität, Latenz, Übertragungsrate und ihrer Funktionsweise unterscheiden. Je nachdem, wie die Speicherhierarchie in den eingesetzten Prozessoren strukturiert ist, kommen noch weitere Bedingungen, wie interne Kommunikationssysteme, die Anzahl der parallelen Schnittstellen sowie Zugriffsbeschleuniger, hinzu, welche ebenfalls einen signifikanten Einfluss auf die Leistungsfähigkeit haben. Nur, wenn die vorhandenen Speicher im System effektiv genutzt werden, kann das volle Potential eines Mikroprozessors mit mehreren Kernen erreicht werden. Faktoren, die bei der performanten Speicherverwaltung berücksichtigt werden sollten, sind die Zugriffshäufigkeiten der unterschiedlichen Prozessorkerne, aber auch die Speicherkapazität, welche zur Ablage von Programm-Code und Daten benötigt wird. Dabei gilt es zu beachten, dass die derzeitig verfügbaren Speicher mit einer hohen Zugriffsgeschwindigkeit und geringer Latenz nur über eine geringe Speicherkapazität verfügen, weswegen die effektive Nutzung essentiell ist [35] [9].

Daher wird im Rahmen dieser Zielstellung eine Möglichkeit zur optimierten Nutzung der verfügbaren Speicherhierarchie erarbeitet. Der Fokus liegt dabei auf der Verringerung der WCET, wodurch mehr Rechenleistung für zusätzliche Aufgaben zur Verfügung steht. Zur Erreichung dieser Zielstellung werden folgende Aspekte realisiert:

1. Durch die gezielte Allokation besonders rechenintensiver Funktionen und Daten mit einer hohen Aufrufhäufigkeit in die schnellen, aber kleinen Speicher wird die Ausführungsgeschwindigkeit signifikant gesteigert. Dabei liegt zusätzlich ein Fokus auf dem Speicherverbrauch der jeweiligen Funktionen und Daten. Das Ziel dieses Vorgehens besteht darin, eine möglichst große Anzahl der Speicherzugriffe durch die performanten Speicher zu realisieren. Aus diesem Grund wird ein Algorithmus erarbeitet, welcher die Zugriffshäufigkeit in Relation zum Speicherverbrauch setzt und eine optimierte Verwendung der schnellen aber kleinen Speicher ermöglicht.
2. Je nach Aufbau der integrierten Speicherhierarchie unterscheidet sich die Anbindung der Speicher deutlich untereinander. Verschiedene Faktoren, wie die Zuordnung zu bestimmten Prozessorkernen, die Art und Geschwindigkeit des internen Kommunikationssystems, die Arbitrierungsrichtlinien, die Anzahl der

Speicherschnittstellen oder die Möglichkeit zur Nutzung von Direktanbindungen, haben einen Einfluss auf die maximale Ausführungsgeschwindigkeit. Nur, wenn diese Faktoren im Speicherlayout effektiv berücksichtigt werden, kann die WCET zusätzlich reduziert werden. Zu diesem Zweck werden die Besonderheiten der Speicherhierarchie bei der Speicherverwaltung mit einbezogen, wodurch die Ausführungsgeschwindigkeit signifikant gesteigert wird.

3. Zur Beschleunigung von Speicherzugriffen sowie zur Reduzierung der Anzahl von konkurrierenden Zugriffen nutzen moderne Mehrkernprozessoren verschiedene Arten der Optimierung, wie beispielsweise das spekulative Vorladen oder das gezielte Alignment. Durch diese Implementierungen kann bei effektiver Nutzung die langsame Zugriffszeit bestimmter Speichertechnologien kompensiert werden. Zu diesem Zweck erfolgt im Rahmen der Speicherverwaltung ebenfalls eine Berücksichtigung dieser Beschleunigungstechnologien. Zusätzlich wird bei der optimierten Allokation darauf geachtet, dass es nicht zu einem Fragmentierungsproblem kommt, welches den Speicherverbrauch sonst deutlich erhöhen würde.

2.1.4 Einhaltung der Reaktionszeit

Das Ziel des optimierten Speicherlayouts besteht darin, dass die besonders laufzeitintensiven Software-Komponenten in die schnellen Speicher allokiert und konkurrierenden Zugriffe möglichst vermieden werden. Dabei gilt es zu beachten, dass die weniger kritischen Funktionalitäten nicht so im Speicher abgelegt werden, dass die geforderten Reaktionszeiten des Echtzeitsystem nicht mehr eingehalten werden können. Gerade asynchrone Events, wie beispielsweise Interrupts, können innerhalb eines definierten Beobachtungszeitraums unter Umständen gar nicht stattfinden. Ein bekanntes Beispiel für solch ein Problem stellt die Funktionalität zur Auslösung des Airbags in Automobilen dar. In der Regel wird die entsprechende Software-Komponente über die Lebensdauer eines Fahrzeugs nie aufgerufen, weswegen in diesem Falle die Zugriffshäufigkeit bei null liegt. Nichtsdestotrotz ist es essentiell, dass die Funktionalität im Bedarfsfall fristgerecht ausgeführt wird, damit bei einem Unfall ein Personenschaden verhindert werden kann. Daher ist der Fokus in dieser Zielstellung die Einhaltung der geforderten Reaktionszeiten. Des Weiteren wird in diesem Arbeitspaket ebenfalls überwacht, dass Funktionen, welche eine hohe Zugriffshäufigkeit, aber keinerlei Anforderungen hinsichtlich ihres Fertigstellungszeitpunkts haben, nicht zu hoch priorisiert werden. In diese Kategorie fallen unter anderem Hintergrundaufgaben, welche immer dann von dem Echtzeitbetriebssystem gestartet werden, wenn derzeit keine Zeitscheibe oder ein Interrupt verarbeitet werden muss. Je nach Auslastung des Systems mit zeitkritischen Funktionalitäten können diese Leerlaufaufgaben mitunter einen Großteil der Rechenzeit in Anspruch nehmen, was zu einer hohen Aufrufhäufigkeit führt.

Zur Erreichung dieser Zielstellung werden die folgenden Aspekte im Detail umgesetzt:

1. Sicherheitskritische Funktionen mit einem hohen Automotive Safety Integrity Level (ASIL) werden bei der optimierten Speicherverwaltung trotz geringer Zu-

griffshäufigkeit mit einer hohen Priorität versehen. Durch dieses Vorgehen wird verhindert, dass diese Software-Komponenten in kritischen Situationen ihre Zeitziele nicht einhalten können.
2. Zeitunkritische Funktionen ohne ein entsprechendes Sicherheitslevel werden trotz hoher Zugriffe nicht bevorzugt allokiert. Bei dieser Art von Hintergrundaufgaben stellen zeitliche Verzögerungen durch eine ungünstige Allokation oder konkurrierende Zugriffe kein Problem dar.

2.2 Aufbau der Arbeit

Die Struktur dieser Arbeit orientiert sich an dem allgemeinen Vorgehen zur Erarbeitung des beschriebenen Beitrags. Nach der Einleitung wird der aktuelle Stand der Technik präsentiert, wobei der Fokus auf dem grundlegenden Aufbau von echtzeitfähigen Mehrkernsystemen für sicherheitskritische Anwendungen in automobilen Systemen liegt. Des Weiteren erfolgt eine Beschreibung der derzeit bekannten statischen und dynamischen Speicherverwaltungsverfahren für die entsprechende Art von Mikrocontrollern. In dem sich anschließenden Kapitel wird das entwickelte Konzept zur optimierten Speicherallokation beschrieben und wichtige Designentscheidungen detailliert erläutert. Zur Validierung des Ansatzes werden im Anschluss umfangreiche Untersuchungen durchgeführt, welche die Funktionsweise verifizieren. Im abschließenden Kapitel werden die Ergebnisse resümiert und ein Ausblick auf potentielle Erweiterungen gegeben.

2.3 Kapitelzusammenfassung

Im Rahmen des zweiten Kapitels dieser Arbeit werden die Motivation sowie Zielstellungen für die weiteren Untersuchungen definiert. Diese gliedern sich grundsätzlich in vier Haupt- sowie dazugehörige Teilziele. Zu den vier Hauptzielen gehören die Verbesserung der temporalen- sowie räumlichen Isolierung, die Steigerung der Ausführungsgeschwindigkeit sowie die Einhaltung der Reaktionszeiten, welche besonders in sicherheitskritischen Echtzeitsystemen essentiell sind. Zur Erreichung der genannten Ziele werden verschiedene Teilziele beschrieben, welche sich primär auf die Priorisierung der Komponenten der Software sowie die gezielte Nutzung der verfügbaren Möglichkeiten der Hardware fokussieren.

3

Echtzeitfähige Mehrkernsysteme

In dem folgenden Kapitel werden technische Grundlagen, welche als Basis für das in dieser Arbeit entwickelte Konzept dienen, beschrieben. Zu diesem Zweck wird zu Beginn der prinzipielle Aufbau sowie die entsprechende Funktionsweise von eingebetteten Mehrkernsystemen für Anwendungen mit einer harten Echtzeitanforderung erläutert, wobei ein besonderer Fokus auf die Speicher sowie deren Anbindung gelegt wird. Im Anschluss werden Verfahren zur optimierten Speicherallokation vorgestellt, welche den derzeitigen Stand der Technik repräsentieren. Zur besseren Vergleichbarkeit der beschriebenen Ansätze werden diese dahingehend gruppiert, ob der Speicher statisch oder dynamisch verwaltet wird.

3.1 Hardware

Bedingt durch den gestiegenen Bedarf an Rechenleistung in sicherheitskritischen Systemen mit einer harten Echtzeitanforderung erfolgt in diesem Bereich verstärkt der Einsatz von eingebetteten Mehrkernprozessoren [43]. Dieser Umstand ist darin begründet, dass eine Leistungssteigerung durch die Erhöhung der Taktfrequenz oder die Erweiterung der zugrundeliegenden Prozessorarchitektur aufgrund von Beschränkungen in der Leistungsaufnahme sowie der Echtzeitfähigkeit nicht mehr in dem benötigten Maße möglich sind [22]. Die Limitierung der maximalen Taktfrequenz ist dabei bedingt durch die sogenannte *Power Wall*, welche das Verhältnis zwischen dem Performance-Gewinn durch Taktsteigerung und der damit verbundenen Zunahme der Leistungsaufnahme beschreibt. In anderen Bereichen der Prozessorentwicklung konnte eine Reduzierung der Leistungsaufnahme durch die fortschreitende Miniaturisierung der genutzten Schaltkreise erreicht werden, was jedoch in sicherheitskritischen Systemen nur eine eingeschränkte Option darstellt [44]. Der Grund dafür besteht in der höheren Fehleranfälligkeit von Prozessoren mit kleineren Strukturbreiten gegenüber äußeren Einflüssen, wie beispielsweise elektromagnetischer Strahlung. Zur Kompensation dieses Umstands werden redundante Schaltungen integriert, welche solche Fehlerfälle detektieren und korrigieren sollen [45]. Jedoch muss dabei berücksichtigt werden, dass diese zusätzlichen Schaltungen ebenfalls die Leistungsaufnahme steigern, weswegen eine genaue Abwägung des tatsäch-

lichen Performance-Gewinns erfolgen muss [46].

Eine weitere Begrenzung der Leistungsfähigkeit stellt die Anforderung nach vollständigem Determinismus des Prozessors dar [47]. Aus diesem Grund können moderne Techniken zur Steigerung der Ausführungsgeschwindigkeit, wie beispielsweise spekulatives Laden, spekulative Ausführung, lastabhängige Taktraten, geteilte Caches, simultanes Multithreading oder Out-of-Order-Execution, lediglich eingeschränkt genutzt werden, da sie die Komplexität der Bewertung der Echtzeitfähigkeit deutlich erhöhen [48] [20] [27] [49]. Aus den genannten Gründen stellt die Integration weiterer Prozessorkerne eine zusätzliche Möglichkeit zur Bewältigung des gestiegenen Bedarfs an Rechenleistung dar.

3.1.1 Aufbau

In automobilen Anwendungen gibt es derzeit vier Anbieter für eingebettete Mehrkernprozessoren, welche in sicherheitskritischen Systemen, je nach konkretem Anwendungsfall, bis zur höchsten Safety-Stufe eingesetzt werden können. Die Einteilung der Sicherheitslevel erfolgt dabei anhand der Norm ISO 26262, welche in automobilen Systemen anhand ihrer Kritikalitätsstufe definiert werden [38]. Je kritischer das Versagen eines Systems für das Leben der Personen ist, desto höher ist das entsprechende ASIL, wobei ASIL D die höchste Sicherheitsstufe darstellt. Eine Übersicht der derzeit am Markt befindlichen eingebetteten Mehrkernprozessoren, welche für Anwendungen bis einschließlich ASIL D geeignet sind, findet sich in Tabelle 3.1. Die Tabelle verzichtet bewusst auf hybride Architekturen, wie beispielsweise die Renesas R-Car V3U-Familie, welche sowohl echtzeitfähige Mikrocontrollerkerne als auch leistungsstarke Prozessorkerne auf Basis der Cortex-A-Architektur enthalten. Diese Systeme sind primär für Anwendungen, wie das automatisierte Fahren, gedacht und nicht Teil dieser Betrachtung.

Tabelle 3.1: Übersicht eingebetteter Mehrkernprozessoren für sicherheitskritische Systeme mit einer harten Echtzeitanforderung [50] [51] [52] [53] [54] [55] [56] [57]

Hersteller	**Serie**	**Anzahl CPUs**	**Architektur**	**ASIL-Level**
Infineon	AURIX	1 - 6	TriCore	D
NXP	MPC55xx	1 - 3	PowerPC	D
NXP	S32Z/E/K	9	Cortex-M, Cortex-R	D
ST	SPC5x	1 - 3	PowerPC	D
ST	Stellar G7	6	Cortex-R	D
Renesas	RH850	1 - 6	V850	D

Je nach Hersteller und Derivat unterscheiden sich die Anzahl der genutzten Prozessorkerne sowie die zugrundeliegende Prozessorarchitektur voneinander. Dabei ist zu beachten, dass die Tabelle 3.1 keine Unterscheidung zwischen den einzelnen Generationen vornimmt. Beispielsweise existieren bei der AURIX-Serie von Infineon drei unterschiedliche Generation, welche sich hinsichtlich der maximal verfügbaren Anzahl an Prozessorkernen unterscheiden. Während die erste Generation, welche

von Infineon als TC2xx-Familie vermarktet wird, Derivate mit einem bis maximal drei Prozessorkernen bietet, werden beispielsweise in der dritten Generation (TC4xx-Familie) Derivate mit zwei bis sechs Kernen angeboten [58] [59]. Zur besseren Übersicht wird in der Tabelle 3.1 auf eine separate Betrachtung der verschiedenen AURIX-Generationen verzichtet.

Die derzeit verfügbaren eingebetteten Mehrkernprozessoren für Anwendungen mit einer harten Echtzeitanforderung nutzen grundlegend ein ähnliches Layout, welches in Abbildung 3.2 zu sehen ist. Je nach Hersteller und Derivat besitzen die in Tabelle 3.1 genannten Mikrocontroller eine unterschiedliche Anzahl an Prozessorkernen, welche nach der modifizierten Harvard-Architektur aufgebaut sind. Daher besitzt jeder Prozessorkern ein separates Interface für den Zugriff auf Code und Daten. Innerhalb dieser Interfaces liegen die lokalen Speicher, welche durch ein Scratchpad und einen Cache realisiert werden. Die Anbindung der Prozessorkerne untereinander sowie mit den globalen Speichern erfolgt über ein Network-on-Chip (NoC), welches als Kommunikationsschnittstelle fungiert. Die globalen Speicher werden durch den nichtflüchtigen Flash-Speicher sowie einen globalen Random-Access Memory (RAM) repräsentiert, welche von allen Prozessorkernen geteilt werden. Durch die externen Speicher wird die dritte Speicherstufe beschrieben, welche mittels eines externen Speicherinterfaces angebunden werden kann.

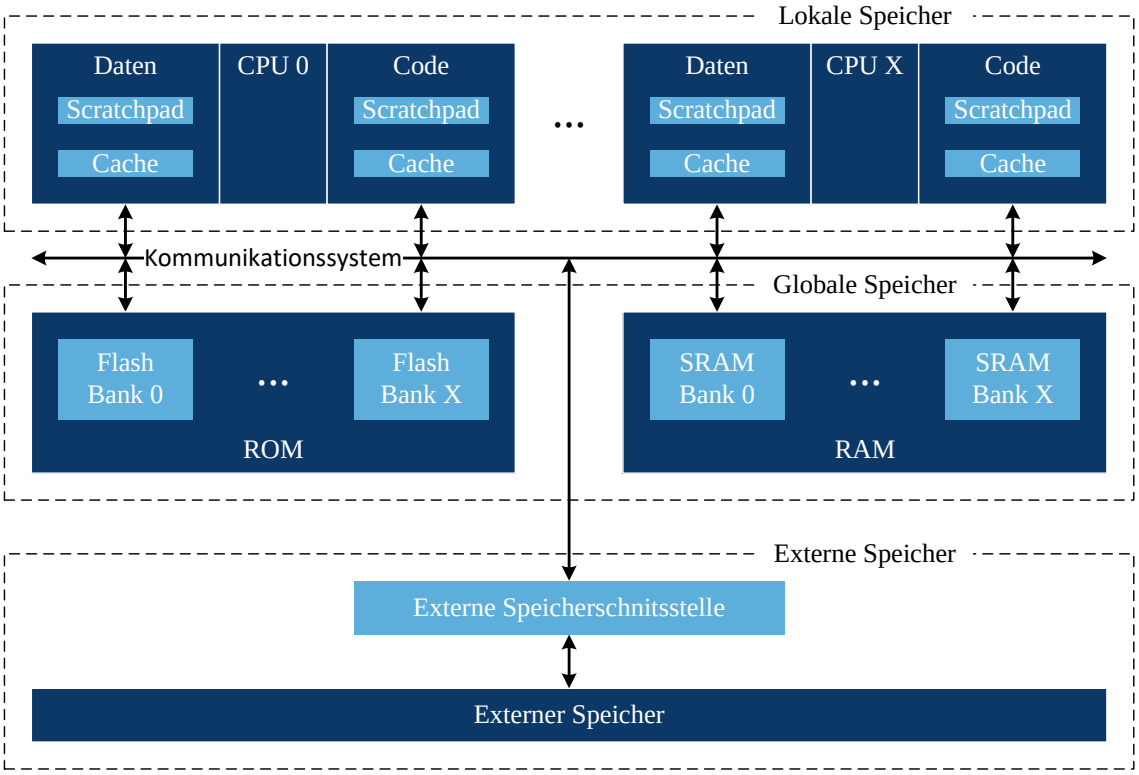

Abbildung 3.2: Grundlegendes Speicherlayout eines eingebetteten Mehrkernprozessors für sicherheitskritische Systeme mit einer harten Echtzeitanforderung

3.1.2 Prozessorkern

Wie in der Tabelle 3.1 dargestellt, nutzen die derzeit am Markt befindlichen Hersteller unterschiedliche Prozessorarchitekturen zur Realisierung ihrer Hauptprozessorkerne. Interessant ist dabei zu beobachten, dass die Hersteller bei den ersten Mehrkernmikrocontrollerfamilien vorwiegend auf proprietäre Architekturen wie TriCore, PowerPC oder V850 gesetzt haben. Während Infineon und Renesas weiterhin auf diese Prozessorarchitekturen aufbauen und diese auch weiterentwickeln, haben NXP und ST den Wechsel zu ARM-basierten Mikrocontrollern vorgenommen und nutzen vorwiegend die Cortex-M und Cortex-R-Kerne in verschiedenen Ausbau- und Leistungsstufen [60] [61].

Wie in Abbildung 3.3 dargestellt, können grundsätzlich alle Mehrkernmikrocontroller in drei Kategorien anhand ihrer Prozessorkerne einsortiert werden [62]:

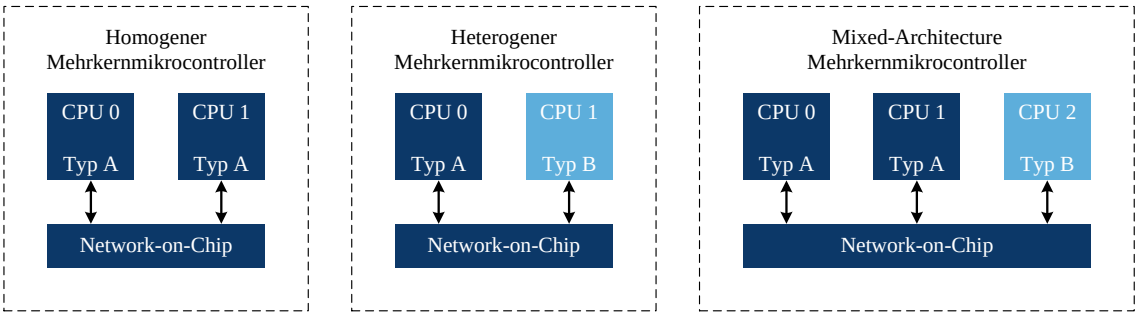

Abbildung 3.3: Grundlegende Typen von eingebetteten Mehrkernprozessoren für sicherheitskritische Systeme mit einer harten Echtzeitanforderung

Homogene Mehrkernmikrocontroller Bei den Homogenen Derivaten nutzen alle Prozessorkerne eine identische Architektur. Der Vorteil von diesem Ansatz besteht in dem geringeren Programmieraufwand, da für alle Prozessorkerne dieselbe Code-Basis genutzt werden kann. Des Weiteren können entsprechende Mikrocontrollerfamilien über die Anzahl der Kerne deutlich leichter in ihrer Leistungsfähigkeit skaliert werden. Nachteilig ist hingegen die geringe Adaption an spezielle Aufgaben, welche von einer angepassten Prozessorarchitektur profitieren würden. Beispielsweise können hier Digital Signal Processors (DSPs) oder Vektoreinheiten genannt werden, welche bei der Signalverarbeitung oder bei Matrizenoperationen deutlich performanter und energieeffizienter arbeiten [62] [63] [7] [64].

Heterogene Mehrkernmikrocontroller Im Gegensatz zu den homogenen Mehrkernmikrocontrollern kommen bei heterogenen Derivaten ausschließlich unterschiedliche Prozessorkernarchitekturen zum Einsatz. Aus diesem Umstand ergeben sich die Vorteile der erhöhten Rechenleistung sowie besseren Energieeffizienz bei entsprechenden Arbeitslasten [65] [66]. Nachteilig ist hingegen die komplexere Software-Architektur im Vergleich zu den homogenen Systemen, welche auf die unterschiedlichen Prozessorkerntypen angepasst sein muss. Des Weiteren stellt die Erweiterbarkeit solcher Systeme ein Problem dar, da für jeden Aufgabentyp eine bestimmte Prozessorarchitektur vorgesehen ist. Sollte der entsprechende

Kern jedoch bereits ausgelastet sein, können unter Umständen geplante Funktionalitäten nicht mehr umgesetzt werden [62] [63] [7].

Mixed-Architecture Mehrkernmikrocontroller Eine Mischung aus beiden Ansätzen stellen die Mixed-Architecture Mehrkernmikrocontroller dar, welche mehrere unterschiedliche Prozessorkerne verwenden. Das Ziel dieses Ansatzes ist die Kombination der Vorteile der homogenen- und heterogenen Architekturen, ohne dabei deren Nachteile zu übernehmen. Aus diesem Grund basieren derzeit alle auf dem Markt befindlichen Mehrkernmikrocontroller auf diesem Konzept. So integriert beispielsweise Infineon in seine AURIX-Mikrocontrollerfamilie neben den Hauptprozessorkernen auf Basis der TriCore-Architektur unter anderem ein HSM mit Cortex-M3-Kern [67], eine Generic Timer Module (GTM) mit fünf separaten Kernen zur Ansteuerung von komplexen Pulsweitenmodulations (PWMs)-Mustern [68], eine Vektoreinheit auf Basis der EV7x-Architektur sowie mehrere DSPs zur effizienten Verarbeitung von Radarsignalen und analogen Messgrößen [62] [63] [51] [69] [7] [70].

Allen Prozessorkernarchitekturen aus Tabelle 3.1 ist gemein, dass sie ein Reduced Instruction Set Computer (RISC)-basiertes Design mit mehrstufigen, superskalaren Pipelines nutzen, welche die Instruktionen In-Order verarbeiten [71]. Die einzige Ausnahme bilden die Infineon AURIX Derivate der TC2xx-Serie mit Efficiency-Kernen sowie ausgewählte SPC5x-Modelle, welche auf ein superskalares Design verzichten [72] [54] [73]. Der Vorteil eines superskalaren Aufbaus besteht darin, dass pro Takt mehrere skalare Instruktionen verarbeitet werden können, wodurch die Rechenleistung eines Prozessorkerns bei entsprechenden Arbeitslasten steigt. Im Gegensatz zu Mikroprozessoren nutzen Mikrocontroller für echtzeitfähige Anwendungen vorwiegend In-Order basierte Architekturen, welche die Instruktionen in der Reihenfolge verarbeiten, wie der Compiler diese vorgibt. Diese Designentscheidung resultiert daraus, dass In-Order-basierte Ansätze ein deterministisches Verhalten aufweisen, welches Out-of-Order-Architekturen nicht in diesem Maße zur Verfügung stellen. Eine weitere Besonderheit in Echtzeitsystemen stellt die Verwendung einer Sprungvorhersage dar, welche aufgrund ihres indeterminierten Verhaltens eher in Mikroprozessoren zum Einsatz kommt. Nichtsdestotrotz sind moderne Mikrocontrollerarchitekturen, bedingt durch ihre mehrstufigen Pipelines, auf eine Sprungvorhersage angewiesen, um die geforderte Rechenleistung zur Verfügung zu stellen. Aus diesem Grund nutzen fast alle der in Tabelle 3.1 aufgelisteten Architekturen eine entsprechende Einheit. Die in Tabelle 3.1 dargestellten Mikrocontrollerfamilien nutzen für die Hauptprozessorkerne 32 Bit-breite Register. Daraus resultiert der Umstand, dass die Speicherzugriffe auf Adressen mit einem 32 Bit-Alignment besonders effizient umgesetzt werden können. Dahingegen sind Zugriffe auf Speicheradressen mit einem kleineren Alignment häufig nur indirekt möglich, indem ein 32 Bit-Zugriff vorgenommen wird und die nicht benötigten Informationen im Anschluss ausmaskiert werden. Diese zusätzlichen Schritte benötigen ebenfalls Rechenzeit, weswegen entsprechende Speicherzugriffe mehr Laufzeit benötigen. Eine Möglichkeit zur Reduzierung dieser Effekte ist die Verwendung eines 32 Bit-Alignments für alle Daten, was jedoch zu einem erhöhten Speicherverbrauch führt.

Um den eigentlichen Prozessorkern von Schreibzyklen auf die globalen Speicher und den damit verbundenen Wartezyklen zu entkoppeln, verfügen die meisten modernen Mikrocontrollerdesigns pro Prozessorkern über einen separaten Store-Buffer, welcher die Schreibzugriffe autark realisiert. Sollte der Prozessorkern ein Datum aktualisieren, bevor der Store-Buffer den Wert zurückschreiben konnte, wird dieser direkt im Store-Buffer überschrieben, wodurch die Anzahl der Schreibzugriffe auf die globalen Speicher reduziert werden kann. Ein weiterer Vorteil entsteht durch die Entkopplung der Schreibzugriffe von der eigentlichen Ausführung des Prozessorkerns. Sollte ein Speicher gerade durch einen konkurrierenden Zugriff belegt sein, wird die Berechnung auf dem zugeordneten Kern nicht verzögert. Durch diesen Umstand können die Effekte von konkurrierenden Zugriffen reduziert werden. Nachteilig ist hingegen das ebenfalls indeterminierte Laufzeitverhalten, da der Füllgrad des Store-Buffers während der eigentlichen Ausführung nicht zuverlässig vorhergesagt werden kann. Die in Tabelle 3.1 dargestellten Prozessorkernarchitekturen nutzen fast alle einen Store-Buffer, wobei sich die Anzahl der speicherbaren Schreibzugriffe unterscheidet.

Mit der Einführung der neuen Mikrocontrollerfamilien in Form der Stellar G7-Serie von ST sowie der TC4xx-Serie von Infineon gibt es zum ersten Mal Hardware-Unterstützung für die Virtualisierung [74]. Im Gegensatz zu den gängigen Ansätzen aus der Computertechnik erfolgt bei echtzeitfähigen Mikrocontrollern keine Virtualisierung der Speicherzugriffe. Die Software auf den Prozessorkernen nutzt für die Zugriffe weiterhin die physikalischen Adressen der Speicherbereiche. Die Virtualisierung beschränkt sich daher lediglich auf die Rechenzeit, welche auf dem jeweiligen Prozessorkern zur Verfügung steht. Zu diesem Zweck ergänzen die Mikrocontrollerhersteller die Prozessorkerne um zusätzliche Register, welche die Nutzung von mehreren virtuellen Maschinen sowie des dazugehörigen Hypervisors unterstützen. Des Weiteren werden das Interrupt-Routing sowie der Speicherschutz dahingehend erweitert. Durch das modifizierte Routing können bestimmte Ereignisse direkt zu der entsprechenden virtuellen Maschine weitergeleitet werden, wodurch die Antwortzeit im Gegensatz zu einem software-basierten Ansatz reduziert werden soll. Mittels des erweiterten Speicherschutzes können die Virtuellen Maschinen voneinander isoliert werden, was besonders bei Implementierungen unterschiedlicher Kritikalität relevant ist [49]. Das Ziel der Hersteller ist dabei die Entwicklungszeiten von eingebetteter Software zu reduzieren, was besonders die Bereitstellung von Updates angeht [75]. Dies soll durch die stärkere Isolierung der virtuellen Maschinen hinsichtlich der Rechenzeit ermöglicht werden. Zu diesem Zweck sollen bei Software-Updates nur noch die angepassten virtuellen Maschinen hinsichtlich ihres Laufzeitverhaltens validiert werden und nicht mehr die gesamte Software des Prozessorkerns. Voraussetzung hierfür ist die korrekte Funktionsweise des Hypervisors.

Eine Besonderheit von sicherheitskritischen Echtzeitsystemen ist das Vorhandensein von redundanten Prozessorkernen für die Detektion von Rechenfehlern. Diese als *Lockstep*-Kern bezeichneten Schaltungen führen mit einem zeitlichen Versatz dieselben Instruktionen wie der Hauptprozessorkern aus. Durch eine anschließende Vergleichslogik werden die beiden Ergebnisse miteinander verglichen und bei einem Unterschied eine entsprechende Fehlersignalisierung eingeleitet. Bei der Umsetzung

von *Lockstep*-Kernen existieren derzeit zwei verschiedene Ansätze. Bei dem ersten Konzept können zwei separate Prozessorkerne so konfiguriert werden, dass einer der beiden als Haupt- und der andere als *Lockstep*-Kern fungiert. Der Vorteil dieser Umsetzung besteht in der hohen Flexibilität, welche es ermöglicht, sowohl alle Prozessorkerne für separate Rechenaufgaben zu nutzen als auch bei Bedarf die *Lockstep*-Funktionalität einzuschalten. Nachteilig hingegen ist die halbierte theoretische Rechenleistung bei Nutzung der Redundanzeigenschaften. Im Gegensatz dazu gibt es den Ansatz der dedizierten *Lockstep*-Kerne, welche den Prozessorkernen direkt zu geordnet und welche ausschließlich für die Realisierung von Sicherheitsfunktionen vorgesehen sind. Der Vorteil besteht darin, dass die theoretische Rechenleistung eines solchen Systems nicht von der Aktivierung der *Lockstep*-Funktionalität abhängig ist. Im Gegensatz dazu können diese dedizierten *Lockstep*-Kerne nicht für andere Aufgaben, außer zur Realisierung der Redundanzeigenschaften genutzt werden [46] [47] [76] [8].

3.1.3 Hardware-Beschleuniger

Eine Möglichkeit zur Steigerung der Leistungsfähigkeit eines eingebetteten Mehrkernprozessors stellt die Integration zusätzlicher Prozessorkerne dar. Jedoch muss dabei berücksichtigt werden, dass weitere Kerne nicht in jedem Falle die nutzbare Rechenleistung steigern können. Wie Gene Amdahl in dem nach ihm benannten Gesetz bereits aufgezeigt hat, muss eine Software einen hohen Anteil an parallelisierbaren Bestandteilen aufweisen, damit weitere Prozessorkerne einen Geschwindigkeitsvorteil bringen können [77]. Einen alternativen Ansatz stellt die Bereitstellung spezialisierter Beschleuniger dar, welche ausgewählte Aufgaben signifikant schneller ausführen können. Als Beispiel für solche Beschleuniger können unter anderen DMA-Controller, Vektorprozessoren oder Acceleratoren für kryptografische Operationen, wie Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA) oder True Random Number Generator (TRNG), genannt werden [41] [78] [79] [80] [81].

Wichtig ist dabei zu beachten, dass diese Beschleuniger auf die Speicher im System zugreifen, wodurch diese ebenfalls konkurrierende Zugriffe erzeugen können, welche zu potentiellen Wartezyklen führen. Diese Beeinflussung der Laufzeit muss in einer entsprechenden WCET-Analyse berücksichtigt werden. Ein weiterer Faktor, welcher beim Einsatz solcher Acceleratoren beachtet werden muss, ist der Speicherschutz in sicherheitskritischen Anwendungen. Bedingt durch den Umstand, dass beispielsweise DMA-Controller große Datenmengen zwischen verschiedenen Speicherbereichen verschieben können, besteht die Gefahr einer ungewollten Manipulation im Fehlerfall. Aus diesem Grund sind solche Beschleuniger ebenfalls beim Speicherschutzkonzept zu berücksichtigen [12].

3.1.4 Network-on-Chip

Die performante Anbindung mehrerer Prozessorkerne sowie einer Vielzahl von Hardware-Beschleunigern und Speichern erfordert ein leistungsstarkes NoC. Zusätzlich

setzen sicherheitskritische Echtzeitsysteme eine vorhersagbare und deterministische Funktionsweise voraus [82]. Aus diesem Grund haben sich in eingebetteten Mehrkernsystemen Crossbars durchgesetzt, welche im Vergleich zu Bussystemen parallele Verbindungen von unterschiedlichen Mastern zu unterschiedlichen Slaves erlauben [77]. Als Master fungieren in solchen Systemen in der Regel die Prozessorkerne sowie die Hardware-Beschleuniger, welche aktiv einen Zugriff auf die Speicher vornehmen können, wohingegen die Speicher als Slaves umgesetzt sind. Crossbars kommen in derzeit allen am Markt erhältlichen echtzeitfähigen Mehrkernmikrocontrollern für die performante Anbindung von verschiedenen Kommunikationsteilnehmern zum Einsatz [83]. Für die Anbindung von Peripheriemodulen wird dahingegen weiterhin auf klassische Bussysteme gesetzt, welche lediglich eine sequenzielle Übertragung ermöglichen. Der Vorteil von Bussystemen im Vergleich zu den performanteren Crossbars besteht in ihrem geringen Bedarf an Chipfläche. Bei einer Crossbar sind alle Kommunikationsteilnehmer direkt miteinander verbunden, weswegen mit jedem weiteren Master beziehungsweise Slave der Bedarf an Chipfläche exponentiell ansteigt. Aus diesem Grund werden in den meisten Mikrocontrollerarchitekturen mehrere Crossbars verwendet, welche über Bridges miteinander verbunden sind, was in Abbildung 3.4 dargestellt ist. Der Vorteil dieser Umsetzung besteht darin, dass weitere Kommunikationsteilnehmer hinzugefügt werden können, ohne dass der Bedarf an Chipfläche exponentiell ansteigt. Nachteilig an dieser Lösung ist die zusätzliche Latenz, welche durch die Bridge erzeugt wird sowie die potentielle Limitierung der Bandbreite aufgrund von konkurrierenden Zugriffen [84]. Zur Reduzierung dieser Effekte haben die Hersteller solcher Mikrocontroller die Crossbar-Domains so konzipiert, dass diese für bestimmte Aufgaben prädestiniert sind. So nutzt beispielsweise ST bei der SPC58-Familie zwei Crossbar-Domains, wobei an der ersten Domain die beiden leistungsstärkeren Prozessorkerne zu einem Rechencluster zusammengefasst werden. Die zweite Crossbar fungiert hingegen als Peripheriecluster und bindet einen leistungsschwächeren Prozessorkern zur Verwaltung sowie diverse breitbandige Kommunikationsschnittstellen an [54]. Das Ziel dieser Separierung ist die Minimierung von Zugriffen über die Crossbar-Grenzen hinweg.

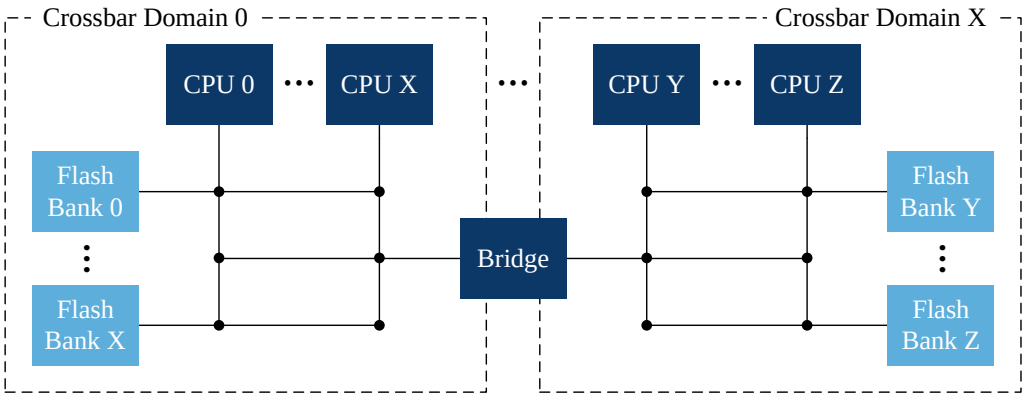

Abbildung 3.4: Grundlegender Aufbau einer Crossbar eines eingebetteten Mehrkernprozessors für sicherheitskritische Systeme mit einer harten Echtzeitanforderung

Neben der Anbindung über die Crossbar gibt es in verschiedenen Mikrocontrollerfamilien noch weitere Verbindungen zwischen den Prozessorkernen und den Speichern im System. Beispielsweise hat Infineon bei der AURIX TC3xx-Serie eine Direktverbindung von jedem Prozessorkern zu jeweils einem Bereich des Flash-Speichers integriert [85] [51] [86]. Der Vorteil dieses Ansatzes besteht darin, dass die Anzahl der Crossbar-Zugriffe auf die Slaves reduziert werden können sowie die Zugriffsgeschwindigkeit durch die fehlende Arbitrierung gesteigert werden kann [4]. Nachteilig ist hingegen die gesteigerte Komplexität bei der Speicherallokation, da entsprechende Verbindungen berücksichtigt werden müssen. In Abbildung 3.5 ist dieser Aufbau exemplarisch dargestellt.

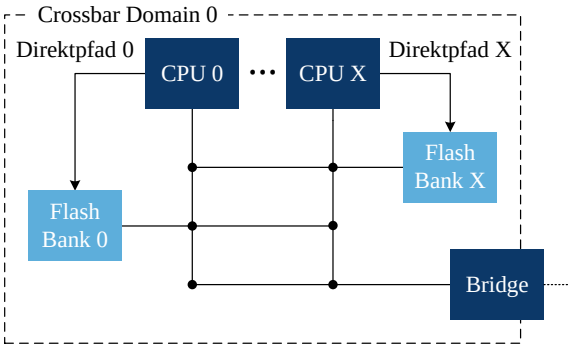

Abbildung 3.5: Grundlegender Aufbau einer Crossbar mit Direktanbindung zu den Flash-Speichern eines eingebetteten Mehrkernprozessors für sicherheitskritische Systeme mit einer harten Echtzeitanforderung

Alle in Tabelle 3.1 aufgelisteten Mikrocontrollerfamilien nutzen ein 32 Bit-breites Design für die Prozessorkerne. Nichtsdestotrotz sind die dazugehörigen Crossbars häufig breitbandiger ausgelegt, wie beispielsweise bei den Infineon AURIX-Derivaten, welche mit einer 64 Bit-breiten Crossbar aufwarten. Der Vorteil besteht in der höheren Übertragungsgeschwindigkeit von Programm-Code und Daten von den globalen Speichern zu den jeweiligen Prozessorkernen pro Transfer. Dieser Umstand kann durch *Burst*-Transfers zusätzlich gesteigert werden, indem die Crossbar für eine gewisse Anzahl an Übertragungen von einem Master zu einem Slave reserviert wird. Durch die fehlende Arbitrierung zwischen den einzelnen Übertragungen eines *Burst*-Transfers kann die verfügbare Bandbreite effizienter genutzt werden. Sobald mehrere Master konkurrierend auf einen Slave zugreifen wollen, ist eine Arbitrierungseinheit zur Koordinierung zwingend erforderlich. Die Aufgaben der Arbitrierung sind dabei die Sortierung der Anfragen sowie die Überwachung, dass kein Master zu lange warten muss, was ansonsten die Echtzeitfähigkeit gefährden könnte. Die Sortierung kann dabei nach verschiedenen Ansätzen erfolgen, wie beispielsweise Round-Robin oder prioritätsbasiert. Bei dem prioritätsbasierten Ansatz bekommt jeder Master eine separate Wichtigkeit zugewiesen anhand derer die Arbitrierungseinheit die Sortierung der Anfragen vornimmt. Dabei ist zu beachten, dass auch Master mit einer geringen Priorität nicht zu lange auf einen Zugriff warten dürfen. Aus diesem Grund zählen die Arbitrierungseinheiten mit, wie lange ein Master schon auf seinen Zugriff war-

tet. Ist ein bestimmter Schwellschwert überschritten, erhält dieser niederpriorisierte Master kurzzeitig die höchste Dringlichkeit. Durch diesen als *Starvation Protection* bezeichneten Mechanismus kann jedem Master eine maximale Antwortzeit garantiert werden, was besonders in echtzeitfähigen Systemen essentiell ist.

3.1.5 Speicher

Moderne Mehrkernmikroprozessoren für sicherheitskritische Anwendungen nutzen eine modifizierte Harvard-Architektur mit integrierten, separaten Speichern für den Programm-Code sowie für die Daten als auch Mischspeicher, welche für beides genutzt werden können. Als Speichertechnologie kommt dabei Static Random-Access Memory (SRAM) für die lokalen Speicher der Prozessorkerne sowie für die globalen RAM-Speicher zum Einsatz, während für die nichtflüchtigen, globalen Speicher vorwiegend eingebetteter Flash genutzt wird. Dieser komplexe Aufbau aus verschiedenen Speicherhierarchien und -Technologien ist erforderlich, da die Leistungsfähigkeit der Prozessorkerne in den letzten Jahrzehnten deutlich stärker angestiegen ist als bei den Speichertechnologien. Diese Leistungsdifferenz wird als *Memory-Performance-Gap* bezeichnet und ist einer der Hauptgründe für die immer komplexer werdenden Speichertopologien [33] [9].

Der Zugriff auf die unterschiedlichen Speicher der jeweiligen Speichertopologien erfolgt bei Mikrocontrollern anhand der physikalischen Speicheradresse auf Basis eines globalen Adressraums. Dieser Ansatz wird in der Literatur auch als Non-Uniform Memory Access (NUMA)-Architektur definiert. Einige Mikrocontrollerarchitekturen nutzen zusätzlich Caches zur Beschleunigung von sich wiederholenden Zugriffen auf globale Speicher. Da diese Caches aufgrund der Echtzeitanforderungen auf ein Kohärenzprotokoll verzichten, wird dies als Non-Cache Coherent Non-Uniform Memory Access (NCCNUMA) bezeichnet [77].

3.1.5.1 Lokale Speicher

Als lokale Speicher oder auch Tightly Coupled Memory (TCM) werden, wie in Abbildung 3.2 dargestellt, alle Speicher in einem System bezeichnet, welche einem Prozessorkern direkt zugeordnet sind. Diese Definition schließt nicht aus, dass andere Prozessorkerne ebenfalls auf diese Speicherbereiche zugreifen können, jedoch mit reduzierter Geschwindigkeit [87]. Aus dieser Eigenschaft wird ersichtlich, dass die lokalen Speicher primär durch den ihnen zugeordneten Prozessorkern exklusiv genutzt werden sollten. Durch lokale Kopien in den jeweiligen lokalen Speichern kann die Ausführungsgeschwindigkeit der Prozessorkerne gesteigert und die Effekte von konkurrierenden Zugriffen reduziert werden [1]. Als Basis für die lokalen Speicher wird SRAM genutzt, welcher eine hohe Zugriffsgeschwindigkeit bietet. Nachteilig ist hingegen der große Bedarf an Chipfläche und den damit verbundenen Herstellungskosten, weswegen SRAM-Speicher in der Regel nur wenige Kilobyte an Speicherkapazität bieten. Aus diesem Grund ist es essentiell für die optimierte Nutzung der verfügbaren Rechenleistung, die kleinen, aber performanten Speicher effizient zu nutzen [88] [9].

Scratchpad Als Scratchpads werden prozessorkern-nahe SRAM-Speicher bezeichnet, welche zusammen mit den Caches die höchste Speichergeschwindigkeit im System bieten. Zusätzlich zeichnen sich Scratchpads durch ein absolut deterministisches Verhalten aus, weswegen sie besonders häufig in Echtzeitsystemen zum Einsatz kommen. Im Gegensatz zu den nichtflüchtigen Speichern müssen die benötigten Daten sowie der Programm-Code nach dem Systemstart in die Scratchpads kopiert werden. Zur Laufzeit halten die Scratchpads dann eine lokale Kopie des entsprechenden Programm-Codes vor. Daraus ergibt sich der Vorteil, dass bei geteilten Funktionen, welche von mehreren Prozessorkernen genutzt werden, die Anzahl der konkurrierenden Zugriffe auf die globalen Speicher reduziert werden können. Je nach Implementierung der Scratchpads verfügen diese über eine beziehungsweise zwei Schnittstellen, über welche ein Zugriff erfolgen kann. Mittels des ersten Interfaces greift der zugeordnete Prozessorkern direkt auf das Scratchpad zu, während die zweite Anbindung das Scratchpad mit der Crossbar verbindet. Durch diese Anbindung können die anderen Prozessorkerne ebenfalls auf diesen lokalen Speicher zugreifen. Dieser Ansatz wird in der Fachliteratur als *Dualported* bezeichnet und kommt in den meisten Mikrocontrollerarchitekturen aus Tabelle 3.1 zum Einsatz.

Cache Neben den Scratchpads fungieren die Caches als lokale Zwischenspeicher für den Programm-Code sowie für die Daten. Die Aufgabe der Caches ist dabei, sich wiederholende Speicherzugriffe auf die globalen Speicher zu beschleunigen, indem sie Kopien der benötigten Informationen lokal vorhalten. Die Verwaltung der Cache-Operationen erfolgt dabei durch den Prozessorkern selbst, weswegen die Funktionsweise für die Software vollständig transparent ist. Aufgrund der Tatsache, dass Caches SRAM als Speichertechnologie nutzen, ist ihre Speicherkapazität analog zu den Scratchpads ebenfalls stark limitiert. Im Gegensatz zu den Mikroprozessoren kommen bei echtzeitfähigen Mikrocontrollern keine mehrstufigen Cache-Hierarchien mit inklusivem oder exklusivem Aufbau zum Einsatz. Des Weiteren wird in solchen Systemen auf Cache-Kohärenzprotokolle verzichtet, da diese ein schwer vorhersagbares Laufzeitverhalten aufweisen.

Die Caches in echtzeitfähigen Mehrkernmikrocontrollern sind in sogenannten Cache-Lines organisiert, welche sowohl die lokal vorgehaltenen Informationen als auch den Status der jeweiligen Cache-Lines beinhalten. Die Größe der Cache-Line ist dabei abhängig von der Mikrocontrollerarchitektur und kann deutlich größer als die Registerbreite des Prozessorkerns sein. Beispielsweise nutzt Infineon in der AURIX-Familie eine 256 Bit breite Cache-Line, während der Prozessorkern über Register mit 32 Bit verfügt. Neben den Nutzdaten werden in den Status-Informationen zum einen der Index der Cache-Line, der Gültigkeitsstatus sowie die globale Adresse der gecachten Daten vorgehalten. Anhand des Indexes wird die Cache-Line innerhalb des Caches eindeutig identifiziert, während der Gültigkeitsstatus angibt, ob die Inhalte in der Cache-Line noch verwendet werden können. In einem Echtzeitsystem mit Caches wird bei einem Speicherzugriff auf die globalen Speicher im ersten Schritt überprüft, ob die benötigten Daten beziehungsweise der Programm-Code als lokale Kopie im Cache bereits vorliegen. Diese Überprüfung erfolgt anhand der globalen Adresse, welche zu

jeder Cache-Line im Statussegment vorgehalten wird. Sollte diese Überprüfung erfolgreich sein, wird im zweiten Schritt der Gültigkeitsstatus kontrolliert. Sobald diese Prüfung ebenfalls erfolgreich ist, kann die lokal vorgehaltene Kopie aus dem Cache genutzt werden, was als *Cache-Hit* bezeichnet wird. Sollte jedoch eine diese Überprüfungen fehlschlagen, muss direkt auf den globalen Speicher zugegriffen werden. Damit bei einem erneuten Zugriff auf die entsprechende Adresse nicht wieder auf den globalen Speicher zugegriffen werden muss, legt der Cache eine lokale Kopie für die entsprechende Adresse an. Dieses Szenario wird als *Cache-Miss* bezeichnet und resultiert in einer deutlich längeren Zugriffszeit.

Aus der Funktionsweise sowie der limitierten Speicherkapazität ergibt sich der Umstand, dass der Cache-Inhalt dynamisch zur Laufzeit umgeladen wird. Dabei ist der aktuelle Cache-Inhalt von verschiedenen Faktoren abhängig, wie dem Programmablauf, der Größe einer Cache-Line, der Verfügbarkeit von Cache-Locks sowie der Möglichkeit zur Auswahl von cache-baren Sektionen im globalen Speicher. Da die Caches immer die letzten Speicherzugriffe lokal vorhalten, hat der Programmablauf einen direkten Einfluss auf den aktuellen Inhalt. Dieser Umstand ist besonders kritisch bei asynchronen Ereignissen, wie beispielsweise Interrupts, welche ein Umladen des Caches zur Folge haben können. Sobald der Prozessorkern zur Bearbeitung der ursprünglichen Aufgabe zurückkehrt, kann es erforderlich sein, dass die benötigten Inhalte erneut aus dem globalen Speicher nachgeladen werden müssen, was einen direkten Einfluss auf die Laufzeit hat. Neben dem Programmablauf ist die Größe der Cache-Line ein wichtiger Faktor für den aktuellen Inhalt. Mittels der Größe der Cache-Line wird definiert, wie viele Informationen bei einem einzelnen Speicherzugriff auf die globalen Speicher vorgeladen werden. Gerade bei einem sequenziellen Programmablauf und entsprechender Allokation im Speicher können große Cache-Lines einen Vorteil bei der Ausführungsgeschwindigkeit bieten. Nachteilig ist dieser Ansatz, sobald der Programmablauf viele Sprunginstruktionen aufweist, wodurch der Cache permanent zum Umladen gezwungen wird. Zur Steuerung der Cache-Inhalte zur Laufzeit bieten die Mikrocontrollerarchitekturen aus Tabelle 3.1 unterschiedliche Ansätze, wie beispielsweise Cache-Locks oder die Definition von cache-baren Bereichen in den globalen Speichern. Mittels eines Cache-Locks können einzelne Cache-Lines gesperrt werden, wodurch ein Umladen effektiv verhindert wird. Im Gegensatz dazu können mit Hilfe der cache-baren Bereiche bestimmte Sektionen im Speicher von der Funktionalität des Caches ausgeschlossen werden. Bei einem Zugriff auf einen entsprechenden Speicherbereich wird der Cache direkt umgangen, wodurch das Umladen des Caches ebenfalls verhindert wird.

Der Einsatz von Caches ist in Echtzeitsystemen stark umstritten, da ihr Laufzeitverhalten aufgrund ihrer Funktionalität nur schwer vorhersagbar ist. Je nach aktuellem Cache-Inhalt kann die Zugriffszeit stark variieren, weswegen es zu Laufzeitanomalien kommt. Diese Besonderheit von Caches werden unter dem Begriff Cache-Interferenzen zusammengefasst. In Echtzeitsystemen werden drei Arten von Cache-Interferenzen unterschieden. Mittels der Intra-Task-Verdrängung werden Cache-Umladungen beschrieben, welche innerhalb der Ausführung einer Task entstehen. Da in Echtzeitbetriebssystemen häufig mehrere Task auf einem

Prozessorkern ausgeführt werden, können auch Verdrängungseffekte aufgrund eines Kontextwechsels entstehen, welche als Inter-Task-Verdrängung bezeichnet werden. Die letzte Variante stellen die Cache-Umladungen dar, welche durch asynchrone Ereignisse, wie beispielsweise Interrupts, verursacht werden. Im Gegensatz zu den Inter-Task-Verdrängungseffekten sind diese deutlich schlechter planbar. Zusätzlich zu den genannten Interferenzen existieren in der Fachliteratur noch weitere Cache-Verdrängungseffekte, welche bei mehrstufigen Cache-Hierarchien in Mehrkernsystemen auftreten. Da mehrstufige Cache-Hierarchien in echtzeitfähigen Systemen derzeit nicht vorkommen, wird auf eine weitere Betrachtung verzichtet.

In einem idealen Cache kann jede Cache-Line einen anderen Bereich des globalen Speichers lokal vorhalten, was das Maximum an Flexibilität darstellt. Der Nachteil dieses Konzepts besteht jedoch darin, dass bei einem Speicherzugriff jede Cache-Line überprüft werden muss, ob die benötigten Informationen schon lokal vorgehalten werden. Bei einer sequenziellen Prüfung wäre dies mit einem großen Zeitaufwand verbunden, weswegen ein paralleler Ansatz zwingend erforderlich ist. Dies erfordert jedoch eine Vergleichslogik für jede Cache-Line, welche die aktuell vorgehaltene Adresse mit der des Speicherzugriffs vergleicht. Da jede Vergleichslogik Chipfläche benötigt und Energie verbraucht, kommen vorwiegend Systeme mit mengenassoziativen Caches zum Einsatz [89]. Bei einem mengenassoziativen Aufbau wird der Cache in sogenannte *Pages* unterteilt, welche jeweils einen Teil des globalen Speichers vorhalten können. Durch dieses Vorgehen müssen nur so viele Vergleichslogiken vorgehalten werden, wie es *Pages* im Cache gibt.

Wie bereits beschrieben, halten Caches die letzten Zugriffe auf die globalen Speicher lokal vor, um diese bei einer erneuten Anfrage schneller zur Verfügung zu stellen. Sobald ein Cache jedoch umgeladen werden muss, ist es erforderlich, eine entsprechende Page im Cache auszuwählen. Für die Auswahl der entsprechenden Page gibt es verschiedene Konzepte, wobei sich bei den Echtzeitsystemen der First-In-First-Out (FIFO)- sowie der Least recently used (LRU)-Ansatz durchgesetzt haben. Bei der FIFO-Ersetzungsstrategie wird die Cache-Page als erstes überschrieben, welche als erstes geladen wurde. Weitere Randbedingungen, wie die Zugriffshäufigkeit oder die Dauer seit dem letzten Zugriff, werden nicht berücksichtigt. Grundlegend ist der FIFO-Ansatz für Echtzeitsysteme nur bedingt geeignet, weswegen die Hersteller der in Tabelle 3.1 dargestellten Mehrkernmikrocontroller zusätzlich einen Cache-Lock integrieren. Bei dem LRU-Ansatz wird als erstes die Cache-Page überschrieben, auf welche am längsten kein Zugriff stattgefunden hat.

In Systemen mit einer modifizierten Harvard-Architektur kommen separate Caches für den Programm-Code und die Daten zum Einsatz. Eine Besonderheit von Daten-Caches ist die Möglichkeit, dass aktualisierte Informationen in den Cache zurückgeschrieben werden können. Bevor der Cache umgeladen werden kann, müssen die aktualisierten Informationen in die globalen Speicher zurückgeschrieben werden, um Datenverlust zu vermeiden. Je nach Mikrocontrollerarchitektur kommen dabei zwei verschiedene Ansätze zum Einsatz. Bei dem

Write-Back-Verfahren werden die Daten nur im Cache aktualisiert und erst dann zurückgeschrieben, sobald die entsprechende Cache-Page umgeladen werden soll. Zu diesem Zweck wird ein entsprechendes *Dirty-Bit* im Statusbereich der Cache-Line gesetzt, welches angibt, dass die Daten im Cache aktueller sind als im globalen Speicher. Sobald die Cache-Line umgeladen werden soll, müssen als erstes die aktualisierten Daten aus dem Cache zurückgeschrieben werden, bevor die neuen Informationen geladen werden können. Der Vorteil dieses Konzepts besteht darin, dass nicht bei jedem Schreibzugriff auf den globalen Speicher gewartet werden muss. Daraus resultiert eine kürzere Zugriffszeit sowie der Umstand, dass die Anzahl der konkurrierenden Zugriffe auf die globalen Speicher reduziert werden kann. Der Nachteil dieses Ansatzes besteht darin, dass es zur Laufzeit zu inkonsistenten Daten kommen kann, da unter Umständen die Informationen im Cache aktueller als die im globalen Speicher sind. Eine Alternative stellen *Write-Through*-Caches dar, welche alle Schreibzugriffe sowohl im Cache als auch im globalen Speicher zeitgleich vornehmen [13].

3.1.5.2 Globale Speicher

Unter dem Begriff der globalen Speicher werden alle Speicher innerhalb des Mikrocontrollers zusammengefasst, welche nicht einem Prozessorkern direkt zugeordnet werden können. Dazu zählen die nicht-flüchtigen Speicher, wie Flash sowie die geteilten RAM-Speicher. Mit der Einführung von Mehrkernmikrocontrollern hat sich die interne Struktur der globalen Speicher verändert. Während es in Mikrocontrollern mit nur einem Prozessorkern häufig nur einen Flash- sowie globalen RAM-Speicher gab, wird seit der Einführung von Mehrkernmikrocontrollern der globale Speicher partitioniert, was in Abbildung 3.2 dargestellt ist. Der Vorteil dieses Vorgehens besteht darin, dass jede Partition über eine separate Anbindung an die Crossbar verfügt. Durch eine optimierte Allokation können durch diesen Aufbau konkurrierende Zugriffe deutlich minimiert werden. Nachteilig ist hingegen die komplexere Allokation, welche die Prozessorkerne zu den jeweiligen Partitionen zuordnen muss.

ROM Als nicht-flüchtiger Speicher wird in den meisten Mikrocontrollerarchitekturen Flash-Speicher verwendet. Die einzige Ausnahme werden ausgewählte Modelle der Infineon AURIX TC4xx-Serie darstellen, welche auf Resistive Random Access Memory (RRAM) setzen werden. Die globalen Read-Only Memory (ROM)-Speicher sind wie die Caches häufig in Lines organisiert, welche deutlich größer als die Registerbreite der eigentlichen Prozessorkerne sind. Wie bei den Cache-Lines soll dieser Ansatz die Ausführung von linearem Programm-Code beschleunigen. Auf Basis dieser Informationen wird deutlich, dass die Mikrocontrollerhersteller die Speicherzugriffe insgesamt deutlich breitbandiger auslegen als es die Registerbreite der Prozessorkerne erfordern. So sind beispielsweise bei der Infineon AURIX-Mikrocontrollerfamilie sowohl die Flash- als auch die Cache-Lines 256 Bit breit. Zusätzlich kann die Crossbar mit einer Breite von 64 Bit einen *Burst*-Transfer realisieren, welcher aus vier Paketen besteht und damit ebenfalls 256 Bit ergibt.

Aufgrund der geringen Zugriffsgeschwindigkeit der globalen, nicht-flüchtigen Speicher werden sogenannte *Prefetch-Buffer* integriert, welche auf Basis der Adresse des letzten Speicherzugriffs bereits die folgenden Adressen in einen kleinen SRAM-Speicher vorladen, ähnlich wie ein Caches. Dies erfolgt ohne Beteiligung des Prozessorkerns im Hintergrund und soll bei einem weiteren Zugriff auf die Folgeadresse den Speicherzugriff beschleunigen. Analog zu den bisherigen Konzepten zur Optimierung der Speicherzugriffsgeschwindigkeit profitiert auch dieser Ansatz von einer linearen Programm-Code-Ausführung.

RAM Neben den lokalen Scratchpads gibt es in Mehrkernsystemen mit einer harten Echtzeitanforderung ebenfalls globale RAM-Speicher auf welche alle Prozessorkerne mit derselben Zugriffsgeschwindigkeit lesend- als auch schreibend zugreifen können. Die globalen RAM-Speicher basieren ebenfalls auf SRAM, bieten aber nicht die schnellen Zugriffszeiten der Scratchpads, da sie keinem Prozessorkern direkt zugeordnet sind, sondern nur über die Crossbar erreichbar sind.

3.1.5.3 Externe Speicher

Als dritte Speicherhierarchiestufe fungieren die externen Speicher, welche über ein externes Speicherinterface an den Mikrocontroller angebunden werden. Je nach Mikrocontrollerfamilie variieren die Umsetzungen hinsichtlich Adress- und Datenbusbreite. Im Gegensatz zu den internen Speichern sind die externen Speicher hinsichtlich ihrer Zugriffsgeschwindigkeit deutlich limitiert, weswegen diese nur bedingt für laufzeitkritische Anwendungen geeignet sind. Des Weiteren verfügen externe Speicher nicht über die Fehlererkennungsmechanismen, wie beispielsweise Error-Correcting Code (ECC), über die die internen Speicher abgesichert sind. Aus diesem Grund sind externe Speicher ebenfalls nur bedingt für sicherheitskritische Anwendungen geeignet. Die Nutzung von externen Speichern ist bei echtzeitfähigen Steuergeräten aufgrund der beschriebenen Limitierungen stark eingeschränkt. Verwendung findet diese dritte Speicherhierarchie häufig für die Ablage von *Freeze Frames* zur Analyse von komplexen Fehlerbildern im RAM oder für die Zwischenspeicherung von kompletten Software-Updates im externen ROM.

ROM Über das externe Speicherinterface kann zusätzlicher nicht-flüchtiger Speicher angebunden werden. Im Gegensatz zu dem internen Speicherkonzept kann die Größe der Flash-Line jedoch deutlich kleiner sein, was einen direkten Einfluss auf die Ausführungsgeschwindigkeit von linearem Programm-Code hat. Des Weiteren verfügen externe ROM-Speicher nicht über einen *Prefetch-Buffer*, was ebenfalls zur reduzierten Zugriffsgeschwindigkeit beiträgt.

RAM Analog zum externen ROM-Speicher kann über das entsprechende Speicherinterface ebenfalls zusätzlicher RAM-Speicher angebunden werden. Bedingt durch die verringerte Bandbreite aufgrund der reduzierten Datenbusbreite ist dieser externe RAM-Speicher ebenfalls ungeeignet für laufzeitkritische Anwendungen.

3.1.5.4 Speicherschutz

Ein elementarer Bestandteil in sicherheitskritischen Systemen ist der Speicherschutz, welcher alle Funktionalitäten zur Sicherstellung der Speicherintegrität, der Speicherkonsistenz sowie des Zugriffsschutzes umfasst. Zur Umsetzung dieser Anforderungen verfügen die im Rahmen dieser Arbeit betrachteten Mehrkernmikrocontroller über eine integrierte Erkennung von Speicherfehlern, eine mehrstufige MPU zur Sicherstellung der Zugriffsbeschränkungen sowie die Möglichkeit von atomaren Zugriffen, welche für die Realisierung von Synchronisierungsmechanismen zur Absicherung der Speicherkonsistenz benötigt werden.

Fehlererkennung Für die korrekte Funktionsweise von echtzeitfähigen Mehrkernsystemen ist es essentiell, dass Speicherfehler zuverlässig erkannt und korrigiert werden können. Zu diesem Zweck integrieren die Hersteller solcher Systeme eine Fehlererkennung auf Basis von ECC für alle internen Speicher sowie NoCs.

Zugriffsschutz Zur Sicherstellung der Isolation von Software unterschiedlicher Kritikalität werden in modernen Mehrkernmikrocontrollern für sicherheitskritische Anwendungen mehrstufige Zugriffsschutzhierarchien genutzt. Die erste Stufe stellt die sogenannte System-MPU dar, welche für jeden internen Speicher im System zur Verfügung steht. Die System-MPU entscheidet anhand der Identifikationsnummer, welche jeder Kommunikationsteilnehmer am NoC hat, ob ein Zugriff auf einen Speicherbereich erlaubt ist. Dabei können moderne System-MPUs zwischen verschiedenen Zugriffsarten, wie beispielsweise lesend oder schreibend, unterscheiden. Mit Hilfe der System-MPU können bestimmte Prozessorkerne oder Hardware-Beschleuniger mit Speicherzugriff von kritischen Speicherbereichen ausgeschlossen werden. Wichtig ist hierbei zu beachten, dass die verschiedenen Speicher mitunter unterschiedliche Speicherbereichsgranularitäten unterstützen, welche separiert werden können. Bei der Allokation ist also darauf zu achten, dass kritische Funktionalitäten in einem zusammenhängenden Speicherbereich liegen, welche separat schützbar sind. Neben der Sytem-MPU existiert zusätzlich eine Prozessorkern-MPU als zweite Stufe, welche es ermöglicht, dass Funktionalitäten unterschiedlicher Kritikalität auf einem Prozessorkern ausgeführt werden können. Zu diesem Zweck besitzen die Prozessorkerne verschiedene Ausführungslevel, welche jeweils eine separate Konfiguration der Prozessorkern-MPU besitzen. So kann beispielsweise eine Funktionalität mit einem niedrigeren Ausführungslevel nicht auf die Speicherbereiche der anderen Ausführungslevel zugreifen. Durch diese beiden MPU-Stufen können die Speicherbereiche von Funktionalitäten unterschiedlicher Kritikalität sowohl systemweit als auch innerhalb eines Prozessorkerns effektiv voneinander isoliert werden. Dies ist erforderlich, um die Anforderungen der ISO 26262 hinsichtlich der Safety sowie der ISO 21434 in Bezug auf die Security im automobilen Umfeld zu erfüllen [90] [38] [91].

Speicherverwaltung Wie in dem Abschnitt Prozessorkern bereits beschrieben, unterstützen die neuen Mikrocontrollergenerationen die Virtualisierung der Rechenzeit in Hardware. Das gilt jedoch nicht für die Speicherzugriffe, weswegen es derzeit keine Memory Management Units (MMUs) in echtzeitfähigen Systemen gibt. Dieser Umstand ist darauf zurückzuführen, dass MMUs für Echtzeitsyste-

me aufgrund ihrer variierenden Laufzeit ungeeignet sind. Die Ursache für diese variable Laufzeit besteht darin, dass für die Virtualisierung der Speicherzugriffe immer eine Umrechnung von der virtuellen in die physikalische Speicheradresse erforderlich ist. Um diese Umrechnung zu beschleunigen, nutzen moderne Mikroprozessoren einen sogenannten Translation Lookaside Buffer (TLB), welcher die letzten Adressumrechnungen wie ein Cache zwischenspeichert. Je nach aktuellem Inhalt des TLB kann die Zugriffszeit auf den Speicher stark variieren. Da in einem Echtzeitsystem immer die WCET die Basis für alle Betrachtungen bildet, muss bei jedem Speicherzugriff davon ausgegangen werden, dass der TLB die entsprechende Umrechnung nicht vorhält. Die Folge wäre eine erhöhte Zugriffszeit durch die zusätzliche Adressumrechnung, was bei Mikrocontrollern aufgrund ihrer deutlich reduzierten Rechenleistung im Vergleich zu Mikroprozessoren nicht vertretbar ist.

3.2 Software

Der grundlegende Aufbau von Software in echtzeitfähigen Steuergeräten für sicherheitskritische Anwendungen im automobilen Umfeld ist in Abbildung 3.6 dargestellt. Auf einigen Prozessorkernen läuft ein Hypervisor, welcher eine unterschiedliche Anzahl an virtuellen Maschinen ausführt [92] [93]. Jede virtuelle Maschine verfügt wiederum über ein separates Betriebssystem, welches die Zeitscheiben zyklisch ausführt [94]. Die Anzahl der Tasks ist dabei abhängig von der jeweiligen Funktionalität sowie deren Kritikalität [95]. Neben den Prozessorkernen mit Hypervisor gibt es auch Kerne, welche ohne Hypervisor arbeiten und das Operating System (OS) direkt ausführen. Wichtig ist hierbei zu beachten, dass jeder Prozessorkern seine separate Instanz des Betriebssystems ausführt [96]. Ein kernübergreifendes Scheduling inklusive Task-Migration, wie es beispielsweise bei Mikroprozessoren genutzt wird, ist in Echtzeitsystemen nicht verbreitet. Die Prozessorkerne sollen möglichst unabhängig funktionieren, damit der Ausfall eines Kerns nicht zu einem kompletten Systemversagen führt. Als letzte Option existieren noch sogenannte *Bare-Metal*-Umsetzungen, welche ohne zusätzliche Abstraktion durch einen Hypervisor oder ein Betriebssystem direkt auf der Hardware ausgeführt werden [97]. In automobilen Systemen ist AUTOSAR der vorherrschende Standard für die Software-Architektur. Neben der Variante für Echtzeitsysteme mit dem Namen AUTOSAR Classic gibt es seit 2017 noch eine Version für Mikroprozessoren mit der Bezeichnung AUTOSAR Adaptive [98] [99]. AUTOSAR Adaptive setzt auf ein Portable Operating System Interface (POSIX)-kompatibles Betriebssystem mit einem service-orientierten Ansatz und wird vorwiegend in Steuergeräten eingesetzt, welche Funktionalitäten ohne eine harte Echtzeitanforderung bereitstellen [100] [37] [101]. Im Gegensatz dazu setzt AUTOSAR Classic auf ein Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK)-konformes Echtzeitbetriebssystem und wird vorwiegend in sicherheitskritischen Anwendungen mit einer harten Echtzeitanforderung eingesetzt. Da der Fokus dieser Arbeit auf echtzeitfähigen Systemen liegt, wird in der weiteren Betrachtung AUTOSAR Adaptive nicht berücksichtigt.

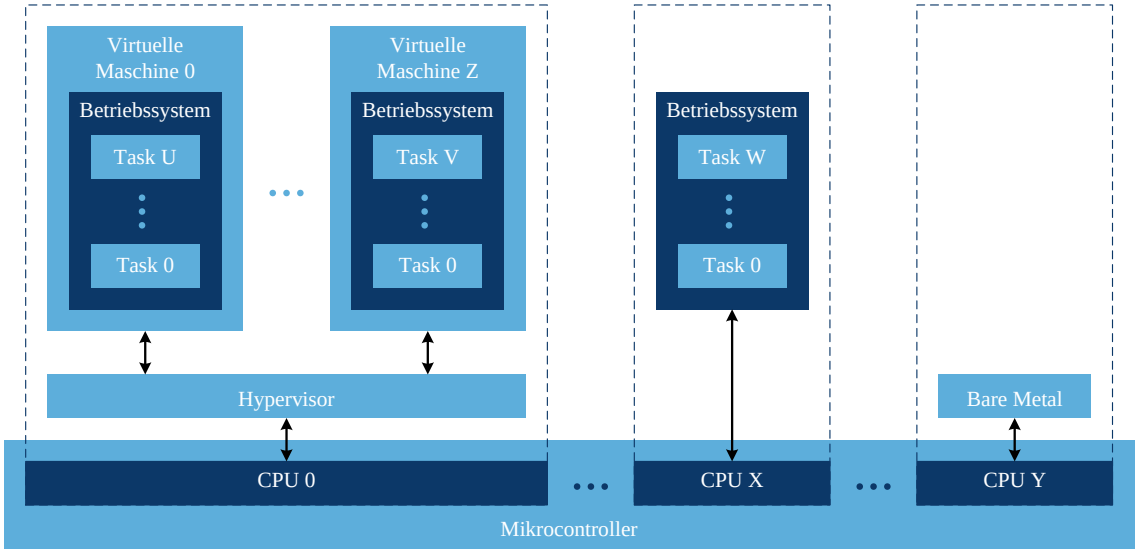

Abbildung 3.6: Grundlegende Systemarchitektur eines sicherheitskritischen Systems mit einer harten Echtzeitanforderung auf Basis eines eingebetteten Mehrkernprozessors

Der Standard AUTOSAR Classic ist in heutigen automobilen Steuergeräten allgegenwärtig und wird herstellerübergreifend für eine Vielzahl von sicherheitskritischen Systemen, wie die Steuerung des Motors, die Realisierung der Bremse oder für die Sicherstellung der Fahrdynamik in Form von Antiblockiersystem (ABS) und ESP, eingesetzt. Die Entwicklung des AUTOSAR-Standards hat bereits 2003 begonnen und wird bis heute kontinuierlich fortgesetzt. Im Lauf der Jahre wurden immer mehr Funktionalitäten, wie beispielsweise das Thema Security, breitbandige Kommunikationsschnittstellen sowie die Unterstützung für Mehrkernmikrocontroller, hinzugefügt.

Grundlegend unterteilt AUTOSAR Classic die Software horizontal nach Abstraktionslevel und vertikal nach Funktionalität, was in Abbildung 3.7 dargestellt ist. Zu den horizontalen Abstraktionsschichten gehören der *Microcontroller Abstraction Layer (MCAL)*, der *ECU Abstraction Layer*, der *Service Layer*, die *Runtime Environment (RTE)* sowie der *Application Layer*. Mittels des *MCAL* werden die Funktionalitäten des genutzten Mikrocontrollers für die darüberliegenden hardwareunabhängigen Schichten über standardisierte Schnittstellen bereitgestellt [102]. Im Anschluss erfolgt die Abstraktion der genutzten ECU über den *ECU Abstraction Layer*. Durch diese Schicht werden die weiteren Bauteile der ECU außerhalb des Mikrocontrollers für die darüberliegenden Software-Komponenten bereitgestellt. Zu diesen externen Bauteilen gehören beispielsweise die Transceiver für Kommunikationsschnittstellen, die Halbbrücken-Treiber zur Ansteuerung von Leistungshalbleitern [103] [104] oder System Basis Chips (SBCs) [105] [106]. In dem *Service Layer* werden alle Funktionalitäten der ECU sowie des Mikrocontrollers als Dienst für die Applikationen aufbereitet. Zu dem *Service Layer* gehören auch wichtige Systemfunktionen, wie das Betriebssystem oder der Startup-Code, welcher die Grundinitialisie-

rung des Mikrocontrollers vornimmt. Mittels der *RTE* werden die Funktionen der Basis-Software als Signale der darüberliegenden Applikations-Software zur Verfügung gestellt. Das Ziel der *RTE* ist dabei die Basis-Software so weit zu abstrahieren, dass Applikationen ohne große Probleme auf verschiedene Steuergeräteplattformen portiert werden können. In der obersten Schicht ist die Applikations-Software vorgesehen, welche die eigentliche Funktionalität des Steuergeräts darstellt. Zusätzlich zu den genannten Schichten existieren noch die *Complex Device Drivers (CDDs)*, über welche Spezialfunktionalitäten abgebildet werden können, welche im AUTOSAR-Standard nicht definiert sind [107].

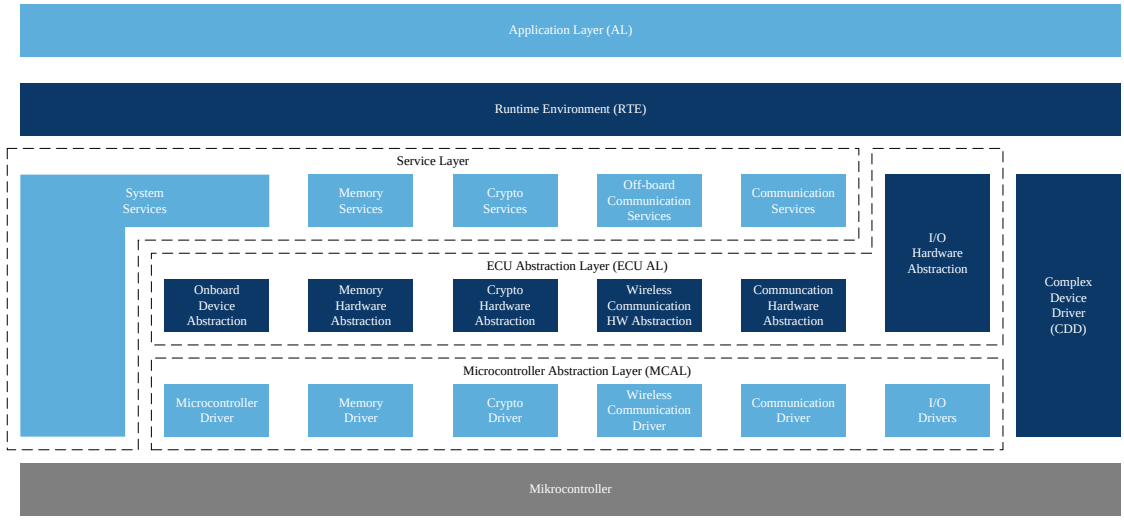

Abbildung 3.7: Grundlegende Software-Architektur von AUTOSAR Classic R22-11 [107]

Die vertikale Unterteilung erfolgt anhand der funktionalen Eigenschaften, welche im AUTOSAR Classic die *System Services*, die *Memory Services*, die *Crypto Services*, die *Off-board Communication Services*, die *Communication Services* sowie die *I/O Services* darstellen. Durch die *System Services* werden die Systemfunktionalitäten, wie das Betriebssystem, aber auch diverse Safety-Funktionen, wie beispielsweise der Speicherschutz sowie die Watchdog- und *Lockstep*-Verwaltung, realisiert. Mittels der *Memory Services* erfolgt die Administration der nichtflüchtigen Speicher, wie beispielsweise dem Daten-Flash. Aufgrund der steigenden Anforderungen hinsichtlich der Absicherung von Steuergeräten und deren Kommunikation gegen unbefugten Zugriff erfolgte die Integration von kryptografischen Diensten sowie der dazugehörigen Schlüsselverwaltung in den AUTOSAR-Standard [108] [109] [110]. Für die Umsetzung von diversen Funkstandards, wie Wireless Local Area Network (WLAN) oder Bluetooth, gibt es die *Off-board Communication Services*. Analog zu den kabellosen Kommunikationsschnittstellen existieren im AUTOSAR-Standard ebenfalls Services für die Bereitstellung des kabelgebundenen Informationsaustausches in Form von Controller Area Network (CAN), FlexRay, Ethernet, Local Interconnect Network (LIN) und Serial Peripheral Interface (SPI) [111] [11] [112]. In der *I/O Hardware Abstraction* werden alle digitalen und analogen Schnittstellen, wie General Purpose

Input/Outputs (GPIOs) oder Analog-to-Digital Converter (ADC), gebündelt und als aufbereitete Signale der *RTE* zur Verfügung gestellt [107].

Die Ausführung der Applikationssoftware sowie der Basissoftware erfolgt zeitscheibenbasiert durch das Betriebssystem [113]. Zu diesem Zweck werden die Tasks in Abhängigkeit ihrer Aufrufhäufigkeit innerhalb einer Hyperperiode priorisiert, wobei die Zeitscheiben mit der höchsten Aufrufanzahl die höchste Priorität erhalten, was als RMS bezeichnet wird. Die Anzahl und Priorität der Tasks sind bereits während des Kompilierens bekannt und zur Laufzeit statisch. Die Isolierung von Applikationen unterschiedlicher Kritikalität erfolgt durch die Nutzung von separaten Partitionen, in welcher die Zeitscheiben ausgeführt werden [114]. Je nach Anzahl der Partitionen und der benötigten Tasks kann es durchaus vorkommen, dass es mehrere Zeitscheiben mit einem identischen Zeitraster gibt. Mittels des Betriebssystems erfolgt dann die Umschaltung des Zugriffsschutzes während des Kontextwechsels. Die Task mit der niedrigsten Priorität und Kritikalität stellt die Hintergrund-Task dar, welche immer dann ausgeführt wird, wenn keine andere Zeitscheibe die Rechenzeit benötigt. Je nach Auslastung des Prozessorkerns kann die Aufrufhäufigkeit der Hintergrund-Task stark variieren. Neben den zeitsynchronen Task gibt es auch asynchrone Events, welche durch Interrupts repräsentiert werden. Diese schwer vorhersagbaren Ereignisse müssen bei der Bewertung der Rechenauslastung ebenfalls mitberücksichtigt werden [115]. Zur einfacheren Integration in bestehende Ansätze zur Scheduling-Analyse werden die asynchronen Events auf Basis ihrer maximalen Auftrittswahrscheinlichkeit wie zeitsynchrone Ereignisse behandelt.

Die Bereitstellung der Funktionalität erfolgt wie bereits beschrieben durch den *Application Layer*, welcher durch die *Software Components (SWCs)* repräsentiert wird. Eine *SWC* stellt dabei eine bestimmte Funktion dar und besteht wiederum aus *Runnables*, welche die kleinste schedul-bare Einheit darstellen [116]. Eine *SWC* kann also durch ihre *Runnables* auf mehrere Zeitscheiben verteilt sein, wobei die Kritikalität durch die *SWC* definiert wird [117]. Daraus folgt, dass alle *Runnables* einer *SWC* ebenfalls diese Kritikalität besitzen und daher in derselben Partition verortet werden müssen. Analog zu den Zeitscheiben erfolgt die Zuweisung der *SWCs* beziehungsweise *Runnables* zur Kompilierzeit und ist daher zur Laufzeit ebenfalls statisch.

3.3 Speicherverwaltung

In dem folgenden Abschnitt werden verschiedene Ansätze zur Speicherverwaltung vorgestellt, welche den aktuellen Stand der Technik repräsentieren. Die Speicherverwaltung in einem echtzeitfähigen Mehrkernmikrocontroller ist essentiell für die Erreichung wichtiger Ziele, wie beispielsweise die temporale und räumliche Isolation von Software-Komponenten unterschiedlicher Kritikalität, die Performance sowie die geforderte Reaktionszeit [118] [1]. Dabei stellt besonders die zeitliche Isolierung eine Herausforderung in Mehrkernsystemen dar. Dies ist darauf zurückzuführen, dass in sicherheitskritischen Echtzeitsystemen die Prozessorkerne möglichst autark arbeiten sollen, damit der Ausfall eines Kerns nicht zum Versagen des Gesamtsystems führt

[119]. Aus diesem Grund arbeiten die Prozessorkerne unabhängig und häufig ohne Synchronisation, wodurch die Speicherzugriffe nicht aufeinander abgestimmt werden können [120]. Daraus können gleichzeitige Zugriffe auf geteilte Speicher resultieren, wodurch es zu Verzögerungen im Ablauf der Programmausführung auf den betroffenen Kernen kommt. Diese Effekte müssen bei der Bewertung der Echtzeitfähigkeit berücksichtigt werden und können daher zu einer pessimistischen WCET führen, was die nutzbare Rechenzeit deutlich einschränkt [121].

Daher werden in diesem Abschnitt die derzeit bekannten Ansätze zum Speichermanagement vorgestellt und miteinander verglichen. Grundsätzlich lassen sich alle Konzepte in die drei Kategorien der statischen- und dynamischen Speicherverwaltung sowie der Intercore-Kommunikation einteilen. Je nach Umsetzung erfolgt noch eine Separierung in Ansätze, welche eine Hardware-Unterstützung voraussetzen sowie rein software-basierte Verfahren.

3.3.1 Statische Speicherverwaltung

In der ersten Kategorie werden alle Speicherverwaltungsverfahren beschrieben, welche einen statischen Ansatz verfolgen. Bei einem statischen Konzept wird die Allokation des Programm-Codes sowie der Daten zur Kompilierzeit vorgenommen, weswegen das Speicherlayout zur Laufzeit konstant ist [122]. Der Vorteil dieses Ansatzes besteht darin, dass eine deutlich einfachere Betrachtung der WCET möglich ist, da zur Laufzeit der Inhalt eines jeden Speichers bekannt ist. Nachteilig ist hingegen die suboptimale Nutzung der lokalen Speicher im System. Bedingt durch die statische Allokation können die Inhalte dieser Speicher zur Laufzeit nicht an den Programmablauf angepasst werden, was eine suboptimale Nutzung der geringen Speicherkapazität zur Folge hat [9].

3.3.1.1 Software-basiert

Mit der grundsätzlichen Geschwindigkeit von Speicherhierarchien in echtzeitfähigen Mehrkernmikrocontrollern für sicherheitskritische Anwendungen und den Effekten von konkurrierenden Zugriffen beschäftigen sich die Autoren in [2], [4] und [84]. Dabei wird ersichtlich, dass lediglich bei der Ausführung aus den Scratchpads bei exklusiver Nutzung durch den jeweiligen Prozessorkern die maximal mögliche Performance erreicht werden kann. Nur, wenn sowohl der Programm-Code als auch die dazugehörigen Daten lokal vorgehalten werden, können die superskalaren Architekturen moderner Mikrocontroller voll ausgelastet werden.

Prinzipiell beschäftigen sich die Artikel zur Optimierung der statischen Speicherallokation mit der effizienten Nutzung der lokalen Speicher in Form der Scratchpads sowie der Caches. Während es bei den Scratchpads primär um die optimierte Auswahl der Speicherinhalte geht, versuchen die Arbeiten zu den Caches die Interferenzen zur Laufzeit zu minimieren. Zur Bestimmung der Auswahl der Speicherinhalte haben sich in der Fachliteratur zwei Ansätze durchgesetzt, welche entweder auf Basis der Aufrufhäufigkeit oder auf Basis des Worst-Case Execution Path (WCEP) die Allokation vornehmen. In [88] nutzen die Autoren zur Bestimmung der Aufrufhäu-

figkeit einen Graphen, mit dessen Hilfe während der Kompilierung eine Abschätzung über die Menge der Zugriffe vorgenommen wird. Anhand dieser Informationen erfolgt im Anschluss eine Aufteilung des Stacks und der globalen Variablen auf das interne Scratchpad sowie auf den externen Dynamic Random Access Memory (DRAM). Das Ziel ist dabei, möglichst viele Zugriffe über die schnellen Scratchpads abzudecken. In [10], [11] und [6] bestimmen die Autoren die Zugriffshäufigkeit mittels eines *Trace* und nutzen die Erkenntnis zur Verteilung des Programm-Codes sowie der Daten auf die schnellsten Speicher im System. Dabei fokussieren sich die Autoren in [10] auf besonders laufzeitintensive *Basic Blocks*, um die geringe Speicherkapazität der Scratchpads möglichst effizient zu nutzen. Problematisch an diesem Vorgehen ist der mangelnde Support durch den genutzten Compiler, weswegen die Autoren die Adresse im kompilierten Code manipulieren müssen. Im Gegensatz dazu wird in [11] die Verarbeitung von Ethernet-Nachrichten auf einem echtzeitfähigen Mehrkernmikrocontroller verbessert, indem gezielt die laufzeitintensiven Funktionen sowie die dazugehörigen Daten in die lokalen Speicher des entsprechenden Prozessorkerns allokiert werden. Durch das gezeigte Vorgehen können die Autoren die Übertragungsrate der Ethernet-Kommunikation signifikant steigern und die Laufzeitschwankungen nachweislich reduzieren. Jedoch ist dabei zu beachten, dass alle anderen Funktionalitäten auf dem Steuergerät von den gezeigten Optimierungen nicht profitieren. In [6] zeigen die Autoren einen Ansatz zur Reduzierung der Interferenzen durch konkurrierende Zugriffe auf geteilte Speicher, indem der Programm-Code sowie die Daten anhand verschiedener Faktoren, wie der Aufrufhäufigkeit, der Laufzeit oder der Anzahl der darauf zugreifenden Kerne, priorisiert werden. Auf Basis dieser Informationen erfolgt im Anschluss die Auswahl des Codes sowie der Daten, welche als lokale Kopien in den lokalen Speicher vorgehalten werden. Dabei beschränken sich die Autoren auf Systeme, welche auf jedem Prozessorkern eine separate Instanz eines Echtzeitbetriebssystems mit einem Task-basierten Scheduling ausführen. Alternative Ansätze, wie Hypervisor oder *Bare-Metal* Applikationen, werden nicht weiter berücksichtigt. Des Weiteren werden speicherrelevante Beschleuniger, wie beispielsweise Co-Prozessoren oder DMA-Controller, ebenfalls nicht in die Betrachtung mit einbezogen.

Zu den Konzepten, welche den WCEP als Basis für die optimierte Allokation nutzen, zählen die Arbeiten [123], [124], [125], [126], [127] und [35]. In [123] und [124] nutzen die Autoren einen Kontrollflussgraphen zur Bestimmung des WCEP, mit dessen Hilfe im Anschluss die Daten für das lokale Daten-Scratchpad des Prozessorkerns bestimmt werden. Bedingt durch den Einsatz der ganzzahligen linearen Optimierung zur Detektion des WCEP innerhalb des Graphen benötigt die Suche mit steigender Komplexität der zu analysierenden Code-Basis unverhältnismäßig viel Zeit, weswegen die Autoren den Einsatz eines heuristischen Ansatzes empfehlen. Im Gegensatz dazu präferieren die Autoren in [125] ein Vorgehen, in welchem zur Kompilierzeit Speicherblöcke ermittelt werden, welche dann zur Laufzeit, je nach Bedarf, nachgeladen werden. Die Auswahl der Speicherblöcke erfolgt dabei wieder über einen Kontrollflussgraphen, jedoch weisen die Autoren ebenfalls auf den hohen Bedarf an Rechenleistung hin und nutzen daher bei großen Software-Projekten einen heuristischen Ansatz. Die Autoren in [126] und [127] nutzen analog zu den bishe-

rigen Fachartikeln einen Kontrollflussgraphen, jedoch für die optimierte Allokation des Programm-Scratchpads. Die Besonderheit in [127] stellt die Reduzierung des Analyseaufwands des Kontrollflussgraphen durch eine Vereinfachung mittels eines gerichteten azyklischen Teilgraphen dar, weswegen hier kein heuristischer Ansatz genutzt wird.

Ein kombiniertes Vorgehen für Scratchpads und Caches beschreiben die Autoren in [35]. Zu diesem Zweck erfolgt die Analyse des Programm-Codes zur Kompilierzeit ebenfalls mittels Kontrollflussgraphen. Dabei werden die Inhalte für die Scratchpads so ausgewählt, dass diese zur Laufzeit statisch vorgehalten werden. Im Gegensatz dazu kann der Inhalt der Caches dynamisch während der Programmausführung variieren. Damit die Vorhersagbarkeit der Caches erhöht wird, erfolgt während des Kompilierens die Auswahl von Vorladepunkten, an denen die Cache-Inhalte geändert werden können. Zwischen diesen Vorladepunkten werden die Caches mit einem *Lock* vor dem Überschreiben geschützt, wodurch die Vorhersagbarkeit signifikant erhöht wird. Einen ähnlichen Ansatz verfolgen die Autoren in [128], jedoch mit dem Ziel zur Verringerung von Cache-Interferenzen. In [14] wiederum erfolgt das Vorladen an bestimmten Punkten in die Scratchpads mittels DMA-Controller, wodurch der Prozessorkern bei der Ausführung nicht unterbrochen werden soll. Aus diesem Grund unterteilen die Autoren die Scratchpads in zwei Regionen, wobei eine Region für die aktuelle Ausführung und der andere Bereich zum Vorladen durch den DMA-Controller genutzt werden soll. Nachteilig an den genannten Ansätzen ist die Erhöhung der Reaktionszeit, da die Speicherinhalte der lokalen Speicher nur an bestimmten Vorladepunkten geändert werden können. Je nach aktuellem Zustand des Systems kann dadurch bei zeitkritischen Anwendungen die geforderte Antwortzeit nicht eingehalten werden, weswegen im Rahmen dieser Arbeit auf Konzepte mit einer Vorladetechnik verzichtet wird.

3.3.1.2 Hardware-basiert

Neben den software-basierten Ansätzen zur optimierten statischen Speicherallokation existieren zusätzlich noch Konzepte, welche einen dedizierten Hardware-Support voraussetzen, die im folgenden Abschnitt beschrieben werden.

Einen Ansatz zur Vereinfachung des Speicherschutzes bei der statischen Speicherallokation beschreiben die Autoren in [23]. Dort wird eine Kombination aus MPU und MMU vorgeschlagen, welche zentral alle Speicheranfragen validiert. Der Vorteil liegt dabei im reduzierten Validierungsaufwand, da die komplette Konfiguration des Speicherschutzes zentral erfolgt. Im Gegensatz dazu beschreiben die Autoren in [29] eine Arbitrierungseinheit für statische Speicherzugriffe in Echtzeitsystemen, welche zwischen Anwendungen mit einer harten Echtzeitanforderung und ohne unterscheiden kann. Zusätzlich können die Anwendungen mit einer strikten Echtzeitanforderung systemweit priorisiert werden, wodurch die Laufzeitinterferenzen bei konkurrierenden Zugriffen von unterschiedlichen Prozessorkernen auf geteilte Ressourcen weiter reduziert werden sollen. Derzeit gibt es keinen Mehrkernmikrocontroller für sicherheitskritische Anwendungen im automobilen Umfeld, welcher einen entsprechenden Speicherschutz oder eine passende Arbitrierungseinheit zur Verfü-

gung stellt. Aus diesem Grund wird von einer weiteren Betrachtung im Rahmen dieser Arbeit abgesehen.

3.3.2 Dynamische Speicherverwaltung

Zu den Verfahren der dynamischen Speicherallokation zählen alle Ansätze, welche zur Laufzeit die Verortung des Programm-Codes sowie der Daten anpassen können. Der Vorteil der dynamischen Konzepte besteht darin, dass die Speicherinhalte in Abhängigkeit des Programmablaufs angepasst werden können. Dadurch kann die Reaktionszeit des Systems sinken und die Ausführungsgeschwindigkeit signifikant steigen [14]. Nachteilig ist hingegen die gestiegene Komplexität bei der Verifikation, ob die WCET in allen Fällen eingehalten werden kann. Aus diesem Grund gibt es in einigen Bereichen der Industrie Standards und Normen, welche den Einsatz von dynamischer Speicherallokation kategorisch untersagen [38] [129]. Daher werden die dynamischen Ansätze im Rahmen dieser Arbeit nicht weiterverfolgt. Der folgende Abschnitt dient lediglich der Vorstellung alternativer Speicherverwaltungsmodelle für eine umfassende Übersicht über den aktuellen Stand der Technik.

3.3.2.1 Software-basiert

Die software-basierten Konzepte zur dynamischen Speicherverwaltung können sich grundsätzlich auf Basis der genutzten Speicher gruppieren lassen. Aus diesem Grund werden im ersten Abschnitt die Fachartikel vorgestellt, welche eine dynamische Nutzung der Scratchpads vorsehen. Im Anschluss erfolgt die Beschreibung der Arbeiten, welche durch verschiedene Ansätze das Laufzeitverhalten der Caches vorhersagbarer und effizienter gestalten wollen. Im letzten Abschnitt werden die Verfahren dargelegt, welche mit Hilfe eines optimierten Speicherzugriffsalgorithmus die Speicherinterferenzen reduzieren.

Die Autoren in [130], [131], [132], [133] und [134] beschreiben Ansätze, welche eine dynamische Speicherverwaltung für die Scratchpads vorschlagen. In [130] werden mit Hilfe eines Kontrollflussgraphen die nachladbaren Einheiten bestimmt, welche zur Laufzeit dynamisch in das Scratchpad geladen werden. Das Ziel der Autoren ist dabei primär die Reduzierung des Energieverbrauchs, was auch für das Konzept in [131] gilt. In [131] werden ebenfalls mit Hilfe eines Graphen Elemente gesucht, welche sich zur Laufzeit dynamisch vorladen lassen. Zu diesem Zweck werden mittels des Compilers zusätzliche Instruktionen vor den entsprechenden Block eingefügt, welche das Vorladen veranlassen. Im Gegensatz dazu versuchen die Autoren in [132] mit ihrem Vorgehen die WCET zu verbessern. Dies erfolgt mittels eines graphenbasierten Ansatzes, bei welchem die *Basic Blocks* detektiert werden, die auf dem WCEP liegen. Während der eigentlichen Programmausführung erfolgt das Vorladen der detektierten *Basic Blocks* in die lokalen Scratchpads. In [133] wird ein Konzept für Mikrocontroller beschrieben, in denen die Prozessorkerne nur auf ihre lokalen Speicher direkt zugreifen können. Für den Zugriff auf die globalen Speicher wird ein separater DMA-Controller genutzt, welcher die benötigten Informationen in die lokalen Speicher vorlädt. Das Ziel der Autoren ist daher das effektive Zusammenfassen von

laufzeitintensiven Funktionen, damit die DMA-Transfers möglichst effizient realisiert werden können. Die Autoren in [134] erweitern ein echtzeitfähiges Betriebssystem dahingehend, dass vor der eigentlichen Task-Ausführung der komplette Inhalt in die lokalen Scratchpads vorgeladen wird, so dass während der eigentlichen Ausführung kein Zugriff auf die globalen Speicher erforderlich ist. Dabei ist jedoch zu beachten, dass die maximale Task-Größe durch die Speicherkapazität des Scratchpads limitiert wird. Einen Ansatz zur optimierten dynamischen Nutzung von Caches in Echtzeitbetriebssystemen wird in [135] beschrieben. Die Grundidee besteht darin, durch das gezielte Setzen von Unterbrechungspunkten das Scheduling des Betriebssystems dahingehend zu optimieren, dass ein Umladen der Caches nur zu definierten Zeitpunkten erfolgt. Das Ziel ist dabei, die Vereinfachung der Laufzeitanalyse aufgrund der reduzierten Cache-Interferenzen. In [136] beschreiben die Autoren ein Verfahren, welches mit Hilfe von Cache-Coloring und gezielten Cache-Locks ebenfalls die Vorhersagbarkeit von Caches erhöhen soll. Zu diesem Zweck werden mittels Cache-Coloring zeitkritische Aufgaben in zusammenhängende Cachelines zusammengefasst und während der eigentlichen Ausführung gesperrt, wodurch ein Umladen effektiv verhindert wird. Ein ähnlicher Ansatz wird in [137] beschrieben, jedoch nutzen die Autoren den Kontextwechsel des Betriebssystems für das Vorladen der Caches. Mittels Cache-Coloring und Cache-Locks wird auch in diesem Falle die Räumlichkeit verbessert und ein Umladen zur Laufzeit effektiv verhindert. Eine Besonderheit stellt das Konzept in [138] dar, da es einen Mehrkernmikrocontroller voraussetzt, welcher über ein Cache-Kohärenzprotokoll verfügt. Zur Minimierung der Laufzeitinterferenzen durch das Kohärenzprotokoll schlagen die Autoren eine Cache-Partitionierung vor, welche mit Hilfe von Hardware-Zählern sich gegenseitig behindernde Tasks detektiert und darauf entsprechend reagiert. Zur Minimierung der Interferenzen von konkurrierenden Zugriffen können ebenfalls Algorithmen zur Drosselung des Speicherzugriffs genutzt werden. Aus diesem Grund beschreiben die Autoren in [30] ein Verfahren, welches Speicherinterferenzen detektiert und im Anschluss zugunsten der laufzeitkritischen Task eine Priorisierung vornimmt. Ein ähnliches Konzept beschreiben die Autoren in [139], jedoch mit dem Fokus auf Fairness und dem Speicherdurchsatz. Zu diesem Zweck werden alle Threads in zwei Kategorien eingeteilt, wobei zwischen latenz-empfindlichen und durchsatz-empfindlichen Threads unterschieden wird. Auf Basis dieser Informationen bildet ein Speicherzugriffsalgorithmus eine optimierte Zugriffszuteilung, welche Fairness und Speicherdurchsatz für die Threads verbessern soll.

3.3.2.2 Hardware-basiert

In dem folgenden Abschnitt sind dynamische Speicherkonzepte beschrieben, welche eine dedizierte Hardware-Unterstützung voraussetzen. Eine spezielle Funktion einiger ARM-Prozessoren stellt der sogenannte Auto-Lock für die Caches dar. Mittels eines hardware-internen Algorithmus bestimmt der Cache-Controller des Prozessorkerns besonders relevante Inhalte in den Cache-Lines und sperrt diese gegen ein automatisches Überschreiben durch beispielsweise einen Kontextwechsel. Der Vorteil dieses Ansatzes besteht in der völligen Transparenz gegenüber der auszuführenden

Software. Nachteilig ist hingegen die schlechte Vorhersagbarkeit und das Sicherheitsrisiko bei Seitenkanalangriffen auf den Cache [140]. In [89] beschreiben die Autoren einen Ansatz zur optimierten Nutzung der Scratchpads, in dem oft genutzte *Basic Blocks* dynamisch zur Laufzeit mittels eines DMA-Controllers vorgeladen werden. Die Adressübersetzung erfolgt dabei mittels einer MMU, welche eine Voraussetzung für dieses Konzept darstellt. Im Gegensatz dazu schlagen die Autoren in [141] ein Scratchpad vor, welches vollständig von der Hardware verwaltet wird. Der Vorteil gegenüber eines Caches soll dabei in der Granularität auf Basis der implementierten Funktionen liegen. Zur Reduzierung von Speicherzugriffsinterferenzen aufgrund von Engpässen in der verfügbaren Bandbreite wird in [142] ein intelligenter Speichercontroller diskutiert, welcher automatisch auf Basis des aktuellen Lastprofils die Bandbreite auf die globalen Speicher einschränkt. Das Ziel ist dabei, die verfügbare Bandbreite so auf die Speichercontroller der jeweiligen Prozessorkerne zu verteilen, dass kein Thread komplett blockiert wird.

3.3.3 Intercore-Kommunikation

Eine Sonderform der Speicherverwaltung stellt die Intercore-Kommunikation zwischen den Prozessorkernen dar. Da die Interaktionen der Kerne über geteilte Daten erfolgen, müssen die Prozessorkerne zwangsläufig auf gemeinsame Speicherbereiche zugreifen. Die Folge ist, dass konkurrierende Zugriffe auf geteilte Speicher nicht verhindert werden können, was die Besonderheit dieser Art des Speichermanagements darstellt. Aus diesem Grund werden in der Literatur eine Vielzahl von Ansätzen diskutiert, welche die Auswirkungen von Intercore-Kommunikation reduzieren sollen [143].

3.3.3.1 Software-basiert

In die erste Gruppe der software-basierten Ansätze zählen alle Verfahren, welche durch das Zusammenlegen von Wirkketten auf einzelne Prozessorkerne die kernübergreifende Kommunikation schon während der Architekturphase verhindern wollen. So beschreiben die Autoren in [144] ein Konzept, mit dem die Bindung zwischen verschiedenen Software-Komponenten eines Systems bewertet wird. Je nach Stärke dieser Bindung erfolgt im Anschluss ein gezieltes Zusammenlegen auf die jeweiligen Prozessorkerne, wodurch die Menge an Intercore-Kommunikation reduziert werden soll. Die Autoren in [145] stellen hingegen einen Ansatz vor, bei welchem schon während der Designphase die Intercore-Kommunikation sowie deren Auswirkung simuliert wird. Auf Basis dieser Informationen kann im Anschluss im Rahmen der Architekturphase eine optimierte Zuweisung der Software-Komponenten auf die vorhandenen Prozessorkerne erfolgen. Der Vorteil der beiden Ansätze besteht darin, dass schon während der Entwicklungsphase Wirkketten erkannt und bei der Verteilung auf die Prozessorkerne berücksichtigt werden können. Nachteilig ist hingegen, dass weitere Faktoren, wie beispielsweise die Zuweisung von geteilten Ressourcen oder die Partitionierung von Software-Komponenten unterschiedlicher Kritikalität, nicht berücksichtigt werden.

Mittels der zweiten Gruppe werden die Konzepte beschrieben, welche mit Hilfe von Priorisierungsansätzen die Häufigkeit sowie die Menge der Intercore-Kommunikation reduzieren wollen. In [146] richten die Autoren für alle Variablen Schwellwerte ein, ab denen eine Aktualisierung in den Speichern der anderen Prozessorkerne erfolgt. Dabei unterscheiden die Autoren zwischen zwei Arten von Schwellwerten, welche entweder anhand der Aktualisierungshäufigkeit oder anhand des Wertebereichs umgesetzt sind. Der Vorteil dieser Umsetzung besteht darin, dass nur dann eine Intercore-Kommunikation ausgelöst wird, sobald ein bestimmter Schwellwert überschritten wurde. Dabei ist jedoch zu beachten, dass besonders die Aktualisierungen anhand des Wertebereichs nur schwer vorhersagbar sind, was zu einer pessimistischen Abschätzung der Häufigkeit führen kann. Im Gegensatz dazu präsentieren die Autoren in [13] ein Konzept, welches ein software-basiertes Cache-Kohärenzprotokoll auf einem Echtzeitsystem umsetzt. Zu diesem Zweck werden die geteilten Daten in drei Kategorien aufgeteilt, welche sich anhand ihrer Kritikalität definieren. Je nach Priorität wird dabei der Aktualisierungszeitpunkt definiert, an welchem das Kohärenzprotokoll arbeitet. Die Besonderheit ist dabei, dass für jeden lesenden Prozessorkern eine separate Kritikalität definiert werden kann, wodurch die Last der eigentlichen Intercore-Kommunikation weiter reduziert wird. Nachteilig ist hingegen der Einsatz von prozessorkernübergreifenden Interrupts, welche zur Signalisierung einer Aktualisierung genutzt werden. Je nach Häufigkeit der Aktualisierung kann dadurch die Interrupt-Last signifikant ansteigen, was besonders in echtzeitfähigen Systemen zu Laufzeitanomalien führen kann. In [76] fassen die Autoren die Intercore-Kommunikation in Form von *Burst*-Transfers zusammen, welche zum einen die Häufigkeit reduzieren und gleichzeitig den Overhead durch die Arbitrierung sowie durch *Locks* verringern. Dabei muss jedoch berücksichtigt werden, dass, je nach Größe der *Bursts*, die Granularität bei den Speicherzugriffen verringert wird, was mitunter zu längeren Wartezeiten führen kann. Des Weiteren können nur Daten in *Bursts* zusammengefasst werden, welche dieselbe Aktualisierungshäufigkeit aufweisen.

In der letzten Gruppe sind alle Konzepte enthalten, welche die Zugriffe auf geteilte Speicher für die Intercore-Kommunikation über die Prozessorkerne hinweg synchronisieren [147] [22]. Mit der Einführung des Logical Execution Time (LET)-Paradigmas in den AUTOSAR-Standard hat dieser Ansatz eine breite Verwendung in automobilen Steuergeräten für echtzeitfähige Anwendungen gefunden, was die Vielzahl an Fachartikeln zu diesem Thema darlegen [148] [149]. In dem Artikel [150] wird ein Ansatz vorgestellt, bei dem die Lese- und Schreibphase am Anfang und Ende der Task-Ausführung so strukturiert wird, dass Größen für die Intercore-Kommunikation bevorzugt behandelt werden [151]. Das Ziel dieses Ansatzes ist dabei, die Wartezeit der anderen Prozessorkerne zu verkürzen, indem der dazugehörige *Lock* möglichst früh aufgehoben wird. Zu diesem Zweck werden die Daten feingranularer aufgeteilt und so gruppiert, dass diese möglichst zusammenhängend für die nächstfolgende Task mit einem *Lock* geschützt sind. Im Gegensatz dazu schlagen die Autoren in [117] und [96] ein Verfahren vor, bei dem neben der zeitbasierten Synchronisierung zusätzlich eine eventbasierte Intercore-Kommunikation möglich ist. Mit diesem Vorgehen soll der Nachteil kompensiert werden, welcher

bei einer zeitbasierten Synchronisierung und Wirkketten mit einer kurzen Worst-Case Response Time (WCRT) entstehen kann. Je nach Auftreten eines kritischen Events und dem aktuellen Status des Scheduling kann die Bearbeitungszeit deutlich verzögert werden, was besonders bei sicherheitskritischen Systemen inakzeptabel sein kann. Durch die zusätzliche Möglichkeit von eventbasierter Ausführung kann die WCRT reduziert werden, jedoch muss dabei berücksichtigt werden, dass die Bewertung der Echtzeitfähigkeit durch dieses Vorgehen aufwendiger wird. Den Ansatz der synchronisierten Intercore-Kommunikation nutzen auch die Autoren in [152] und [153], welche das *MOSSCA*-Betriebssystem verwenden. Mittels dieses Betriebssystems wird die Intercore-Kommunikation in Zeitschlitzen zusammengefasst, welche als statische Kommunikationsmatrix existiert. Durch dieses Konzept kann die Intercore-Kommunikation zuverlässig vorhergesagt werden, was die Bewertung der Echtzeitfähigkeit vereinfacht. Ein alternatives Verfahren beschreiben die Autoren in [119], welche die drei Phasen des LET-Paradigmas voneinander zeitlich entkoppeln. Zu diesem Zweck wird ein *Double-Buffering* implementiert, welches Wartezeiten durch *Locks* signifikant reduzieren kann. Dabei muss jedoch berücksichtigt werden, dass sich durch dieses Vorgehen die Reaktionszeit deutlich verlängern kann, was abhängig von dem jeweiligen Anwendungsfall betrachtet werden muss. Grundsätzlich eignet sich das LET-Paradigma zur Synchronisierung der Speicherzugriffe auf geteilte Daten. Jedoch muss dabei beachtet werden, dass die zentrale Synchronisierung aller Prozessorkerne nicht zu einem Single Point of Failure (SPOF) werden darf.

3.3.3.2 Hardware-basiert

Zur Optimierung der Intercore-Kommunikation existieren auch hardware-basierte Ansätze, welche im folgenden Abschnitt erläutert werden. Eine Möglichkeit, die Gleichzeitigkeit bei den Zugriffen zu erhöhen, stellen transaktionale Speicher dar. Diese besondere Art der Speicher ermöglichen gleichzeitige Lese- und Schreibzugriffe von unterschiedlichen Prozessorkernen auf dieselben Daten. Zur Realisierung dieser Funktionalität werden innerhalb des Speichers lokale Kopien angelegt, welche über die Laufzeit des Transfers die Datenintegrität gewährleisten [154]. Der Nachteil dieser Speicher liegt in ihrem indeterministischen Verhalten, weswegen transaktionale Speicher in Systemen mit einer harten Echtzeitanforderung derzeit nicht vorhanden sind [155]. Einen alternativen Ansatz präsentieren die Autoren in [29], in dem die Arbitrierungseinheiten am Bussystem des Mehrkernmikrocontrollers so erweitert werden, dass diese zwischen Anwendungen mit einer harten Echtzeitanforderung und zeitunkritischen Funktionalitäten unterscheiden können. In Abhängigkeit der Kritikalität erhält dabei die Anwendung Vorrang, welche die höchste Priorität aufweist. In den derzeit am Markt erhältlichen Mehrkernmikrocontrollern für echtzeitfähige Anwendungen verfügen die Arbitrierungseinheiten lediglich über eine *Starvation-Protection*, weswegen ein breiter Einsatz dieses Verfahrens derzeit nicht möglich ist. Ein weiteres Konzept zur Realisierung der Intercore-Kommunikation mittels der verwendeten Hardware stellt die Implementierung eines Cache-Kohärenzverfahrens dar. Der Vorteil dieser Umsetzung besteht darin, dass jeder Prozessorkern bei je-

dem Speicherzugriff immer das aktuelle Datum erhält, was das Design der Software signifikant vereinfacht. Da derzeit kein Cache-Kohärenzprotokoll existiert, welches die strengen Anforderungen an das deterministische Laufzeitverhalten von echtzeitfähigen Systemen erfüllt, wird auf eine weitere Betrachtung im Rahmen dieser Arbeit verzichtet [156]. Einen weiteren Ansatz präsentieren die Autoren in [3], in dem sie die *Dualported*-Fähigkeit von bestimmten Speichern in echtzeitfähigen Mehrkernsystemen gezielt nutzen. Zu diesem Zweck werden die geteilten Daten so auf die verschiedenen Speicher aufgeteilt, dass maximal zwei Prozessorkerne gleichzeitig darauf zugreifen. Durch die Nutzung eines *Double-Bufferings* werden zusätzlich die Wartezeiten bei aktiven *Locks* reduziert. Bedingt durch den Umstand, dass in modernen Echtzeitsystemen nicht genügend separate Speicher zur Verfügung stehen, um die Kommunikation zwischen allen Prozessorkernen mittels dieses Konzepts zu realisieren, wird auf eine Integration in das Konzept dieser Arbeit verzichtet.

3.4 Kapitelzusammenfassung

In dem Kapitel zu den echtzeitfähigen Mehrkernsystemen werden die notwendigen Grundlagen für diese Arbeit beschrieben. Zu diesem Zweck erfolgt zu Beginn die Beschreibung des Aufbaus aktueller Mikrocontroller für sicherheitskritische Echtzeitsysteme, wobei ein besonderer Fokus auf alle speicherrelevanten Hardware-Komponenten gelegt wird. Im Anschluss erfolgt eine Erläuterung des Aufbaus und der Funktionsweise von Software-Umsetzungen für sicherheitskritische Echtzeitsysteme im automobilen Kontext. Im letzten Teilabschnitt werden die derzeit gängigen Verfahren zur optimierten Speichernutzung in Echtzeitsystemen beschrieben, wobei hier eine Separierung anhand von statischen oder dynamischen Verfahren sowie zu Konzepten der optimierten Intercore-Kommunikation erfolgt. Neben der reinen Beschreibung der vorhandenen Verfahren wird zusätzlich eine Einordnung vorgenommen sowie die Abgrenzung zu dieser Arbeit aufgezeigt.

4

Konzept

In dem folgenden Kapitel wird auf Basis der in Abschnitt 2.1 formulierten Zielstellungen ein Konzept entwickelt, welches die statische Speicherverwaltung für echtzeitfähige Mehrkernsysteme mit einer harten Echtzeitanforderung optimiert. Dafür wird zu Beginn eine Priorisierung von Funktionen und Daten entwickelt, welche auf Basis der Kritikalität, der WCET der jeweiligen Funktion, der maximalen Aufrufhäufigkeit sowie des Speicherverbrauchs berechnet wird. Im nächsten Schritt erfolgt die Beschreibung für das generelle Vorgehen bei der Systemanalyse von eingebetteten Mehrkernsystemen in Bezug auf die Speicherzugriffe. Anhand dieser Informationen erfolgt im letzten Teilaspekt dieses Kapitels eine Beschreibung zur optimierten Nutzung der integrierten Speicherhierarchie auf Basis der kalkulierten Priorisierung.

Der grundlegende Aufbau einer Gesamt-Software für echtzeitfähige Mehrkernmikrocontroller für sicherheitskritische Anwendungen ist exemplarisch in Abbildung 4.1 für die Funktionen dargestellt. Es gibt eine Menge von Funktionen, welche von verschiedenen Funktionalitäten zur Realisierung ihrer Aufgabe benötigt werden. Die Funktionalitäten werden wiederum im Kontext eines Betriebssystems ausgeführt oder als sogenannte *Bare-Metal*-Anwendung direkt auf dem Prozessorkern. Die Betriebssysteminstanzen können wiederum im Kontext einer Virtualisierung ausgeführt werden oder ebenfalls direkt auf einem Kern. Die Darstellung für Variablen und Konstanten ist identisch zu der der Funktionen.

4.1 Priorisierung

Ein wichtiger Grundstein für die optimierte Speicherverwaltung stellt die Priorisierung von allen Funktionen und Daten im Gesamtsystem dar. Der Hintergrund für die Auswahl von Funktionen und Daten als kleinste allokierbare Einheit ist darauf zurückzuführen, dass dies bereits von gängigen Compilern sowie der dazugehörigen Linker unterstützt wird. In diversen wissenschaftlichen Publikationen werden ebenfalls Ansätze auf Basis von sogenannten *Basic Blocks* untersucht, welche die Granularität deutlich erhöhen können. Zu diesem Zweck werden Funktionen so zerlegt, dass lineare Code-Segmente als kleinste logische Einheit betrachtet werden. Der Vorteil dieses Konzepts besteht darin, dass dadurch nur die rechenintensiven

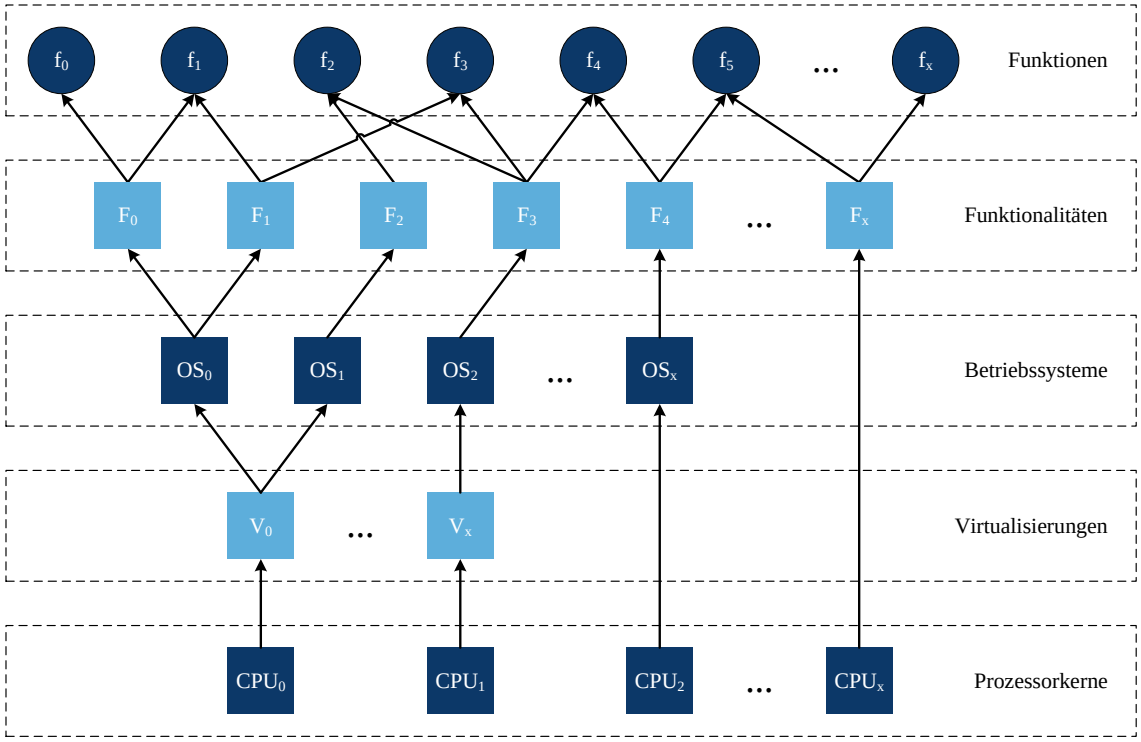

Abbildung 4.1: Grundlegender Aufbau einer Gesamt-Software eines eingebetteten Mehrkernprozessors für sicherheitskritische Systeme mit einer harten Echtzeitanforderung für Funktionen

Abschnitte, wie beispielsweise Schleifen, in die lokalen Speicher allokiert werden, wodurch die begrenzte Speicherkapazität effektiver genutzt wird. Der Nachteil dieses Ansatzes besteht in der gesteigerten Komplexität sowie in der Problemstellung, dass bisherige Compiler und Linker diese Funktionalität nicht unterstützten, wodurch eine Überführung in einen realen Entwicklungsprozess mit großem Aufwand verbunden wäre [10].

Neben der Definition der kleinsten allokierbaren Einheit stellt ebenfalls die Bestimmung der Aufrufhäufigkeit ein Problem bei der Berechnung der Prioritäten dar. In den meisten eingebetteten Systemen ist der Ablauf innerhalb einer Zeitscheibe abhängig von den Eingangsgrößen und dem derzeitigen Systemzustand. Daraus ergeben sich unter Umständen unterschiedliche Zugriffsfrequenzen, je nach Analysezeitpunkt und der dazugehörigen Dauer. Da der Fokus dieses Konzepts jedoch auf der Einhaltung der Echtzeitfähigkeit liegt, ist lediglich die maximale Aufrufhäufigkeit, welche innerhalb einer Hyperperiode möglich ist, relevant für die Beurteilung. Durch dieses Vorgehen kann sichergestellt werden, dass in jedem Falle der schlimmste anzunehmende Zustand bewertet wird. Ein Vorteil, welcher sich aus diesem Ansatz ergibt, ist die Reduzierung der maximalen Analysedauer auf eine Hyperperiode. Die Integration von asynchronen Events, wie beispielsweise Interrupts, erfolgt auf Basis ihrer maximalen Aufrufhäufigkeit innerhalb einer Hyperperiode, wodurch diese ebenfalls als zyklische Task betrachtet werden können. Nachteilig an

diesem Vorgehen ist die fehlende Berücksichtigung von sich gegenseitig ausschließenden Programmpfaden, da für jede Funktion separat die maximale Zugriffshäufigkeit ermittelt wird. Ansätze zur automatisierten Detektion solcher Pfade im Programm-Code wurden in [157] und [158] untersucht, jedoch können diese Konzepte keine dynamischen Aspekte, wie beispielsweise Funktionszeiger, berücksichtigen, welche erst zur Laufzeit initialisiert werden. Einen alternativen Ansatz stellt die Optimierung des WCEP dar, jedoch haben Untersuchungen in diesem Bereich gezeigt, dass durch die Optimierung des WCEP unter Umständen ein neuer WCEP entsteht, wodurch eine erneute Betrachtung vorgenommen werden muss [35]. Durch eine iterative Optimierung kann zwar eine Verbesserung erreicht werden, jedoch ist diese aufgrund des Umfangs der Software-Basis von modernen, echtzeitfähigen Steuergeräten mit einem hohen Rechenaufwand verbunden, weswegen im Rahmen dieser Arbeit auf einen solchen Ansatz verzichtet wird. Einen ähnlichen Ansatz haben die Autoren in [125], [123] und [127] untersucht. Dort wird die Gesamt-Software in Form eines Kontrollflussgraphen dargestellt, welcher im Anschluss mit Hilfe der ganzzahligen linearen Optimierung nach dem WCEP durchsucht wird. Bedingt durch den ebenfalls hohen Rechenaufwand kommen alle Autoren zu dem Ergebnis, dass diese Umsetzung nur für Codebasen mit einem geringen Umfang in Frage kommt, weswegen dieser Ansatz im Rahmen dieser Arbeit ebenfalls nicht weiter verfolgt, sondern stattdessen ein heuristischer Ansatz gewählt wird.

Eine Randbedingung stellt die Zuweisung der Funktionalitäten auf die Prozessorkerne dar, welche im Rahmen dieser Arbeit als fix angesehen wird. Dieser Umstand ist darauf zurückzuführen, dass in der Fachliteratur eine Vielzahl von Ansätzen diskutiert werden, welche für unterschiedliche Anwendungsfälle geeignet sind [159] [77]. Zu diesen Ansätzen gehören beispielsweise die Zuweisung auf Basis der Prozessorauslastung [159], die Abhängigkeiten aufgrund von Wirkketten [144] sowie die Nutzung von geteilten Ressourcen [153]. Da der in dieser Arbeit entwickelte Algorithmus zur Speicherverwaltung möglichst universell einsetzbar sein soll, wird auf eine anwendungsspezifische Ausrichtung verzichtet. Des Weiteren ist die optimierte Speicherallokation, welche in dieser Arbeit vorgestellt wird, grundsätzlich mit vielen Konzepten zur optimierten Zuweisung von Funktionalitäten auf die Prozessorkerne kompatibel.

4.1.1 Systemdefinition

Grundlegend kann die Software eines echtzeitfähigen Steuergeräts als eine Menge von Funktionen, Variablen und Konstanten beschrieben werden, welche die benötigten Funktionalitäten realisieren. Exemplarisch für die Funktionen ist der Zusammenhang in der Formel (4.1) beschrieben [160].

$$M_f := \{(f_0; ...; f_x);\ x \in \mathbb{N}_0\} \tag{4.1}$$

M_f Menge aller Funktionen im System
f_x Funktion

Die Umsetzung einer Funktionalität erfolgt auf Basis einer Teilmenge der im System verfügbaren Elemente in Form von Funktionen, Variablen und Konstanten, was in Formel (4.2) für die Funktionen beispielhaft dargestellt ist.

$$F_{fx} := \{(f_0; ...; f_y);\ x, y \in \mathbb{N}_0\} \tag{4.2}$$

F_{fx} Menge aller Funktionen einer Funktionalität
f_y Funktion einer Funktionalität

Da jedes Element innerhalb des Gesamtsystems von mindestens einer Funktionalität verwendet werden muss, kann die Formel (4.3) aufgestellt werden. Dabei ist jedoch zu beachten, dass die Elemente zur Umsetzung einer Funktionalität nicht zwingend exklusiv für diese sind. So können beispielsweise Funktionen von mehreren Funktionalitäten genutzt oder Variablen zur Kommunikation zwischen den Funktionalitäten verwendet werden.

$$M_f := \{(F_{f0} \cup ... \cup F_{fx});\ x \in \mathbb{N}_0\} \tag{4.3}$$

Zur Unterscheidung der geteilten und der exklusiven Elemente werden zwei Teilmengen gebildet, welche zum einen die Schnittmenge und zum anderen die symmetrische Differenz beinhalten. Die Unterscheidung ist dahingehend wichtig, da geteilte Elemente das Risiko von konkurrierenden Zugriffen erhöhen, was bei der späteren Allokation berücksichtigt werden muss. In den folgenden Formeln (4.4) und (4.5) ist die Bildung der Teilmengen exemplarisch für die Funktionen beschrieben.

$$M_{fE} := \{(F_{f0} \triangle ... \triangle F_{fx});\ x \in \mathbb{N}_0\} \tag{4.4}$$

$$M_{fS} := \{(F_{f0} \cap ... \cap F_{fx});\ x \in \mathbb{N}_0\} \tag{4.5}$$

M_{fE} Teilmenge aller exklusiven Funktionen
M_{fS} Teilmenge aller geteilten Funktionen

4.1.2 Systemfunktionalität

Wie bereits in Abschnitt 3.2 beschrieben, realisiert ein Steuergerät in sicherheitskritischen Anwendungen mit einer harten Echtzeitanforderung eine Vielzahl von Funktionalitäten unterschiedlicher Kritikalität. Die Bewertung der Kritikalität einer Funktionalität erfolgt dabei in der Regel auf Basis der Eintrittswahrscheinlichkeit sowie der potentiellen Schwere der Folgen beim Eintreten eines Fehlers. Je nach Kritikalitätsstufe müssen entsprechende Maßnahmen im Steuergerät umgesetzt werden, welche die Eintrittswahrscheinlichkeit oder die Schwere einer Fehlfunktion reduzieren. Zur Umsetzung dieser Sicherheitsmaßnahmen gehören ebenfalls die Realisierung von geschützten Speicherbereichen, wodurch eine Manipulation des entsprechenden Inhalts verhindert werden kann sowie die Vermeidung von Laufzeitanomalien durch konkurrierende Zugriffe auf geteilte Speicher. Aus diesem Grund wird im Rahmen dieser Arbeit die Kritikalität einer Funktionalität als Basis für die Bewertung von Funktionen, Konstanten und Variablen im System vorgenommen. Zu diesem Zweck wird für jeden Funktionsaufruf sowie für jeden Zugriff auf Konstanten und Variablen eine Priorisierung auf Basis der aktuellen Funktionalität vorgenommen. Neben diesen Aspekten hat noch die WCET einer Funktion sowie deren Speicherverbrauch einen Einfluss auf die Priorisierung. Sollte eine Funktion eine hohe WCET aufweisen, kann dies Einfluss auf die Einhaltung der Echtzeitfähigkeit haben. Da die schnellen Speicher im System, wie beispielsweise die Scratchpads, nur eine geringe Speicherkapazität aufweisen, sollten diese möglichst effizient genutzt werden. Das Ziel ist dabei, eine möglichst große Anzahl an Zugriffen durch die lokalen Speicher der Prozessorkerne zu realisieren, was zum einen die Ausführungsgeschwindigkeit steigert und zum anderen das Potential von konkurrierenden Zugriffen reduziert. Aus diesem Grund wird die Größe der Funktion ebenfalls berücksichtigt, wobei kleinere Funktionen durch die Verwendung des Kehrwerts des Speicherverbrauchs eine höhere Priorität erhalten. Die genaue Berechnung der Priorität für jede Funktion über alle Funktionalitäten ist in der Formel (4.6) exemplarisch für die Funktionen beschrieben.

$$p_F(f) = \sum_{F=0}^{F_{\max}} \left(p(F) \cdot e_F(f) \cdot r_{max}(f) \cdot \frac{1}{s(f)} \right) \tag{4.6}$$

$p_F(f)$	Priorität einer Funktion f über alle Funktionalitäten F
$p(F)$	Priorität einer Funktionalität F
$e_F(f)$	Maximale Aufrufhäufigkeit einer Funktion f innerhalb einer Funktionalität F
$r_{max}(f)$	Maximale Laufzeit (WCET) einer Funktion f
$s(f)$	Speicherverbrauch einer Funktion f

Wie in der Gleichung (4.6) zu sehen ist, werden alle Faktoren, welche einen Einfluss auf die Priorisierung haben, miteinander multipliziert. Je nach Art der benötigten Gewichtung können dabei die Wertebereiche der einzelnen Faktoren angepasst werden, was jedoch spezifisch für das reale Gesamtsystem zu definieren ist. Alternativ

kann auch zu jedem Faktor ein separater, optionaler Verstärkungsfaktor definiert werden, welcher ebenfalls die Möglichkeit zur Gewichtung bietet. In Formel (4.7) ist dieser Ansatz exemplarisch für die Funktionen dargestellt. Die Notwendigkeit der zusätzlichen Gewichtung besteht darin, dass manche Funktionalitäten beispielsweise aufgrund ihrer Aufrufhäufigkeit falsch bewertet werden könnten. Exemplarisch kann hier der Airbag genannt werden, welcher im Idealfall über die gesamte Lebensdauer eines Fahrzeugs nie ausgelöst wird. Um diesem Umstand entgegenzusteuern, könnte jetzt die Kritikalität so sehr erhöht werden, dass dies in der Gesamtpriorität kompensiert wird. Jedoch dient die Kritikalität ebenfalls dazu, Funktionalitäten mit identischem Sicherheitsniveau zu identifizieren und in gemeinsame Speicherbereiche zu allokieren. Durch eine Anpassung der Kritikalität wäre dies in der späteren Allokation nicht mehr möglich, weswegen hier ein Verstärkungsfaktor genutzt werden sollte. Zur besseren Übersicht wird im weiteren Verlauf dieser Arbeit auf die Darstellung aller Gewichtungsfaktoren verzichtet.

$$p_F(f) = \sum_{F=0}^{F_{\max}} \Big((p(F)\cdot w(p_F))\cdot(e_F(f)\cdot w(e_F))\cdot(r_{max}(f)\cdot w(r_{max}))\cdot(\frac{1}{s(f)}\cdot w(s_f)) \Big) \tag{4.7}$$

$w(p_F)$	Gewichtungsfaktor für die Kritikalität
$w(e_F)$	Gewichtungsfaktor für die maximale Aufrufhäufigkeit innerhalb einer Funktionalität F
$w(r_{max})$	Gewichtungsfaktor für die maximale Laufzeit
$w(s_f)$	Gewichtungsfaktor für den Speicherverbrauch

Im Gegensatz zu den Funktionen verursachen die Variablen und Konstanten bei der Ausführung keine direkte Rechenlast. Jedoch hängt die Zugriffsgeschwindigkeit und damit die Laufzeit von der Allokation der Werte im Speicher ab. Da jedoch die Allokation der Elemente in Abhängigkeit der ermittelten Priorität erfolgt, kann dieser Aspekt nicht berücksichtigt werden. Daher ergibt sich für die Konstanten und Variablen eine reduzierte Formel, welche exemplarisch für die Konstanten in Formel (4.8) dargestellt ist.

$$p_F(c) = \sum_{F=0}^{F_{\max}} \Big(p(F) \cdot e_F(c) \cdot \frac{1}{s(c)} \Big) \tag{4.8}$$

$p_F(c)$	Priorität einer Konstanten c über alle Funktionalitäten F
$e_F(c)$	Maximale Aufrufhäufigkeit einer Konstanten c innerhalb einer Funktionalität F
$s(c)$	Speicherverbrauch einer Konstanten c

4.1.3 Betriebssystem

Auf Basis der hergeleiteten Formeln im Abschnitt Systemfunktionalität lässt sich die Priorität eines Elements im Kontext der Funktionalitäten bewerten. Jedoch können Funktionalitäten mit unterschiedlichen Zyklen auf einem echtzeitfähigen Steuergerät berechnet werden, wodurch deren Einfluss auf die Auslastung und Reaktionszeit von der bisher ermittelten Priorität abweichen kann. Beispielsweise erfolgt die Berechnung des Reglers für die Leistungselektronik einer Windkraftanlage in einem deutlich schnelleren Zeitraster als die Bestimmung der Umgebungstemperatur. Aus diesem Grund ist es essentiell, dass der Zyklus der Ausführung ebenfalls berücksichtigt wird. Als Basis für die Bewertung wird die Hyperperiode des Betriebssystems genutzt, welche das vollständige Scheduling einmal beinhaltet. Zur Integration von asynchronen Events in die Berechnung der Priorität wird deren maximale Auftrittswahrscheinlichkeit als Grundlage verwendet. Für die Nutzung der entwickelten Priorisierung ist es erforderlich, dass jede Funktionalität nur einem Zyklus zugeordnet werden kann. Sollte dies aufgrund der gewählten Systemarchitektur nicht möglich sein, wird die entsprechende Funktionalität in Teilfunktionalitäten zerlegt, welche diese Anforderung erfüllen. Wichtig ist dabei zu beachten, dass ein Element ebenfalls in mehreren Teilfunktionen genutzt werden kann. Der entsprechende Zusammenhang ist in Formel (4.9) dargestellt.

$$F_{fx} := \{(PF_{f0} \cup ... \cup PF_{fy});\ x, y \in \mathbb{N}_0\} \tag{4.9}$$

PF_{fy} Menge aller Teilfunktionalitäten einer Funktionalität F

Sobald alle Funktionalitäten beziehungsweise Teilfunktionalitäten die Bedingung der eindeutigen Zuordnung zu einem Zyklus erfüllen, kann die folgende Formel (4.10) zur Bestimmung der Priorität genutzt werden.

$$p_{OS}(f) = \sum_{F=0}^{F_{\max}} \left(p(F) \cdot (e_F(f) \cdot e_{OS}(F)) \cdot r_{max}(f) \cdot \frac{1}{s(f)} \right) \tag{4.10}$$

$p_{OS}(f)$ Priorität einer Funktion f über alle Funktionalitäten F innerhalb eine Hyperperiode des Betriebssystems OS

$e_{OS}(F)$ Maximale Aufrufhäufigkeit einer Funktionalität F innerhalb einer Hyperperiode des Betriebssystems OS

Zur Berechnung der Priorität eines Elements innerhalb einer Hyperperiode wird die Aufrufhäufigkeit innerhalb der Funktionalität mit der Aufrufhäufigkeit der Funktionalität innerhalb der Hyperperiode multipliziert. Daraus ergibt sich der Gesamtwert aller Aufrufe einer Funktion innerhalb einer Hyperperiode durch eine Funktionalität. In [6] wird nicht nur den Funktionalitäten, sondern auch den Tasks eine Priorität zugewiesen, welche bei der Berechnung berücksichtigt wird. Dieses Konzept wird

im Rahmen dieser Arbeit nicht berücksichtigt, da in Systemen mit einer harten Echtzeitanforderung Funktionalitäten mit identischer Priorität häufig in einer Task zusammengefasst werden. Durch diesen Ansatz kann das Umkonfigurieren des Speicherschutzes zu Beginn der Task-Ausführung erfolgen und muss nicht separat für jede Funktionalität ausgeführt werden. Bedingt durch dieses Vorgehen erhalten die Tasks automatisch ihre Priorität durch die Funktionalitäten, welche sie ausführen. Daher würde durch die zusätzliche Integration einer Task-Priorität eine doppelte Berücksichtigung erfolgen, was unter Umständen das Ergebnis der finalen Priorität verfälschen würde. Sollte ein Prozessorkern eine *Bare-Metal*-Umsetzung nutzen, ist eine Priorisierung auf Basis des Betriebssystems nicht erforderlich. Der entsprechende Faktor kann daher auf 1 gesetzt werden.

4.1.4 Virtualisierung

Wie in Abschnitt 3.2 beschrieben, bieten immer mehr Mikrocontroller die Möglichkeit zur Virtualisierung von Software-Funktionen an. Im Gegensatz zu Mikroprozessoren verwenden die Mikrocontroller aber keine MMU, weswegen lediglich eine zusätzliche Abstraktion der Rechenzeit erfolgt. Die Zuordnung der vorhandenen Rechenzeit wird dabei durch einen Hypervisor vorgenommen, von welchem auf jedem Prozessorkern eine separate Instanz berechnet wird. Jeder Hypervisor kann mehrere virtuelle Maschinen ausführen, welche jeweils ihr eigenes Betriebssystem besitzen. Daraus ergibt sich eine weitere Ebene des Scheduling mit eigener Hyperperiode, welche ebenfalls bei der Bestimmung der Aufrufhäufigkeit berücksichtigt werden muss. Aus diesem Grund ist die Formel (4.10) um einen entsprechenden Faktor erweitert, sodass sich Formel (4.11) ergibt. Wichtig ist dabei zu beachten, dass dieser Priorisierungsschritt nur für die Prozessorkerne umgesetzt werden muss, welche die Virtualisierung tatsächlich nutzen und dabei mehr als eine virtuelle Maschine ausführen. Falls dies nicht der Fall sein sollte, wird der entsprechende Faktor auf den Wert 1 gesetzt.

$$p_V(f) = \sum_{F=0}^{F_{\max}} \left(p(F) \cdot (e_F(f) \cdot e_{OS}(F) \cdot e_V(OS)) \cdot r_{max}(f) \cdot \frac{1}{s(f)} \right) \tag{4.11}$$

$p_V(f)$ Priorität einer Funktion f über alle Funktionalitäten F innerhalb eine Hyperperiode des Hypervisors V

$e_V(OS)$ Maximale Aufrufhäufigkeit eines Betriebssystems OS innerhalb einer Hyperperiode des Hypervisors V

4.1.5 Gesamtsystem

Im letzten Schritt erfolgt die Betrachtung der Aufrufhäufigkeit im Kontext des Gesamtsystems. Analog zu den bisherigen Erweiterungen der Formel für die Berücksichtigung des Betriebssystems sowie der Virtualisierung wird in dieser Phase die Betrachtung der Hyperperiode über das Gesamtsystem vorgenommen. Neben den Hauptprozessorkernen werden dabei auch die Co-Prozessoren sowie alle Hardware-Beschleuniger mit Speicherzugriff berücksichtigt. Zur Integration dieser Zusatzmodule in den bestehenden Ansatz werden diese als weitere Prozessoren betrachtet, welche ebenfalls eine Funktionalität mit einem festen Zyklus realisieren. Sollte beispielsweise ein DMA-Controller die Messwerte einer Spannungserfassung in den lokalen Speicher eines Prozessorkerns kopieren, werden die Werte bei der Berechnung der Priorität wie Größen aus einer Intercore-Kommunikation behandelt. Der vollständige Ansatz ist exemplarisch für die Funktion in Formel (4.12) dargestellt. Wie in Gleichung (4.7) aufgezeigt, können die optionalen Gewichtungsfaktoren ebenfalls bei der Priorisierung der Elemente im Kontext des Gesamtsystems verwendet werden.

$$p_S(f) = \sum_{F=0}^{F_{\max}} \left(p(F) \cdot (e_F(f) \cdot e_{OS}(F) \cdot e_V(OS) \cdot e_S(V)) \cdot r_{max}(f) \cdot \frac{1}{s(f)} \right) \quad (4.12)$$

$p_S(f)$ Priorität einer Funktion f über alle Funktionalitäten F innerhalb eine Hyperperiode des Gesamtsystems S

$e_S(V)$ Maximale Aufrufhäufigkeit eines Hypervisors V innerhalb einer Hyperperiode des Gesamtsystems S

4.2 Systemanalyse

Im Rahmen der Systemanalyse erfolgt die Betrachtung der Hardware, welche für das System genutzt wird. Das Ziel ist dabei, eine Übersicht über alle wichtigen Faktoren zu erhalten, welche das Speichermanagement betreffen. Zu diesem Zweck werden Eigenschaften zum Prozessorkern, zu den internen Kommunikationssystemen sowie zu allen Speichern im Mikrocontroller erhoben. Dabei wird ein besonderer Fokus auf die Anbindung der Prozessorkerne an die Speicher im System sowie auf den Speicherschutz gelegt [161].

Die ermittelten Informationen werden in einer generischen Übersicht abgelegt, welche als Basis für die optimierte Speicherverwaltung dient. Zur besseren Beschreibung der Struktur wird in diesem Abschnitt schrittweise der Aufbau sowie die möglichen Inhalte beschrieben. Der Start bildet die Beschreibung des Mikrocontrollers, was in dem Listing 4.1 dargestellt ist.

```
<Mikrocontroller>
    <Beschreibung>
        <ID>
        <Name>
        <Version>
        <Mikrocontrollertyp>
    </Beschreibung>
</Mikrocontroller>
```

Listing 4.1: Beschreibung des Systems

Wie in der allgemeinen Beschreibung des Mikrocontrollers zu sehen ist, gibt es vier Parameter zur Beschreibung. Bei dem ersten Wert handelt es sich um eine eindeutige ID, welche das zu optimierende System definiert. Über den Parameter Name wird dem Mikrocontroller neben der eindeutigen ID ein beschreibender Wert hinzugefügt, welcher bei der Unterscheidung von mehreren Systemen helfen kann. Zur Versionierung der Systembeschreibung ist ein entsprechender Parameter vorgesehen. Mit dem letzten Wert wird der genutzte Mikrocontrollertyp definiert, was zur Verifizierung der Systemanalyse genutzt werden kann.

4.2.1 Prozessorkern

Um den Prozessorkernen im System den benötigten Programm-Code sowie die dazugehörigen Daten vorhersagbar und performant zur Verfügung zu stellen, ist es erforderlich, die speicherrelevanten Elemente zu analysieren. Wie bereits im Abschnitt Gesamtsystem beschrieben, werden neben den Prozessorkernen auch speicherrelevante Peripherien, wie DMA-Controller oder Ethernet-Module, als entsprechende Einheiten in der Systemanalyse berücksichtigt, was in dem Listing 4.2 dargestellt ist.

```
<Mikrocontroller>
    <Beschreibung>...</Beschreibung>
    <Prozessorkern>
        <ID>
        <Name>
        <Architektur>
        <Speicherausrichtung>
    </Prozessorkern>
</Mikrocontroller>
```

Listing 4.2: Beschreibung der Prozessorkerne im System

Die ersten beiden Parameter ID und Name entsprechen grundlegend ihrer Funktionalität im System. Durch die ID wird der Prozessorkern eindeutig identifiziert, was besonders bei Mehrkernsystemen einen wichtigen Aspekt darstellt. Durch den Namen kann dem Prozessorkern eine Bezeichnung gegeben werde, welche als weiterführende Beschreibung fungiert.

Einen weiteren Aspekt bei der Analyse der Prozessorkerne stellt die zugrundeliegende Architektur dar, wobei zwischen der Von-Neumann-, der Harvard- sowie der modifizierten Harvard-Architektur unterschieden wird. Dieser Umstand hat direkten Einfluss auf die Nutzbarkeit der Speicher, denn während die Von-Neumann-Architektur Code und Daten in dieselben Speicher ablegt, erfolgt bei der Harvard-Architektur eine strikte Trennung zwischen Code und Daten. Eine Zwischenlösung stellt die modifizierte Harvard-Architektur dar, welche sowohl exklusive- als auch gemischte Speicher unterstützt. Sollte es sich bei dem zu betrachteten speicherrelevanten Element um ein Peripheriemodul handeln, wird dies in der Architektur entsprechend vermerkt.

Neben der verwendeten Architektur spielt die Speicherausrichtung ebenfalls eine wichtige Rolle. Moderne Mikrocontroller nutzen häufig eine 32 Bit-Architektur und sind dementsprechend auf dieses Alignment hin optimiert. Die Folge ist, dass Zugriffe auf Elemente, welche nicht an einer entsprechenden Adresse liegen, deutlich länger dauern. Moderne Compiler bieten aus diesem Grund Optionen zur Definition des Verhältnisses aus Speicherverbrauch und Performance an [162]. Beispielsweise können in Systemen mit einer geringen Code-Basis und einer schnellen Zykluszeit die Speicher so allokiert werden, dass es die Speicherausrichtung des Prozessorkerns unterstützt. Dahingegen muss bei Systemen mit einer hohen Speicherauslastung von dieser Art der Optimierung abgesehen werden. Diese Beispiele zeigen den Einfluss der entsprechenden Eigenschaft, weswegen diese ebenfalls bei der Systemanalyse berücksichtigt wird.

Da die Zuordnung der Funktionalitäten auf die Prozessorkerne im Rahmen dieser Arbeit als gegeben angesehen wird, ist eine dedizierte Betrachtung der Sicherheitsfunktionen, wie beispielsweise *Lockstep*-Fähigkeit, nicht erforderlich.

4.2.2 Speicher

Im nächsten Schritt erfolgt die Analyse der im Mikrocontroller verfügbaren Speicher. Dabei werden die für die Speicherverwaltung relevanten Informationen extrahiert und anhand einer Struktur beschrieben, welche in dem Listing 4.3 exemplarisch dargestellt ist. Wie diesem Listing zu entnehmen ist, erfolgt in diesem Abschnitt keine Beschreibung der Speichergeschwindigkeit. Dieser Umstand ist darauf zurückzuführen, dass die Speicher, je nach Anbindung im System, eine unterschiedliche Zugriffsgeschwindigkeit aufweisen können. Beispielsweise sind die lokalen Speicher, welche einem Prozessorkern direkt zugeordnet sind, für diesen Kern besonders schnell zu erreichen. Für die anderen Prozessorkerne im System gibt es jedoch keine direkte Anbindung, weswegen die Zugriffsgeschwindigkeit limitiert ist. Aus diesem Grund erfolgt die Betrachtung der Speichergeschwindigkeit in einem separaten Abschnitt.

```
1    <Mikrocontroller>
2        <Beschreibung>...</Beschreibung>
3        <Prozessorkern>...</Prozessorkern>
4        <Speicher>
5            <ID>
6            <Name>
```

```
7             <Gesamtkapazität>
8             <Granularität>
9             <Speicherinhalt>
10            <Typ>
11            <Schnittstelle>
12                <ID>
13                <Name>
14            </Schnittstelle>
15            <Bereich>
16                <Adresse>
17                </Adresse>
18            </Bereich>
19        </Speicher>
20    </Mikrocontroller>
```

Listing 4.3: Beschreibung der Speicher im System

Analog zu den bisherigen Beschreibungen bietet die Struktur der Speicher ebenfalls eine ID zur eindeutigen Identifizierung sowie einen Namen, welcher als beschreibendes Element fungiert. Neben diesen beiden Attributen verfügt jeder Speicher zusätzlich über den Wert Kapazität, welcher die Größe des Speichers in Byte angibt. Für die optimierte Allokation von Funktionalitäten unterschiedlicher Kritikalität ist neben der Speichergröße auch die Granularität der MPU relevant. Nur auf Basis dieser Informationen können der Code sowie die Daten so in die Speicher abgelegt werden, dass eine räumliche Trennung durch die MPU gewährleistet werden kann. Zur Unterstützung der verschiedenen, grundlegenden Architekturen wird die Definition der möglichen Speicherinhalte benötigt. Zu diesem Zweck erfolgt für jeden Speicher die Festlegung, ob dieser Code, Daten oder beides unterstützt. Mittels des Typs wird die verwendete Speichertechnologie beschrieben, wie beispielsweise Cache, Scratchpad oder Flash-Speicher. Neben des Speichertyps ist zusätzlich die Speicheranbindung ein wesentlicher Aspekt zur Beschreibung des Mikrocontrollers. Die separate Definition der Anbindung ist notwendig, da in derzeitig am Markt erhältlichen Multicore-Mikrocontrollern sich teilweise mehrere Speicherbänke eine Schnittstelle teilen oder virtuelle Speicherbereiche innerhalb einer Speicherbank zur Isolierung von geschützten Elementen zur Anwendung kommen. Aus den genannten Gründen ist daher eine eindeutige ID zur Beschreibung der Anbindung essentiell und zusätzlich ein Name, welcher die Interpretation erleichtert.

Die Beschreibung der allokierbaren Sektionen eines Speichers werden als Bereiche in der Analyse erfasst. Dabei können mehrere Adressbereiche für einen Speicher hinterlegt werden. Dies ist erforderlich, da einige Mikrocontrollerhersteller virtuelle Adressen für die Realisierung von Funktionen, wie beispielsweise der Unterscheidung von cache- und nicht-cache-baren Bereichen, nutzen [51].

4.2.3 Network-on-Chip

Einen weiteren wichtigen Aspekt für die optimierte Speicherverwaltung stellt das in dem Listing 4.4 dargestellte Network-on-Chip dar. Je nach eingesetztem Typ ermöglicht es parallele Speicherzugriffe von unterschiedlichen Prozessorkernen auf unterschiedliche Speicher. Des Weiteren hat die Anzahl der integrierten Kommunikationssysteme sowie deren Verbindung untereinander einen signifikanten Einfluss auf die Zugriffsgeschwindigkeit zwischen Teilnehmern, welche über verschiedene Network-on-Chips mit dem Rest des Systems verbunden sind [2].

```
1  <Mikrocontroller>
2      <Beschreibung>...</Beschreibung>
3      <Prozessorkern>...</Prozessorkern>
4      <Speicher>...</Speicher>
5      <Network-on-Chip>
6          <ID>
7          <Name>
8          <Typ>
9      </Network-on-Chip>
10 </Mikrocontroller>
```

Listing 4.4: Beschreibung der Kommunikationssysteme im System

Zur eindeutigen Identifizierung der Kommunikationssysteme verfügen diese über eine eindeutige ID sowie einen beschreibenden Namen. Über den Typ wird definiert, um welche Art Network-on-Chip es sich handelt, was einen signifikanten Einfluss auf die spätere Speicherverwaltung hat. Sollte es sich beispielsweise um ein Bussystem handeln, ist die Allokation auf verteilte Speicher zu Vermeidung von konkurrierenden Zugriffen unnötig, da bereits die Anbindung der Speicher den Engpass darstellt. Eine Crossbar hingegen erlaubt parallele Zugriffe, jedoch nur von unterschiedlichen Prozessorkernen zu unterschiedlichen Speichern. Dieser Umstand muss bei der Allokation berücksichtigt werden, damit die Anzahl der konkurrierenden Zugriffe reduziert werden kann.

4.2.4 Speicheranbindungen

Im letzten Analyseschritt erfolgt die Beschreibung der Speicheranbindungen von jedem Prozessorkern zu jedem Speicher für jeden Zugriffstyp, was in dem Listing 4.5 dargestellt ist. Aus diesem Grund besteht jede Speicheranbindung aus den beiden Parametern Prozessorkern und Speicher, welche die Referenz auf das entsprechende Analyseergebnis darstellen. Durch die beiden Elemente Zugriffstyp und Zugriffsgeschwindigkeit erfolgt die Abbildung der eigentlichen Performance der jeweiligen Speicheranbindung. Bedingt durch den Umstand, dass Prozessorkerne, welche auf der Harvard-Architektur basieren, über zwei separate Schnittstellen für den Zugriff auf Code und Daten verfügen, kann die Geschwindigkeit, je nach Zugriffstyp, variieren, weswegen diese Angabe essentiell für die Bewertung ist. Zur Bestimmung der Zugriffsgeschwindigkeit wird die Performance bei exklusiver Nutzung durch den aus-

gewählten Prozessorkern als Basis angenommen, da die Menge der konkurrierenden Zugriffe erst nach der Allokation bestimmt werden kann. Die Zugriffsgeschwindigkeit wird dabei als normierter Wert hinterlegt und dient der späteren Allokation als wichtiger Faktor bei der Zuordnung der Speicher zu den jeweiligen Prozessorkernen.

```
<Mikrocontroller>
    <Beschreibung>...</Beschreibung>
    <Prozessorkern>...</Prozessorkern>
    <Speicher>...</Speicher>
    <Network-on-Chip>...</Network-on-Chip>
    <Speicheranbindung>
        <Prozessorkern>
        <Speicher>
        <Zugriffstyp>
        <Zugriffsgeschwindigkeit>
    </Speicheranbindung>
</Mikrocontroller>
```

Listing 4.5: Beschreibung der Speicheranbindungen im System

4.3 Speicherverwaltung

Im dritten Schritt des hier vorgestellten Konzepts erfolgt nach der Priorisierung sowie der Systemanalyse die eigentliche Zuweisung des Programm-Codes und der Daten im Rahmen der Speicherverwaltung. Zu diesem Zweck werden im ersten Abschnitt die Speicher auf Basis ihrer Zugriffsgeschwindigkeit den jeweiligen Kernen zugeordnet. Es wird also anhand der Systemanalyse eine Bewertung vorgenommen, welche Speicher im System primär für die Verwendung durch einen bestimmten Prozessorkern vorgesehen sind. Im Anschluss erfolgt die Ermittlung des Speicherbedarfs für jeden Kern unter Verwendung der zugeordneten Funktionalitäten. Im Rahmen dieser Arbeit wird die Zuordnung der Funktionalitäten als fix angenommen, weswegen hier keine weiteren Optimierungsschritte anhand des Speicherverbrauchs vorgesehen sind. Dieser Umstand ist darauf zurückzuführen, dass für eine optimierte Verteilung der Funktionalitäten auf die Prozessorkerne weitere Randbedingungen, wie beispielsweise die benötigte Rechenzeit, die Abhängigkeit zu anderen Funktionalitäten oder auch die Verwendung von bestimmten Peripherien, mit einbezogen werden sollte. Da die Verteilung der Funktionalitäten auf die Prozessorkerne von anderen Faktoren als die Speicheroptimierung abhängig ist, kann eine entsprechende Optimierung im Rahmen dieser Arbeit nicht erfolgen. Im letzten Teilabschnitt erfolgt dann die eigentliche Allokation des Programm-Codes sowie der Daten auf Basis der errechneten Priorität auf die entsprechenden Speicher.

4.3.1 Speicherzuweisung

Die Speicherzuweisung erfolgt durch die Analyse der Systembeschreibung. Zu diesem Zweck wird für jeden Prozessorkern die Anbindung zu jedem Speicher ausgewertet und die Speicher-IDs anhand ihrer Zugriffsgeschwindigkeit für den jeweiligen Kern sortiert. Das Ziel ist dabei, die direkt zugeordneten lokalen Speicher sowie die besonders performant angebundenen globalen Speicher für jeden Prozessorkern zu identifizieren und diese vorwiegend für die Allokation der Funktionalitäten zu nutzen. Sollte die Speicherkapazität dieser Speicher nicht ausreichen, kann zusätzlich der nächste leistungsfähigere Speicher aus der Liste verwendet werden. Beispielhaft kann hier die Infineon AURIX TC3xx-Serie genannt werden, bei welcher jeder Prozessorkern vier lokale Speicher besitzt. Zusätzlich sind jeweils eine Bank des globalen Flash-Speichers sowie des globalen RAMs über eine Direktanbindung mit dem Kern verbunden. Über diese Direktanbindung kann die Zugriffsgeschwindigkeit signifikant erhöht werden, weswegen diese Speicher vorwiegend genutzt werden sollten [4]. Falls alle Speicher eines Typs dieselbe Zugriffsgeschwindigkeit bieten, erfolgt die Zuordnung anhand der Speicher-ID. Der erste Prozessorkern im System erhält dabei den ersten Speicher des entsprechenden Typs, was für alle fortlaufenden Kerne ebenfalls gilt. Der Pseudocode in dem Listing 4.6 verdeutlicht exemplarisch die Bewertung der Speicheranbindung für jeden Prozessorkern.

```
/* Liste mit allen Speichertypen pro Prozessorkern */
Speicherliste[Prozessorkern.Anzahl][Speicher.Typ.Anzahl]

/* Iteration über alle Prozessorkerne */
for i = 0 to Prozessorkern.Anzahl
{
    /* Iteration über alle Speicher */
    for j = 0 to Speicher.Anzahl
    {
        /* Bestimmung des Zugriffstyps */
        switch(Speicheranbindung.Zugriffstyp)
        {
            /* Programm-Code */
            case: Code
            {
                Speicherliste[i][Speicher[j].Typ.Code].Add(Speicher[j])
            }
            /* Daten */
            case: Daten
            {
                Speicherliste[i][Speicher[j].Typ.Daten].Add(Speicher[j])
            }
        }
    }

    /* Sortierung nach Zugriffsgeschwindigkeit */
    for k = 0 to Speicher.Typ.Anzahl
    {
        Speicherliste[i][k].Sort(Speicheranbindung.Zugriffsgeschwindigkeit)
```

```
30
31          /* Sind alle Zugriffsgeschwindigkeiten identisch? Vergleich der
32          Zugriffsgeschwindigkeit von dem ersten und letzten Element */
33          if(Speicherliste[i][k][0].Zugriffsgeschwindigkeit ==
34          Speicherliste[i][k][Anzahl].Zugriffsgeschwindigkeit)
35          {
36              // Sortiere nach Speicher-ID anhand der Prozessorkern-ID
37              Speicherliste[i][k].Sort(Speicher.ID, Prozessorkern[i].ID)
38          }
39      }
40  }
```

Listing 4.6: Pseudocode zur Zuweisung und Sortierung der Speicher anhand der Zugriffsgeschwindigkeit zu den jeweiligen Prozessorkernen

4.3.2 Speicherbedarf

Für die Berechnung des Speicherbedarfs von jedem Prozessorkern wird ein iterativer Ansatz gewählt, welcher bei jedem Schritt prüft, ob genügend Speicherplatz vorhanden ist. Das Ziel ist dabei, möglichst jede Funktionalität in einem separaten Bereich in einem von dem jeweiligen Prozessorkern exklusiv genutzten Speicher zu allokieren. Dabei ist die Größe eines Bereichs von der Granularität der MPU abhängig. Der Vorteil dieses Ansatzes besteht darin, dass grundlegend jede Funktionalität von den anderen räumlich isoliert wird, was beispielsweise für partielle Software-Updates relevant sein kann. Die Voraussetzung dafür ist, dass in jedem MPU-Bereich alle Funktionen, Variablen und Konstanten vorhanden sein müssen, welche von der jeweiligen Funktionalität benötigt werden. Je nach Gesamt-Software kann sich daraus ein hoher Bedarf an Kopien ergeben, welche in den zugeordneten Speichern vorgehalten werden müssen. Sollte die vorhandene Speicherkapazität in den zugewiesenen Speichern des jeweiligen Prozessorkerns nicht ausreichen, werden im ersten Schritt der Code sowie die Daten mit identischer Kritikalität zusammengelegt. Dabei erfolgt das Zusammenlegen schrittweise und es wird mit den Funktionen, Variablen und Konstanten begonnen, welche die geringste Priorität aufweisen. Nach jeder Fusion erfolgt eine Prüfung, ob der benötigte Speicherplatz nun ausreichend ist.

Sollte der gesamte Code sowie die dazugehörigen Daten mit identischer Kritikalität eines Prozessorkerns vollständig zusammengelegt sein und die Speicherkapazität ist immer noch nicht ausreichend, wird im zweiten Schritt überprüft, ob in den exklusiv zugewiesenen Speichern der anderen Prozessorkerne noch Speicherplatz vorhanden ist. Die Überprüfung erfolgt dabei anhand der Reihenfolge aus der im Abschnitt Speicherzuweisung beschriebenen Priorisierung. Um die Zugriffe auf die zugewiesenen Speicher der anderen Prozessorkerne möglichst gering zu halten, werden dort vorwiegend Funktionen, Variablen und Konstanten mit der höchsten Priorität allokiert, welche zur Laufzeit zusätzlich in den lokalen Speichern vorgehalten werden. Durch dieses Vorgehen erfolgt nur während der Startphase ein Zugriff auf die Speicher der anderen Prozessorkerne. Falls mehr Speicherkapazität benötigt

wird als in den lokalen Speichern zur Verfügung steht, werden im Anschluss nur Code und Daten mit der geringsten Priorität in die Speicher der anderen Prozessorkerne allokiert, um die konkurrierenden Zugriffe auf ein Minimum zu beschränken. Des Weiteren kann mit Hilfe der Arbitrierungsrichtlinie eine Priorisierung der Prozessorkerne für den Zugriff auf den jeweiligen Speicher erfolgen, wodurch diese Effekte weiter reduziert werden können.

Falls nach den genannten Optimierungsschleifen weiterhin zu wenig Speicher im System vorhanden ist, werden im letzten Schritt der Code sowie die Daten von Funktionalitäten mit identischer Kritikalität auf verschiedenen Prozessorkernen zusammengelegt. Dabei wird analog zum Vorgehen im letzten Schritt mit den Funktionen, Variablen und Konstanten begonnen, welche die höchste Priorität aufweisen. Da der dazugehörige Code sowie die Daten zur Laufzeit in den lokalen Speichern vorgehalten werden, entstehen konkurrierende Zugriffe nur während der Startphase. Sollte der Speicherbedarf weiterhin die verfügbare Kapazität überschreiten, werden im Anschluss der Code sowie die Daten zusammengelegt, welche die geringste Priorität haben. Dieser Vorgang wird so lange fortgesetzt, bis der Speicherbedarf der verfügbaren Kapazität entspricht, wobei die räumliche Isolierung von Funktionalitäten unterschiedlicher Kritikalität eingehalten werden muss. Das iterative Vorgehen ist in dem Listing 4.7 exemplarisch für die Funktionen dargestellt.

```
1  /* Liste mit dem Speicherbedarf pro Prozessorkern */
2  Speicherbedarf[Prozessorkern.Anzahl]
3
4  /* Iteration über alle Prozessorkerne */
5  for i = 0 to Prozessorkern.Anzahl
6  {
7      /* Bestimmung des Speicherbedarfs für Prozessorkern i,
8      jede Funktionalität erhält einen separaten Speicherbereich */
9      for j = 0 to Funktionen.Anzahl
10     {
11         /* Wird die aktuelle Funktionalität von dem
12         aktuellen Prozessorkern ausgeführt? */
13         if i == Funktionen[j].Prozessorkern
14         {
15             /* Prüfung, ob es bereits einen MPU-Bereich von derselben
16             Funktionalität gibt */
17             if exist(Funktionen[j].Funktionalität in MPU[i].Bereiche)
18             {
19                 /* Hinzufügen der Funktion zu dem entsprechenden
20                 MPU-Bereich */
21                 add(Funktion[j] to MPU[i].Bereiche[Funktion.Funktionalität])
22             }
23             else
24             {
25                 /* Hinzufügen des MPU-Bereichs */
26                 add(MPU.Bereich[Funktion.Funktionalität] to MPU[i].Bereiche)
27                 /* Hinzufügen der Funktion zu dem entsprechenden
28                 MPU-Bereich */
29                 add(Funktion[j] to MPU[i].Bereiche[Funktion.Funktionalität])
30             }
```

```
31          }
32      }
33
34      /* Prüfung, ob die Speicherkapazität der zugeordneten Speicher
35      des jeweiligen Prozessorkerns ausreichend sind */
36      if MPU[i].Bereiche > Prozessorkern[i].Speicher.Kapazität
37      {
38          /* Zusammenführen aller Funktionalitäten, welche auf einem Prozessorkern
39          ausgeführt werden */
40
41          /* Sortierung aller MPU-Bereiche nach Priorität
42          (Start mit der niedrigsten Priorität) */
43          sort(MPU[i].Bereiche, 0)
44
45          /* Iteratives Zusammenführen von niederprioren MPU-Bereichen mit
46          derselben Kritikalität */
47          for j = 0 to MPU[i].Bereiche
48          {
49              /* Zusammenführen der MPU-Bereiche */
50              combine(MPU[i].Bereiche, MPU[i].Bereiche.Kritikalität[j])
51
52              /* Prüfung, ob Speicherkapazität ausreichend ist */
53              if MPU[i].Bereiche.Kapazität <= Prozessorkern[i].Speicher.Kapazität
54              {
55                  /* Speicherkapazität ist ausreichend */
56                  break
57              }
58          }
59      }
60  }
61
62  /* Prüfung, ob Gesamtspeicherkapazität ausreicht */
63  if MPU.Bereiche.Kapazität <= Speicher.Kapazität
64  {
65      /* Zusammenführen aller Funktionalitäten, welche im Gesamtsystem
66      ausgeführt werden */
67
68      /* Sortierung aller MPU-Bereiche nach Priorität
69      (Start mit der niedrigsten Priorität) */
70      sort(MPU.Bereiche, 0)
71
72      /* Iteratives Zusammenführen von niederprioren MPU-Bereichen mit
73      derselben Kritikalität */
74      for j = 0 to MPU.Bereiche
75      {
76          /* Zusammenführen der MPU-Bereiche */
77          combine(MPU.Bereiche, MPU.Bereiche.Kritikalität[j])
78
79          /* Prüfung, ob Speicherkapazität ausreichend ist */
80          if MPU.Bereiche.Kapazität <= Speicher.Kapazität
81          {
82              /* Speicherkapazität ist ausreichend */
83              break
84          }
```

```
85      }
86  }
87  else
88  {
89      /* Nutzung der freien Speicherkapazitäten in den Speichern
90      der anderen Prozessorkerne */
91  }
```

Listing 4.7: Pseudocode zur Berechnung des Speicherbedarfs für den Code eines jeden Prozessorkerns im System

Grundsätzlich ist die Berechnung des Speicherbedarfs für die Variablen identisch zu der Berechnung des Speicherbedarfs von Funktionen. Einen Unterschied gibt es jedoch hinsichtlich des Alignments. Im Gegensatz zu dem Programm-Codes profitieren die Daten von einer optimierten Alignment-Strategie, welche sich an der Registerbreite des Prozessorkerns orientiert. Aus diesem Grund gibt es bei den Variablen eine weitere Schleife zur Berechnung des Speicherbedarfs. Sollten alle Daten vollständig in die Speicher des jeweiligen Prozessorkerns passen, können die Variablen mit der höchsten Priorität zusätzlich so allokiert werden, dass diese immer auf einer alignment-kompatiblen Adresse liegen. Da dieses Vorgehen zusätzlichen Speicher benötigt, kann dies nur umgesetzt werden, solange freier Speicher im RAM vorhanden ist.

4.3.3 Allokation

Nach der Priorisierung der Speicher für jeden Prozessorkern sowie der Berechnung des Speicherbedarfs für jede Funktionalität erfolgt im letzten Schritt der Speicherverwaltung die eigentliche Allokation. Zu diesem Zweck wird für jeden Prozessorkern eine separate Liste mit allen Funktionen, Variablen und Konstanten erstellt, welche von den Funktionalitäten benötigt werden, die der entsprechende Kern ausführt. Die Elemente in der Liste werden wieder auf Basis ihrer Priorität sortiert, wobei in diesem Falle nicht die Priorität des Gesamtsystems entscheidend ist, sondern die Wichtigkeit für den jeweiligen Kern. Nur falls mehrere Elemente dieselbe Prozessorkernpriorität aufweisen, wird die Gesamtpriorität als Entscheidungskriterium genutzt. Die Berechnung der Prioritäten erfolgt dabei analog zu der Kalkulation in dem Kapitel 4.1. Je nachdem, ob der jeweilige Prozessorkern die Funktionalitäten mittels einer Virtualisierung, eines Betriebssystems oder als sogenannte *Bare-Metal*-Umsetzung ausführt, wird die entsprechende Formel für die Berechnung genutzt. Der Grund für die Nutzung von separaten Kernlisten ist darin begründet, dass sich die Priorität der Funktion, Variablen und Konstanten für die Funktionalitäten auf den unterschiedlichen Prozessorkernen deutlich unterscheiden kann. Anhand der Abbildung 4.2 wird das Vorgehen exemplarisch für die Funktionen grafisch dargestellt.

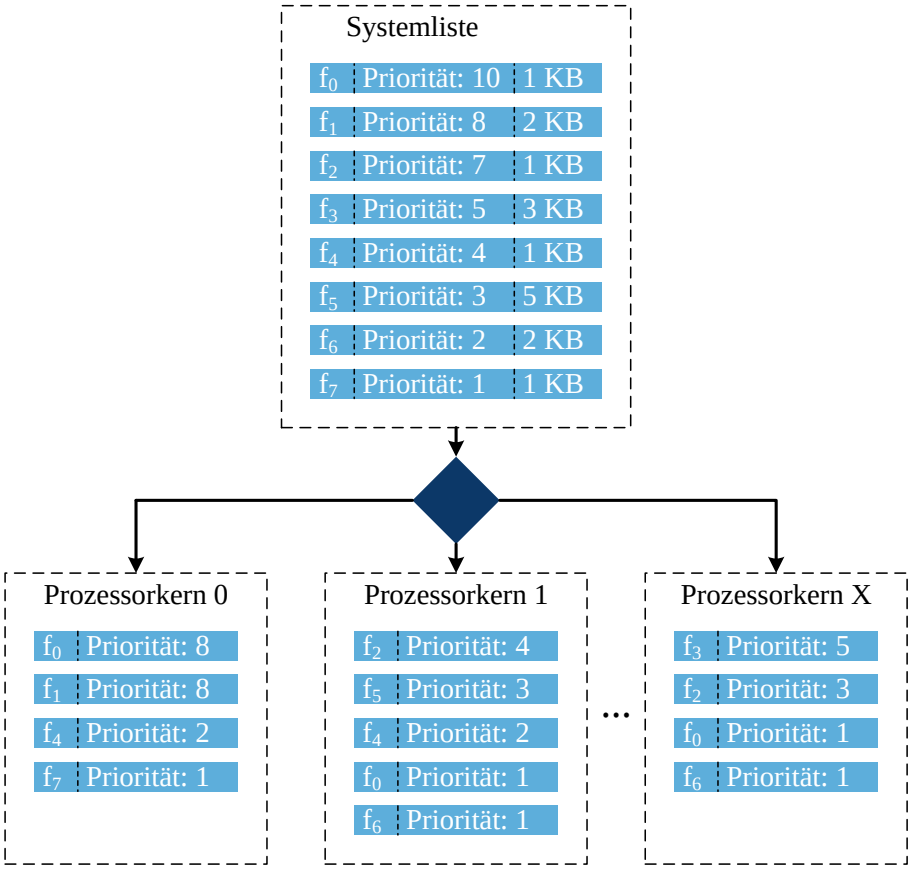

Abbildung 4.2: Separierung der Funktionen aus der Gesamtliste in gesonderte Listen für jeden Prozessorkern

4.3.3.1 Code

Nachdem die Prioritätslisten für alle Prozessorkerne erstellt sind und der Speicherbedarf aller Funktionalitäten eines Prozessorkerns ermittelt ist, erfolgt die Verteilung des Codes sowie der Daten auf die jeweiligen Speicher. Zu diesem Zweck wird als erstes überprüft, ob der zugeordnete Flash-Speicher des jeweiligen Prozessorkerns für die Allokation des gesamten Programm-Codes, der Konstanten sowie der Initialisierungswerte der Variablen ausreicht. Sollte dies der Falle sein, wird mit der Zuweisung des Codes in den dazugehörigen Flash-Speicher begonnen. Dabei wird anhand der Prioritäten der jeweiligen Kernlisten die Abarbeitung vorgenommen. Für die Funktion mit der höchsten Priorität wird ein Bereich im Speicher reserviert, welcher die Größe einer MPU-Granularität besitzt, in welche die Funktion allokiert wird. Im Anschluss wird die Funktion mit der nächsthöheren Priorität verarbeitet. Hier findet zu Beginn eine Überprüfung statt, ob diese Funktion ebenfalls in den ersten Bereich mit allokiert werden kann. Je nach Speicherbedarfsberechnung aus Abschnitt 4.3.2 ist dies der Fall, wenn die Funktion zur selben Funktionalität, zur selben Kritikalität auf dem jeweiligen Prozessorkern oder zur selben Kritikalität im Kontext des Gesamtsystems gehört. Anhand des so berechneten Speicherbedarfs er-

gibt sich die Menge der verfügbaren Kopien, welche im Gesamtsystem vorgehalten werden können. Sollte es sich dabei um eine geteilte Funktion über Prozessorkerngrenzen hinweg handeln, erhält der Kern die Kopie in seinen zugeordneten Speicher, welcher die höchste Priorität in seiner separaten Prozessorkernliste aufweist. Der Vorteil dieses Ansatzes besteht darin, dass dieser Kern den schnellstmöglichen Zugriff auf diese Funktion hat, was besonders bei laufzeitkritischen Anwendungen mit einer hohen Priorität essentiell ist. Sollte es sich hingegen um eine Funktion handeln, welche von mehreren Funktionalitäten auf einem Prozessorkern geteilt wird, erhält die Funktionalität die entsprechende Funktion in ihren Bereich, bei welcher die Prioritätsberechnung aus Abschnitt 4.1.2 den höchsten Wert ergeben hat. Durch dieses Vorgehen wird der Programm-Code anhand seiner Priorität zusammenhängend im Speicher abgelegt, was die Funktionsweise der Programm-Caches sowie der sogenannten *Prefetech-Buffer* unterstützt. Dieses Vorgehen wird analog für alle weiteren Funktionen wiederholt, bis diese vollständig im nichtflüchtigen Flash-Speicher abgelegt sind. Im Anschluss wird für die Funktionen mit der höchsten Priorität zusätzlich noch Speicher im lokalen Scratchpad des entsprechenden Prozessorkerns reserviert, in welchem nach dem Systemstart mit Hilfe der *Copy-Table* der Programm-Code kopiert wird. Das Ziel ist dabei, dass jeder Prozessorkern sein Programm-Scratchpad möglichst exklusiv nutzt, um Laufzeitanomalien, bedingt durch konkurrierende Zugriffe, effektiv zu verhindern. Analog zu dem Flash-Speicher wird bei den lokalen Scratchpads ebenfalls die Granularität der MPU berücksichtigt. Je nach Aufbau des Programm-Codes kann es dabei vorkommen, dass die einzelnen Bereiche aufgrund der Granularität der MPU nicht vollständig befüllt werden können. Diese Fragmentierung lässt sich nicht verhindern und ist ein bekanntes Problem bei der Ausführung von Funktionalitäten unterschiedlicher Kritikalität auf einem Prozessorkern [163].

Der gesamte Ablauf zur Allokation von Programm-Code ist in Abbildung 4.3 für die Funktionen beispielhaft dargestellt. Als Basis für diese Musterumsetzung dienen die Funktionen aus der Abbildung 4.2.

4.3.3.2 Daten

Die Allokation von Daten unterteilt sich in drei separate Abschnitte, welche im Folgenden detailliert erläutert werden. Zur ersten Gruppe gehören die exklusiven Variablen, welche ausschließlich von einem Prozessorkern genutzt werden. Alle Variablen, auf welche mehrere Prozessorkerne zugreifen, werden für die Kommunikation zwischen den Kernen genutzt und unterliegen damit zwangsläufig konkurrierenden Zugriffen. Zu der letzten Gruppe gehören alle Konstanten, auf welche die Kerne zur Laufzeit nur lesend zugreifen.

Exklusive Variablen Für die Allokation der exklusiven Variablen werden primär die lokalen Daten-Scratchpads genutzt, welche im Gegensatz zu den Programm-Scratchpads über eine deutlich höhere Speicherkapazität verfügen. Um die Effekte von konkurrierenden Zugriffen möglichst gering zu halten, erfolgt die Nutzung der lokalen Daten-Scratchpads exklusiv durch die jeweiligen Prozessorkerne. Sollte die Speicherkapazität der lokalen Scratchpads nicht ausreichen, erfolgt

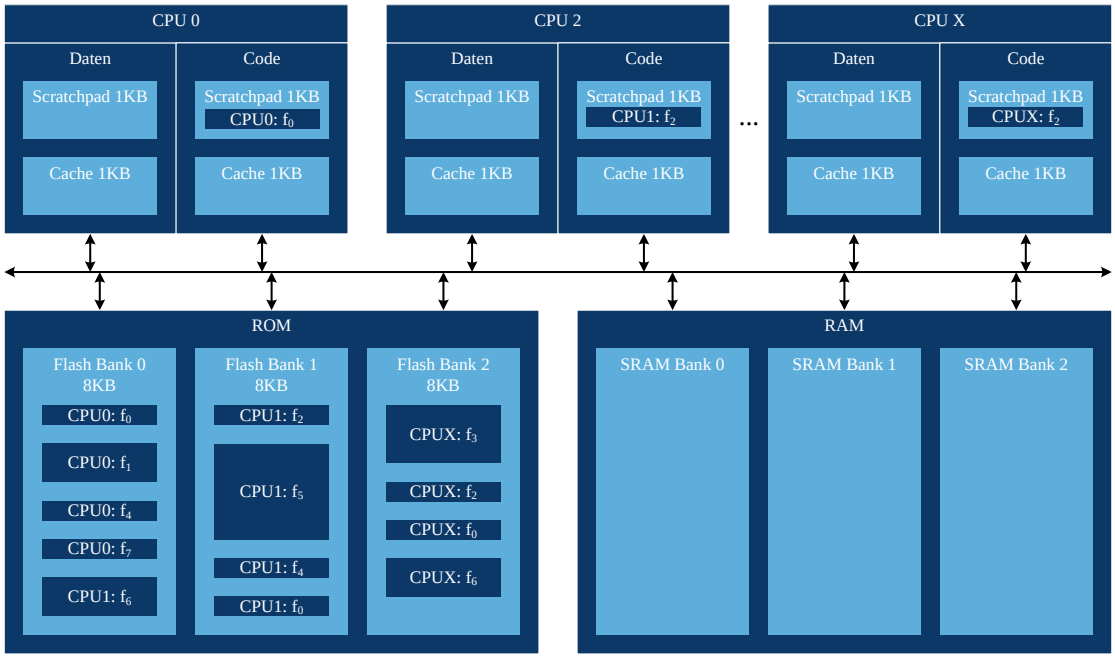

Abbildung 4.3: Exemplarische Darstellung der Allokation von Programm-Code bei einem Beispielsystem mit einer MPU-Granularität von 1 KB

im Anschluss die Nutzung des globalen RAMs. Dabei erfolgt die Nutzung der RAM-Speicher anhand der Speicherzuweisung aus Abschnitt 4.3.1.

Geteilte Variablen Da sich bei den geteilten Variablen die konkurrierenden Zugriffe nicht vermeiden lassen, erfolgt bei dieser Art von Daten die Allokation direkt in die globalen RAM-Speicher. Durch dieses Vorgehen sollen Wartezyklen auf die lokalen Scratchpads durch konkurrierende Zugriffe anderer Prozessorkerne verhindert werden, was ansonsten einen negativen Einfluss auf die WCET des betroffenen Prozessorkerns hätte. Aus diesem Grund werden die lokalen Speicher aus der Speicherzuweisung für die geteilten Variablen ausgeblendet und ausschließlich auf die globalen RAM-Speicher zurückgegriffen. Die Auswahl des globalen RAM-Speicher erfolgt dabei anhand der bekannten Speicherzuweisung, wobei der Prozessorkern entscheidend ist, welcher die höchste Priorität für diese Variable aufweist. Die Bestimmung der geteilten Variablen erfolgt dabei durch die Beschreibung in Abschnitt 4.1.1.

Konstanten Die Allokation der Konstanten erfolgt im ROM-Speicher aufgrund des Umstandes, dass diese zur Laufzeit nicht geändert werden müssen. Um die Zugriffsgeschwindigkeit auf diese Werte zu erhöhen, werden diese analog zu dem Programm-Code so im Speicher abgelegt, dass es die Funktionsweise der Daten-Caches unterstützt. Da die Daten-Caches in echtzeitfähigen Systemen über keinerlei Kohärenz in Hardware verfügen, bietet sich die Nutzung für konstante Werte an. Der Vorteil dieses Ansatzes besteht darin, dass bei LRU-basierten Caches auf das *Dirty-Bit* verzichtet werden kann, wodurch die Betrachtung der Laufzeit vereinfacht wird.

4.4 Kapitelzusammenfassung

Mittels des vierten Hauptabschnitts erfolgt die Beschreibung des im Rahmen dieser Arbeit entwickelten Konzepts zur optimierten statischen Speichernutzung von echtzeitfähigen Mehrkernsystemen. Im ersten Abschnitt wird daher die Priorisierung hergeleitet, welche als Basis für die optimierte Allokation dient. Dazu wird mittels verschiedener Parameter, wie der Laufzeit, der Aufrufhäufigkeit, des Speicherverbrauchs sowie der Kritikalität, für jede Komponente der Software eine Priorität ermittelt anhand welcher die Reihenfolge bei der Allokation berücksichtigt wird. Durch die sich anschließende Systemanalyse wird die genutzte Hardware dargestellt, wobei ausschließlich die speicherrelevanten Bestandteile des Mikrocontrollers dargelegt werden. Anhand der genannten Priorisierung sowie der generischen Hardware-Beschreibung kann im Anschluss die eigentliche Allokation erfolgen. Dabei werden die Bestandteile der Software mittels verschiedener Optimierungsansätze so auf die verfügbaren Speicher verteilt, dass die Effekte von konkurrierenden Zugriffen minimiert und die Ausführungsgeschwindigkeit optimiert wird. Zusätzlich erfolgt die Berücksichtigung der Granularität des Speicherschutzes, was besonders für die räumliche Isolation essentiell ist.

5

Experimentelle Ergebnisse

Für den Nachweis der Funktionalität des in Kapitel 4 vorgestellten Konzepts wird in diesem Abschnitt eine experimentelle Umsetzung vorgenommen. Zu diesem Zweck wird als erstes der Messaufbau sowie die dazugehörige Messmethodik detailliert beschrieben. Im Anschluss erfolgt die Vorstellung des Testsystems, wobei ein besonderer Fokus auf die genutzten Lastszenarien gelegt wird. Im Abschnitt 5.3 wird die genutzte Messauswertung sowie die dafür implementierte Software dargelegt. Den Abschluss dieses Kapitels bildet die Präsentation sowie die Bewertung der erreichten Ergebnisse.

5.1 Messaufbau

Zur Ermittlung der Messergebnisse wird das in Abbildung 5.1 dargestellte Testsystem genutzt.

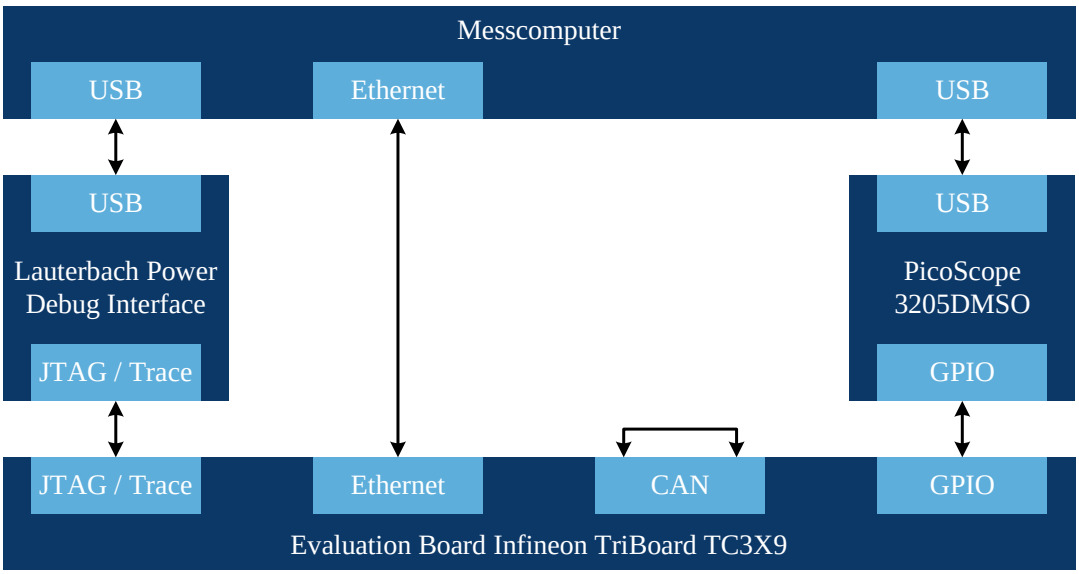

Abbildung 5.1: Grundlegender Aufbau der verwendeten Messumgebung

Die Basis des Testsystems bildet ein Evaluationsboard von der Firma Infineon für die zweite AURIX-Generation. Mittels der integrierten Joint Test Action Group (JTAG)-Schnittstelle erfolgt das Aufspielen der Testsoftware sowie das Debugging und Auslesen des *Trace*-Speichers [164]. Für die Anbindung der JTAG-Schnittstelle an den Messcomputer wird eine Debug-Hardware der Firma Lauterbach genutzt, welche wiederum mittels Universal Serial Bus (USB) angeschlossen wird [165]. Da moderne Steuergeräte häufig in einem Verbund eingesetzt werden, ist es essentiell, dass Kommunikationsschnittstellen für ein realistisches Lastszenario zur Verfügung stehen. Aus diesem Grund werden im Rahmen dieses Messaufbaus Ethernet und Controller Area Network Flexible Data-Rate (CAN-FD) angewendet, wobei die Ethernet-Schnittstelle direkt mit dem Messcomputer verbunden und zur Übertragung von Laufzeitmessungen genutzt wird [11]. Im Gegensatz dazu erfolgt bei der CAN-FD-Schnittstelle nur die Simulation einer Netzwerklast indem die *Loopback*-Funktionalität des verwendeten Mikrocontrollers Anwendung findet. Der Grund für die Auswahl dieser beiden Kommunikationssysteme ist darin begründet, dass diese im Fahrzeug aktuell vorherrschend sind [111]. Wichtig ist dabei zu beachten, dass im automobilen Umfeld für die physikalische Schicht der Ethernet-Verbindung *BroadR-Reach* genutzt wird, welches jedoch keinen Einfluss auf die darüber liegenden Kommunikationsprotokolle und damit auf die implementierte Software hat. Zur Verifizierung der Laufzeitmessungen und zur Simulation von GPIO-Interaktionen werden zusätzlich digitale Ausgänge geschaltet und mit einem Oszilloskop erfasst. Zu diesem Zweck werden die digitalen Eingänge mit Hilfe des Logikanalysators des PicoScope 3205DMSO ausgewertet, welches via USB mit dem Messcomputer verbunden ist. Eine detaillierte Übersicht über die im Testsystem verwendeten Komponenten ist in Tabelle 5.2 dargestellt.

Tabelle 5.2: Definition der verwendeten Messumgebung [166] [165]

Verwendung	**Hardware**	**Software**
Evaluierungsboard	Infineon TriBoard TC3X9	-
Debugging/Tracing	Lauterbach Power Debug Interface	Trace32 V2020-02
GPIO Messung	PicoScope 3205DMSO	PicoScope 6
Messcomputer	HP EliteBook 850 G6	Windows 10 Enterprise 22H2

Zur Validierung des in Kapitel 4 vorgestellten Konzepts ist es essentiell, die Laufzeit des Gesamtsystems sowie die Rechenzeit der einzelnen Funktionen zu ermitteln. Während mittels der Laufzeitmessung des Gesamtsystems der Nachweis für die korrekte Funktionsweise des in dieser Arbeit entwickelten Ansatzes erbracht werden soll, wird die Rechenzeit der einzelnen Funktionen für die eigentliche Priorisierung benötigt.

Zur Ermittlung der Laufzeit werden drei verschiedene Verfahren genutzt, welche sich gegenseitig plausibilisieren. Als primäres Messverfahren kommt dabei ein *Trace* zum Einsatz, welcher die Laufzeit der Funktionen taktgenau erfasst und die Zeitstempel im entsprechenden Speicher ablegt. Zu diesem Zweck wird die *Trace*-Logik des genutzten Infineon AURIX so konfiguriert, dass diese lediglich den Einsprungs-

sowie den Austrittszeitpunkt einer Funktion erfasst, was die zu ermittelnden Datenmengen im Vergleich zu einem vollständigen *Trace* deutlich reduziert. Der Vorteil eines *Traces* besteht dabei in dem non-intrusiven Messverfahren, wodurch es zu keiner Beeinflussung der Laufzeit sowie der Speicherzugriffsmuster kommt [167] [168] [169]. Nachteilig hingegen ist die Notwendigkeit einer *Trace*-Logik sowie der dazugehörigen Schnittstelle im Mikrocontroller. Aufgrund der enormen Komplexität und des zusätzlichen Bedarfs eines *Trace*-Speichers verfügen nur wenige Mikrocontrollerserien über die entsprechende Funktionalität. Beispielsweise nutzt die Infineon AURIX-Mikrocontrollerfamilie als *Trace*-Schnittstelle das Aurora-Interface. Dabei ist jedoch zu beachten, dass nur die sogenannten *Emulation Devices* über diese Logik verfügen, wohingegen alle anderen Derivate lediglich ein Debug-Interface zur Verfügung stellen [51]. Zur Plausibilisierung der primären Messung wird zusätzlich eine software-basierte Laufzeiterfassung implementiert, welche zum Start sowie nach der Beendigung einer Task einen Zeitstempel erzeugt. Diese Daten werden im Anschluss mittels der Ethernet-Schnittstelle an den Messcomputer übertragen und dort ausgewertet. Der Vorteil des software-basierten Ansatzes besteht in der leichten Portierbarkeit auf andere Mikrocontrollerfamilien, da keine spezielle Hardware-Unterstützung erforderlich ist. Nachteilig ist hingegen der Umstand, dass das sekundäre Messverfahren zusätzliche Rechenzeit benötigt und die Speicherzugriffsmuster geringfügig beeinflusst. Durch eine optimierte Implementierung sollen diese Effekte möglichst gering gehalten werden [170]. Für die Messung oder Validierung bestimmter Ablaufmuster können zusätzlich noch digitale Ausgänge geschaltet und mit einem Logikanalysators ausgewertet werden. Analog zu der software-basierten Laufzeitmessung ergeben sich durch diesen Ansatz ebenfalls Veränderungen in der Laufzeit sowie in den Speicherzugriffsmustern. Diese Effekte können mittels einer effizienten Implementierung gleichermaßen reduziert werden.

5.2 Testsystem

In dem folgenden Abschnitt wird das verwendete Testsystem mit allen relevanten Details beschrieben, wobei ein besonderer Fokus auf die verwendete Hardware-Plattform sowie die implementierte Software gelegt wird.

5.2.1 Hardware

Als Plattform für die Testdurchführung wird ein Infineon AURIX SAK-TC399XE 256F300S BC genutzt, welcher für sicherheitskritische Anwendungen mit einer harten Echtzeitanforderung entwickelt wurde. Wie in der Abbildung 5.3 zu sehen ist, bietet der TC399 insgesamt sechs Prozessorkerne, welche auf der proprietären TriCore-Architektur in der Version 1.6.2 basieren. Die TriCore-Architektur nutzt eine 32 Bit Registerbreite, eine für Echtzeitsysteme typische In-Order-Ausführung sowie ein superskalares Design mit je einer Pipeline für Integer-, Speicher- und Schleifenoperationen [27]. Zusätzlich integriert Infineon eine IEEE 754-kompatible Floating Point Unit (FPU) als Co-Prozessor in die Kerne zur effizienten Berechnung von Gleitkom-

mazahlen [171]. Da der AURIX grundlegend einer modifizierten Harvard-Architektur entspricht, besitzt jeder Kern ein separates Interface für den Code- sowie für den Datenzugriff. Diese werden als *Program Memory Interface* und *Data Memory Interface* bezeichnet und beinhalten jeweils zwei lokale Speicher in Form eines Scratchpads sowie eines Zwei-Wege-Assoziativen-Caches mit einer Cacheline-Größe von 256 Bit. Dabei nutzt Infineon für die Caches des AURIX als Ersetzungsstrategie den LRU-Ansatz ohne die Möglichkeit von Cache-Locks, weswegen die Funktionalität der Caches für die auszuführende Software vollständig transparent ist. Zur Reduzierung der Schreibzugriffe auf die globalen RAM-Speicher verwenden die Daten-Caches das *Write-Back*-Verfahren, wodurch die aktualisierten Daten erst dann zurückgeschrieben werden, sobald die Cache-Line anderweitig benötigt wird. Dieser Umstand ist besonders bei der Allokation von Variablen zur Intercore-Kommunikation zu beachten, da es aufgrund der gewählten Schreibstrategie zur Laufzeit zu inkonsistenten Daten kommen kann. Die weiteren Unterschiede zwischen den beiden lokalen Speichertypen sind in Abschnitt 3.1.5.1 detailliert beschrieben [51].

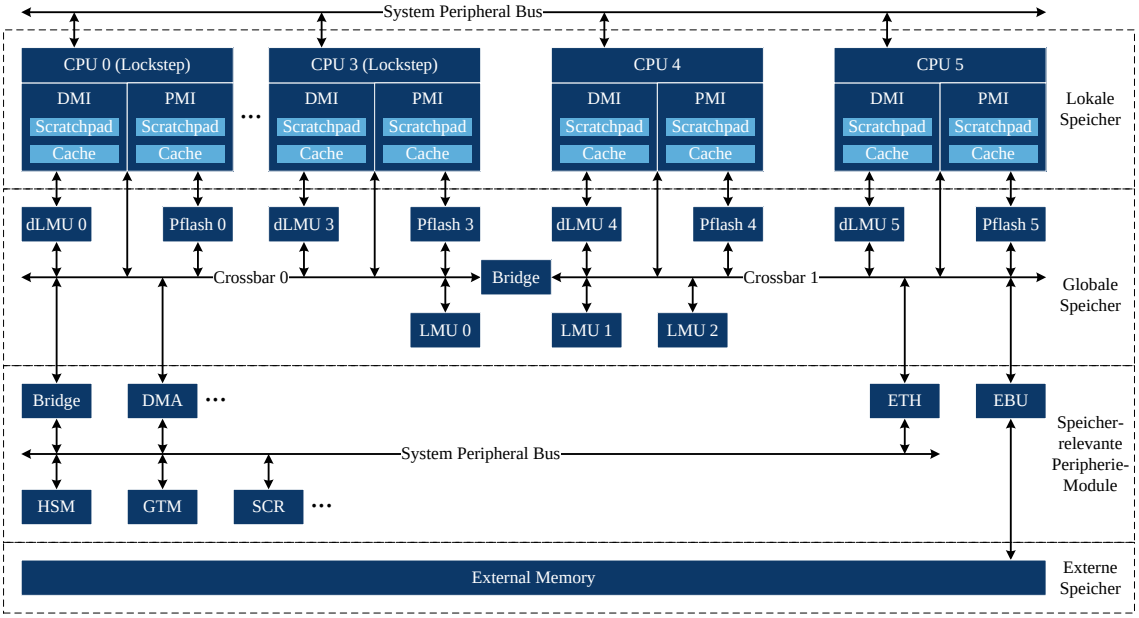

Abbildung 5.3: Grundlegendes Speicherlayout eines Infineon AURIX TC399 mit ausgewählten Peripherien [51]

Neben den lokalen Speichern bietet die AURIX-Familie noch zwei globale Speichertypen an, welche von den Prozessorkernen gemeinsam genutzt werden können. Zu den globalen Speichern zählen sowohl der Flash, welchen Infineon als *Pflash* bezeichnet, sowie der RAM-Speicher, welcher im AURIX als Local Memory Unit (LMU) beziehungsweise Distributed Local Memory Unit (dLMU) definiert ist. Wie in der Abbildung 5.3 zu sehen ist, sind alle globalen Speicher in Bänken organisiert, wodurch die Effekte von konkurrierenden Zugriffen reduziert werden sollen. Zusätzlich besitzt jeder Prozessorkern einen direkten Zugriff auf einen Teil des *Pflash* sowie der dLMU. Diese als *Direct Path* bezeichneten Schnittstellen bieten im Gegensatz

zu dem Zugriff über die Crossbar eine höhere Bandbreite sowie die Möglichkeit zur Entlastung der entsprechenden Crossbar-Schnittstelle. Wichtig ist dabei der Umstand, dass nur jeweils ein Interface des TriCore-Prozessorkerns den direkten Zugriff nutzen kann. So besitzt lediglich das *Program Memory Interface* eine Anbindung an den *Direct Path* des *Pflash*, währenddessen nur das *Data Memory Interface* direkt auf die entsprechende dLMU zugreifen kann. Sollten ein Datenzugriff auf den *Pflash* erforderlich sein oder ein Programmzugriff auf die dLMU, muss die Anbindung über die Crossbar genutzt werden. Neben der dLMU existieren noch drei LMU-Bänke, auf welche kein Prozessorkern einen direkten Zugriff hat, weswegen hier die Anbindung ausschließlich über die Crossbar erfolgen kann. Die Größe der lokalen und globalen Speicher im System sowie die verwendete Speichertechnologie kann der Tabelle 5.4 entnommen werden [51].

Neben den internen Speichern kann der Infineon AURIX zusätzlich externen Speicher über die sogenannte External Bus Unit (EBU) anbinden. Da die EBU nur eine deutlich geringere Bandbreite sowie Zugriffsgeschwindigkeit im Vergleich zu den internen Speichern bietet, erfolgt der Einsatz von externem Speicher nur in laufzeitunkritischen Anwendungen [51].

Zur Anbindung der globalen Speicher im System sowie zur Kommunikation der Prozessorkerne untereinander nutzt der AURIX zwei Crossbars, welche über eine Bridge miteinander verbunden sind. Hierbei ist zu beachten, dass Infineon eine asymmetrische Aufteilung der Prozessorkerne auf die Crossbar Domänen vornimmt, weswegen an Crossbar 0 vier Kerne und über Crossbar 1 lediglich zwei Kerne angebunden sind. Diese Designentscheidung ist darauf zurückzuführen, dass alle Prozessorkerne an der Crossbar 0 über einen dedizierten *Lockstep*-Kern verfügen, weswegen dieser Bereich als Safety-Domain betrachtet werden kann. Der Nachteil dieser Umsetzung besteht in der zusätzlichen Verzögerung, welche bei der Kommunikation zwischen Teilnehmern unterschiedlicher Crossbar-Domains entsteht [51].

Neben den Prozessorkernen haben zusätzlich verschiedene Hardware-Beschleuniger einen Einfluss auf die optimierte Speicherallokation. Zu den speicherrelevanten Peripherieeinheiten zählen das HSM, die GTM, das Ethernet-Module, der Standby Controller (SCR) sowie der DMA-Controller. Mittels des HSM wird im AURIX eine sichere Laufzeitumgebung für kryptografische Anwendungen zur Verfügung gestellt. Das HSM besitzt zwar einen separaten RAM-Speicher, jedoch keinen eigenen Flash. Aus diesem Grund wird ein Teil des *Pflash 0* für die Software des HSM reserviert und vor unbefugtem Zugriff durch die MPU geschützt. Die Folge dieses Ansatzes ist, dass durch das HSM ebenfalls konkurrierende Zugriffe auf die globalen Speicher entstehen können. Neben dem HSM bietet der AURIX noch zwei weitere Co-Prozessoren in Form der GTM sowie des SCR. Die GTM ist ein komplexes Timer-Modul, welches fünf separate Recheneinheiten zur Kalkulation von aufwendigen PWM-Mustern bietet. Zur Reduzierung des Energieverbrauchs in Ruhephasen können die sechs TriCore-Kerne abgeschaltet werden. Damit der AURIX trotzdem auf Reaktivierungsbotschaften reagieren kann, hat Infineon einen zusätzlichen Standby-Prozessorkern integriert, welcher Zugriff auf die Kommunikationsschnittstellen hat. Dieser als SCR bezeichnete Co-Prozessor besitzt, wie die GTM, keinen separaten Flash-Speicher, weswegen hier ebenfalls ein Zugriff auf den Host-Speicher

erforderlich ist. Für das Kopieren von großen Datenmengen und zur Entlastung der TriCore-Kerne besitzt der AURIX einen separaten DMA-Controller sowie ein Ethernet-Modul mit integriertem DMA-Controller. Beide DMAs haben dabei ebenfalls einen Zugriff auf die globalen und lokalen Speicher des Systems und können daher für potentielle Wartezyklen aufgrund von konkurrierenden Zugriffen verantwortlich sein. Aus diesem Grund ist es essentiell, die speicherrelevanten Hardware-Beschleuniger bei einer optimierten Speicherallokation mitzuberücksichtigen [51].

Zur Isolierung von Funktionalitäten unterschiedlicher Kritikalität bietet die AURIX-Mikrocontrollerfamilie einen zweistufigen Speicherschutz. Mittels der systemweiten MPU können bestimmte Crossbar-Teilnehmer von dem Zugriff auf geschützte Speicherbereiche ausgeschlossen werden. Zu diesem Zweck kontrolliert die Arbitrierungseinheit der entsprechenden Crossbar-Slaves bei jedem Zugriff die Master-ID des zugreifenden Prozessorkerns oder Hardware-Beschleunigers. Zusätzlich wird die Zugriffsart überwacht und, je nach eingestellter MPU-Konfiguration, erfolgt im Anschluss eine Freigabe oder Ablehnung des entsprechenden Lese- und Schreibzugriffs. Mittels der MPU-Konfiguration lassen sich, je nach Speicher, unterschiedlich große Speicherbereiche schützen, wodurch bei der AURIX-Mikrocontrollerfamilie die Granularität des Speicherschutzes vom entsprechenden Speichertyp abhängt. Eine Übersicht über die mittels der systemweiten MPU schützbaren Speicher sowie der dazugehörigen Granularität können der Tabelle 5.4 entnommen werden. Für die Ausführung von Funktionalitäten unterschiedlicher Kritikalität auf einem Kern besitzen die AURIX-Mikrocontroller zusätzlich eine prozessorkernlokale MPU. Mit Hilfe dieser MPU können verschiedene Konfigurationssätze hinterlegt werden, welche bei einem Kontextswitch zwischen Funktionalitäten unterschiedlicher Kritikalitätslevel umgeschaltet werden können. Dabei unterstützt jeder Prozessorkern 18 Sektionen für die Daten sowie 10 Sektionen für den auszuführenden Programm-Code. Bei der Zugriffsart kann zwischen einem lesenden, einem schreibenden oder einem ausführenden Zugriff unterschieden und entsprechend limitiert werden. Generell lässt sich die Granularität der schützbaren Speicherbereiche flexibel einstellen, da diese mittels einer oberen- und unteren 32 Bit langen Grenzadresse definiert werden. Lediglich die unteren 5 Bits der Grenzadressen sind fest, wodurch sich eine minimale schützbare Größe von 32 Byte ergibt. Die einzige Ausnahme bildet der *Pflash*, bei welchem die Sektoren eine feste Größe von 16 KB haben [51] [171].

5.2.2 Software

Für den Aufbau eines realistischen Lastszenarios für automobile Anwendungen in sicherheitskritischen Systemen wird eine Test-Software in der Programmiersprache C implementiert, welche sich grundsätzlich an dem AUTOSAR Classic Standard orientiert, vorwiegend Open Source-Lösungen nutzt und deren Aufbau in Abbildung 5.5 dargestellt ist.

Tabelle 5.4: Übersicht der Speicher des Infineon AURIX TC399

Bezeichnung	**Speicher-technologie**	**Größe**	**Granularität der systemweiten MPU**
Daten-Scratchpad (CPU0-CPU1)	SRAM	240 KB	32 Byte (8 Sektionen)
Daten-Scratchpad (CPU2-CPU5)	SRAM	96 KB	32 Byte (8 Sektionen)
Daten-Cache	SRAM	16 KB	-
Programm-Scratchpad	SRAM	64 KB	32 Byte (8 Sektionen)
Programm-Cache	SRAM	32 KB	-
dLMUx	SRAM	64 KB	32 Byte (8 Sektionen)
LMUx	SRAM	256 KB	32 Byte (16 Sektionen)
Pflash (0-4)	Flash	3072 KB	16 KB (192 Sektionen)
Pflash (5)	Flash	1024 KB	16 KB (64 Sektionen)

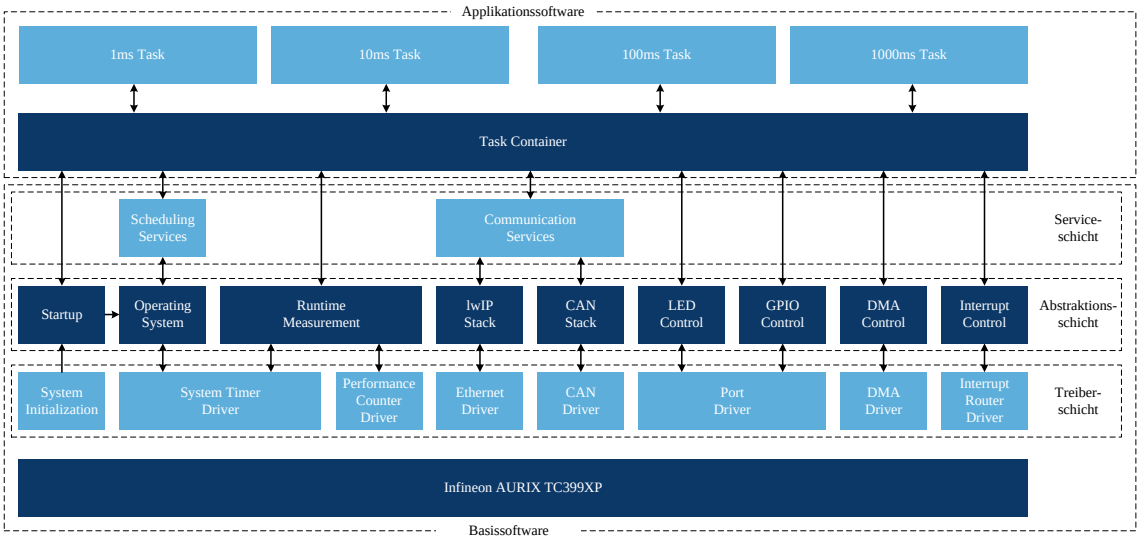

Abbildung 5.5: Grundlegende Architektur der implementierten Testsoftware

5.2.2.1 Basissoftware

Analog zu dem AUTOSAR Classic Standard verfügt die implementierte Test-Software über eine klassische Teilung zwischen Basis- und Applikationssoftware. Die Basissoftware gliedert sich dabei in drei Schichten, welche eine Treiber-, eine Abstraktions- und eine Serviceschicht beinhalten. Für die Umsetzung der Treiberschicht wird auf die Infineon Low Level Drivers (iLLD) zurückgegriffen, welche eine kostenlose und quelloffene Treiberbibliothek von Infineon für die AURIX-Familie darstellt. Für jedes Peripheriemodul des AURIX gibt es ein separates Treibermodul, welches die jeweiligen Funktionalitäten bereitstellt. Die Struktur ist dabei an den MCAL von AUTOSAR angelehnt [172]. Mittels der Abstraktionsschicht werden die benötigten Funktionalitäten für die Zeitmessung, die Kommunikationsprotokolle sowie die vereinfachte Ansteuerung der genutzten Peripherien realisiert. Durch die Serviceschicht erfolgt die Bereitstellung der Zeitbasis für die Task Container sowie das zur Verfügung stellen der empfangenen Botschaften als Signale. Die Umsetzung der

verschiedenen Lastszenarien erfolgt dabei durch die Applikationsschicht, welche eine Vielzahl unterschiedlicher Benchmarks enthält. Das Ziel ist dabei möglichst verschiedene Lastprofile abzubilden, um das in Kapitel 4 vorgestellte Konzept umfangreich zu evaluieren. Die Verbindung der Basis- mit der Applikationssoftware erfolgt mittels der Task Container, welche die verschiedenen Zeitscheiben auf den Prozessorkernen darstellen und die Interaktion zwischen den Tasks selbst, den Prozessorkernen untereinander sowie mit der Basissoftware als Signale realisieren [107].

Neben der horizontalen Unterteilung, je nach Abstraktionsgrad von der verwendeten Hardware, wird die Basissoftware vertikal zusätzlich in die unterschiedlichen Funktionalitäten separiert. Dazu gehört zum einen die gesamte Startup-Routine, welche die Grundinitialisierung des Mikrocontrollers vornimmt, sowie das Echtzeitbetriebssystem, welches das Scheduling der Task Container übernimmt. Als Betriebssystem kommt das Erika Enterprise Real-Time Operating System (RTOS) in der Version 3.0 zum Einsatz, welches dem AUTOSAR Classic Standard entspricht, als Open Source-Lösung zur Verfügung steht und bereits für die Infineon AURIX TC3xx-Serie umgesetzt ist.

Zur Bewertung der korrekten Funktionsweise des entwickelten Konzepts ist die Zeitmessung ein essentieller Bestandteil, weswegen diese sowohl auf Basis der System Timer des AURIX als auch mit Hilfe der Performance Counter umgesetzt ist. Für die Zeitmessung auf Task-Ebene werden die System Timer verwendet, da diese einen 64 Bit großen Zähler nutzen, welcher lange Messperioden ermöglicht und als einheitliche Zeitbasis im System fungiert. Für die taktgenaue Bestimmung von Laufzeiten einzelner Funktionalitäten werden hingegen die Performance Counter genutzt, welche als Teil des Prozessorkerns die präziseste Messung ermöglichen, jedoch in ihrer Messdauer beschränkt sind [171].

Zur Bereitstellung der benötigten Kommunikationsschnittstellen werden zwei Treibermodule für CAN-FD und Ethernet verwendet. Mittels dazugehöriger Abstraktionsmodule werden die Protokolle für die verschiedenen Kommunikationssysteme umgesetzt. Für die Bereitstellung der Ethernet-Protokolle wird die quelloffene Bibliothek lightweight IP (lwIP) genutzt, welche Infineon bereits für den AURIX portiert hat und als entsprechendes Projekt zur Verfügung stellt. Mittels des letzten Moduls der Communication Services werden die Botschaftsinhalte als unabhängige Signale bereitgestellt, wodurch die Applikationsschicht von dem eigentlichen Kommunikationssystem und den dazugehörigen Protokollen vollständig entkoppelt wird. Zur Ansteuerung der GPIOs, welche für die Light-Emitting Diodes (LEDs) und die Plausibilisierung der Zeitmessung genutzt werden, wird ein ähnlicher Aufbau wie bei den bisherigen Funktionalitäten genutzt. Durch ein entsprechendes Treibermodul wird das Port-Module des AURIX abstrahiert, wodurch die Ansteuerung der LEDs und GPIOs über hardware-unabhängige Schnittstellen ermöglicht wird. Zur Simulation von praxisnahen Speicherlasten wird zusätzlich der DMA-Controller mit eingebunden, mit welchem sowohl synchrone als auch asynchrone Kopiervorgänge zwischen den Speichern umgesetzt werden. Zur Einbindung des DMA-Controllers wird analog zu den anderen Umsetzungen ein Treibermodul auf Basis der iLLD verwendet und der Zugriff über ein Kontrollmodul abstrahiert.

Für die Generierung von asynchronen Events wird eine Kombination aus zy-

klischen Interrupts und der General Purpose Service Requests (GPSRs) genutzt, welche mittels eines entsprechenden Treiber- sowie des dazugehörigen Kontrollmoduls angesprochen werden. Mittels der GPSRs können in der AURIX-Familie von Infineon Software-Interrupts ausgelöst werden, welche im Anschluss von einem Prozessorkern verarbeitet werden. In der aktuellen Umsetzung, welche in Abbildung 5.6 dargestellt ist, wird mittels eines konfigurierbaren Timer-Interrupts zyklisch eine Interrupt Service Routine (ISR) aufgerufen, in welcher mit Hilfe eines Pseudozufallszahlengenerators einer von acht GPSRs ausgewählt wird. Durch die Aktivierung des entsprechenden GPSR erfolgt die Aktivierung des dazugehörigen Interrupts. Mittels der aktuellen Implementierung wird pro Prozessorkern ein Timer genutzt, welcher wiederum bis zu acht GPSRs ansprechen kann. Durch die Einstellung der Frequenz des Timer-Interrupts kann die maximale Aufrufhäufigkeit sowie die Granularität der jeweiligen GPSRs definiert und durch den Pseudozufallszahlengenerators eine gewisse Asynchronität realisiert werden. Je nach benötigtem Lastszenario können dabei alle Freiheitsgerade, wie beispielsweise die Aufrufhäufigkeit, die Anzahl der GPSRs pro Prozessorkern sowie die Laufzeit der jeweiligen ISR, definiert werden [51].

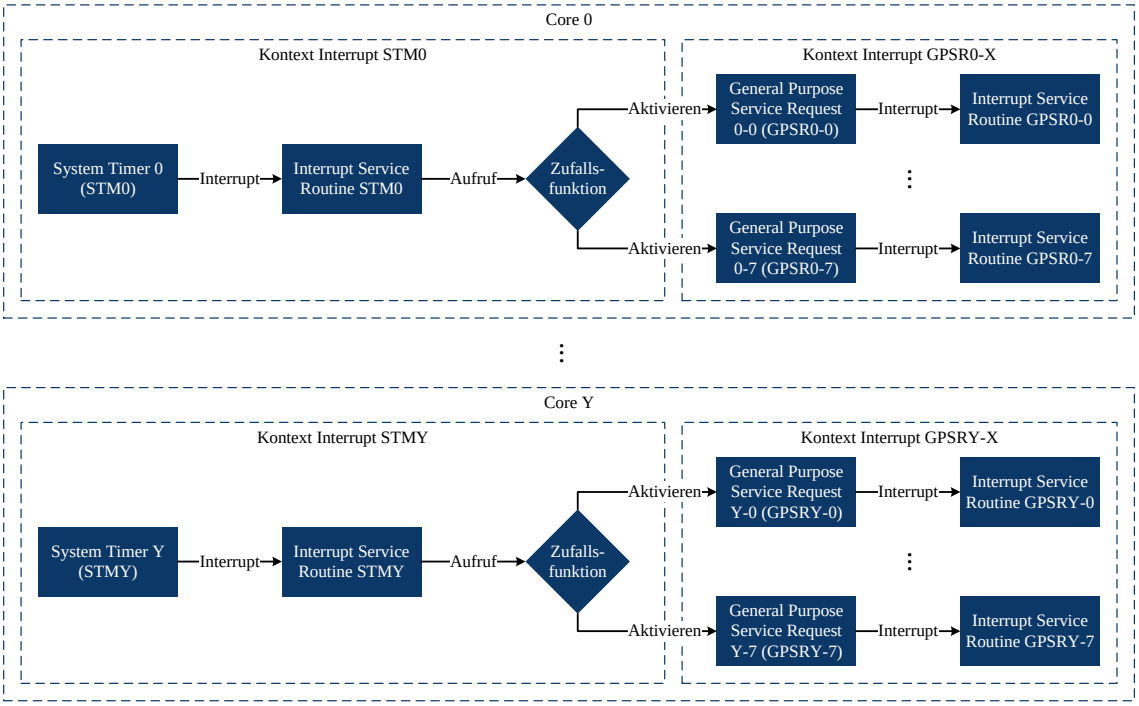

Abbildung 5.6: Grundlegende Funktionsweise der Generierung der asynchronen Events

In der Tabelle 5.7 sind alle genutzten Basissoftware-Bestandteile sowie die verwendete Version, welche im Rahmen der Testsoftware zum Einsatz kommt, aufgeschlüsselt.

Tabelle 5.7: Übersicht der genutzten Basissoftware-Bestandteile für die Software der Testumgebung [172]

Bezeichnung	Verwendung	Version
Erika Enterprise	Echtzeitbetriebssystem	V3.0
Infineon Low Level Driver	Treiber	V1.0.1.16.0
lightweight IP	Ethernet Stack	2.1.2 Stable

5.2.2.2 Applikationssoftware

Für die Umsetzung eines realistischen und reproduzierbaren Lastszenarios werden verschiedene Benchmarks für eingebettete System als Applikationen genutzt. Durch die unterschiedliche Ausrichtung der verschiedenen Benchmarks kann ein breites Portfolio an Anwendungsfällen simuliert werden. In dem folgenden Abschnitt werden die gewählten Benchmarks kurz beschrieben und die Auswahl entsprechend begründet.

Tasking Dhrystone Der Dhrystone Benchmark ist schon seit mehreren Jahrzehnten ein Lastszenario, mit welchem die Leistungsfähigkeit verschiedener Mikrocontroller und Mikroprozessoren miteinander verglichen werden soll. Dabei liegt der Fokus auf Ganzzahlberechnungen und Zeichenkettenoperationen. Im Rahmen dieser Arbeit wird auf eine Umsetzung der Firma Tasking für die AURIX Mikrocontrollerfamilie zurückgegriffen, welche als Beispielcode mit dem verwendeten Compiler mitgeliefert wird.

EEMBC CoreMark Der CoreMark von EEMBC wird in diversen Publikationen als Lastszenario verwendet, da eine Vielzahl von gängigen Aufgaben von eingebetteten Systemen dargestellt werden. Dazu gehören Matrixoperationen, verschiedene Anwendungen mit verketteten Listen sowie Zustandsmaschinen. Nachteilig ist hingegen, dass der CoreMark ausschließlich Ganzzahloperationen nutzt und keine Berechnung auf Basis von Gleitkommazahlen. Aufgrund der Bereitstellung des Programm-Codes als Open-Source Lösung findet der CoreMark trotz der genannten Limitierung breiten Einsatz, wodurch die Ergebnisse dieser Arbeit mit anderen Fachpublikationen verglichen werden können [173].

Embench IoT Mit der Benchmark-Suite Embench IoT werden insgesamt 22 verschiedene Lastszenarien bereitgestellt, welche ein breites Spektrum an möglichen Aufgaben für eingebettete Systeme darstellen, wie beispielsweise komplexe mathematische Berechnungen mit Ganzzahl- und Gleitkommaarithmetik, kryptografische Algorithmen, Zustandsmaschinen und Sortieralgorithmen. Zusätzlich ist der Embench IoT als Open-Source Lösung verfügbar und kann durch eine geringe Hardware-Abhängigkeit mit geringen Anpassungen in bestehende Testsysteme integriert werden, weswegen ein Einsatz im Rahmen dieser Arbeit erfolgt.

Infineon Cyclic Redundancy Check (CRC) Bibliothek Im AUTOSAR-Standard erfolgt die Abstraktion des Mikrocontrollers anhand des MCAL, wie in Abschnitt 3.2 beschrieben. Neben den eigentlichen Treibermodulen zur Ansteuerung der im Mikrocontroller integrierten Peripherie liefern die Hersteller häufig Software-Bibliotheken mit, welche für die genutzte Hardware-Plattform optimiert sind. Zu die-

sen Bibliotheken gehört unter anderem die Berechnung von CRCs, welche in sicherheitskritischen Systemen von großer Bedeutung sind, um die Integrität von Speicherinhalten zyklisch zu überprüfen. Infineon bietet für die AURIX Mikrocontrollerfamilie einen passenden MCAL an, welcher ebenfalls eine Bibliothek zur effizienten Berechnung von CRCs beinhaltet. Zur Erstellung eines realistischen Lastszenarios mit einer integrierten zyklischen Speicherprüfung wird daher auf die entsprechende Bibliothek von Infineon zurückgegriffen. Im Gegensatz zu den anderen Benchmarks steht der MCAL des AURIX nicht als Open-Source Lösung zur Verfügung [107] [174] [175].

Infineon Bitmanipulation Bibliothek Analog zur Berechnung von CRCs bietet der MCAL des AURIX eine Bibliothek zur effizienten Verwendung von Bitoperationen auf unterschiedlich großen Datentypen. Hierzu nutzt die entsprechende Bibliothek spezielle Instruktionen, welche Infineon mit der zweiten AURIX-Generation eingeführt hat. Da die Bibliothek zur Bitmanipulation ein Teil des MCALs ist, steht der Programm-Code ebenfalls nicht quelloffen zur Verfügung [107] [174] [175].

IAV quantumSAR Mit der kommerziellen Verfügbarkeit von Quantencomputern können mathematische Probleme, auf denen die moderne asymmetrische Kryptografie basiert, effizient gelöst werden, wodurch die notwendige Sicherheit nicht mehr gegeben ist [176]. Um auf diese Bedrohungslage rechtzeitig zu reagieren, hat das National Institute of Standards and Technology (NIST) bereits 2017 mit einem Auswahlprozess begonnen, um kryptografische Algorithmen zu standardisieren, welche im Zeitalter des Quantencomputers die notwendige Sicherheit bieten, aber weiterhin auf klassischen Mikrocontrollern beziehungsweise Mikroprozessoren berechnet werden können [177]. Nach der dritten Selektionsrunde hat das NIST vier Algorithmen ausgewählt, welche in der Open-Source Bibliothek IAV quantumSAR in der Programmiersprache C umgesetzt sind. Diese kryptografisch Bibliothek stellt eine Adaption der Open-Source Lösung *PQClean* dar, wobei der Fokus von IAV quantumSAR ressourcenlimitierte Systeme sind. Mit Hilfe dieser Bibliothek soll die Leistungsfähigkeit von eingebetteten Systemen hinsichtlich der Post-Quantum-Kryptografie evaluiert werden, um die Update-Fähigkeit im Kontext der kryptografischen Agilität zu bewerten. Da es in den aktuellen Mikrocontrollergenerationen derzeit keine Hardware-Unterstützung für die genannten Algorithmen gibt, ist eine Umsetzung in Software zwingend erforderlich, weswegen IAV quantumSAR als Lastszenario in das Testsystem mit integriert wird [178] [179] [180] [181] [182] [183] [184].

IAV CopyData Bibliothek Zur Simulation einer Intertask- und einer Intercore-Kommunikation wird auf eine selbstentwickelte Bibliothek mit dem Namen IAV CopyData zurückgegriffen, welche nicht quelloffen zur Verfügung steht. Über diese können unterschiedliche Kopierszenarien zwischen verschiedenen Speicherbereichen durchgeführt werden. Zu diesem Zweck werden diverse Buffer angelegt, welche mit Hilfe des Linker-Skripts in die entsprechenden Speicherbereiche allokiert werden. Durch verschiedene Schnittstellen können unterschiedliche Datentypen kopiert sowie die Transferrichtung definiert werden. Grundsätzlich ist diese Testbibliothek dem Konzept zur Intercore-Kommunikation aus der Quelle [3] nachempfunden.

SystemMD SipHash-2-4 Für die gesicherte Übertragung in Automobilen sieht der AUTOSAR-Standard mehrere verschiedene Message Authentication Codes (MACs) vor, welche von vielen Mikrocontrollern bereits in Hardware unterstützt werden. Lediglich der SipHash-2-4 stellt hier eine Ausnahme dar, weswegen dieser kryptografische Algorithmus bei Bedarf in Software umgesetzt werden muss. Da es für den SipHash-2-4 eine Vielzahl an Open-Source Implementierungen in der Programmiersprache C gibt, wird auf eine entsprechende Umsetzung von SystemMD zurückgegriffen [107].

Universität zu Lübeck AES-256/SHA-256 Im Rahmen einer Kooperation zur Entwicklung eines alternativen Konzepts zum klassischen HSM zwischen der IAV GmbH und der Universität zu Lübeck sind verschiedene kryptografische Algorithmen in Software umgesetzt worden. Das Ziel des sogenannten Firmware Security Module (FSM)-Konzepts besteht darin, die kryptografische Agilität von Echtzeitsystemen zu erhöhen, in dem die entsprechenden Algorithmen als update-fähige Lösung in Software umgesetzt werden. Die integrierten Hardware-Beschleuniger der Mikrocontroller bieten zwar eine höhere Performance, können jedoch nur durch einen aufwendigen Hardware-Tausch aktualisiert werden. Zum Aufzeigen dieser software-basierten Option wurden verschiedene kryptografische Algorithmen umgesetzt, wobei der Hauptfokus auf dem AES-256 und SHA-256 liegt, welche auch als Lastszenarien im Rahmen dieser Arbeit verwendet werden. Im Gegensatz zu den anderen Benchmarks stehen die beiden kryptografischen Algorithmen nicht als Open-Source Lösung zur Verfügung [41] [185].

In der Tabelle 5.8 sind alle genutzten Applikationssoftware-Bestandteile sowie die verwendete Version, welche im Rahmen der Testsoftware zum Einsatz kommt, aufgeschlüsselt.

Tabelle 5.8: Übersicht der genutzten Applikationssoftware-Bestandteile für die Software der Testumgebung [172]

Bezeichnung	**Repository**	**Version**
Tasking Dhrystone	Tasking Compiler Examples	TC63r1p2
EEMBC CoreMark	https://github.com/eembc/coremark	d5fad6b
Embench IoT	https://github.com/embench/embench-iot	54fd9a0
Infineon CRC Bibliothek	Infineon MCAL AS440 TC39x	2.0.0
Infineon Bitmanipulation Bibliothek	Infineon MCAL AS440 TC39x	2.0.0
IAV quantumSAR	https://github.com/iavofficial/IAV_quantumSAR	5a4f5e1
IAV CopyData Bibliothek	-	1.0
SystemMD SipHash-2-4	https://github.com/systemd/systemd	cf4deea
Universität zu Lübeck AES-256/SHA-256	-	1.0

Eine detaillierte Aufschlüsselung aller Testfälle aus den genutzten Benchmarks mit dem dazugehörigen Ressourcenbedarf ist im Abschnitt 7.1 hinterlegt.

Neben den bereits beschriebenen Benchmarks erfolgt zusätzlich die Bewertung und Evaluierung weiterer Benchmarks, welche aber aufgrund verschiedener Faktoren nicht genutzt werden können. Eine Übersicht dieser alternativen Lastszenarien sowie die Begründung für die Devalidierung erfolgt in dem kommenden Abschnitt.

TACLe Benchmark Bei der Open-Source Bibliothek TACLeBench handelt es sich um eine Sammlung von verschiedenen Benchmarks, welche den Fokus auf Bild-, Video- und Audioverarbeitung legen. Dabei sollen entsprechende Systeme hinsichtlich ihrer Echtzeitfähigkeit mittels der integrierten Testfälle bewertet werden können. Neben den genannten Testfällen unterstützt die TACLeBench-Suite zusätzlich noch Standardtests, welche aber durch eine Vielzahl von bereits ausgewählten Benchmarks abgedeckt werden. Da die Verarbeitung von Bild-, Video- und Audiosignalen kein Anwendungsszenario für eingebettete Mehrkernsysteme mit einer harten Echtzeitanforderung ist, wird diese Benchmark-Sammlung nicht als Lastszenario im Rahmen dieser Arbeit eingesetzt [186].

PapaBench Die Benchmark-Suite PapaBench steht ebenfalls als Open-Source Lösung bereit und soll auf Basis einer Drohnensteuerung ein realistisches Anwendungsszenario für Echtzeitsysteme abbilden. Mit einer Kombination aus zyklischen Aufgaben und Events in Form von Interrupts sollen sowohl synchrone als auch asynchrone Lastszenarien dargestellt werden. Derzeit existiert lediglich eine Umsetzung für den PapaBench für die ATMEL AVR-Familie, welche zudem eine starke Hardware-Abhängigkeit aufweist. Das Auflösen dieser Hardware-Abhängigkeiten hätte eine umfassende Anpassung der Programm-Code-Basis zur Folge, wodurch die Ergebnisse nicht mehr vergleichbar wären. Aus diesem Grund wird auf eine Nutzung des PapaBench als Lastszenario für das beschriebene Testsystem verzichtet [187].

EMSBench Mit dem EMSBench hat die Universität Augsburg eine Benchmark-Suite erstellt, welche als Lastszenario auf einer Motorsteuerung für einen Verbrennungsmotor setzt. Analog zum PapaBench bietet auch der EMSBench sowohl synchrone als auch asynchrone Lasten, wodurch ein breites Spektrum an zeitlichem Verhalten abgebildet werden soll. Leider weist auch der EMSBench eine starke Hardware-Abhängigkeit auf, da auf eine Vielzahl von Peripherien, wie beispielsweise ADCs, PWMs oder Ports, zugegriffen wird. Die Auflösung der entsprechenden Hardware-Schnittstellen zum ST STM32F4 würde ebenfalls eine umfassende Anpassung der Codebasis erfordern, weswegen auf eine Nutzung im Kontext dieser Arbeit verzichtet wird [188].

Embench RT Die Lastszenarien des Embench RT sind ein Teil der Embench-Benchmark Suite, zu welcher auch der genutzte Embench IoT gehört. Im Gegensatz zum Embench IoT hat der Embench RT eine deutlich stärkere Abhängigkeit zur genutzten Hardware, da die Laufzeiten von Interrupts dediziert gemessen werden sollen. Des Weiteren existiert bisher nur eine Portierung für die RISC-V-Architektur, weswegen eine Umsetzung für die TriCore-Architektur nur mit größeren Modifikationen des Programm-Codes möglich ist. Da im Rahmen des implementierten Testsystems bereits eine umfassende Zeitmessung für synchrone als auch asynchrone Rechenlasten umgesetzt ist, wird auf eine Portierung des

Embench RT verzichtet.

Daphne Die Benchmark-Suite DAPHNE der Technischen Universität Darmstadt ist speziell für automobile Systeme entwickelt, wobei ein Fokus auf die parallele Verarbeitung von Lasten liegt. Analog zu den bisherigen Benchmarks ist DAPHNE ebenfalls als Open-Source Lösung verfügbar, setzt bei der Umsetzung aber auf Frameworks zur parallelen Verarbeitung, wie beispielsweise Open Multi-Processing (OpenMP) oder Compute Unified Device Architecture (CUDA), welche auf Mikrocontrollern nicht zur Verfügung stehen. Aus diesem Grund ist ein Einsatz des im Rahmen dieser Arbeit entwickelten Testsystem nicht möglich. [189]

EEMBC CoreMark Pro Der Benchmark CoreMark Pro von EEMBC stellt eine erweiterte Fassung des CoreMarks dar, welcher als Lastszenario genutzt wird. Im Gegensatz zur genutzten Variante erweitert der CoreMark Pro das Spektrum an Lastszenarien um Zugriffe auf Dateisysteme und die Bildverarbeitung. Diese zusätzlichen Testfälle stellen jedoch keine klassischen Lasten auf eingebetteten Mehrkernsystemen mit einer harten Echtzeitanforderung dar, weswegen im Rahmen dieser Arbeit auf eine Nutzung verzichtet wird.

EEMBC MLPer Tiny Bei der Benchmark-Suite MLPer Tiny von EEMBC handelt es sich um eine Sammlung von Testfällen zur Bewertung von Systemen hinsichtlich ihrer Leistung beim maschinellen Lernen. Aus diesem Grund werden vorwiegend Lastszenarien genutzt, bei welcher die Klassifizierung von Bildern im Vordergrund steht. Da dies ebenfalls keine klassischen Aufgaben eines Echtzeitsystems in sicherheitskritischen Anwendungen ist, wird auf eine Integration in das entwickelte Testsystem verzichtet [190].

EEMBC SecureMark Das Ziel des SecureMarks ist die Bewertung von Systemen hinsichtlich ihrer Leistung bei kryptografischen Berechnungen, weswegen eine Vielzahl unterschiedlicher Algorithmen vermessen wird. Im Gegensatz zu den anderen Benchmark-Suiten von EEMBC werden die entsprechenden Algorithmen nicht direkt als Teil der Codebasis mitgeliefert, sondern müssen aus anderen Bibliotheken, wie beispielsweise wolfSSL, hinzugefügt werden. Zur leichteren Integration liefert EEMBC in dem dazugehörigen Open-Source Archiv zwei Beispielumsetzungen mit. Jedoch hat sich bei der Analyse der entsprechenden Codebasis gezeigt, dass durch den Verzicht einer eigenen Implementierung die Messwerte zu anderen Systemen nur bedingt vergleichbar sind. Je nach integrierter Bibliothek für die kryptografischen Algorithmen kann das Ergebnis der Messung stark variieren. Aus diesem Grund und bedingt durch den Umstand, dass bereits eine große Anzahl von kryptografischen Algorithmen Teil des Testsystems sind, wird auf die Nutzung des SecureMarks verzichtet.

wolfSSL Embedded SSL/TLS Library Mit der Embedded SSL/TLS Library Bibliothek stellt wolfSSL eine umfassende Sammlung von kryptografischen Algorithmen bereit, welche für die Umsetzung einer sicheren Kommunikation zwischen verschiedenen Steuergeräten benötigt werden. Durch die Bereitstellung dieser Umsetzung als quelloffene Implementierung ist eine Adaption an bestehende Systeme möglich, setzt jedoch die Auflösung verschiedener Abhängigkeiten zu anderen Lösungen von wolfSSL voraus. Aufgrund der komplexen Anpassung der Code-

basis zur Auflösung der genannten Abhängigkeiten sowie des Umstands, dass bereits eine Vielzahl von kryptografischen Algorithmen Teil des Testsystems sind, wird auf einen Einsatz der Lösung von wolfSSL verzichtet.

5.2.2.3 Speicherlayout

Neben der Allokation des eigentlichen Programm-Codes und der dazugehörigen Daten werden vier zusätzliche Speicherbereiche benötigt, welche dynamisch vom Prozessorkern selbst und von der Software verwaltet werden. Infineon sieht bei der TriCore-Architektur insgesamt drei Speicherbereiche vor, welche als *USTACK*, *ISTACK* und Context Save Area (CSA) bezeichnet werden. Der *USTACK* beinhaltet den Stack-Bereich, welcher von der zyklischen Software verwendet wird. Im Gegensatz dazu wird der *ISTACK* ausschließlich im Kontext von ISRs genutzt, wodurch die Stack-Verwaltung vereinfacht wird [171]. Wichtig ist hierbei zu beachten, dass Infineon bei der TriCore-Architektur die Registerinhalte bei einem Funktionsaufruf nicht im Stack sichert, sondern separat im sogenannten CSA. Das Ziel dieses Vorgehens ist dabei, dass ein Stack-Überlauf nicht die Rücksprungadressen verändern kann, was besonders in sicherheitskritischen Anwendungen fatale Folgen haben kann. Der vierte Speicherbereich ist der sogenannte Heap, welcher von der Software zur Laufzeit für dynamische Speicherallokationen genutzt werden kann. Generell ist die dynamische Speicherallokation, wie in Abschnitt 3.3.2 beschrieben, in Systemen mit einer harten Echtzeitanforderung häufig untersagt, da die Bewertung der Echtzeitfähigkeit deutlich aufwendiger ist. Jedoch nutzen einige der Testfälle aus dem Embench IoT den Heap, weswegen dieser im Rahmen dieser Lastszenarien genutzt wird. Zugunsten eines möglichst breiten Spektrums an verschiedenen Lastszenarien wird im Rahmen dieser Arbeit eine auf die Testfälle des Embench IoT beschränkte dynamische Speicherallokation genutzt. Die Größen und physikalischen Adressen der vier beschriebenen Speicherbereiche können der Tabelle 5.9 entnommen werden [51] [171].

Wie in der Tabelle 5.9 zu sehen ist, sind sowohl CSA, *USTACK* und *ISTACK* der jeweiligen Prozessorkerne in dem dazugehörigen lokalen Daten-Scratchpad allokiert, was optimal für die Zugriffsgeschwindigkeit ist. Im Gegensatz dazu wird der Heap, welcher von allen Prozessorkernen gemeinschaftlich genutzt wird, in der LMU 0 positioniert. Die Nutzung eines prozessorkernlokalen Speichers für den Heap würde unter Umständen die Zugriffszeit auf lokale Daten durch konkurrierende Zugriffe negativ beeinflussen [2]. Die Bestimmung der benötigten Speicherkapazitäten ist im Rahmen der Inbetriebnahme des Testsystems erfolgt. Die Größe des CSA und des *ISTACK* sind dabei für alle Prozessorkerne gleich. Lediglich der *USTACK* ist bei der Central Processing Unit (CPU) 1 deutlich größer, was auf die Ausführung der Post-Quantum-Kryptografie zurückzuführen ist. Im speziellen handelt es sich dabei um den speicherintensiven Algorithmus Stateless Hash-Based Digital Signature Algorithm (SLH-DSA), welcher den Stack-Bedarf deutlich erhöht und von der CPU 1 im Kontext der Hintergrund-Task ausgeführt wird [191] [51] [171].

Neben den genannten Speicherbereichen, welche vollständig im RAM-Speicher vorgehalten werden, gibt es noch zwei weitere Sektionen, welche jedoch im ROM-

Tabelle 5.9: Übersicht der festen Speicherbereiche [51] [171]

Speicherbereich		Größe in Byte	Physikalische Startadresse
CSA	CPU 0	8192	0x70031000 (Daten-Scratchpad 0)
	CPU 1	8192	0x60029000 (Daten-Scratchpad 1)
	CPU 2	8192	0x5000d000 (Daten-Scratchpad 2)
	CPU 3	8192	0x4000d000 (Daten-Scratchpad 3)
	CPU 4	8192	0x3000d000 (Daten-Scratchpad 4)
	CPU 5	8192	0x1000d000 (Daten-Scratchpad 5)
USTACK	CPU 0	32768	0x70034000 (Daten-Scratchpad 0)
	CPU 1	65536	0x6003c000 (Daten-Scratchpad 1)
	CPU 2	32768	0x50018000 (Daten-Scratchpad 2)
	CPU 3	32768	0x40018000 (Daten-Scratchpad 3)
	CPU 4	32768	0x30018000 (Daten-Scratchpad 4)
	CPU 5	32768	0x10018000 (Daten-Scratchpad 5)
ISTACK	CPU 0	4196	0x70034000 (Daten-Scratchpad 0)
	CPU 1	4196	0x6002c000 (Daten-Scratchpad 1)
	CPU 2	4196	0x50010000 (Daten-Scratchpad 2)
	CPU 3	4196	0x40010000 (Daten-Scratchpad 3)
	CPU 4	4196	0x30010000 (Daten-Scratchpad 4)
	CPU 5	4196	0x10010000 (Daten-Scratchpad 5)
Heap	CPU x	32768	0x90040000 (LMU 0)

Speicher des Testsystems allokiert sind. Dabei handelt sich zum einen um die Interrupt Vector Table (IVT) und die Trap Vector Table (TVT). Mittels der IVT werden die Sprungadressen für die ISRs und mit Hilfe der TVT die Sprungadressen für die Trap-Behandlung gespeichert. Jeder Prozessorkern besitzt dabei eine separate IVT und TVT. Die Größe der jeweiligen Sprungtabelle sowie die dazugehörigen physikalischen Adressen können für alle Prozessorkerne der folgenden Tabelle 5.10 entnommen werden. Die Größe der IVT und der TVT sind für alle Prozessorkerne gleich und die Allokation erfolgt in dem jeweiligen Segment des *Pflash*, auf welchen die Prozessorkerne direkten Zugriff über den sogenannten *Direct Path* haben [51] [171].

5.2.2.4 Toolchain

Für die Erstellung der Testsoftware wird auf ein Framework der Firma Infineon zurückgegriffen, welches die nötigen Skripte zur Einbindung verschiedener Compiler, Linker und Debugger bereitstellt. Das sogenannte *BiFACES* stellt dabei ein Basisprojekt dar, welches durch modulare Schnittstellen für verschiedene Anwendungsfälle spezifisch erweitert werden kann und speziell für die AURIX-Mikrocontrollerfamilie bereitgestellt wird [192]. Mittels des Frameworks von Infineon erfolgt die Einbindung des verwendeten Compilers sowie Linkers von der Firma Tasking in der Version 6.3r1, welche in dieser Variante bereits die zweite AURIX-Generation mit der TriCore 1.6.2-Architektur unterstützt [162]. In der Tabelle 5.11 sind die genauen Bezeichnungen sowie die dazugehörigen Einstellungen für die genutzte Build-Umgebung detailliert beschrieben.

Tabelle 5.10: Übersicht der Sprungtabellen [51] [171]

Sprungtabelle		Größe in Byte	Physikalische Startadresse
IVT	CPU 0	8192	0x802fe000 (*Pflash 0*)
	CPU 1	8192	0x805fe000 (*Pflash 1*)
	CPU 2	8192	0x808fe000 (*Pflash 2*)
	CPU 3	8192	0x80bfe000 (*Pflash 3*)
	CPU 4	8192	0x80efe000 (*Pflash 4*)
	CPU 5	8192	0x80ffe000 (*Pflash 5*)
TVT	CPU 0	242	0x80000100 (*Pflash 0*)
	CPU 1	242	0x80300000 (*Pflash 1*)
	CPU 2	242	0x80600000 (*Pflash 2*)
	CPU 3	242	0x80900000 (*Pflash 3*)
	CPU 4	242	0x80c00000 (*Pflash 4*)
	CPU 5	242	0x80f00000 (*Pflash 5*)

Tabelle 5.11: Übersicht der genutzten Software für die Build-Umgebung zur Erstellung der Software der Testumgebung [192] [162]

Bezeichnung	Verwendung	Version	Einstellungen
Infineon *BiFACES*	Basis-Framework	V1.0.1.16.0	Tasking-Compiler
Tasking C-Compiler for TriCore	C-Compiler	6.3r1p2	–core=tc1.6.x -D__CTRI –iso=99 –language=-gcc,-strings –switch=auto –align=4 –default-near-size=1 –default-a0-size=1 –default-a1-size=1 -ONRpfceogvIlywakmsU –tradeoff=0 –compact-max-size=200 –max-call-depth=-1 –inline-max-incr=35 –inline-max-size=10 -g –misrac-version=2004 –immediate-in-code
Tasking Assembler-Compiler for TriCore	Assembler-Compiler	6.3r1p2	–list-format=Ldegilmnpqrsvwxyz –optimize=gs –debug-info=+hll -il
Tasking Linker for TriCore	Linker	6.3r1p2	-D___CPU___=tc39x –map-file -OTcXYL –core=mpe:vtc

5.2.2.5 Lastszenarien

Für die Abbildung eines breiten Spektrums an Anwendungsfällen werden auf Basis der Ergebnisse aus dem Fachartikel [193] insgesamt drei Lastszenarien definiert, welche sich hinsichtlich ihrer Auslastung auf den Prozessorkernen unterscheiden. Zur Erreichung der jeweiligen Auslastung in Abhängigkeit des Lastszenarios werden die Benchmark-Testfälle so auf die Zeitscheiben und ISRs der Prozessorkerne verteilt, dass diese die erforderliche Rechenzeit benötigen. In der Tabelle 5.12 ist für jeden Prozessorkern dargestellt, welche Tasks und asynchronen Events zur Verfügung ste-

hen.

Tabelle 5.12: Übersicht der Tasks und asynchronen Events für alle Prozessorkerne

Name	Typ	Intervall	Verwendung
Initialisierungs-Task	Task	Systemstart	Task-Container
1ms Task	Task	1ms	Task-Container
10ms Task	Task	10ms	Task-Container
100ms Task	Task	100ms	Task-Container
1000ms Task	Task	1000ms	Task-Container
Idle-Task	Task	Hintergrund	Task-Container
ISR 0	Category 1 Interrupt	20ms	Interrupt-Container
ISR 1	Category 1 Interrupt	20ms	Interrupt-Container
ISR 2	Category 1 Interrupt	20ms	Interrupt-Container
ISR 3	Category 1 Interrupt	20ms	Interrupt-Container
ISR 4	Category 1 Interrupt	20ms	Interrupt-Container
ISR 5	Category 1 Interrupt	20ms	Interrupt-Container
ISR 6	Category 1 Interrupt	20ms	Interrupt-Container
ISR 7	Category 1 Interrupt	20ms	Interrupt-Container
OS Timer Interrupt	Category 2 Interrupt	0,5ms	OS Zeitbasis

Wie in der Tabelle 5.12 dargestellt ist, besitzt jeder Prozessorkern eine Initialisierungs-Task, welche direkt nach dem Systemstart einmalig ausgeführt wird. Hinzu kommen vier zyklische Tasks mit einem unterschiedlichen Intervall sowie eine *Idle*-Task für die Ausführung von Hintergrundaufgaben. Zusätzlich gibt es acht Interrupts, welche mit Hilfe des Konzepts aus Kapitel 5.2.2.1 erzeugt werden. Zu diesem Zweck löst ein Timer-Interrupt zyklisch alle 2,5 ms ein Event aus, in welchem mittels einer Pseudozufallsfunktion einer der acht ISRs ausgelöst wird. Durch eine ungefähre Gleichverteilung über die Gesamtlaufzeit des Systems ergibt sich ein durchschnittliches Intervall von ca. 20 ms für jeden Interrupt. Für die genannten acht ISRs wird die Kategorie 1 genutzt, weswegen die Ausführung dieser asynchronen Events nicht im Kontext des Betriebssystems erfolgt. Lediglich die ISR für die Zeitbasis des Betriebssystems läuft im entsprechenden Kontext und ist daher ein Kategorie 2 Interrupt.

Die Bestimmung der Auslastung der Prozessorkerne erfolgt über die Laufzeit, welche für die Hintergrundaufgabe zur Verfügung steht. Diese sogenannte *Idle*-Task wird immer dann ausgeführt, wenn keine Zeitscheibe oder ISR berechnet werden muss, weswegen diese Task auch als Leerlaufroutine bezeichnet wird. Neben den Tasks und ISRs benötigt auch das Betriebssystem Rechenzeit zur Ausführung des Schedulers, zur Überwachung des Stacks und zur Überwachung der Laufzeit aller Tasks im System. Die Leerlauflaufzeit kann anhand der folgenden Formel 5.1 beschrieben werden:

$$t_{free} = t_{all} - t_{task} - t_{ISR} - t_{OS} \tag{5.1}$$

t_{free}	Leerlauflaufzeit
t_{all}	Gesamtlaufzeit
t_{task}	Task-Laufzeit
t_{ISR}	ISR-Laufzeit
t_{OS}	Betriebssystemlaufzeit

Zur Untersuchung der Auswirkungen eines optimierten Speichermanagements auf eine unterschiedliche Anzahl an Prozessorkernen werden neben den drei Lastszenarien zusätzliche Konfigurationen erstellt, welche sich hinsichtlich der Anzahl der aktiven Kerne unterscheiden. Der in dieser Arbeit verwendete Mikrocontroller vom Typ Infineon AURIX TC39x bietet insgesamt sechs TriCore-Kerne, wobei in der Initialisierungsphase bestimmt werden kann, wie viele der sechs Prozessorkerne gestartet werden sollen. Mit Hilfe dieser Option werden neben den drei Lastszenarien drei Konfigurationen mit zwei, vier und sechs Kernen evaluiert, woraus sich in Summe neun Testszenarien ergeben, welche in Tabelle 5.13 beschrieben sind. Wie in Tabelle 5.13 zu sehen ist, sinkt die Auslastung der Prozessorkerne trotz identischem Lastszenario bei abnehmender Anzahl von aktiven Kernen. Dieser Umstand ist darauf zurückzuführen, dass durch die reduzierte Menge an Prozessorkernen die Häufigkeiten von konkurrierenden Zugriffen minimiert wird, wodurch Wartezyklen effektiv verhindert werden [51] [4].

Tabelle 5.13: Übersicht der umgesetzten Lastszenarien (Die Bestimmung der Auslastung der Prozessorkerne erfolgt mittels der Standardeinstellung des Linkers sowie aktivierten Programm-Caches.)

Lastszenario		**Auslastung der Prozessorkerne in %**					
		CPU0	CPU1	CPU2	CPU3	CPU4	CPU5
65 % CPU Last	2 Prozessorkerne	60,54	56,07	-	-	-	-
	4 Prozessorkerne	60,98	57,48	59,84	60,85	-	-
	6 Prozessorkerne	61,41	62,23	62,68	65,16	72,34	71,01
80 % CPU Last	2 Prozessorkerne	77,50	72,89	-	-	-	-
	4 Prozessorkerne	78,03	74,48	74,79	76,11	-	-
	6 Prozessorkerne	78,47	79,74	78,36	80,42	81,29	80,79
90 % CPU Last	2 Prozessorkerne	89,54	82,13	-	-	-	-
	4 Prozessorkerne	90,13	84,07	85,30	85,11	-	-
	6 Prozessorkerne	89,24	90,19	89,82	90,20	92,18	92,35

Für den Nachweis der korrekten Funktionsweise der optimierten Allokation wird jeder der neun Testfälle mit drei verschiedenen Linker-Skripten übersetzt, welche jeweils einen unterschiedlichen Ansatz des Speichermanagements nutzen. In der ersten Variante wird die Standardeinstellung vom Linker verwendet, bei welcher die beiden globalen Speicher *Pflash* und LMU bevorzugt und linear befüllt werden. Bedingt durch diesen Umstand werden die Separierung der globalen Speicher sowie die lokalen Speicher nicht genutzt, was die Ausführungszeit erhöht und die Effekte von konkurrierenden Zugriffen maximiert. Zur Kompensation dieser Effekte werden die Programm-Caches aktiviert, welche bedingt durch ihre Funktionsweise für die auszuführende Software völlig transparent sind [2]. Auf eine optimierte Allokation zur Reduzierung von Cache-Interferenzen wird in diesem Referenzszenario verzichtet. Mit Hilfe des zweiten Linker-Skripts wird der Stand der Technik reproduziert, welcher Applikationen als kleinste allokierbare Einheit ansieht. Eine Applikation oder auch Funktionalität wird durch einen Benchmark-Testfall beschrieben und kann daher aus mehreren Funktionen, Konstanten und Variablen bestehen. Hierbei ist jedoch zu beachten, dass eine mehrfache Allokation von Funktionalitäten im Flash-Speicher nicht vorgesehen ist. Lediglich in den Scratchpads können lokale Kopien pro Prozessorkern angelegt werden. Die Kombination von verschiedenen Applikationen in einem MPU-Bereich ist nur dann erlaubt, wenn diese dieselbe Kritikalitätsstufe aufweisen. Mittels des letzten Linker-Skripts erfolgt die Umsetzung des in dieser Arbeit vorgestellten Konzepts, welches Funktionen, Konstanten und Variablen als kleinste allokierbare Einheit betrachtet und für die Priorisierung und Allokation eine Vielzahl von Faktoren berücksichtigt.

Ein detaillierte Aufschlüsselung aller Testszenarien mit den dazugehörigen Benchmarks sowie deren Prioritäts- und Kritikalitätsstufe pro Prozessorkern und Zeitscheibe ist im Abschnitt 7.2 hinterlegt. Des Weiteren sind dort ebenfalls die asynchronen Events mit den entsprechenden Lasten beschrieben.

5.3 Speicheroptimierung

Aufgrund der Komplexität der Software von modernen Steuergeräten kann eine manuelle Auswertung und Priorisierung auf Basis des in Kapitel 4 vorgestellten Konzepts nur mit großem Zeitaufwand erfolgen. Da bei jeder Änderung der Gesamtsoftware dieser Prozess wiederholt werden muss, stellt dies keine akzeptable Situation dar. Aus diesem Grund wird im Rahmen dieser Arbeit ein Programm entwickelt, welches mittels von vier Eingangsvektoren die Auswertung, Priorisierung und optimierte Allokation vornimmt. Die Umsetzung erfolgt dabei mit Hilfe des Microsoft Visual Studio in der Programmiersprache C#. Der Grundlegende Aufbau der implementierten Software ist in Abbildung 5.14 exemplarisch dargestellt.

Wie in Abbildung 5.14 dargestellt, ist die Gesamtsoftware für die optimierte Allokation in sieben Schichten unterteilt, welche unterschiedliche Funktionalitäten realisieren. In der ersten Schicht sind die vier Eingangsgrößen enthalten, welche die erforderlichen Informationen beinhalten. Mittels der Systembeschreibung erfolgt die Definition des verwendeten Mikrocontrollers mit allen Prozessorkernen, Speichern

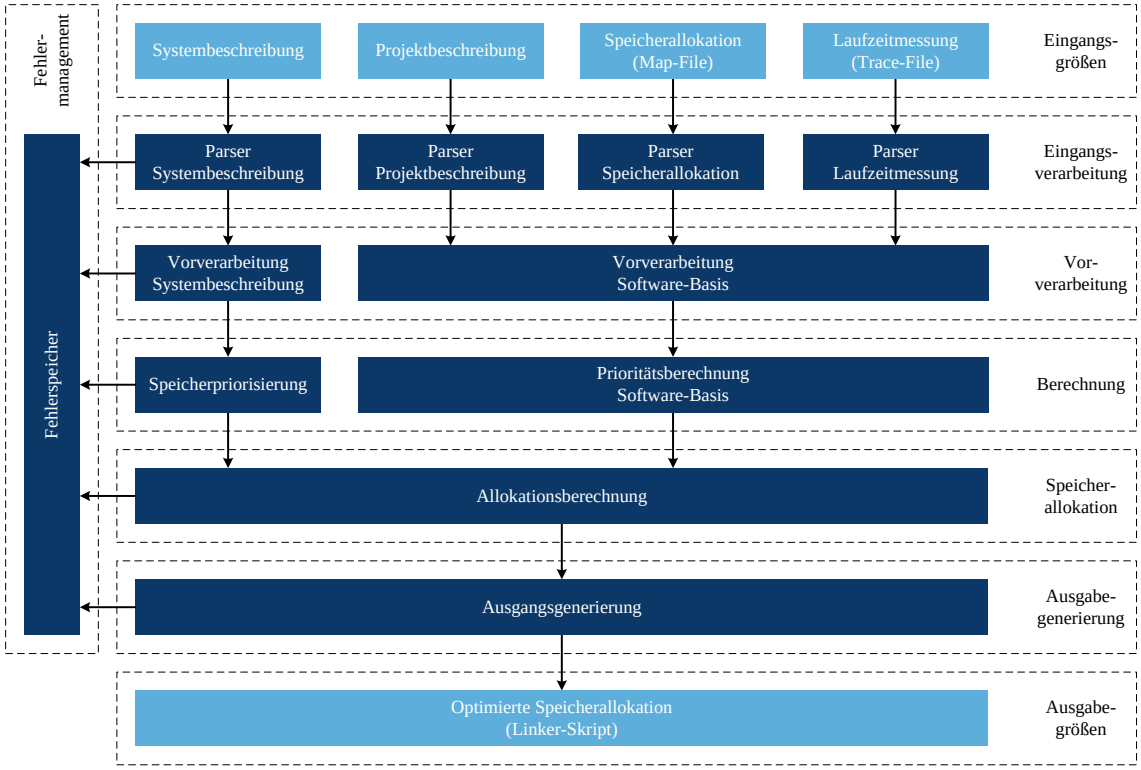

Abbildung 5.14: Grundlegender Aufbau der implementierten Messauswertung

sowie den dazugehörigen Schnittstellen im System. Der Aufbau dieser Extensible Markup Language (XML)-Datei entspricht dabei der Beschreibung in Abschnitt 4.2. Für die Definition der Prioritäten der einzelnen Funktionalitäten im System wird die Projektbeschreibung benötigt, welche ebenfalls als XML-Datei umgesetzt ist. Innerhalb dieser Eingangsgröße sind alle Funktionalitäten mit ihrer Priorität, der Prozessorkernzugehörigkeit sowie der verwendeten Funktionen enthalten. Zur Bestimmung der vorhandenen Funktionen, Variablen und Konstanten in der Gesamtsoftware wird das Map-File ausgewertet, welches der Linker während des Erstellungsprozesses der Zielsoftware erstellt. Der Grund für die Nutzung des Map-Files besteht darin, dass moderne Compiler häufig eine Vielzahl von Optimierungen verwenden, wodurch Funktionen oder Variablen durch intelligentes Management zusammengelegt oder separiert werden können. Um diese Vorteile weiterhin zu nutzen, setzt das entwickelte Programm zur optimierten Allokation erst nach dem Erstellungsprozess der Gesamtsoftware an. Bei der letzten Eingangsgröße handelt es sich um den Mitschnitt der Programmausführung auf der Zielhardware, welcher mindestens eine Hyperperiode beinhalten sollte. Mit Hilfe dieses Mitschnitts kann die Aufrufhäufigkeit sowie die Laufzeit beziehungsweise die Zugriffsgeschwindigkeit von Funktionen, Variablen und Konstanten ermittelt werden. Für jeden Eingangsvektor gibt es in der folgenden Schicht einen separaten Parser für die Eingangsverarbeitung. Der Grund für diese Eingangsverarbeitung besteht darin, dass sich, je nach verwendetem Linker und Debugger, das Format des Map-Files sowie des *Trace*-Mitschnitts

unterscheiden können. Damit in diesem Falle nicht das gesamte Programm zur optimierten Allokation angepasst werden muss, überführen diese Eingangsparser die entsprechenden Eingangsgrößen in generische Formate, welche intern weiterverarbeitet werden können. In der anschließenden Vorverarbeitung wird in einem separaten Modul die Systembeschreibung analysiert und die Speicher anhand ihrer Zugriffsgeschwindigkeit für jeden Prozessorkern priorisiert. In dem zweiten Vorverarbeitungsmodul erfolgt die Zusammenführung aller Informationen aus der Projektbeschreibung, des Map-Files sowie der Laufzeitmessung. Im Anschluss existiert für jede Funktion, Variable und Konstante eine Übersicht darüber, welcher Funktionalität auf welchem Prozessorkern mit welcher Zugriffshäufigkeit innerhalb einer Hyperperiode auf das entsprechende Element zugreift. Im nächsten Schritt kann nun die Berechnung der Priorität eines jeden Elements erfolgen, was auf Basis der in Abschnitt 4.1 erarbeiteten Formel erfolgt und Teil der Priorisierungsschicht ist. Mittels der priorisierten Speicherhierarchie sowie der gewichteten Funktionen, Variablen und Konstanten kann anschließend die optimierte Allokation berechnet werden, wobei sich das Vorgehen nach dem Algorithmus in Abschnitt 4.3 richtet. Analog zu der Eingangsverarbeitung erfolgt die Generierung des optimierten Linker-Skripts über ein separates Modul, welches je nach Anwendungsfall und Linker-Hersteller ausgetauscht werden kann. Parallel zu den genannten Schichten existiert zusätzlich ein Fehlermanagement inklusive Fehlerspeicher, welches auftretende Events protokolliert und zur späteren Analyse abspeichert.

5.4 Testergebnisse

In dem folgenden Abschnitt werden die drei Allokationsansätze auf Basis der insgesamt neun Lastszenarien miteinander verglichen und ausgewertet. Die Untersuchung erfolgt dabei anhand der in Abschnitt 2.1 beschriebenen Zielstellung, welche sich auf vier Bereiche aufteilt.

5.4.1 Temporale Isolation

Im Rahmen der ersten Zielstellung werden die Zuweisungen der Speicher zu den Prozessorkernen untersucht, wobei der Fokus darauf liegt, dass eine möglichst exklusive Nutzung vorgenommen wird. Des Weiteren wird die Berücksichtigung von Caches sowie die Einbindung von speicherrelevanten Hardware-Beschleunigern untersucht.

5.4.1.1 Speicherzuweisung

Für die Zuweisung der Speicher zu den jeweiligen Prozessorkernen wird im ersten Schritt die Zugriffsgeschwindigkeit für jeden Speicher ermittelt. Dabei wird für die Programm-Code- und Datenspeicher ein unterschiedliches Lastszenario gewählt, wobei für die Ermittlung der Performance der Code-Speicher der EEMBC CoreMark und für die Bestimmung der Leistungsfähigkeit der Datenspeicher auf den IAV CopyData Benchmark zurückgegriffen wird. Der Anlass für die Auswahl dieser beiden Testfälle ist darin begründet, dass der CoreMark nur einen sehr geringen Bedarf

an Variablen hat, welche hauptsächlich für die Berechnung der Punktzahl am Ende der Ausführung benötigt werden. Daher bestimmen primär die Programm-Code-Speicher das Endergebnis des CoreMarks. Im Gegensatz dazu liegt der Fokus beim Benchmark IAV CopyData auf den Datenspeichern und der gezielten Allokation der Lese- und Schreibbuffer, welche für die Kopiervorgänge genutzt werden. Durch die geringe Größe des Programm-Codes dieser Bibliothek kann dieser vollständig in den Programm-Caches vorgehalten werden, wodurch die Laufzeit vorwiegend durch die verwendeten Datenspeicher bestimmt wird.

Für die Bestimmung der Speichergeschwindigkeiten wird Prozessorkern 0 vermessen, welcher mittels Crossbar 0 angebunden ist. Die Ergebnisse können analog für die anderen Prozessorkerne an der jeweiligen Crossbar übernommen werden, wobei die Zuordnung der Speicher beachtet werden muss, welche beispielsweise über eine Direktanbindung verfügen. Das Ergebnis der Performance der Programmspeicher ist in der Abbildung 5.15 dargestellt.

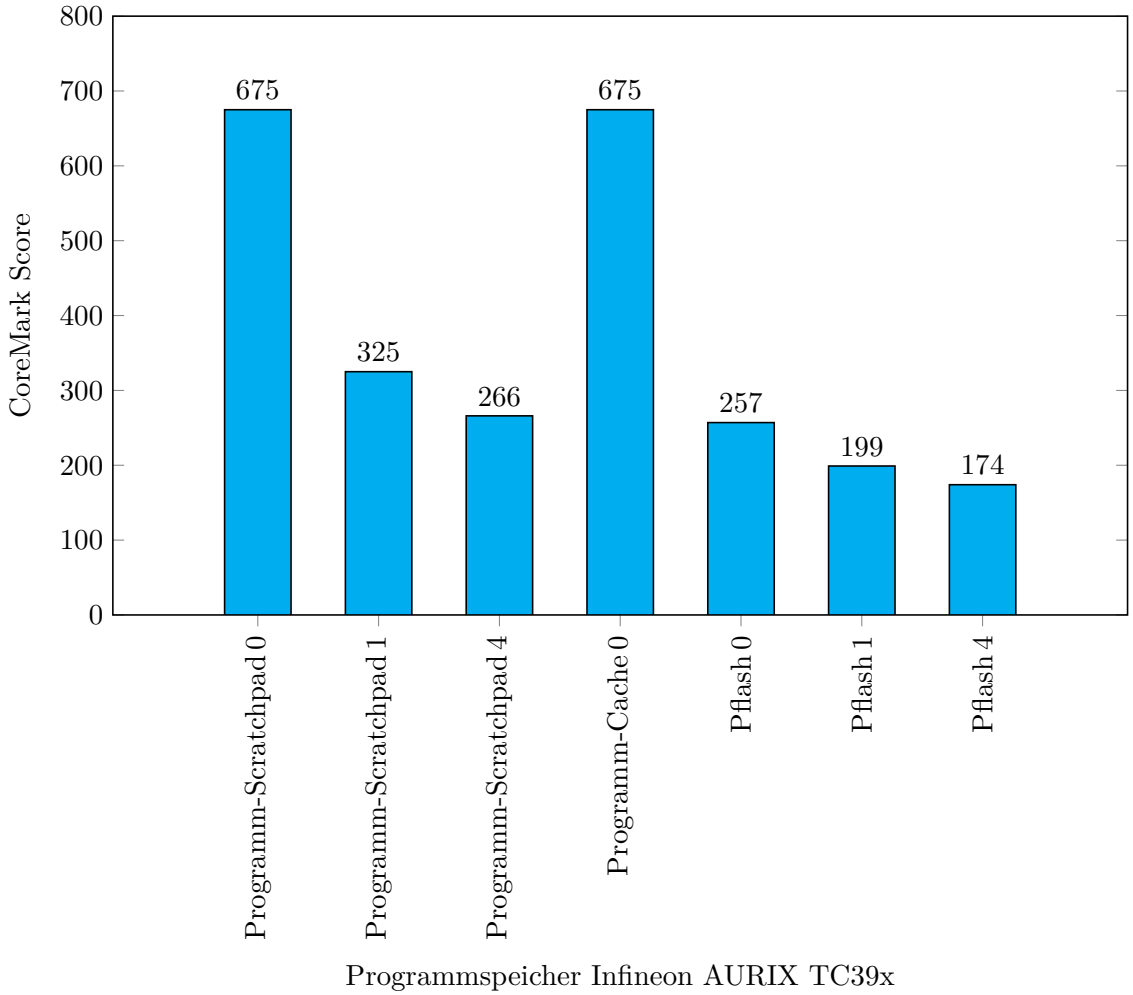

Abbildung 5.15: Performance der Programmspeicher des Infineon AURIX TC39x bei exklusiver Nutzung von Prozessorkern 0

Wie in der Messung in Abbildung 5.15 dargestellt, erreicht der Prozessorkern 0 des Infineon AURIX TC39x die maximale Ausführungsgeschwindigkeit bei der Nutzung der lokalen Speicher in Form des Programm-Scratchpad 0 sowie des Programm-Cache 0. Beide Speicher erreichen in diesem isolierten Szenario eine identische Performance, da der Code des CoreMarks vollständig in den Programm-Cache geladen werden kann. Des Weiteren treten keine Cache-Interferenzen durch Task-Wechsel oder asynchrone Events auf, weswegen der Inhalt des Caches über die gesamte Ausführungszeit des CoreMarks konstant bleibt. In einer optimierten Speicherzuordnung sollten die lokalen Speicher ausschließlich durch die dazugehörigen Prozessorkerne genutzt werden. Zur Vollständigkeit werden die Scratchpads der anderen Prozessorkerne bei der Nutzung durch Kern 0 ebenfalls vermessen. Bei der Ausführung aus dem Scratchpad von Prozessorkern 1 profitiert Kern 0 von der Anbindung über die gleiche Crossbar, weswegen die Ausführungsgeschwindigkeit höher als bei der Nutzung des Programm-Scratchpad 4 ist. Für den Zugriff auf Programm-Scratchpad 4 muss der Prozessorkern 0 einen Transfer über beide Crossbars sowie die verbindende Bridge initiieren, was zusätzliche Laufzeit benötigt und sich direkt auf den Punktestand des CoreMarks auswirkt. Ein ähnliches Verhalten wie die Programm-Scratchpads zeigen die verschiedenen Flash-Bänke. Auf den *Pflash 0* erfolgt der Zugriff mittels des *Direct Paths*, weswegen hier die Ausführungsgeschwindigkeit im Kontext der globalen Speicher am höchsten ist. Für den Zugriff auf *Pflash 1* ist hingegen die Nutzung einer Crossbar und bei *Pflash 4* von beiden Crossbars erforderlich, weswegen hier die Performance geringer ist. Die Messungen für Prozessorkern 0 können für die anderen Kerne im System übertragen werden, wobei die Anbindung an die jeweilige Crossbar entscheidend ist. Beispielsweise ist bei Prozesskern 4 und 5 die Anbindung über den Direktpfad zur jeweiligen Flash-Bank besonders schnell und der Zugriff auf die benachbarten Flash-Speicher der Crossbar 1 performanter als die Nutzung der Flash-Bänke, welche über die Crossbar 0 angebunden sind [2] [84].

Für die Messung der Zugriffsgeschwindigkeit auf die Datenspeicher im System werden zwei 4 KB große Arrays angelegt und in verschiedenen Speichern des Infineon AURIX TC39x abgelegt. Im Anschluss wird ein Kopiervorgang gestartet und die Laufzeit mittels der Performance Counter ermittelt. Das Ergebnis dieser Messung ist in Tabelle 5.16 dargestellt [3].

Wie bei der Testreihe zur Bestimmung der Programmspeicherperformance wird in diesem Szenario ebenfalls exklusiv Prozessorkern 0 genutzt und die anderen Kerne deaktiviert. Des Weiteren sind alle Caches abgeschaltet und die Allokation des Programm-Codes erfolgt im Programm-Scratchpad 0. Wie in den Messreihen in Tabelle 5.16 zu sehen ist, zeigen die lokalen Datenspeicher in Form des Daten-Scratchpads 0 sowie der dLMU 0 die höchste Performance. Eine Besonderheit stellt der Kopiervorgang des Test-Arrays innerhalb der dLMU dar, welcher deutlich mehr Laufzeit benötigt. Dieser Umstand ist darauf zurückzuführen, dass das Daten-Scratchpad im Gegensatz zur dLMU breitbandiger angebunden ist. Wie die Messungen zeigen, eignet sich das Daten-Scratchpad der Prozessorkerne primär für die exklusive Nutzung durch den jeweiligen Kern, da beispielsweise eine Intertask-Kommunikation, welche innerhalb eines Speichers abläuft, nicht ausgebremst wird. Im Gegensatz dazu

Tabelle 5.16: Performance der Datenspeicher des Infineon AURIX TC39x bei exklusiver Nutzung von Prozessorkern 0 [3]

Quelle	Ziel	Dauer in Ticks
Daten-Scratchpad 0	Daten-Scratchpad 0	2076
Daten-Scratchpad 0	dLMU 0	2076
dLMU 0	Daten-Scratchpad 0	2076
dLMU 0	dLMU 0	4113
Daten-Scratchpad 0	LMU 0	2076
LMU 0	Daten-Scratchpad 0	9239
LMU 0	LMU 0	9239
Daten-Scratchpad 0	Daten-Scratchpad 1	2076
Daten-Scratchpad 1	Daten-Scratchpad 0	9239
Daten-Scratchpad 1	Daten-Scratchpad 1	9239
Daten-Scratchpad 0	Daten-Scratchpad 4	3090
Daten-Scratchpad 4	Daten-Scratchpad 0	12311
Daten-Scratchpad 4	Daten-Scratchpad 4	12311

verfügt die dLMU über eine separate Anbindung an die Crossbar, wodurch diese für die anderen Komponenten des Systems besser erreichbar ist. Dieser Umstand prädestiniert die dLMU für die Nutzung zur Intercore-Kommunikation. Die restlichen Messergebnisse entsprechen den Erwartungswerten. Die Speicher, welche ebenfalls über die Crossbar 0 angebunden sind, sind für den Prozessorkern 0 deutlich schneller zu erreichen als die Speicher, welche mittels der Crossbar 1 mit dem Rest des Systems verbunden werden [3] [63] [51].

Auf Basis der durchgeführten Messungen ergibt sich die Speicherzuweisung für die jeweiligen Prozessorkerne, welche in Tabelle 5.17 aufgeschlüsselt ist. Auf Basis des in Abschnitt 4.3.1 beschriebenen Verfahrens wird für jeden Speicher ein Index vergeben. Je niedriger der Index ist, desto höher ist die Speichergeschwindigkeit für den jeweiligen Prozessorkern. Auf eine gesonderte Betrachtung der Caches wird an dieser Stelle verzichtet, da diese zur Laufzeit dynamisch durch die Hardware selbst verwaltet werden. Eine Einflussnahme auf die Funktionsweise der Caches ist nur indirekt über die Auswahl der Inhalte, welche von den Caches vorgehalten werden können, sowie über die Allokation in den globalen Speicher möglich [13].

5.4.1.2 Cache-Nutzung

Für die optimierte Nutzung der Caches der verwendeten Hardware-Plattform wird für die Daten- und Programm-Caches ein ähnliches Konzept als reiner Lesespeicher gewählt. Der Vorteil dieses Vorgehens liegt in einem einheitlichen Ansatz, wodurch bei einer Speicheroptimierung beide Cache-Typen gleichermaßen profitieren. Nachteilig ist hingegen der Umstand, dass die Daten-Caches ausschließlich für Konstanten genutzt werden können und nicht für Variablen. Sobald der lokale Daten-Cache eine Variable vorhält, kann diese zur Laufzeit auch aktualisiert werden, wodurch es bei der TriCore-Architektur durch das genutzte LRU-Verfahren zu inkonsistenten Daten kommen kann. Gerade bei Variablen, welche für die Intercore-Kommunikation

Tabelle 5.17: Übersicht der Speicherzuweisung für den Infineon AURIX TC39x

Speicher		**Prozessorkern**					
		CPU0	CPU1	CPU2	CPU3	CPU4	CPU5
Programm-speicher	Scratchpad X	0	0	0	0	0	0
	Pflash 0	1	2	2	2	3	3
	Pflash 1	2	1	3	3	4	4
	Pflash 2	3	3	1	4	5	5
	Pflash 3	4	4	4	1	6	6
	Pflash 4	5	5	5	5	1	2
	Pflash 5	6	5	5	5	2	1
Daten-speicher	Scratchpad X	0	0	0	0	0	0
	dLMU 0	1	2	2	2	5	5
	dLMU 1	2	1	3	3	6	6
	dLMU 2	3	3	1	4	7	7
	dLMU 3	4	4	4	1	8	8
	dLMU 4	6	6	6	6	1	2
	dLMU 5	7	7	7	7	2	1
	LMU 0	5	5	5	5	9	9
	LMU 1	8	8	8	8	3	3
	LMU 2	9	9	9	9	4	4

genutzt werden, ist dies ein potentielles Risiko, da der lesende Prozessorkern unter Umständen mit veralteten Werten die Berechnung startet. Ein weiterer Aspekt, welcher die Nutzung als reinen Lesespeicher begünstigt, ist der Umstand der weniger komplexen Laufzeitbetrachtung, da das Rückschreiben von aktualisierten Werten ausgeschlossen werden kann.

Zur optimierten Nutzung der Caches werden zwei Faktoren berücksichtigt, welche für die Allokation relevant sind. Dies betrifft zum einen die Anzahl der gleichzeitig vorhaltbaren *Pages* sowie deren Größe. Das Ziel der optimierten Allokation besteht darin, alle cache-baren Inhalte möglichst kompakt in den globalen Speicher abzulegen, damit die Größe der Cache *Pages* effektiv genutzt werden kann. Des Weiteren werden in Abhängigkeit der verfügbaren *Pages* die MPU-Sektionen vorgehalten, welche die Funktionalitäten mit der höchsten Kritikalität aufweisen und nicht bereits im Scratchpad vorhanden sind. Das Ziel ist dabei, nur Funktionalitäten mit einer identischen Kritikalität durch die Caches vorzuhalten, wodurch eine Anforderung der ISO 26262 erfüllt wird, welche Interferenzfreiheit für Funktionalitäten unterschiedlicher Kritikalität fordert [38].

Der verwendete Infineon AURIX TC39x nutzt Zwei-Wege-Assoziative Caches, wobei sich die Speicherkapazitäten der *Pages* beim Programm- und Daten-Cache unterscheiden. Während der Programm-Cache über *Pages* mit je 16 KB verfügt, bietet der Daten-Cache lediglich 8 KB pro *Page*. Die Größe der Programm-Cache *Page* entspricht damit exakt der Größe einer MPU-Sektion des *Pflashs*, was die optimierte Allokation vereinfacht. Bei dem Daten-Cache kann hingegen nur eine halbe MPU-Sektion vorgehalten werden, was die effektive Nutzung der geringen Speicherkapazität umso wichtiger macht.

Für die Vermessung des Einflusses der Caches auf die Ausführungsgeschwindigkeit wird ein rudimentäres Testszenario genutzt, welches 16 Funktionen verwendet, die sich nacheinander aufrufen. Zur verbesserten Darstellung der Ergebnisse wird dieses Testszenario 10 Millionen Mal wiederholt und die Laufzeit mit Hilfe der Performance Counter des Infineon AURIX taktgenau erfasst. Für die Durchführung der Messung sind alle Prozessorkerne, abgesehen von dem ausführenden Kern 0, deaktiviert, damit der isolierte Einfluss des Programm-Caches ermittelt werden kann. Im Rahmen der ersten Messung werden die 16 Funktionen in einer MPU-Sektion allokiert, was der Funktionsweise des Programm-Caches entspricht, da alle 16 Funktionen in einer Cache *Page* vorgehalten werden können. In der zweiten Messung werden die 16 Funktionen auf 16 verschiedene MPU-Sektionen verteilt, so dass der Programm-Cache permanent zum Umladen gezwungen wird, was sich auch in der gemessenen Laufzeit niederschlägt. Während bei einer optimierten Allokation die komplette Ausführung nach 6,5 Sekunden abgeschlossen ist, benötigt eine suboptimierte Allokation annährend 18 Sekunden, was in Abbildung 5.18 dargestellt ist. Da für die Programm- und Daten-Caches das selbe Konzept als reiner Lesespeicher im System verwendet wird, wird auf eine separate Messung der Geschwindigkeit der Daten-Caches verzichtet [4].

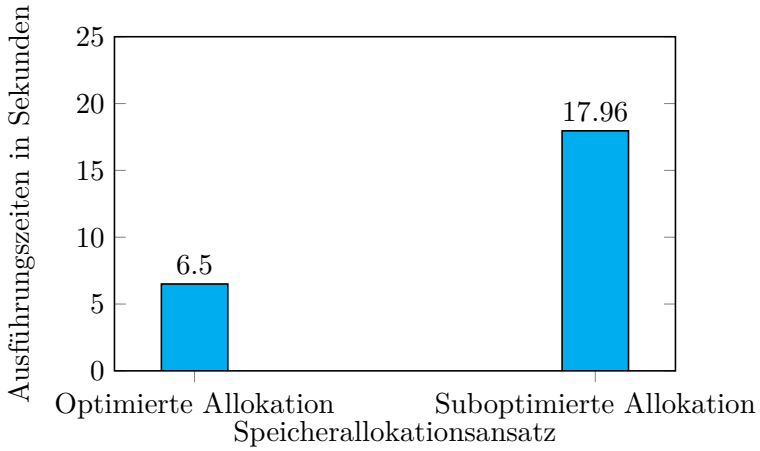

Abbildung 5.18: Performance des Programm-Caches des Infineon AURIX TC39x [4]

5.4.1.3 Berücksichtigung von Hardware-Beschleunigern

Für die Integration von Hardware-Beschleunigern in die optimierte Allokation ist es essentiell, deren Speicherzugriffe ebenfalls zu berücksichtigen, weswegen eine Betrachtung als zusätzlicher Prozessorkern im Rahmen dieser Arbeit erfolgt. Dabei ist jedoch zu beachten, dass Hardware-Beschleuniger primär dafür konzipiert sind, die Hauptprozessorkerne bei Standardaufgaben zu entlasten. Aus diesem Grund werden die Hardware-Beschleuniger bei der Bestimmung der Speicherlast sowie bei der Verteilung der Speicher als Prozessorkerne betrachtet, jedoch bei der optimierten Allokation die Anforderungen der Kerne priorisiert, welchen die Beschleuni-

ger zuarbeiten. Dies erfolgt über die Prioritätsberechnung der Variablen für die Intercore-Kommunikation im System mit Hilfe der Priorität des Betriebssystems. Da ein Hardware-Beschleuniger über kein separates Betriebssystem verfügt, kann die entsprechende Priorität mit dem Wert 1 angenommen werden. Im Gegensatz dazu verfügt das Betriebssystem des Prozessorkerns über eine höhere Priorität, weswegen die Speicherzuordnung des ausführenden Kerns Vorrang hat. An dieser Stelle kann beispielhaft das Ethernet-Modul des Infineon AURIX TC39x genannt werden, welches über einen separaten DMA-Controller verfügt. Die Speicherzugriffe dieses DMA-Controllers können zu konkurrierenden Zugriffen führen, weswegen eine Betrachtung im Rahmen der Priorisierung der entsprechenden Variablen im Speicher sowie bei der Speicherzuordnung essentiell ist. Bei der Allokation der Variablen, welche für die Ethernet-Kommunikation benötigt werden, ist es wiederum wichtig, dass diese möglichst ideal für den Prozessokern 0 liegen, welche in dem implementierten Testsystem verantwortlich für die Kommunikation, wie Ethernet und CAN, ist. Aus diesem Grund wird der Speicher des Kern 0 für die Ablage der Sende- und Empfangsbuffer verwendet, welcher ebenfalls für die Intercore-Kommunikation genutzt wird. Ein analoges Vorgehen gilt für den separaten DMA-Controller des AURIX, weswegen alle Transferbuffer in der dLMU des jeweiligen Prozessorkerns allokiert sind. Durch dieses Vorgehen kann das Daten-Scratchpad eines jeden Kerns exklusiv durch den zugeordneten Prozessorkern genutzt werden, während die dLMU für die Intercore-Kommunikation mit allen anderen Systemkomponenten vorgehalten wird [3] [63] [51].

5.4.2 Räumliche Isolation

Im Kontext der zweiten Untersuchung wird die räumliche Isolation der drei Verfahren miteinander verglichen. Der Fokus liegt dabei auf der effektiven Nutzung der MPU-Bereiche, aber auch auf der gezielten Duplikation hochpriorisierter Funktionen und Variablen zur Vermeidung von konkurrierenden Zugriffen auf geteilte Speicherbereiche.

5.4.2.1 MPU-Nutzung

Die Berücksichtigung der MPU-Granularität ist in sicherheitskritischen Systemen relevant für die räumliche Trennung von Funktionalitäten unterschiedlicher Kritikalität. Aus diesem Grund berücksichtigt das im Rahmen dieser Arbeit entwickelte Verfahren diese Eigenschaft der genutzten Hardware-Plattform. Bedingt durch den Umstand, dass der Infineon AURIX eine flexible prozessorkernlokale MPU aufweist, können die lokalen Speicher sehr feingranular separiert werden. Die Folge dieser flexiblen Hardware-Umsetzung ist jedoch, dass die Vorteile der Berücksichtigung der MPU-Granularität nicht zum Tragen kommen. Gleiches gilt für die globalen Speicher, welche mit Hilfe der systemweiten MPU geschützt werden. Da die einzelnen Bänke der globalen Speicher mehr Kapazitäten zur Verfügung stellen als von den implementierten Lastszenarien benötigt werden, können auch hier nicht die Vorteile aufgezeigt werden. Aus diesem Grund sollte zum Nachweis der entsprechenden Op-

timierung eine Evaluierung auf einer anderen Mikrocontrollerfamilie erfolgen. Alternativ könnten die Speichergrößen der genutzten AURIX-Plattform durch den Linker künstlich verkleinert werden, wodurch eine Betrachtung ebenfalls ermöglicht werden würde.

5.4.2.2 Gezielte Duplikation

Bedingt durch den Umstand, dass die gesamte Code-Basis der erstellten Testsoftware weniger Speicherplatz benötigt als jede einzelne *Pflash*-Bank zur Verfügung stellt, kann die gezielte Duplikation von geteiltem Programm-Code und Konstanten vollumfänglich ausgeschöpft werden. Durch die mehrfache Ablage geteilter Software-Bestandteile können die konkurrierenden Zugriffe auf den *Pflash* vollständig verhindert werden, wodurch die Leistung mit jedem weiteren Prozessorkern fast linear ansteigt. Dabei muss jedoch berücksichtigt werden, dass die Steigerung der Ausführungsgeschwindigkeit auch von Variablen abhängt, welche im Rahmen der Intercore-Kommunikation benötigt werden, weswegen der Performance-Gewinn nicht vollständig mit der Anzahl der Prozessorkerne skaliert.

In der Abbildung 5.19 ist exemplarisch dargestellt, wie sich die Entwicklung der Ausführungsgeschwindigkeit bei der ausschließlichen Nutzung des *Pflash* darstellt, wenn die gezielte Duplikation vollumfänglich genutzt werden kann. Als Lastszenario wird für diese Messung auf den CoreMark von EEMBC zurückgegriffen, welcher mit den drei Allokationsansätzen im Speicher abgelegt wird. Da der Stand der Technik keine Duplikationen von Programm-Code und Konstanten in den globalen Speichern vorsieht, zeigt dieser ein identisches Verhalten wie die Standardallokation des Linkers. Auf die Nutzung der lokalen Speicher in Form des Programm-Scratchpads und Programm-Caches wird im Rahmen dieser Messung bewusst verzichtet, um den Einfluss von konkurrierenden Zugriffen auf globale Speicher aufzuzeigen.

Wie die Messreihen in Abbildung 5.19 zeigen, steigt die Gesamtpunktzahl des CoreMarks mit jedem weiteren Prozessorkern bei exklusiver Nutzung der entsprechenden *Pflash*-Bank X linear an. Da der CoreMark keine Abhängigkeiten zwischen den Kernen aufweist, können keine Effekte von konkurrierenden Zugriffen aufgrund von Intercore-Kommunikation auftreten. Im Gegensatz dazu skaliert die Punktzahl des CoreMarks bei der Nutzung von lediglich einer *Pflash*-Bank nur marginal mit jedem weiteren Kern, was die Messreihe *Pflash*-Bank 0 aufzeigt. Für eine bessere Einordnung der Möglichkeiten der gezielten Duplikation sollten in weiterführenden Untersuchungen kleinere Speichergrößen evaluiert werden, welche nur eine teilweise Verdopplung geteilter Software-Komponenten erlauben.

5.4.3 Steigerung der Ausführungsgeschwindigkeit

Im dritten Bereich der Untersuchung erfolgt die Auswertung der Auswirkungen einer optimierten Allokation auf die Ausführungsgeschwindigkeit. Das Ziel ist dabei, die performanten, aber kleinen Scratchpads des Mikrocontrollers effektiv zu nutzen. Des Weiteren werden zusätzliche Faktoren, wie die Speicheranbindung oder das Alignment, ebenfalls mitberücksichtigt.

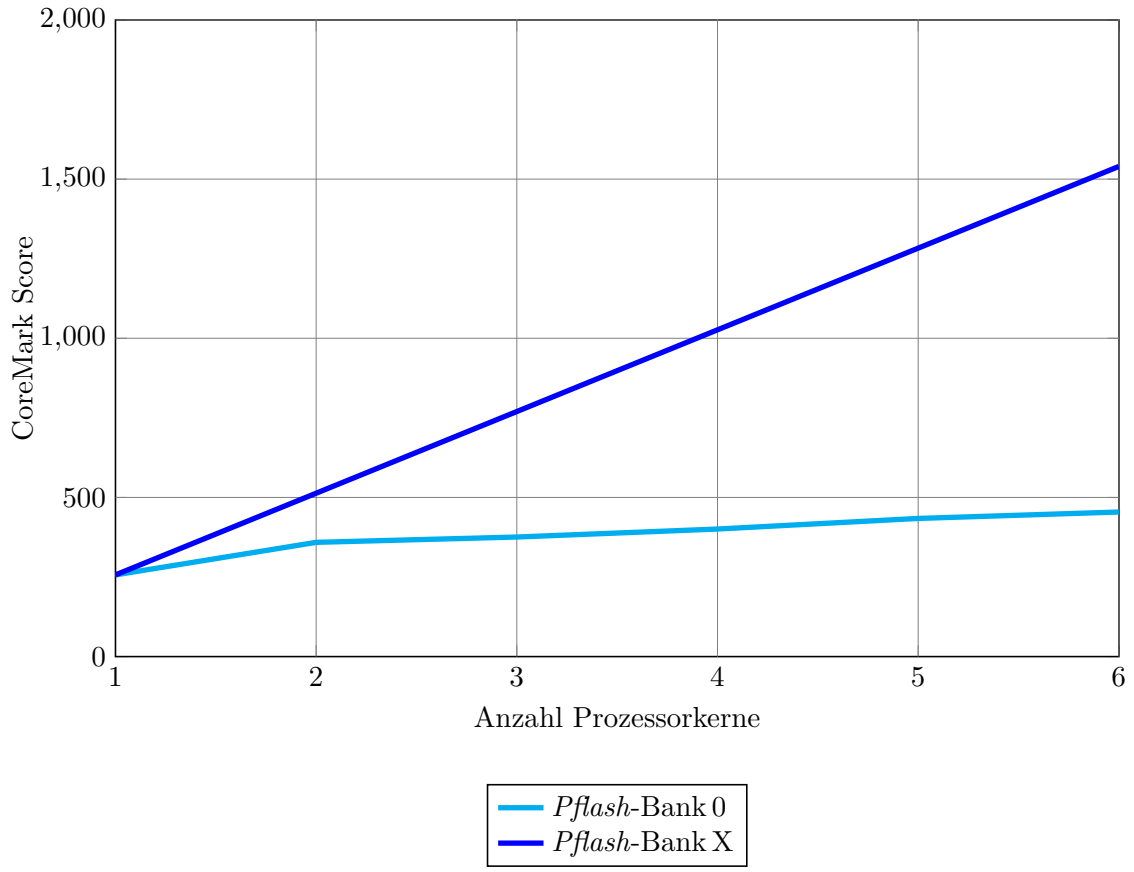

Abbildung 5.19: Performance des *Pflash* des Infineon AURIX TC39x

5.4.3.1 Speichernutzung

Für den Nachweis der optimierten Speichernutzung werden die in Tabelle 5.13 beschriebenen Lastszenarien mit den beiden Allokationsansätzen auf Basis des Stands der Technik sowie mit dem hier vorgestellten Ansatz zur optimierten Speicherallokation miteinander verglichen. Auf eine separate Beschreibung der Standardverteilung des Linkers wird an dieser Stelle verzichtet, da dies bereits in Abschnitt 5.2.2.5 erfolgt ist.

Wie an den Messungen in Tabelle 5.20 zu sehen ist, profitiert die implementierte Testsoftware bereits signifikant durch die Allokation auf Basis des Stands der Technik. Die Auslastung pro Prozessorkern sinkt deutlich im Vergleich zum Standardansatz des Linkers. Durch die Bewertung der Auslastung des Gesamtsystems ist eine Einordnung, welche einzelne Optimierung wie stark die Prozessorlast reduziert, nur bedingt möglich. Für diesen Aspekt wird im Rahmen der Auswertung der weiteren Zielstellungen eine detailliertere Betrachtung jedes einzelnen Optimierungsansatzes vorgenommen.

Für die Messreihen in Tabelle 5.21 werden die vorgestellten Lastszenarien mit Hilfe des im Kontext dieser Arbeit entwickelten Allokationsansatzes auf die vorhandenen Speicher im System verteilt. Wie die Messungen zeigen, kann die Auslastung

Tabelle 5.20: Übersicht der umgesetzten Lastszenarien auf Basis des Stands der Technik Allokation

Lastszenario		Auslastung der Prozessorkerne in %					
		CPU0	CPU1	CPU2	CPU3	CPU4	CPU5
65 % CPU Last	2 Prozessorkerne	58,95	48,15	-	-	-	-
	4 Prozessorkerne	59,31	48,74	50,39	59,00	-	-
	6 Prozessorkerne	59,48	49,11	50,57	59,11	38,73	69,29
80 % CPU Last	2 Prozessorkerne	76,17	63,81	-	-	-	-
	4 Prozessorkerne	76,58	64,37	61,93	74,93	-	-
	6 Prozessorkerne	76,91	64,97	62,11	75,12	77,36	79,99
90 % CPU Last	2 Prozessorkerne	87,72	70,91	-	-	-	-
	4 Prozessorkerne	88,39	71,58	70,71	82,32	-	-
	6 Prozessorkerne	88,96	72,31	71,05	82,69	87,16	89,68

der Prozessorkerne im Vergleich zum Stand der Technik weiter reduziert werden. Dieser Umstand ist primär auf die effizientere Nutzung der lokalen Speicher sowie die Nutzung der gezielten Duplikationen zurückzuführen, wie die detaillierteren Messungen der einzelnen Optimierungsansätze in den folgenden Zielstellungen zeigen.

Tabelle 5.21: Übersicht der umgesetzten Lastszenarien auf Basis der optimierten Allokation

Lastszenario		Auslastung der Prozessorkerne in %					
		CPU0	CPU1	CPU2	CPU3	CPU4	CPU5
65 % CPU Last	2 Prozessorkerne	59,17	35,72	-	-	-	-
	4 Prozessorkerne	59,19	35,78	37,44	41,14	-	-
	6 Prozessorkerne	59,21	35,86	37,57	41,20	34,69	51,72
80 % CPU Last	2 Prozessorkerne	76,09	48,03	-	-	-	-
	4 Prozessorkerne	76,09	48,11	45,18	55,79	-	-
	6 Prozessorkerne	76,24	48,19	45,24	55,91	69,72	57,71
90 % CPU Last	2 Prozessorkerne	82,55	54,53	-	-	-	-
	4 Prozessorkerne	82,58	54,66	53,94	63,97	-	-
	6 Prozessorkerne	82,94	54,76	54,05	64,15	78,72	64,83

Durch die optimierte Allokation auf Basis der umgesetzten Software sowie durch die Einbeziehung der Eigenschaften der genutzten Hardware kann die Auslastung des Gesamtsystems signifikant reduziert werden. Dies ermöglicht es, mehr Funktionalitäten auf einem Mikroprozessor zu vereinen, was die Integrationsdichte erhöht und die Regelung sowie Steuerung von komplexeren Systemen ermöglicht.

5.4.3.2 Speicheranbindung

Die Bewertung der Speicheranbindungen an die Prozessorkerne ist bereits in Abschnitt 5.4.1.1 im Kontext der Speicherzuweisung detailliert beschrieben. Daher wird auf eine separate Beschreibung im Rahmen dieser Teilzielstellung verzichtet.

5.4.3.3 Gezieltes Alignment

Moderne Mikrocontroller, wie der Infineon AURIX TC39x, nutzen eine Architektur mit 32 Bit Registerbreite für die Berechnung der auszuführenden Funktionalitäten. Der Vorteil einer höheren Registerbreite liegt darin, dass auch größere Zahlenwerte ohne aufwendige Emulation berechnet werden können. Nachteilig ist hingegen der Umstand, dass der Speicherzugriff nur dann die volle Performance erreicht, wenn die Daten entsprechend der Registerbreite im Speicher abgelegt sind. Bei dem genutzten Mikrocontroller bedeutet dies, dass die Ablage von Variablen und Konstanten im Speicher im Idealfall ein 4 Byte Alignment aufweist. Während dieser Umstand bei entsprechenden Werten mit 32 Bit Größe keine Herausforderung darstellt, ist dies bei kleineren Datentypen deutlich aufwendiger. Durch eine optimierte Allokation können beispielsweise Byte-Arrays ebenfalls mit einem 4 Byte Alignment abgelegt werden, dies erhöht jedoch den Speicherverbrauch signifikant. Aus diesem Grund kann diese Art der Optimierung nur dann genutzt werden, wenn die lokalen Speicher des Prozessorkerns noch freie Kapazitäten aufweisen. Zur Bewertung des Einflusses dieser Optimierungsart erfolgt im kommenden Abschnitt eine Vergleichsmessung der beiden Ablageoptionen. Zu diesem Zweck werden zwei Byte-Arrays mit einer Größe von 256 Elementen angelegt, welche als Sende- und Empfangsbuffer fungieren. Für die erste Messung, welche in Abbildung 5.22 dargestellt ist, erfolgt die speicherplatzoptimierte Allokation mit einem 8 Bit-Alignment und in der zweiten Messung dann die laufzeit-fokussierte Ablage mit einem 32 Bit-Alignment. Die Ablage der Sende- und Empfangsbuffer erfolgt in der LMU 0 [3].

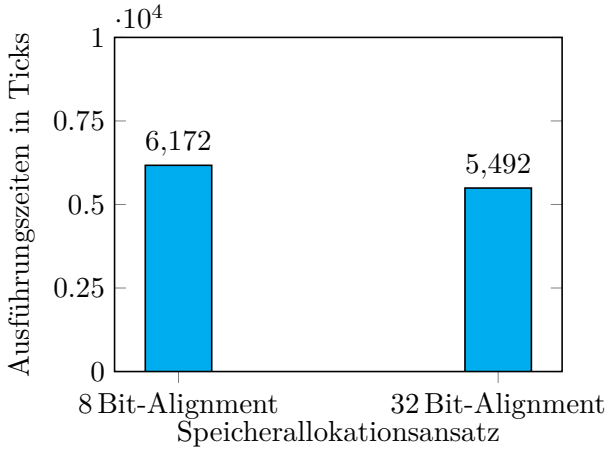

Abbildung 5.22: Speicherperformance des Infineon AURIX TC39x mit unterschiedlichem Alignment

Wie die beiden Messreihen in Abbildung 5.22 zeigen, hat das Alignment einen direkten Einfluss auf die Ausführungsgeschwindigkeit. Dabei muss jedoch berücksichtigt werden, dass das 32 Alignment zwar schneller ist, den Speicherverbrauch der beiden Arrays jedoch vervierfacht. Aus diesem Grund stellt das optimierte Alignment nur dann eine Option dar, wenn genügend Speicherplatz in den Datenspeichern des Mikrocontrollers vorhanden ist.

Der geringe Unterschied von ca.11% in der Laufzeit ist darauf zurückzuführen, dass die TriCore-Architektur die Zugriffe auf Speicheradressen ohne ein 4-Byte Alignment grundsätzlich emulieren muss, dies aber durch den Compiler bereits optimiert wird. Mit einer geringeren Optimierungsstufe bei der Übersetzung des Programm-Codes sind die Effekte wahrscheinlich höher, jedoch stellt dies kein realistisches Szenario dar [162] [171].

5.4.4 Einhaltung der Reaktionszeit

Im letzten Abschnitt erfolgt die Betrachtung der Reaktionszeit, welche besonders bei asynchronen Events, wie beispielsweise Interrupts, von großer Bedeutung ist. Zu diesem Zweck werden die Laufzeiten vom Eintreten des Interrupts bis zur vollständigen Bearbeitung der dazugehörigen ISR vermessen und bewertet.

5.4.4.1 Priorisierung von zeitkritischen Elementen

Die Einhaltung von Zeitzielen in Systemen mit einer harten Echtzeitanforderung ist essentiell. Ein Nichteinhalten einer entsprechenden Frist wird als vollständiges Versagen des Gesamtsystems betrachtet, weswegen eine optimierte Allokation dies berücksichtigen muss. Für die Bestimmung der Reaktionszeit werden die drei Allokationsansätze miteinander verglichen, wobei jeweils drei Messwerte aufgenommen werden. Mit Hilfe vom ersten Messwert wird die minimale Verzögerung vom Eintreten des Interrupts bis zur vollständigen Verarbeitung der dazugehörigen ISR vermessen. Mittels der zweiten Erhebung wird hingegen der Maximalwert ermittelt, welcher in Echtzeitsystemen von besonderer Relevanz ist. Zusätzlich wird ein dritter Wert in Form des Mittelwerts über 32 Messungen gebildet, durch welchen die Streuung beurteilt werden kann. Die Bewertung der Reaktionszeit erfolgt dabei für alle acht ISRs, welche in Tabelle 5.12 beschrieben sind. Zur besseren Übersicht werden nur die Messergebnisse von Prozessorkern 0 für die ISR 0 dargestellt. Die Laufzeiten können jedoch für die anderen Prozessorkerne und ISRs adaptiert werden, da diese auf allen Kernen gleich aufgebaut sind.

Wie in der Messung in Tabelle 5.23 für die ISR 0 auf Prozessorkern 0 exemplarisch dargestellt ist, zeigen die drei Allokationsansätze deutliche Unterschiede in der minimalen, der durchschnittlichen und der maximalen Laufzeit. Für die erste Messreihe wird der Standardallokationsansatz des verwendeten Compilers genutzt, welcher den kompletten Programm-Code sowie die dazugehörigen Konstanten in den *Pflash 0* allokiert und die verwendeten Variablen in die LMU 0. Zur Kompensation der Effekte von konkurrierenden Zugriffen wird der Programm-Cache aktiviert, welcher die Zugriffshäufigkeit auf die globalen Programmspeicher reduzieren kann. Was alle Lastszenarien auf allen Prozessorkernen gemeinsam haben, ist die minimale Laufzeit, welche im Optimalfall 48 Ticks beträgt. Im Gegensatz dazu korreliert die maximale sowie die durchschnittliche Laufzeit mit der Anzahl an aktiven Kernen, welche konkurrierend auf den *Pflash 0* zugreifen. Die Laufzeitschwankungen sind dabei relativ gering, was darauf zurückzuführen ist, dass die Cache-Line des AURIX 256 Bit lang ist, wodurch die komplette Funktion, welche im Kontext der ISR

Tabelle 5.23: Übersicht der Laufzeiten der ISR 0 für den Prozessorkern 0 auf Basis verschiedener Lastszenarien

Allokationsansatz	Lastszenario	Laufzeit in Ticks	Anzahl Prozessorkerne		
			2 Kerne	4 Kerne	6 Kerne
Standardeinstellung des Linkers	65% CPU Last	Minimum	48	48	48
		Durchschnitt	49	50	52
		Maximum	57	59	70
	80% CPU Last	Minimum	48	48	48
		Durchschnitt	49	51	53
		Maximum	58	61	75
	90% CPU Last	Minimum	48	48	48
		Durchschnitt	50	53	55
		Maximum	64	68	79
Stand der Technik	65% CPU Last	Minimum	46	46	46
		Durchschnitt	49	50	51
		Maximum	53	54	65
	80% CPU Last	Minimum	45	45	45
		Durchschnitt	50	52	54
		Maximum	53	55	67
	90% CPU Last	Minimum	45	45	45
		Durchschnitt	52	53	57
		Maximum	54	56	68
Optimierter Ansatz	65% CPU Last	Minimum	39	39	39
		Durchschnitt	39	39	39
		Maximum	42	42	42
	80% CPU Last	Minimum	39	39	39
		Durchschnitt	39	39	39
		Maximum	44	44	44
	90% CPU Last	Minimum	39	39	39
		Durchschnitt	39	39	39
		Maximum	45	45	45

ausgeführt wird, mit einem Zugriff vorgeladen werden kann. Neben der Anzahl der aktiven Kerne steigt die Laufzeit der ISR ebenfalls mit der CPU-Last, was darauf zurückzuführen ist, dass die damit verbundenen Cache-Interferenzen ansteigen.

Im zweiten Allokationsansatz wird die komplette Funktionalität, welche als Lastszenario für die ISR fungiert, lokal im Programm-Scratchpad von Prozessorkern 0 vorgehalten, was einen positiven Einfluss auf die Ausführungszeiten in Summe hat. Sowohl die minimale, die durchschnittliche als auch die maximale Laufzeit sind deutlich niedriger im Vergleich zum ersten Allokationsansatz. Jedoch sind auch hier Schwankungen in der Laufzeit, in Abhängigkeit der aktiven Prozessorkerne und des gewählten Lastszenarios, zu beobachten, was darin begründet ist, dass zwar die komplette Funktionalität mit den dazugehörigen Software-Bestandteilen im Scratchpad vorgehalten wird, jedoch nicht die Hilfsfunktionen, welche zur Ausführung benötigt werden. Zu diesen Hilfsfunktionen gehören zum einen die eigentliche ISR, welche nach dem Auftritt des asynchronen Events angesprungen wird, sowie Abstraktions-

schichten, welche im Rahmen einer geschichteten Software-Architektur vorhanden sind. Diese Zwischenschritte müssen daher vom Programm-Cache vorgeladen werden, was die Laufzeitschwankungen erklärt.

Mittels der dritten Messreihe wird der in dieser Arbeit vorgestellte Allokationsansatz genutzt, welcher den kompletten Programm-Code der ISR inklusive aller Hilfsfunktionen aufgrund des genutzten Priorisierungsansatzes detektiert und im Scratchpad des Prozessorkerns allokiert. Die Folge sind taktgenaue Laufzeiten, welche nicht variieren und damit deutlich einfacher in einer Laufzeitanalyse berücksichtigt werden können. Die in der Messung zu sehenden Laufzeitschwankungen sind auf das implementierte Verfahren zur Laufzeitbestimmung zurückzuführen. Sollte der Interrupt während einer laufenden Task-Ausführung auftreten, muss die entsprechende Messung zuerst pausiert werden, damit das Gesamtergebnis nicht verfälscht wird. Da die Laufzeitmessung ein Teil der ISR ist, sind die entsprechenden Effekte zu sehen. Je höher dabei die CPU-Last ist, desto häufiger treten entsprechende Effekte auf. In einem realen System ohne eine Bestimmung der Laufzeit sind diese Effekte jedoch nicht existent.

Aufgrund des Umstands, dass in der ISR 0 eine Funktionalität aus der Bitmanipulationsbibliothek von Infineon aufgerufen wird, welche keine Konstanten oder Variablen beinhaltet, ist primär der Programmspeicher für die Laufzeit verantwortlich.

5.4.4.2 Priorisierung von zeitunkritischen Elementen

Ein wichtiger Aspekt bei Echtzeitsystemen ist der Umstand, dass hochfrequentierte Funktionen nicht zwingend zeitkritisch sind. Ein Beispiel für diese Art von Aufgaben ist die Hintergrund-Task, welche immer dann ausgeführt wird, wenn keine zyklischen Tasks oder asynchrone Events bearbeitet werden müssen. Je nach Auslastung des Systems und des Umfangs der *Idle*-Task kann daher die Aufrufhäufigkeit sehr hoch sein im Vergleich zu anderen, zeitkritischen Funktionalitäten. Um diesem Umstand gerecht zu werden, kann mit Hilfe der Prioritätsberechnung der Wert der *Idle*-Task durch die Priorität des Betriebssystems entsprechend reduziert werden. Die korrekte Funktionsweise des im Rahmen dieser Arbeit entwickelten Algorithmus kann darüber sichergestellt werden, dass die Funktionen, Konstanten und Variablen, welche exklusiv von der *Idle*-Task genutzt werden, eine entsprechend niedrige Priorität aufweisen. Nichtsdestotrotz ist das Ziel der optimierten Allokation bei der Aufteilung der Software-Bestandteile auch die niederpriorisierten Funktionalitäten so zu allokieren, dass konkurrierende Zugriffe reduziert werden. Des Weiteren werden freie Speicherkapazitäten in den lokalen Scratchpads der Prozessorkerne ebenfalls für die *Idle*-Task genutzt, wenn diese vorhanden sind.

In der Tabelle 5.24 ist die Laufzeit sowie die Allokation im Speicher der *Idle*-Task exemplarisch für den Prozessorkern 1 für alle drei Allokationsansätze dargestellt. Aufgrund der Tatsache, dass bei einer optimierten Allokation die Auslastung der Prozessorkerne sinkt, können die Lastszenarien nur indirekt miteinander verglichen werden. Dieser Umstand ist darauf zurückzuführen, dass durch die optimierte Allokation die Berechnung der synchronen Tasks sowie der asynchronen Events deutlich

Tabelle 5.24: Übersicht der Laufzeiten der *Idle*-Task für den Prozessorkern 1 auf Basis verschiedener Allokationsansätze

Ergebnis		**Allokationsansatz**		
		Standard-einstellung des Linkers	**Stand der Technik**	**Optimierter Ansatz**
Speicher	**Programm-Code**	*Pflash 0* Programm-Cache 1	*Pflash 1* Programm-Cache 1	*Pflash 1* Programm-Cache 1 Programm-Scratchpad 1
	Konstanten	*Pflash 0*	*Pflash 1* Daten-Cache 1	*Pflash 1* Daten-Cache 1 Daten-Scratchpad 1
	Variablen	LMU	Daten-Scratchpad 1	Daten-Scratchpad 1
Laufzeit in Ticks	**Minimum**	72795036	34038256	20904145
	Durchschnitt	130915991	44135280	43743353
	Maximum	179152793	99252649	56209657

beschleunigt wird, weswegen mehr Rechenzeit für die Hintergrundaufgaben zur Verfügung steht. Um trotzdem eine Vergleichbarkeit herzustellen, werden ausschließlich die Ausführungszeiten der *Idle*-Task exklusiv aller Unterbrechungen gemessen und für das Lastszenario mit ursprünglich 90% Auslastung und sechs aktiven Prozessorkernen aufgeschlüsselt. Des Weiteren wird für jeden Allokationsansatz aufgezeigt, auf welche Speicher die *Idle*-Task aufgeteilt wird.

Wie den Messreihen in Tabelle 5.24 zu entnehmen ist, verbessert sich die Laufzeit der *Idle*-Task mit der Optimierung der Allokationsstrategie signifikant. Im Rahmen der ersten Messreihe wird die Standardeinstellung des Linkers genutzt, welcher lediglich die globalen Speicher für die Allokation verwendet. Da alle Prozessorkerne konkurrierend auf diese Speicher zugreifen, sind die Laufzeiten mit Abstand am höchsten und variieren auch am stärksten im Vergleich zu anderen Messreihen. Mittels des Stands der Technik wird die *Pflash*-Bank 1 für den Programm-Code sowie für die Konstanten genutzt, was die Laufzeit deutlich verbessert. Zur Reduzierung der langsameren Speicherzugriffe auf die globalen Speicher werden die beiden Caches verwendet, welche wiederholte Zugriffe durch das Vorhalten einer lokalen Kopie beschleunigen. Die Ablage aller Variablen der *Idle*-Task erfolgt im Daten-Scratchpad von Prozessorkern 0, da dieses genügende Speicherkapazität aufweist. Im optimierten Ansatz werden die freien Bereiche im Programm-Scratchpad zusätzlich genutzt, was die Ausführungszeit sowie die Laufzeitschwankungen reduziert. Durch die optimierte

Ablage der Konstanten im Speicher können zusätzlich die Cache-Interferenzen reduziert werden, was ebenfalls einen positiven Einfluss auf die Ausführungsgeschwindigkeit hat. Aufgrund von freien Kapazitäten im Daten-Scratchpad von Prozessorkern 0 können zusätzlich stark frequentierte Konstanten ebenfalls lokal vorgehalten werden.

5.5 Kapitelzusammenfassung

Zum Nachweis der korrekten Funktionalität des im Rahmen dieser Arbeit entwickelten Konzepts wird auf Basis eines selbstentwickelten Testsystems drei verschiedene Allokationsansätze miteinander verglichen. Als Hardware-Plattform wird dabei ein Mikrocontroller der Firma Infineon mit sechs Kernen genutzt sowie eine an den AUTOSAR-Standard angelehnte Testsoftware. Als repräsentative Lastszenarien werden eine Vielzahl bekannter Benchmarks für eingebettete Systeme genutzt. Wie die Testergebnisse auf Basis der neun verschiedenen Lastszenarien aufzeigen, kann das im Rahmen dieser Arbeit entwickelte Konzept die beiden anderen Allokationsverfahren deutlich überbieten. Die Bewertung erfolgt dabei anhand der vier Hauptziele, welche im zweiten Kapitel beschrieben sind. Dank der optimierten Allokation auf Basis der berechneten Priorisierung sowie der effizienten Nutzung der vorhandenen Speicher kann die temporale- als auch die räumliche Isolation verbessert, die Ausführungsgeschwindigkeit gesteigert sowie die Reaktionszeit reduziert werden, was anhand separater Messungen dargelegt wird.

6

Diskussion und Ausblick

In dem sich anschließenden Kapitel dieser Arbeit werden die erreichten Ergebnisse resümiert und in einen Kontext zur Zielstellung gesetzt. Des Weiteren werden potentielle Erweiterungen des entwickelten Konzepts zur optimierten statischen Speicherallokation erläutert und bewertet.

6.1 Diskussion

Nach Abschluss der durchgeführten Untersuchungen zeigt sich, dass die aufgestellten Zielstellungen aus Abschnitt 2.1 im Rahmen dieser Arbeit umgesetzt worden sind. Durch die Umsetzung einer Allokationsstrategie, welche das Gesamtsystem betrachtet, sind alle vier Hauptziele erfüllt sowie die dazugehörigen Unterpunkte vollständig umgesetzt. Die Basis für die optimierte Allokation ist dabei die Priorisierung aller Bestandteile der Software in Form der Funktionen, Variablen und Konstanten. Durch den Ansatz, nicht mehr die Funktionalitäten als kleinste allokierbare Einheit zu betrachten, können mehrere Vorteile gegenüber dem aktuellen Stand der Technik erreicht werden.

Durch die feingranularere Betrachtung der auszuführenden Software können besonders laufzeitrelevante Bestandteile extrahiert und in die schnellen lokalen Speicher der verwendeten Hardware-Plattform allokiert werden. Der Vorteil dieses Vorgehens besteht darin, dass die lokalen Speicher der Prozessorkerne ausschließlich von den zugeordneten Kernen genutzt werden, was die Anzahl von konkurrierenden Zugriffen reduziert und die Ausführungsgeschwindigkeit steigert. Neben den lokalen Speichern wird ebenfalls die Partitionierung der globalen Speicher genutzt, indem jedem Prozessorkern möglichst exklusiv eine Bank zur Verfügung gestellt wird. Durch die gezielte Allokation von Duplikationen können parallele Zugriffe auf globale Speicher reduziert werden. Jedoch ist dabei zu beachten, dass eine vollständige Verhinderung von konkurrierenden Zugriffen nur dann möglich ist, wenn in jeder Speicherbank genügend Speicherkapazität zum Vorhalten der notwendigen Duplikationen zur Verfügung steht. Ist dies nicht der Fall, können die Laufzeitanomalien, bedingt durch konkurrierende Zugriffe, minimiert, aber nicht vollständig verhindert werden. Eine Ausnahme bilden dabei die Variablen zur Intercore-Kommunikation,

welche zwingend in einem geteilten Speicher abgelegt werden müssen. Jedoch berücksichtigt das in dieser Arbeit entwickelte Konzept auch diesen Umstand und versucht Speicher zu nutzen, welche sich durch ihre Anbindung an das Gesamtsystem besonders gut für parallele Zugriffe eignen. Des Weiteren berücksichtigt die optimierte Allokation die Priorität der Variablen für die Intercore-Kommunikation für den jeweiligen Prozessorkern und versucht diese so abzulegen, dass die Funktionalität mit der höheren Priorität die kürzeren Zugriffszeiten hat.

Für die verbesserte Integration der Caches wird deren technische Umsetzung bei der Allokation berücksichtigt. Das Ziel ist dabei, die Größe der Cache *Pages* und die Anzahl der Assoziationen bei der optimierten Allokation einzubeziehen, wodurch Cache-Interferenzen signifikant reduziert werden können. Des Weiteren erfolgt bei der Nutzung der Caches die Berücksichtigung der Kritikalität der lokal vorgehaltenen Funktionalitäten, so dass nur eine Kritikalitätsstufe in den Caches des jeweiligen Prozessorkerns bereitgehalten wird. Dieser Umstand ist erforderlich, um der temporalen Isolation von Funktionalitäten unterschiedlicher Kritikalität gerecht zu werden. Durch die Festlegung, dass Daten-Caches lediglich als Lesespeicher für das lokale Vorhalten von Konstanten genutzt werden können, vereinfacht sich außerdem die Laufzeitanalyse.

Durch die fortwährende Integration speicherrelevanter Hardware-Beschleuniger ist deren Betrachtung bei einer optimierten Speicherallokation für das Gesamtsystem essentiell. Die Berücksichtigung erfolgt dabei durch den Ansatz, dass die Hardware-Beschleuniger als Prozessorkerne mit einer niedrigen Priorität betrachtet werden. Durch dieses Vorgehen werden die Hardware-Beschleuniger bei der Zuweisung der Speicherbänke berücksichtigt, jedoch liegt der Fokus weiterhin darauf die Hauptprozessorkerne möglichst zu entlasten. Neben der temporalen Isolation ist ebenfalls die räumliche Isolation ein signifikanter Bestandteil bei der optimierten Allokation. Zu diesem Zweck wird die Granularität des Speicherschutzes bei der Verteilung der Software-Bestandteile berücksichtigt.

In modernen Mehrkernmikrocontrollern werden verschiedene Speichertechnologien und komplexe Hierarchien zur Steigerung der Ausführungsgeschwindigkeit verwendet. So zeichnen sich beispielsweise die lokalen Speicher durch eine hohe Zugriffsgeschwindigkeit aus, verfügen dafür aber nur über eine geringe Speicherkapazität. Für eine möglichst effiziente Nutzung dieser geringen Speichergrößen werden die Zugriffshäufigkeiten und die Laufzeit der Software-Bestandteile bei der Priorisierung mitberücksichtigt. Des Weiteren erfolgt im Gegensatz zum Stand der Technik eine feingranularere Auswahl der zu allokierenden Elementen. Während derzeit komplette Funktionalitäten als kleinste allokierbare Einheit angesehen werden, nutzt das im Rahmen dieser Arbeit entwickelte Verfahren Funktionen, Konstanten und Variablen als kleinstes Element. Durch dieses Vorgehen können die lokalen Speicher deutlich effizienter genutzt werden, was sich in einer gesteigerten Ausführungsgeschwindigkeit sowie in einer konstanteren Laufzeit widerspiegelt.

Neben der optimierten Nutzung der lokalen Speicher erfolgt im Kontext dieser Arbeit ebenfalls die Betrachtung der globalen Speicher, wobei der Fokus auf deren Anbindung an die jeweiligen Prozessorkerne liegt. Zu diesem Zweck wird für jeden Prozessorkern im System eine Liste der Programm- und Datenspeicher angelegt und

deren Performance ermittelt. Auf Basis dieser Bewertung erfolgt im Anschluss die Zuweisung der Speicher zu den jeweiligen Prozessorkernen, was wiederum bei der optimierten Allokation berücksichtigt wird.

Eine weitere Möglichkeit zur Verbesserung der Ausführungsgeschwindigkeit ist die Untersuchung der Effekte eines optimierten Alignments. Wie die Messungen bei der genutzten AURIX-Plattform aufgezeigt haben, profitiert die TriCore-Architektur von einer auf 32 Bit ausgerichteten Ablagestrategie. Dabei ist jedoch zu beachten, dass ein entsprechendes Alignment den Speicherverbrauch signifikant erhöhen kann, weswegen dieser Ansatz nur dann genutzt werden kann, wenn entsprechende freie Speicherkapazitäten zur Verfügung stehen.

Die Einhaltung der Reaktionszeiten ist gerade in sicherheitskritischen Systemen essentiell, weswegen dies bei der Priorisierung berücksichtigt wird. Durch die Nutzung einer hohen Kritikalität bei der entsprechenden Funktionalität werden die dazugehörigen Software-Bestandteile bei der optimierten Allokation zuerst beachtet, was sich bei der Verarbeitungsgeschwindigkeit der ISRs zeigt. Im Gegensatz dazu kann diese Kritikalität auch dafür genutzt werden, zeitlich unkritische Hintergrundaufgaben erst im letzten Schritt der optimierten Allokation zu berücksichtigen.

6.2 Ausblick

Auf Basis der erreichten Ergebnisse im Rahmen dieser Arbeit wurde das Potential einer optimierten statischen Speicherallokation für echtzeitfähige Mehrkernsysteme aufgezeigt. Jedoch haben sich bei der Erarbeitung des vorgestellten Konzepts potentielle Erweiterungen ergeben, welche im Kontext weiterer Forschungen untersucht werden sollten. Den ersten Bereich betrifft das Thema der Laufzeitbestimmung sowie der Aufrufhäufigkeit innerhalb einer Hyperperiode. Aktuell erfolgt die Ermittlung dieser für die optimierte Allokation essentiellen Werte mittels eines statistischen Ansatzes auf Basis eines *Trace*-Mitschnitts. Dabei muss jedoch berücksichtigt werden, dass in Abhängigkeit der Eingangsgrößen die Applikationen unterschiedliche Programm-Code-Pfade durchlaufen, welche nicht zwingend den WCEP oder den Pfad mit der höchsten Zugriffshäufigkeit darstellen. Dieser Umstand kann zwar durch einen möglichst langen *Trace*-Mitschnitt reduziert werden, jedoch stellt dies keine Garantie dar und ist daher für ein System mit einer harten Echtzeitanforderung eine potentielle Unsicherheit, was in zukünftigen Arbeiten berücksichtigt werden sollte. Auf den Ergebnissen von Arbeiten zur automatischen Detektion von WCEPs und sich gegenseitig ausschließenden Programmpfaden kann hier eine Optimierung des in Kapitel 5.3 vorgestellten Programms vorgenommen werden. Dabei müssen jedoch auch Faktoren, wie *Pointer-Aliasing* oder *Function-Pointer*, berücksichtigt werden, deren finale Adresse erst zur Laufzeit bekannt ist und welche sich, je nach Programmpfad, ändern kann. Ein weiterer interessanter Aspekt ist die Anzahl der Wiederholungen, welche bei der statischen Speicheroptimierung vorgenommen werden. Durch einen iterativen Ansatz kann die Optimierung gegebenenfalls weiter verbessert werden, wobei alle Faktoren bei der Bewertung berücksichtigt werden müssen, wie beispielsweise die Auslastung der Prozessorkerne, die Reaktionszeit auf

asynchrone Events, aber auch die Menge der konkurrierenden Zugriffe auf geteilte Ressourcen. In zukünftigen Arbeiten könnte zu diesem Zweck eine Bewertungsmatrix erstellt werden, auf deren Basis die Optimierung so oft wiederholt wird, bis keine Verbesserung mehr eintritt [35].

Neben den genannten Erweiterungen könnte die Reduzierung der kleinsten allokierbaren Einheit der optimierten Allokation weitere Freiheitsgerade ermöglichen. Zu diesem Zweck würden statt vollständigen Funktionen nur rechenintensive *Basic Blocks* in die schnellen Scratchpad allokiert werden. Dadurch könnten noch mehr Zugriffe über die performanten lokalen Speicher umgesetzt werden, wodurch die geringe Speicherkapazität der Scratchpads effektiver genutzt werden würde. Dabei ist jedoch zu berücksichtigen, dass die Detektion von *Basic Blocks* nur auf Basis des kompilierten Programm-Codes erfolgen kann, da moderne Compiler eine Vielzahl von Optimierungen vornehmen. Diese aufwendige Analyse ist in Kombination mit der deutlich gesteigerten Anzahl an allokierbaren Einheiten eine rechenintensive Aufgabe, weswegen die Machbarkeit evaluiert werden muss. Des Weiteren nutzen moderne Mikrocontrollerarchitekturen relative Sprünge, welche die Codegröße reduzieren. Im Gegensatz zu absoluten Sprunginstruktionen sind diese aber in ihrer Reichweite limitiert, was durch verschiedene Ansätze kompensiert werden kann, jedoch die Komplexität weiter steigert [10].

Im Gegensatz zu den Scratchpads sowie den globalen Speichern zeichnen sich die Caches durch ein dynamisches Verhalten aus. Ihr Speicherinhalt wird zur Laufzeit in Abhängigkeit des Programmablaufs umgeladen, wobei dies von den umgesetzten Ersetzungsstrategien sowie der Verfügbarkeit von Cache-Locks beeinflusst wird. Bedingt durch diese Faktoren sind die Caches am schwersten in Laufzeitanalysen zu berücksichtigen, weswegen ihr Einsatz in Echtzeitsystemen umstritten ist [21] [20] [31]. Für die bessere Integration von Caches in die optimierte Speicherallokation sollen daher in Zukunft Konzepte untersucht werden, welche die Speicherinhalte von Cache-Inhalten durch Critical Sections absichern. Dieser Ansatz würde einem softwarebasierten Cache-Lock entsprechen und das Umladen der Cache-Inhalte durch asynchrone Events oder Kontextswitches bei Task-Wechseln vorbeugen. Dabei müssen jedoch die Reaktionszeiten berücksichtigt werden, welche weiterhin den Anforderungen entsprechen müssen. Neben diesem Ansatz könnte das im Rahmen dieser Arbeit entwickelte Konzept dahingehend erweitert werden, dass bei der Speicherauswahl zusätzlich das Verdrängungspotential für den jeweiligen Programm- oder Daten-Cache berücksichtigt wird. Beispielsweise könnten für asynchrone Events vorwiegend die Scratchpads genutzt werden, wodurch diese keine Cache-Interferenzen verursachen. Im Gegensatz dazu könnten dann die höchstpriorisierten Tasks auf dem jeweiligen Prozessorkern die Caches vorwiegend nutzen, da diese durch niederpriorisierte Tasks nicht unterbrochen werden können, was Cache-Interferenzen effektiv verhindert. Dieser Ansatz kann zusätzlich um die Möglichkeiten der verfügbaren Cache *Pages* erweitert werden, so dass bestimmten Tasks eine Cache *Pages* durch die optimierte Allokation indirekt zugewiesen wird. Durch diese Erweiterung könnten dann mehrere Tasks auf einem Prozessorkern von den Vorteilen der Caches profitieren, ohne das die Menge der Cache-Interferenzen signifikant ansteigt, was wiederum eine vereinfachte Laufzeitanalyse ermöglicht.

Bedingt durch den Umstand, dass auf echtzeitfähigen Mehrkernmikrocontrollern für sicherheitskritische Anwendungen Funktionalitäten unterschiedlicher Kritikalität ausgeführt werden, ist die Separierung des Speichers durch die MPU unumgänglich. Im hier vorgestellten Ansatz wird die MPU-Granularität für die Allokation berücksichtigt, wobei jedoch die kleinste separierbare Einheit der MPU als Basis angenommen wird. Dieses Vorgehen stellt in Kombination mit der Allokation von Funktionen und Variablen statt von Funktionalitäten einen signifikanten Vorteil dar, jedoch findet im Anschluss keine Bewertung statt, ob MPU-Bereiche mit identischer Kritikalität in einem Speicher zusammengelegt werden können. Durch diese Erweiterung wird die Fragmentierung des Speichers reduziert, da Freiräume in den MPU-Bereichen zusammengelegt werden, wodurch mehr Funktionen und Variablen in die schnellen lokalen Speicher allokiert werden könnten.

Eine weitere Herausforderung stellt die Integration von vorkompilierten Bibliotheken dar, welche besonders im kommerziellen Umfeld vorkommen können [194]. Bei diesen vorkompilierten Bibliotheken fehlen die notwendigen Meta-Informationen, welche der in Kapitel 4 vorgestellte Algorithmus zur Bestimmung der Priorität benötigt. Des Weiteren können die Adressen innerhalb des sogenannten *Objects* nicht angepasst werden, was eine Separierung deutlich erschwert. Die einzige Möglichkeit zur Integration entsprechender Funktionalitäten ist lediglich darüber gegeben, dass die gesamte Bibliothek als kleinste allokierbare Einheit angesehen wird und alle Zugriffe auf Funktionen der Bibliothek mitgezählt werden. Diese werden dann als Aufrufhäufigkeit in den Algorithmus gegeben und zusammen mit dem gesamten Speicherverbrauch der Bibliothek zur Berechnung der Priorität verwendet. Zur besseren Integration entsprechender *Objects* sollten daher in zukünftigen Arbeiten Ansätze untersucht werden, welche eine optimierte Berücksichtigung entsprechender vorkompilierter Bibliotheken ermöglichen.

Neben den genannten Faktoren, welche die Verbesserung der Genauigkeit des Bewertungsalgorithmus zur Folge haben, gibt es noch praktische Weiterentwicklungsmöglichkeiten, welche moderne Mikrocontrollerfunktionen besser integrieren. Beispielsweise werden in der Automobilindustrie bei der Erprobung von Fahrzeugen sogenannte Konfigurationsparameter appliziert, welche das Fahrverhalten des Automobils signifikant beeinflussen. Zu diesem Zweck werden statische Konfigurationsparameter in einen RAM-Speicher geladen und können dadurch zur Laufzeit manipuliert werden. Zur Vermeidung von Fehlern, bedingt durch Laufzeitanomalien, welche durch die unterschiedlichen Speicherzugriffe entstehen können, bieten aktuelle Mikrocontroller ein sogenanntes Overlay-Feature. Mittels dieser Funktion werden Anfragen auf den Speicher der Konfigurationsparameter automatisch auf den RAM umgeleitet. Dieser Vorgang ist für die Software vollständig transparent und erfordert daher keine Modifikationen an der Programmausführung. Da dieses Feature häufig für ausgewählte Speicherbereiche limitiert ist, sollten diese bei einer automatisierten Allokation berücksichtigt werden. Eine weitere essentielle Funktionalität von modernen Mikrocontrollern sind sogenannte One-Time Programmable (OTP)-Bereiche. Mit Hilfe dieser Speicherbereiche kann ein Überschreiben in Hardware verhindert werden, wodurch die Inhalte permanent abgelegt sind. Diese Funktion wird häufig für Produktionsmerkmale oder für die Ablage von kryptografischen Zertifika-

ten genutzt, wodurch diese effektiv gegen Manipulationen geschützt sind. Analog zu dem Overlay-Feature sind auch die OTP-Bereiche für ausgewählte Speichersektionen limitiert und sollten daher bei der optimierten Allokation berücksichtigt werden. Für die Reduzierung des Energieverbrauchs können die Prozessorkerne in aktuellen Mikrocontrollerfamilien im Standby abgeschaltet werden. Durch die Abschaltung der dazugehörigen Versorgungsspannung werden jedoch auch die Inhalte der RAM-Speicher geleert, was bei einer Reaktivierung zu ungewollten Zuständen führen kann. Aus diesem Grund haben die Hersteller entsprechender Mikrocontrollerderivate sogenannte Standby-Features integriert, welche den Inhalt der RAM-Speicher auch beim Abschalten der dazugehörigen Prozessorkerne halten. Der Vorteil dieses Ansatzes liegt in einer reduzierten Reaktivierungszeit, da keine Inhalte aus dem Flash nachgeladen werden müssen, und der Möglichkeit auf der Basis bestehender Zustände die Ausführung fortzusetzen. Wie bei den anderen Hardware-Funktionalitäten auch, ist die Standby-Funktion auf bestimmte Speicherbereiche beschränkt. Neben den genannten Sonderfunktionen der verschiedenen Speicher integrieren die Hersteller der Mehrkernmikrocontroller zunehmend Erweiterungen, welche die Effekte von konkurrierenden Zugriffen bereits in der Hardware reduzieren sollen. Beispielsweise besitzt die zweite AURIX-Generation von Infineon pro Flash-Bank insgesamt vier Prefetch-Buffer, welche je einem Prozessorkern zugeordnet werden können. Bei der ersten AURIX-Generation besitzt jede Bank des *Pflash* lediglich einen Prefetch-Buffer, welcher bei einem konkurrierendem Zugriff von verschiedenen Kernen permanent umgeladen wird. Durch die Bereitstellung mehrerer Prefetch-Buffer können die Effekte dieser Zugriffe reduziert werden, was ebenfalls bei der optimierten Allokation berücksichtigt werden sollte. Das Ziel sollte dabei sein, dass nicht mehr Prozessorkerne auf einen Speicher zugreifen als dieser entsprechende Beschleuniger zum Vorladen bereithält. Diese Funktionalität könnte besonders dann relevant werden, wenn die Speicherkapazität zur Bereitstellung lokaler Kopien für jeden Prozessorkern nicht ausreicht und konkurrierende Zugriffe daher unumgänglich sind. Durch die effektive Berücksichtigung der Menge der verfügbaren Prefetch-Buffer pro Speicherbank könnten die geteilten Funktionen so verteilt werden, dass nur maximal soviele Kerne auf die jeweilige Bank zugreifen, wie Prefetch-Buffer vorhanden sind. Für die Berücksichtigung der genannten Hardware-Features soll in zukünftigen Arbeiten die Systembeschreibung sowie der Allokationsalgorithmus aus Kapitel 4 um die entsprechenden Funktionalitäten erweitert werden.

Zusätzlich ist das Ziel für weitere Arbeiten die Evaluierung des hier vorgestellten Konzepts auf verschiedenen Hardware-Plattformen. Im Rahmen dieser Arbeit erfolgte die Erprobung auf einem Infineon AURIX TC39x, welcher auf der proprietären TriCore-Architektur basiert. Auch wenn viele der derzeit am Markt erhältlichen Mehrkernmikrocontoller für sicherheitskritische Anwendungen auf einen ähnlichen Aufbau setzen, können Unterschiede in der konkreten Umsetzung der Mikrocontroller einen Einfluss auf die Effektivität des entwickelten Ansatzes zur statischen Speicherallokation haben. Ein besonderer Fokus sollte dabei auf den Mikrocontrollerfamilien mit einer ARM- beziehungsweise einer RISC-V-Architektur liegen, da immer mehr Hersteller ihre Eigenentwicklungen zugunsten der genannten Architekturen einstellen.

Abschließend kann der hier vorgestellte Ansatz zur statischen Allokation signifikant erweitert werden, in dem auf der Basis der erhobenen Informationen eine optimierte Verteilung der Funktionalitäten auf die Prozessorkerne vorgenommen wird. Durch die Zugriffsdaten aus dem *Trace* kann ermittelt werden, wie hoch die Interaktion von Funktionalitäten untereinander ist und diese Wirkketten können dann auf einem Prozessorkern zusammengefasst werden. Dies würde die Intercore-Kommunikation reduzieren, was wiederum die Allokation vereinfachen würde. Mittels der Wirkketten, der WCET und den verfügbaren MPU-Regionen zur Separierung unterschiedlicher Kritikalitäten pro Prozessorkern würden die Funktionalitäten auf die Prozessorkerne verteilt werden und im Anschluss könnte auf dieser Basis eine optimierte Allokation erfolgen.

6.3 Kapitelzusammenfassung

Im letzten Kapitel dieser Arbeit werden die erreichten Ergebnisse kurz resümiert und im Anschluss ein Ausblick auf potentielle Erweiterungen gegeben. Dabei wird ein besonderer Fokus auf die zur Priorisierung genutzten Aufrufhäufigkeit sowie Ausführungsgeschwindigkeit gelegt. Beide Werte werden derzeit aus einer Messung auf Basis einer Hyperperiode ermittelt, weswegen nicht garantiert werden kann, dass es sich dabei um den WCEP handelt. Des Weiteren werden zusätzliche Verbesserungen vorgeschlagen, welche die Cache-Nutzung optimieren sowie weitere Besonderheiten von Speichern, wie OTP-Funktionalitäten, Overlay-Features oder *Prefetch-Buffer*, berücksichtigen. Neben diesen Aspekten werden weitere Ansätze diskutiert, welche die Integration von vorkompilierten Bibliotheken erleichtern.

7

Appendix

7.1 Benchmarks

In dem sich anschließenden Abschnitt werden die verwendeten Benchmarks sowie die darin enthaltenen Testfälle klassifiziert. Für die Messung der Laufzeiten werden die Einstellungen der Toolkette aus Abschnitt 5.2.2.4 verwendet. Des Weiteren wird von den sechs Prozessorkernen des verwendeten Infineon AURIX TC39x nur CPU0 verwendet und die anderen fünf Prozessorkerne deaktiviert, um Laufzeitanomalien durch konkurrierende Zugriffe zu verhindern. Für die Allokation der Testfälle werden ausschließlich die beiden globalen Speicher *Pflash 0* für die Ablage des Programm-Codes sowie für Konstanten und die LMU für die Allokation der Variablen genutzt. Die beiden Caches des TriCore-Kerns sind deaktiviert, wodurch Laufzeitanomalien aufgrund von Cache-Interferenzen verhindert werden.

7.1.1 Tasking Dhrystone

Tabelle 7.1: Tasking Dhrystone Testfälle

Testfall	**Lauf-zeit/µS**	**Programm-Code/Byte**	**Konstan-ten/Byte**	**Variablen/-Byte**
Dhrystone	32,02	922	124	10242

7.1.2 EEMBC CoreMark

Tabelle 7.2: EEMBC CoreMark Testfälle

Testfall	**Lauf-zeit/µS**	**Programm-Code/Byte**	**Konstan-ten/Byte**	**Variablen/-Byte**
CoreMark	4406,28	3574	128	280

7.1.3 Embench IoT

Tabelle 7.3: Embench IoT Testfälle

Testfall	**Lauf-zeit/µS**	**Programm-Code/Byte**	**Konstan-ten/Byte**	**Variablen/-Byte**
aha-mont64	76,00	1076	0	24
crc32	341,49	76	1024	0
cubic	2474,66	1624	384	40
edn	379,09	1254	1600	1610
huffbench	1442,12	1000	500	8692
matmult-int	869,88	322	1600	8004
md5sum	234,91	552	512	3096
minver	58,98	1032	0	400
nbody	20970,41	794	0	1280
nettle-aes	636,81	2330	9738	2088
nettle-sha256	40,92	5216	288	208
nsichneu	48,30	12388	56	80
picojpeg	7826,07	7682	1894	2401
primecount	23938,58	138	0	0
qrduino	6918,83	5654	861	8228
slre	344,25	2102	57	124
st	3664,50	776	0	1628
statemate	12,39	3610	64	227
tarfind	235,95	220	0	16380
ud	31,91	574	80	1764
wikisort	23624,73	5104	3236	3200

7.1.4 Infineon CRC Bibliothek

Tabelle 7.4: Infineon CRC Bibliothek Testfälle

Testfall	Laufzeit/µS	Programm-Code/Byte	Konstanten/Byte	Variablen/-Byte
CRC 8 Runtime	56,04	88	0	0
CRC 8 Table	9,02	62	256	0
CRC 8H2F Runtime	55,76	88	0	0
CRC 8H2F Table	4,71	62	256	0
CRC 16 Runtime	50,03	76	0	0
CRC 16 Table	5,43	56	512	0
CRC 32 Runtime	116,14	158	0	0
CRC 32 Table	69,19	148	1024	0

7.1.5 Infineon Bitmanipulation Bibliothek

Tabelle 7.5: Infineon Bitmanipulation Bibliothek Testfälle

Testfall	Laufzeit/µS	Programm-Code/Byte	Konstanten/Byte	Variablen/-Byte
SetBit 32 Bit	0,49	20	0	0
ClearBit 32 Bit	0,46	20	0	0
GetBit 32 Bit	0,40	10	0	0
TestBitMask 32 Bit	0,40	8	0	0
TestParityEven 32 Bit	0,47	10	0	0
PutBit 32 Bit	0,45	26	0	0

7.1.6 IAV quantumSAR

Die Trennung des Speicherbedarfs für die einzelnen Testfälle ist im Falle der Bibliothek für Post-Quantum-Kryptografie nur bedingt möglich, da es zwischen den einzelnen Testfällen im Programm-Code diverse Abhängigkeiten gibt. Daher wird der Speicherbedarf nur für das gesamte Modul, welches für die Berechnung des jeweiligen Algorithmus benötigt wird, angegeben.

Tabelle 7.6: IAV quantumSAR Testfälle

Testfall	**Laufzeit/µS**	**Programm-Code/Byte**	**Konstanten/Byte**	**Variablen/-Byte**
ML-KEM 512 Key Pair Gen.	13133,72			
ML-KEM 512 Encapsulate	9063,54			
ML-KEM 512 Decapsulate	10027,94	3290	256	6560
ML-KEM 512 Encryption	7072,14			
ML-KEM 512 Decryption	1873,34			
ML-DSA 2 Key Pair Gen.	19542,10			
ML-DSA 2 Signature Calc.	31559,40			
ML-DSA 2 Verify Signature	21003,06	6242	1024	8773
ML-DSA 2 Sign Message	31559,81			
ML-DSA 2 Verify Message	21003,94			
FN-DSA 1024 Key Pair Gen.	125353,85			
FN-DSA 1024 Signature Calc.	2934605,83			
FN-DSA 1024 Verify Signature	178687,14	26648	29868	115263
FN-DSA 1024 Sign Message	2934606,61			
FN-DSA 1024 Verify Message	180340,71			
SLH-DSA SHA-128f Key Pair Gen.	4053969,57			
SLH-DSA SHA-128f Signature Calc.	1036526,84			
SLH-DSA SHA-128f Verify Signature	16263,86	25872	428	34341
SLH-DSA SHA-128f Sign Message	1036114,39			
SLH-DSA SHA-128f Verify Message	16275,58			

7.1.7 IAV CopyData Bibliothek

Tabelle 7.7: IAV CopyData Bibliothek Testfälle

Testfall	**Laufzeit/µS**	**Programm-Code/Byte**	**Konstanten/Byte**	**Variablen/Byte**
Transfer 64 * 8 Bit	25,99	330	0	4
Transfer 64 * 16 Bit	28,44	352	0	8
Transfer 64 * 32 Bit	29,57	330	0	16
Transfer 64 * 32 Bit (Float)	28,28	330	0	16

7.1.8 SystemMD SipHash-2-4

Tabelle 7.8: SystemMD SipHash-2-4 Testfälle

Testfall	Laufzeit/µS	Programm-Code/Byte	Konstanten/Byte	Variablen/-Byte
SipHash-2-4	18,82	1674	0	448

7.1.9 Universität zu Lübeck AES-256/SHA-256

Tabelle 7.9: Universität zu Lübeck AES-256/SHA-256 Testfälle

Testfall	Laufzeit/µS	Programm-Code/Byte	Konstanten/Byte	Variablen/-Byte
AES-256 CBC	202,10	2986	0	16976
AES-256 CTR	123,50	2898	0	16976
AES-256 ECB	184,68	2910	0	16976
SHA-256	29,16	710	0	704

7.2 Testfälle

Wie in Kapitel 5 beschrieben, werden zum Nachweis des vorgestellten Konzepts drei verschiedene Lastszenarien mit einer unterschiedlichen Prozessorauslastung verwendet. Eine genaue Beschreibung darüber, welcher Kern welche Funktionalität mit welcher Kritikalität ausführt, kann dem folgenden Abschnitt entnommen werden.

Die Realisierung der Intertask- und Intercore-Kommunikation erfolgt durch die IAV CopyData Bibliothek, welche sowohl am Anfang als auch am Ende einer jeden Task-Ausführung verschiedene Kopiervorgänge ausführt. Diese Umsetzung entspricht der Realisierung in vielen Steuergeräten, bei welcher zu Beginn einer Task-Ausführung eine lokale Kopie aller Daten von anderen Tasks angelegt wird, damit während der eigentlichen Task-Ausführung auf konsistente Daten zugegriffen wird. Am Ende der Task werden die berechneten Werte zurückgeschrieben und den anderen Funktionalitäten zur Verfügung gestellt [3].

Ein wichtiger Aspekt ist der Umstand, dass der verwendete Infineon AURIX TC39x pro Prozessorkern lediglich drei unterschiedliche Kritikalitätsstufen umsetzen kann. Diese Limitierung ist auf die MPU der TriCore-Kerne zurückzuführen, welche nur drei MPU-Sets unterstützt. Da das Betriebssystem für die Umschaltung der MPU-Konfigurationssätze verantwortlich ist, wird dieses automatisch in die höchste Kritikalitätsstufe aller Funktionalitäten auf dem jeweiligen Prozessorkern eingruppiert. Neben dem Betriebssystem führt jeder Kern eine Hintergrund-Task aus, welche aufgrund der fehlenden zeitlichen Anforderung keinerlei Kritikalität aufweist, trotzdem aber ein MPU-Set benötigt. Aus diesem Grund können pro Prozessorkern neben der *Idle*-Task nur Funktionalitäten mit maximal zwei unterschiedlichen Kritikalitätsstufen ausgeführt werden. Sollte eine Funktionalität ebenfalls keine Kritikalität aufweisen, kann diese im Kontext der Hintergrund-Task berechnet werden. Die Definition der Sicherheitslevel erfolgt auf Basis der in der ISO 26262 definierten Stufen [38].

Die Umsetzung der Kommunikationsschnittstellen für CAN und Ethernet erfolgt durch den Prozessorkern 0, welcher im Rahmen der *Idle*-Task die Botschaften verschickt. Die entsprechenden Datenpakete werden durch die zyklischen Task vorbereitet und im Anschluss als Hintergrundaufgabe versendet. Daraus folgt, dass die entsprechenden Treiber sowie die dazugehörigen Module zur Protokollumsetzung die niedrigste Priorität im System haben.

Tabelle 7.10: Lastszenario Prozessorkern 0

Testfall	Task	Sicherheits-level	Anzahl Wiederholungen im Lastszenario		
			65% CPU	80% CPU	90% CPU
Dhrystone	1000ms	ASIL D	16	20	24
SipHash-2-4	1000ms	ASIL B	16	20	24
AES-256 CBC	1000ms	ASIL B	16	20	24
AES-256 CTR	1000ms	ASIL B	16	20	24
AES-256 ECB	1000ms	ASIL B	16	20	24
SHA-256	1000ms	ASIL B	16	20	24
CoreMark	100ms	ASIL D	16	20	22
crc32	100ms	ASIL D	16	20	22
cubic	10ms	ASIL B	3	4	4
CRC 16 Table	1ms	QM	2	2	18
SetBit 32 Bit	ISR 0	ASIL D	1	1	1
ClearBit 32 Bit	ISR 1	ASIL D	1	1	1
GetBit 32 Bit	ISR 2	ASIL D	1	1	1
TestBitMask 32 Bit	ISR 3	ASIL D	1	1	1
TestParityEven 32 Bit	ISR 4	ASIL D	1	1	1
PutBit 32 Bit	ISR 5	ASIL D	1	1	1
CRC 8 Runtime	ISR 6	ASIL B	1	1	1
CRC 8 Table	ISR 7	ASIL B	1	1	1
ML-KEM 512	*Idle*	QM	1	1	1

Tabelle 7.11: Lastszenario Prozessorkern 1

Testfall	Task	Sicherheits-level	Anzahl Wiederholungen im Lastszenario		
			65% CPU	80% CPU	90% CPU
Dhrystone	100ms	ASIL C	3	4	6
CoreMark	100ms	ASIL C	3	4	6
edn	100ms	ASIL A	3	4	6
matmult-int	100ms	ASIL A	3	4	6
huffbench	10ms	QM	3	3	3
AES-256 CBC	10ms	QM	3	3	3
AES-256 CTR	10ms	QM	3	3	3
AES-256 ECB	10ms	QM	3	3	3
SipHash-2-4	1ms	ASIL A	2	6	7
SHA-256	1ms	ASIL A	2	6	7
CRC 8 Runtime	ISR 0	ASIL A	1	1	1
CRC 8 Table	ISR 1	ASIL A	1	1	1
CRC 8H2F Runtime	ISR 2; 1000ms	ASIL A	1; 4	1; 4	1; 16
CRC 8H2F Table	ISR 3; 1000ms	ASIL A	1; 4	1; 4	1; 16
CRC 16 Runtime	ISR 4; 1000ms	ASIL A	1; 4	1; 4	1; 16
CRC 16 Table	ISR 5; 1000ms	ASIL A	1; 4	1; 4	1; 16
CRC 32 Runtime	1ms Task	ASIL A	2	1	1
CRC 32 Table	1ms Task	ASIL A	2	1	1
SetBit 32 Bit	ISR 6	ASIL A	1	1	1
ClearBit 32 Bit	ISR 7	ASIL A	1	1	1
ML-DSA 2	*Idle*	QM	1	1	1

Tabelle 7.12: Lastszenario Prozessorkern 2

Testfall	Task	Sicherheits-level	Anzahl Wiederholungen im Lastszenario		
			65% CPU	80% CPU	90% CPU
AES-256 CBC	1000ms	QM	32	32	32
AES-256 CTR	1000ms	QM	32	32	32
AES-256 ECB	1000ms	QM	32	32	32
CRC 8H2F Runtime	1000ms	ASIL B	32	32	32
CRC 8H2F Table	1000ms	ASIL B	32	32	32
CRC 16 Runtime	1000ms	QM	32	32	32
CRC 16 Table	1000ms	QM	32	32	32
CRC 32 Runtime	1000ms	QM	32	32	32
CRC 32 Table	1000ms	QM	32	32	32
CoreMark	100ms	QM	16	16	16
md5sum	100ms	ASIL B	16	16	16
nettle-aes	10ms	QM	3	4	4
minver	1ms	ASIL B	2	5	9
CRC 8 Runtime	ISR 0; 1000ms	ASIL B	1; 32	1; 32	1; 32
CRC 8 Table	ISR 1; 1000ms	ASIL B	1; 32	1; 32	1; 32
SetBit 32 Bit	ISR 2	ASIL D	1	1	1
ClearBit 32 Bit	ISR 3	ASIL D	1	1	1
GetBit 32 Bit	ISR 4	ASIL D	1	1	1
TestBitMask 32 Bit	ISR 5	ASIL D	1	1	1
TestParityEven 32 Bit	ISR 6	ASIL D	1	1	1
PutBit 32 Bit	ISR 7	ASIL D	1	1	1
SLH-DSA SHA-128f	*Idle*	QM	1	1	1

Tabelle 7.13: Lastszenario Prozessorkern 3

Testfall	Task	Sicherheits-level	Anzahl Wiederholungen im Lastszenario		
			65% CPU	80% CPU	90% CPU
nbody	1000ms	ASIL C	2	16	16
CoreMark	100ms	ASIL D	2	2	5
AES-256 CBC	10ms	ASIL C	4	4	4
AES-256 CTR	10ms	ASIL D	4	4	4
AES-256 ECB	10ms	QM	4	4	4
SipHash-2-4	10ms	QM	4	4	4
SHA-256	10ms	QM	4	4	4
CRC 16 Table	1ms	QM	8	8	9
CRC 32 Runtime	1ms	QM	8	8	9
CRC 8 Runtime	ISR 0	ASIL D	1	1	1
CRC 8 Table	ISR 1	ASIL D	1	1	1
CRC 8H2F Runtime	ISR 2	ASIL D	1	1	1
CRC 8H2F Table	ISR 3	ASIL D	1	1	1
CRC 16 Runtime	ISR 4	ASIL D	1	1	1
CRC 16 Table	ISR 5	ASIL D	1	1	1
TestParityEven 32 Bit	ISR 6	ASIL C	1	1	1
PutBit 32 Bit	ISR 7	ASIL C	1	1	1
FN-DSA 1024	*Idle*	QM	1	1	1

Tabelle 7.14: Lastszenario Prozessorkern 4

Testfall	Task	Sicherheits-level	Anzahl Wiederholungen im Lastszenario		
			65% CPU	80% CPU	90% CPU
CoreMark	1000ms	ASIL A	1	4	12
nettle-sha256	1000ms	ASIL A	1	4	12
primecount	1000ms	ASIL C	1	4	12
nsichneu	100ms	QM	1	2	3
AES-256 CBC	100ms	QM	1	2	3
AES-256 CTR	100ms	QM	1	2	3
AES-256 ECB	100ms	ASIL A	1	2	3
picojpeg	10ms	ASIL C	1	2	2
SipHash-2-4	1ms	ASIL A	1	2	2
SHA-256	1ms	ASIL C	1	2	2
CRC 8 Runtime	ISR 0	ASIL A	1	1	1
CRC 8 Table	ISR 1	ASIL C	1	1	1
CRC 8H2F Runtime	ISR 2	ASIL A	1	1	1
CRC 8H2F Table	ISR 3	ASIL C	1	1	1
CRC 16 Runtime	ISR 4	ASIL A	1	1	1
CRC 16 Table	ISR 5	ASIL C	1	1	1
CRC 32 Runtime	ISR 6	ASIL A	2	1	1
CRC 32 Table	ISR 7	ASIL C	2	1	1
primecount	*Idle*	QM	1	1	1
nbody	*Idle*	QM	1	1	1
wikisort	*Idle*	QM	1	1	1

Tabelle 7.15: Lastszenario Prozessorkern 5

Testfall	Task	Sicherheits-level	Anzahl Wiederholungen im Lastszenario		
			65% CPU	80% CPU	90% CPU
CoreMark	1000ms	QM	1	10	20
Dhrystone	1000ms	QM	1	10	20
qrduino	1000ms	ASIL A	1	10	20
slre	1000ms	ASIL A	1	10	20
st	1000ms	ASIL B	1	10	20
statemate	1000ms	ASIL B	1	10	20
AES-256 CBC	1000ms	ASIL B	1	10	20
AES-256 CTR	1000ms	ASIL A	1	10	20
tarfind	100ms	QM	3	3	3
ud	100ms	ASIL A	3	3	3
wikisort	100ms	ASIL B	3	3	3
SipHash-2-4	100ms	QM	3	3	3
SHA-256	100ms	ASIL A	3	3	3
SetBit 32 Bit	10ms	ASIL A	16	16	16
ClearBit 32 Bit	10ms	ASIL B	16	16	16
GetBit 32 Bit	10ms	ASIL A	16	16	16
TestBitMask 32 Bit	10ms	ASIL B	16	16	16
TestParityEven 32 Bit	10ms	ASIL A	16	16	16
PutBit 32 Bit	10ms	ASIL B	16	16	16
AES-256 CBC	1ms	QM	1	1	1
CRC 8 Runtime	ISR 0	ASIL A	1	1	1
CRC 8 Table	ISR 1	ASIL B	1	1	1
CRC 8H2F Runtime	ISR 2	ASIL B	1	1	1
CRC 8H2F Table	ISR 3	ASIL A	1	1	1
CRC 16 Runtime	ISR 4	ASIL A	1	1	1
CRC 16 Table	ISR 5	ASIL B	1	1	1
CRC 32 Runtime	ISR 6	ASIL B	2	1	1
CRC 32 Table	ISR 7	ASIL A	2	1	1
primecount	*Idle*	QM	1	1	1
nbody	*Idle*	QM	1	1	1
wikisort	*Idle*	QM	1	1	1

Tabellenverzeichnis

Abbildungsverzeichnis

Akronyme

ABS Antiblockiersystem. 33
ADC Analog-to-Digital Converter. 35, 80
AES Advanced Encryption Standard. 22, 79, 116, 125
ASIL Automotive Safety Integrity Level. 14, 17
AURIX AUtomotive Realtime Integrated NeXt Generation Architecture. 4, 6, 17, 18, 20, 24, 26, 29, 59, 69–78, 83, 91, 93–96, 99, 100, 107, 110, 112, 117, 124
AUTOSAR AUTomotive Open System ARchitecture. 9, 32–34, 42, 73–75, 77, 79, 104

CAN Controller Area Network. 34, 95, 117
CAN-FD Controller Area Network Flexible Data-Rate. 69, 75
CDD Complex Device Driver. 34
CPU Central Processing Unit. 82, 101, 112
CRC Cyclic Redundancy Check. 77–79, 114, 125
CSA Context Save Area. 82, 83
CUDA Compute Unified Device Architecture. 81

dLMU Distributed Local Memory Unit. 71, 72, 91, 92, 95
DMA Direct Memory Access. 11, 22, 37–41, 53, 54, 72, 73, 75, 95
DRAM Dynamic Random Access Memory. 37
DSP Digital Signal Processor. 19, 20

EBU External Bus Unit. 72
ECC Error-Correcting Code. 30, 31
ECU Electronic Control Unit. 9, 33
ESP Elektronisches Stabilitätsprogramm. 7, 33

FIFO First-In-First-Out. 28
FPU Floating Point Unit. 70
FSM Firmware Security Module. 79

GPIO General Purpose Input/Output. 34, 69, 75
GPSR General Purpose Service Request. 76
GTM Generic Timer Module. 20, 72

HSM Hardware Security Module. 11, 20, 72, 79

iLLD Infineon Low Level Drivers. 74, 75
ISO International Organization for Standardization. 9, 10, 93, 117
ISR Interrupt Service Routine. 76, 82–86, 100–102, 107, 124
IVT Interrupt Vector Table. 83, 84

JTAG Joint Test Action Group. 69

LED Light-Emitting Diode. 75
LET Logical Execution Time. 42, 43
LIN Local Interconnect Network. 34
LMU Local Memory Unit. 71, 72, 82, 87, 99, 100, 103, 112
LRU Least recently used. 28, 66, 71, 92
lwIP lightweight IP. 75

MAC Message Authentication Code. 79
MCAL Microcontroller Abstraction Layer. 33, 74, 77, 78
MMU Memory Management Unit. 31, 38, 41, 52
MPU Memory Protection Unit. 12, 31, 38, 56, 60, 64–66, 72–74, 87, 93–95, 109, 111, 117, 126

NCCNUMA Non-Cache Coherent Non-Uniform Memory Access. 25
NIST National Institute of Standards and Technology. 78
NoC Network-on-Chip. 18, 22, 31
NUMA Non-Uniform Memory Access. 25

OpenMP Open Multi-Processing. 81
OS Operating System. 32, 85
OSEK Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug. 32
OTP One-Time Programmable. 109–111

POSIX Portable Operating System Interface. 32
PWM Pulsweitenmodulation. 20, 72, 80

RAM Random-Access Memory. 18, 29, 30, 59, 63, 66, 71, 82, 109, 110
RISC Reduced Instruction Set Computer. 20
RMS Rate Monotonic Scheduling. 10, 35
ROM Read-Only Memory. 29, 30, 66, 82
RRAM Resistive Random Access Memory. 29
RTE Runtime Environment. 33–35
RTOS Real-Time Operating System. 75

SBC System Basis Chip. 33
SCR Standby Controller. 72
SHA Secure Hash Algorithm. 22, 79, 116, 125
SLH-DSA Stateless Hash-Based Digital Signature Algorithm. 82
SPI Serial Peripheral Interface. 34
SPOF Single Point of Failure. 43
SRAM Static Random-Access Memory. 25, 26, 30
SWC Software Component. 35

TCM Tightly Coupled Memory. 25
TLB Translation Lookaside Buffer. 32

TRNG True Random Number Generator. 22
TVT Trap Vector Table. 83, 84

USB Universal Serial Bus. 69

WCEP Worst-Case Execution Path. 36, 37, 39, 47, 107, 111
WCET Worst-Case Execution Time. 11–14, 22, 32, 36, 39, 45, 49, 66, 111
WCRT Worst-Case Response Time. 43
WLAN Wireless Local Area Network. 34

XML Extensible Markup Language. 88

Publikationen

Fachartikel

1. P. Jungklass und M. Berekovic Performance-Oriented Memory Management for Embedded Multicore Microcontrollers. In: *26th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*. 2018. ISBN: 978-3-902457-49-3
2. P. Jungklass und M. Berekovic Effects of concurrent access to embedded multicore microcontrollers with hard real-time demands. In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, S. 1–9. ISBN: 978-1-5386-4155-2/18
3. P. Jungklass und M. Berekovic Intercore-Kommunikation für Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2018*. 2018. ISBN: 978-3-8343-3447-3
4. P. Jungklass und M. Berekovic Performance-orientiertes Speichermanagement bei embedded Multicore-Mikrocontrollern. In: *Tagungsband Embedded Software Engineering Kongress 2018*. 2018. ISBN: 978-3-8343-3447-3
5. P. Jungklass und M. Berekovic Cache-Kohärenz für embedded Multicore-Mikrocontroller mit harter Echtzeitanforderung. In: *Echtzeit 2019*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 129–138. ISBN: 978-3-658-27808-3
6. P. Jungklass und M. Berekovic MemOpt: Automated Memory Distribution for Multicore Microcontrollers with Hard Real-Time Requirements. In: *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. Okt. 2019, S. 1–7. ISBN: 978-1-7281-2769-9/19. DOI: 10.1109/NORCHIP.2019.8906914
7. P. Jungklass, C. Schmidt, T. Jungklass und M. Berekovic Scheduling-Konzepte für echtzeitfähige embedded Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2019*. 2019. ISBN: 978-3-8343-3463-3
8. P. Jungklass, C. Schmidt und M. Berekovic Redundanzkonzepte für embedded Multicore Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2019*. 2019. ISBN: 978-3-8343-3463-3
9. T. Fricke, C. Uzlu, R. Mallwitz, J. Ries, J. Wussow, J. Brockschmidt, M. Kurrat, B. Engel, P. Jungklass und F. Grieger NetProsum2030: A Contribution to the Solution for Distributed Energy Supply in 2030. In: *PCIM Europe*. 2020. ISBN: 978-3-8007-5245-4
10. P. Jungklass und M. Berekovic Speicherkonzepte für echtzeitfähige Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2020*.

2020. ISBN: 978-3-8343-2415-3

11. P. Jungklass und M. Berekovic Static Allocation of Basic Blocks Based on Runtime and Memory Requirements in Embedded Real-Time Systems with Hierarchical Memory Layout. In: *Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*. Hrsg. von M. Bertogna und F. Terraneo. Bd. 87. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, 3:1–3:14. ISBN: 978-3-95977-178-8. DOI: 10.4230/OASIcs.NG-RES.2021.3. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13479
12. C. Pott, P. Jungklass, D. J. Csejka, T. Eisenbarth und M. Siebert Firmware Security Module. In: *Journal of Hardware and Systems Security*, Apr. 2021. ISSN: 2509-3436. DOI: 10.1007/s41635-021-00114-4. URL: https://doi.org/10.1007/s41635-021-00114-4
13. F. Grieger, K. Koiro, P. Jungklass, A. von Daake und N. Falke Modular Power Electronics Platform for Evaluation and Rapid Prototyping using a NPC Converter. In: *PCIM Europe*. VDE VERLAG GMBH, Mai 2021, S. 1630–1637. ISBN: 978-3-8007-5515-8
14. S. Körür, P. Jungklass und M. Berekovic Echtzeitfähige Ethernet-Kommunikation in automobilen Multicore-Systemen mit hierarchischem Speicherlayout. In: *Echtzeit 2021*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 83–92. ISBN: 978-3-658-37751-9. DOI: https://doi.org/10.1007/978-3-658-37751-9
15. C. Böttcher, P. Jungklass und M. Berekovic Hardware-Beschleuniger für automobile Multicore-Mikrocontroller mit einer harten Echtzeitanforderung. In: *Echtzeit 2021*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 63–72. ISBN: 978-3-658-37751-9. DOI: https://doi.org/10.1007/978-3-658-37751-9
16. P. Jungklass, F. Grieger, C. Elvers und M. Berekovic Cache-Konzepte für echtzeitfähige Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2021*. 2021. ISBN: 978-3-8343-6291-9
17. C. Elvers und P. Jungklass Agile Software-Entwicklung im automobilen Umfeld. In: *Tagungsband Embedded Software Engineering Kongress 2021*. 2021. ISBN: 978-3-8343-6291-9
18. P. Jungklass, F. Grieger und M. Berekovic Predictive Preload at Fixed Preemption Points for Microcontrollers with Hard Real-Time Requirements. In: *Real-time and Autonomous Systems 2022*. Hrsg. von H. Unger und M. Schaible. Cham: Springer Nature Switzerland, 2023, S. 43–51. ISBN: 978-3-031-32700-1
19. P. Jungklass und R. Barg Firmware Security Module - Sicherheit über die gesamte Lebensdauer? In: *Tagungsband Embedded Software Engineering Kongress 2023*. 2023. ISBN: 978-3-8343-6314-5
20. P. Jungklass, C.-P. Stoeber-Schmidt, R. Barg, H. Hansen und M. Siebert Post-Quantum Cryptography on Embedded ECUs. In: *2024 JSAE Annual Congress (Spring)*. JSAE, 2024, S. 1–6
21. P. Jungklass, M. Manthe und D. J. Csejka Post-Quantum-Kryptographie auf eingebetteten Steuergeräten. In: *Tagungsband Embedded Software Engineering*

Kongress 2024. 2024. ISBN: 978-3-8343-6328-2
22. F. Sinell, R. Tschirley und P. Jungklass Evaluation of Post-Quantum Cryptography Signature Scheme on an Automotive Multicore Microcontroller for Real-Time Application. In: *Advances in Information and Communication*. Springer Nature Switzerland, 2025, S. 454–466. ISBN: 978-3-031-85363-0
23. P. Jungklass, C.-P. Stoeber-Schmidt, M. Siebert, J. Rummel und T. Nigoro Firmware Security Module. In: *2025 JSAE Annual Congress (Spring)*. JSAE, 2025
24. C. Elvers, P. Jungklass, C.-P. Stöber-Schmidt, J. Rummel und T. Nigoro Agile Software Development in the Automotive Environment - Scrum and Automotive SPICE, Contradiction in Terms? In: *2025 JSAE Annual Congress (Spring)*. JSAE, 2025
25. C.-P. Stöber-Schmidt, T. Nigoro, M. Siebert, P. Jungklass und J. Rummel Penetration Testing of Automotive Systems - Efficient Security Analysis of Vehicular E/E Systems. In: *2025 JSAE Annual Congress (Spring)*. JSAE, 2025

Fachvorträge

1. P. Jungklass und M. Berekovic Influence of Memory Management on Performance and Real Time Capability of Embedded Multicore Microcontrollers. In: *Embedded Multi-Core Conference*. 2018
2. C.-P. Stöber-Schmidt, P. Jungklass und M. Siebert Post-Quantum Cryptography on Embedded ECUs. In: *International VDI Conference - Cyber Security for Vehicles*. Juni 2024
3. P. Jungklass Post-Quantum Cryptography for Embedded Systems. In: *3rd Charter of Trust Meetup Braunschweig*. Aug. 2024
4. P. Jungklass Firmware Security Module. In: *4th Charter of Trust Meetup Braunschweig*. Nov. 2024

Patente

1. P. Jungklass Verfahren zur Zwischenkernkommunikation in einem Mehrkernprozessor. Pat. 10 2018 123 563.1. 2020
2. P. Jungklass Verfahren und Vorrichtung zur statischen Speicherverwaltungsoptimierung bei integrierten Mehrkernprozessoren. Pat. 10 2019 128 206.3. 2022
3. P. Jungklass Verfahren zur Herstellung der Cachekohärenz in Mehrkernprozessoren. Pat. 10 2019 118 757.5. 2023

Lehrtätigkeit

1. Lehrbeauftragter des Landes Mecklenburg-Vorpommern, Digitale Schaltungen / Hardware-Grundlagen I, Wintersemester 2022/2023, Hochschule Stralsund

2. Lehrbeauftragter des Landes Mecklenburg-Vorpommern, Mikroprozessortechnik / Hardware-Grundlagen II, Sommersemester 2023, Hochschule Stralsund
3. Lehrbeauftragter des Landes Mecklenburg-Vorpommern, Digitale Schaltungen / Hardware-Grundlagen I, Wintersemester 2023/2024, Hochschule Stralsund
4. Lehrbeauftragter des Landes Mecklenburg-Vorpommern, Mikroprozessortechnik / Hardware-Grundlagen II, Sommersemester 2024, Hochschule Stralsund
5. Lehrbeauftragter des Landes Mecklenburg-Vorpommern, Digitale Schaltungen / Hardware-Grundlagen I, Wintersemester 2024/2025, Hochschule Stralsund
6. Lehrbeauftragter des Landes Mecklenburg-Vorpommern, Mikroprozessortechnik / Hardware-Grundlagen II, Sommersemester 2025, Hochschule Stralsund

Betreute studentische Arbeiten

1. S. Körür *Auswahl und Implementierung einer Embedded-Systems-Kommunikationsschnittstelle für einen Mikroprozessor vom Typ Infineon AURIX TC299.* Ostfalia - Hochschule für angewandte Wissenschaften. Bachelor-Thesis. 2018
2. T. Rettemeyer *Untersuchung von Speicherkorrekturverfahren im Bezug auf Performance, Speicherbedarf und Fehlerkorrekturrate im AURIX TriCore 1G.* Ostfalia - Hochschule für angewandte Wissenschaften. Projektarbeit. 2020
3. S. Körür *Entwicklung einer echtzeitfähigen Ethernet-Kommunikation für eine embedded Steuergeräteplattform in der Prototypentwicklung.* Ostfalia - Hochschule für angewandte Wissenschaften. Master-Thesis. 2021
4. T. Rettemeyer *Entwicklung eines universellen Sicherheitskonzepts für eine embedded Steuergeräteplattform für die Prototypenentwicklung.* Ostfalia - Hochschule für angewandte Wissenschaften. Master-Thesis. 2021
5. M. Rogalski *Entwicklung eines Demonstrators zum Vergleich repräsentativer automotive Lastszenarien für embedded Mikrocontroller.* Ostfalia - Hochschule für angewandte Wissenschaften. Bachelor-Thesis. Apr. 2021
6. L. Poeppel *Embedded Programmierung eines Infineon AURIX (Tricore): Implementierung und Laufzeitanalyse von RSA und SipHash-2-4.* Hochschule für Technik und Wirtschaft Berlin. Bachelor-Thesis. 2022

Mitarbeit an Förderprojekten

1. NetProsum2030 - Kompakte modulare Wandler und optimierte Systemlösungen zur Energieflusssteuerung für netzdienlichen Prosumer 2030 mit HV-Fahrzeugbatterien. Förderkennzeichen: 0350021A. Projektlaufzeit: 2017-2021. Gefördert durch das Bundesministerium für Wirtschaft und Klimaschutz
2. StabLe - Stabilität von Verteilnetzen mit vorwiegend leistungselektronisch angekoppelten Speichern, Erzeugungseinheiten und Verbrauchern. Förderkennzeichen: 0324101D. Projektlaufzeit: 2017-2020. Gefördert durch das Bundesministerium für Wirtschaft und Klimaschutz

Literatur

[1] Jungklass, P. und Berekovic, M. Performance-Oriented Memory Management for Embedded Multicore Microcontrollers. In: *26th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*. 2018. ISBN: 978-3-902457-49-3.

[2] Jungklass, P. und Berekovic, M. Effects of concurrent access to embedded multicore microcontrollers with hard real-time demands. In: *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 2018, S. 1–9. ISBN: 978-1-5386-4155-2/18.

[3] Jungklass, P. und Berekovic, M. Intercore-Kommunikation für Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2018*. 2018. ISBN: 978-3-8343-3447-3.

[4] Jungklass, P. und Berekovic, M. Performance-orientiertes Speichermanagement bei embedded Multicore-Mikrocontrollern. In: *Tagungsband Embedded Software Engineering Kongress 2018*. 2018. ISBN: 978-3-8343-3447-3.

[5] Jungklass, P. und Berekovic, M. Cache-Kohärenz für embedded Multicore-Mikrocontroller mit harter Echtzeitanforderung. In: *Echtzeit 2019*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 129–138. ISBN: 978-3-658-27808-3.

[6] Jungklass, P. und Berekovic, M. MemOpt: Automated Memory Distribution for Multicore Microcontrollers with Hard Real-Time Requirements. In: *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. Okt. 2019, S. 1–7. ISBN: 978-1-7281-2769-9/19. DOI: 10.1109/NORCHIP.2019.8906914.

[7] Jungklass, P., Schmidt, C., Jungklass, T. und Berekovic, M. Scheduling-Konzepte für echtzeitfähige embedded Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2019*. 2019. ISBN: 978-3-8343-3463-3.

[8] Jungklass, P., Schmidt, C. und Berekovic, M. Redundanzkonzepte für embedded Multicore Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2019*. 2019. ISBN: 978-3-8343-3463-3.

[9] Jungklass, P. und Berekovic, M. Speicherkonzepte für echtzeitfähige Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2020*. 2020. ISBN: 978-3-8343-2415-3.

[10] Jungklass, P. und Berekovic, M. Static Allocation of Basic Blocks Based on Runtime and Memory Requirements in Embedded Real-Time Systems with Hierarchical Memory Layout. In: *Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*. Hrsg. von M. Bertogna und

F. Terraneo. Bd. 87. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, 3:1–3:14. ISBN: 978-3-95977-178-8. DOI: 10.4230/OASIcs.NG-RES.2021.3. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13479.

[11] Körür, S., Jungklass, P. und Berekovic, M. Echtzeitfähige Ethernet-Kommunikation in automobilen Multicore-Systemen mit hierarchischem Speicherlayout. In: *Echtzeit 2021*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 83–92. ISBN: 978-3-658-37751-9. DOI: https://doi.org/10.1007/978-3-658-37751-9.

[12] Böttcher, C., Jungklass, P. und Berekovic, M. Hardware-Beschleuniger für automobile Multicore-Mikrocontroller mit einer harten Echtzeitanforderung. In: *Echtzeit 2021*. Hrsg. von H. Unger. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, S. 63–72. ISBN: 978-3-658-37751-9. DOI: https://doi.org/10.1007/978-3-658-37751-9.

[13] Jungklass, P., Grieger, F., Elvers, C. und Berekovic, M. Cache-Konzepte für echtzeitfähige Multicore-Mikrocontroller. In: *Tagungsband Embedded Software Engineering Kongress 2021*. 2021. ISBN: 978-3-8343-6291-9.

[14] Jungklass, P., Grieger, F. und Berekovic, M. Predictive Preload at Fixed Preemption Points for Microcontrollers with Hard Real-Time Requirements. In: *Real-time and Autonomous Systems 2022*. Hrsg. von H. Unger und M. Schaible. Cham: Springer Nature Switzerland, 2023, S. 43–51. ISBN: 978-3-031-32700-1.

[15] Koulamas, C. und Lazarescu, M. T. Real-Time Embedded Systems: Present and Future. In: *Electronics* 7(9), 2018. ISSN: 2079-9292. DOI: 10.3390/electronics7090205. URL: https://www.mdpi.com/2079-9292/7/9/205.

[16] Wong, M. The future of Heterogeneous and Parallel Programming for Self-Driving Cars. In: *Embedded Multi-Core Conference*. Juni 2017.

[17] Jamal, R. Disruptive Technologies: A Shock Wave for Existing Ecosystems? In: *Embedded Multi-Core Conference*. National Instruments. Juni 2018.

[18] Ries, J., Reinhold, C., Herr, H. und Engel, B. Modular Control System for Testing of Electrical Power Systems and Energy Management Systems in elenia-energy-labs. In: *International ETG-Congress 2019; ETG Symposium*. 2019, S. 1–6. ISBN: 978-3-8007-4954-6.

[19] Grieger, F., Koiro, K., Jungklass, P., Daake, A. von und Falke, N. Modular Power Electronics Platform for Evaluation and Rapid Prototyping using a NPC Converter. In: *PCIM Europe*. VDE VERLAG GMBH, Mai 2021, S. 1630–1637. ISBN: 978-3-8007-5515-8.

[20] Edwards, S. A. und Lee, E. A. The Case for the Precision Timed (PRET) Machine. In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. New York, NY, USA: Association for Computing Machinery, 2007, S. 264–265. ISBN: 9781595936271. DOI: 10.1145/1278480.1278545. URL: https://doi.org/10.1145/1278480.1278545.

[21] Lee, E. A. *Cyber Physical Systems: Design Challenges*. Techn. Ber. UCB/EECS-2008-8. EECS Department, University of California, Berkeley,

Jan. 2008. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html.

[22] Villaescusa, D. G., Rivas, M. A. und Harbour, M. G. M2OS-Mc: An RTOS for Many-Core Processors. In: *Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*. Hrsg. von M. Bertogna und F. Terraneo. Bd. 87. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, 5:1–5:13. ISBN: 978-3-95977-178-8. DOI: 10.4230/OASIcs.NG-RES.2021.5. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13481.

[23] Hattendorf, A., Raabe, A. und Knoll, A. Shared memory protection for spatial separation in multicore architectures. In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 2012, S. 299–302.

[24] Wolf, W. The future of multiprocessor systems-on-chips. In: *Proceedings. 41st Design Automation Conference, 2004.* 2004, S. 681–685. DOI: 10.1145/996566.996753.

[25] Becker, J. und Bapp, F. K. The ARAMiS Project Initiative - Multicore Systems in Safety- and Mixed-Critical Applications. In: *Embedded Multi-Core Conference*. Karlsruhe Institute of Technology. Juni 2018.

[26] Brunie, N. Manycore Processor for Next-generation Intelligent Vehicles. In: *Embedded Multi-Core Conference*. Kalray SA. Juni 2018.

[27] Mitra, T., Teich, J. und Thiele, L. Time-Critical Systems Design: A Survey. In: *IEEE Design Test* 35(2):8–26, 2018. DOI: 10.1109/MDAT.2018.2794204.

[28] Kramer, S., Ziegenbein, D. und Hamann, A. Real World Automotive Benchmarks For Free. In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2015.

[29] Paolieri, M., Quiñones, E., Cazorla, F. J., Bernat, G. und Valero, M. Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. New York, NY, USA: ACM, 2009, S. 57–68. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555764. URL: http://doi.acm.org/10.1145/1555754.1555764.

[30] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M. und Sha, L. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In: *2012 24th Euromicro Conference on Real-Time Systems*. 2012, S. 299–308. DOI: 10.1109/ECRTS.2012.32.

[31] Bui, D., Lee, E., Liu, I., Patel, H. und Reineke, J. Temporal Isolation on Multiprocessing Architectures. In: *Proceedings of the 48th Design Automation Conference*. DAC '11. New York, NY, USA: Association for Computing Machinery, 2011, S. 274–279. ISBN: 9781450306362. DOI: 10.1145/2024724.2024787. URL: https://doi.org/10.1145/2024724.2024787.

[32] Schneider, R., Jürgens, D. und Kohn, A. Software Parallelization in Automotive Multi-Core Systems. In: *SAE Technical Paper*. Bd. 2015. SAE International, Apr. 2015. DOI: 10.4271/2015-01-0189.

[33] Or-Bach, Z. A 1,000x improvement in computer systems by bridging the processor-memory gap. In: *2017 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2017, S. 1–4. DOI: 10.1109/S3S.2017.8309202.

[34] Wehmeyer, L. und Marwedel, P. Influence of memory hierarchies on predictability for time constrained embedded software. In: *Design, Automation and Test in Europe*. 2005, 600–605 Vol. 1. DOI: 10.1109/DATE.2005.183.

[35] Puaut, I. und Pais, C. Scratchpad memories vs locked caches in hard real-time systems: a qualitative and quantitative comparison. In: *Institut de Recherche en Informatique et Systèmes Aléatoires - Publication Interne No 1818*, Jan. 2006. ISSN: 1166-8687.

[36] Taylor, J. Performance meets safety and security - Future automotive SoC trends. In: *Embedded Multi-Core Conference*. 2018.

[37] Grave, R. Insights in an automotive central computing cluster. In: *Embedded Multi-Core Conference*. Juni 2017.

[38] ISO *ISO26262 - Road vehicles – Functional safety*. ISO 26262. International Organization for Standardization. ISO, Geneva, Switzerland, Dez. 2018.

[39] Wozniak, E., Di Natale, M., Zeng, H., Mraidha, C., Tucci-Piergiovanni, S. und Gerard, S. Assigning Time Budgets to Component Functions in the Design of Time-Critical Automotive Systems. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, S. 235–246. ISBN: 9781450330138. DOI: 10.1145/2642937.2643015. URL: https://doi.org/10.1145/2642937.2643015.

[40] Mubeen, S., Nolte, T., Sjödin, M., Lundbäck, J. und Lundbäck, K.-L. Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints. In: *Software and Systems Modeling* 18(1):39–69, Feb. 2019. ISSN: 1619-1374. DOI: 10.1007/s10270-017-0579-8. URL: https://doi.org/10.1007/s10270-017-0579-8.

[41] Pott, C., Jungklass, P., Csejka, D. J., Eisenbarth, T. und Siebert, M. Firmware Security Module. In: *Journal of Hardware and Systems Security*, Apr. 2021. ISSN: 2509-3436. DOI: 10.1007/s41635-021-00114-4. URL: https://doi.org/10.1007/s41635-021-00114-4.

[42] Wiersma, N. und Pareja, R. *Safety != Security - A security assessment of the resilience against fault injection attacks in ASIL-D certified microcontrollers*. Techn. Ber. Riscure Security Lab, 2017. URL: https://www.riscure.com/uploads/2017/08/Riscure_Whitepaper_Safety_is_not_Security_Automotive.pdf.

[43] Gai, P. und Violante, M. Automotive embedded software architecture in the multi-core age. In: *2016 21th IEEE European Test Symposium (ETS)*. 2016, S. 1–8. DOI: 10.1109/ETS.2016.7519309.

[44] Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., Herkersdorf, A., Kapitza, R., Lohmann, D., Marwedel, P., Platzner, M., Rosenstiel, W., Schlichtmann, U., Spinczyk, O., Tahoori, M., Teich, J., When, N. und Wun-

derlich, H. J. Design and architectures for dependable embedded systems. In: *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS)*. Okt. 2011, S. 69–78. DOI: 10.1145/2039370.2039384.

[45] Silva, F. A. d., Bagbaba, A. C., Hamdioui, S. und Sauer, C. Efficient Methodology for ISO26262 Functional Safety Verification. In: *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2019, S. 255–256. DOI: 10.1109/IOLTS.2019.8854449.

[46] Gizopoulos, D., Psarakis, M., Adve, S. V., Ramachandran, P., Hari, S. K. S., Sorin, D., Meixner, A., Biswas, A. und Vera, X. Architectures for online error detection and recovery in multicore processors. In: *2011 Design, Automation Test in Europe*. März 2011, S. 1–6. DOI: 10.1109/DATE.2011.5763096.

[47] Seifert, G., Hiergeist, S. und Schwierz, A. Entwicklungsvorschläge für ISO 26262 konforme MCUs in sicherheitskritischer Avionik. In: *Echtzeit und Sicherheit*. Hrsg. von H. Unger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, S. 29–38. ISBN: 978-3-662-58096-7.

[48] Saidi, S., Ernst, R., Uhrig, S., Theiling, H. und Dinechin, B. D. de The shift to multicores in real-time and safety-critical systems. In: *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press. 2015, S. 220–229.

[49] Kluge, F., Mische, J., Uhrig, S., Ungerer, T. und Zalman, R. Use of helper threads for OS support in the multithreaded embedded TriCore 2 processor. In: *Proceedings: Work-In-Progress-Session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, 3-6 April, 2007, Bellevue, USA*. Hrsg. von C. Lu. 2007, S. 25 –27. DOI: 10.7936/K7G15Z63. URL: https://openscholarship.wustl.edu/cse_research/918/.

[50] AG, I. T. *AURIX TC29x B-Step User's Manual V1.3*. Infineon Technologies AG. 81726 Munich, Germany, Dez. 2014.

[51] AG, I. T. *AURIX TC3xx User's Manual*. 2021-02. Infineon Technologies AG. 81726 Munich, Germany, Feb. 2021.

[52] N.V., N. S. *MPC5777M Reference Manual Rev. 4.2*. NXP Semiconductors N.V. Eindhoven, Netherlands, Aug. 2016.

[53] N.V., N. S. *S32Z2 - Safe and Secure High-Performance Real-Time Processors - Fact Sheet*. Rev. 0. NXP Semiconductors N.V. Eindhoven, Netherlands, 2022.

[54] STMicroelectronics *SPC58xEx/SPC58xGx 32-bit Power Architecture microcontroller for automotive ASILD applications - Reference Manual*. STMicroelectronics. Schiphol, Amsterdam, Niederlande, Aug. 2018.

[55] STMicroelectronics *RM0500 Reference Manual - Stellar SR6 G7 line*. Rev 1. STMicroelectronics. Schiphol, Amsterdam, Niederlande, Juni 2022.

[56] Corporation, R. E. *RH850/F1KH, RH850/F1KM ICUMD User's Manual: Hardware*. Rev. 1.20. Renesas Electronics Corporation. 3-2-24 Toyosu, Tokyo 135-0061, Japan, Okt. 2020.

[57] STMicroelectronics *RM0482 Reference Manual - SR6P7G7C7*. Rev 2. STMicroelectronics. Schiphol, Amsterdam, Niederlande, Okt. 2021.

[58] Feldmann, C. AURIX Multicore - Continuity across generations. In: *Embedded Multi-Core Conference*. Juni 2017.

[59] AG, I. T. *AURIX 32-bit Microcontroller family - Performance meets Safety*. Infineon Technologies AG. 81726 Munich, Germany, Juni 2018.

[60] Volland, D. *Spoilt for Choice: What is the Right ARM Architecture?* Techn. Ber. Microconsult GmbH, 2014.

[61] Limited, A. *Arm Cortex-R Processor Comparison Table*. Version 2021. Arm Limited. 110 Fulbourn Road, Cambridge, 2021.

[62] Gößler, M. *Multicore im Mikrocontroller - Von der Architektur bis zum Test*. Techn. Ber. Microconsult GmbH, 2015.

[63] Gößler, M. *AURIX TC3xx Workshop: 32-Bit Multicore-Mikrocontroller-Familie (2G - Zweite Generation)*. Techn. Ber. Microconsult GmbH, 2017.

[64] Incorporated, T. I. *TMS320F2838x Microcontrollers Technical Reference Manual*. Texas Instruments Incorporated. Post Office Box 655303, Dallas, Texas 75265, Mai 2019.

[65] AG, I. T. *TC1798 User's Manual V1.2*. Infineon Technologies AG. 81726 Munich, Germany, Mai 2012.

[66] Platzner, M. On-the-fly Computing: Self-aware Heterogeneous Multi-cores. In: *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*:1–2, 2016.

[67] AG, I. T. *2nd Generation AURIX TC3xx Hardware Security Module Target Specification*. 2.0. Infineon Technologies AG. 81726 Munich, Germany, März 2016.

[68] Gößler, M. *Generic Timer Module - GTM V1*. Techn. Ber. Microconsult GmbH, 2015.

[69] AG, I. T. *AURIX TC49x User's Manual*. V0.73. Infineon Technologies AG. 81726 Munich, Germany, Dez. 2021.

[70] GmbH, V. I. Introducing Hardware Security Modules to Embedded Systems. In: *Vector E-Mobility Engineering Day*. März 2017.

[71] Sommerlad, P. Patterns for Parallel Programming. In: *Embedded Multi-Core Conference*. Juni 2017.

[72] AG, I. T. *AURIX TC27x C-Step User's Manual V2.2*. Infineon Technologies AG. 81726 Munich, Germany, Dez. 2014.

[73] AG, I. T. *TriCore TC1.6P & TC1.6E Core Architecture*. Infineon Technologies AG. 81726 Munich, Germany, Feb. 2012.

[74] AG, I. T. *TriCore TC1.8 Core Architecture Manual*. V0.9. Infineon Technologies AG. 81726 Munich, Germany, Nov. 2022.

[75] Bock, T., Kleinwechter, H., Sasse, R., Münzenberger, R., Rehkop, P. und Schmidt, O. Einsatz von Virtualisierung für sichere Softwarearchitekturen - Zentrale Steuergeräte, Mixed Criticality, Testen, Multicore. In: *Tagungsband Embedded Software Engineering Kongress 2017*. 2017. ISBN: 978-3-8343-3426-8.

[76] Reif, S. Fault Tolerance in Multi-Core Systems. In: *Masterseminar Fehlertolerante Systeme (WS 2014/15)*. 2015.

[77] Bengel, G., Baun, C., Kunze, M. und Stucky, K.-U. *Masterkurs Parallele und Verteilte Systeme*. Springer Fachmedien Wiesbaden, 2015. ISBN: 978-3-8348-1671-9. DOI: https://doi.org/10.1007/978-3-8348-2151-5.

[78] Porrmann, M. Neuromorphic Computing - Disruptive Technology for Automotive Applications? In: *Embedded Multi-Core Conference*. Juni 2017.

[79] Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A. und Uhsadel, L. A Survey of Lightweight-Cryptography Implementations. In: *IEEE Design Test of Computers* 24(6):522–533, 2007. DOI: 10.1109/MDT.2007.178.

[80] Killmann, W. und Schindler, W. *A proposal for: Functionality classes for random number generators*. Bundesamt für Sicherheit in der Informationstechnik. Sep. 2011.

[81] Fischer, V. A Closer Look at Security in Random Number Generators Design. In: *Constructive Side-Channel Analysis and Secure Design*. Hrsg. von W. Schindler und S. A. Huss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 167–182. ISBN: 978-3-642-29912-4.

[82] Conti, M., Caldari, M., Vece, G. B., Orcioni, S. und Turchetti, C. Performance analysis of different arbitration algorithms of the AMBA AHB bus. In: *Proceedings of the 41st annual Design Automation Conference*. 2004, S. 618–621.

[83] Shrivastav, A., Tomar, G. und Singh, A. K. Performance Comparison of AMBA Bus-Based System-On-Chip Communication Protocol. In: *2011 International Conference on Communication Systems and Network Technologies*. 2011, S. 449–454. DOI: 10.1109/CSNT.2011.98.

[84] Jungklass, P. und Berekovic, M. Influence of Memory Management on Performance and Real Time Capability of Embedded Multicore Microcontrollers. In: *Embedded Multi-Core Conference*. 2018.

[85] AG, I. T. *AURIX TC3xx Target Specification V2.5.1*. Infineon Technologies AG. 81726 Munich, Germany, Apr. 2018.

[86] Ashok, A. und Harnisch, J. AURIX - Programming close to hardware for best performance. In: *Embedded Multi-Core Conference*. Infineon Technologies AG. 81726 Munich, Germany, Juni 2017.

[87] Kuntz, S. The importance of hardware models in developing applications for MSoCs. In: *Embedded Multi-Core Conference*. Juni 2017.

[88] Avissar, O., Barua, R. und Stewart, D. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. In: *ACM Transactions on Embedded Computing Systems* 1(1):6–26, Nov. 2002.

[89] Chen, Z.-H. und Su, A. A Hardware/Software Framework for Instruction and Data Scratchpad Memory Allocation. In: *ACM Transactions on Architecture and Code Optimization* 7, Apr. 2010. DOI: 10.1145/1736065.1736067.

[90] Gößler, M. *ISO/SAE DIS 21434:2020(E) - Road vehicles Cybersecurity engineering*. Techn. Ber. Microconsult GmbH, 2020.

[91] Barakat, A. und Iseler, R. Automatic Verification of Safety Criterias of ASIL D Basic Software. In: *Embedded Multi-Core Conference*. Altium Europe GmbH. Juni 2018.

[92] Hohmuth, M. L4Re Microhypervisor - Towards a dynamic service architecture for automotive applications. In: *Embedded Multi-Core Conference*. Juni 2017.

[93] Lampka, K. Mastering security and resource sharing in the future HPC platforms. In: *Embedded Multi-Core Conference*. Elektrobit. Juni 2018.

[94] GmbH, E. *ETAS RTA Lightweight Hypervisor - User Manual (SPC58ECxxGHS)*. ETAS GmbH. 70469 Stuttgart, Germany, Mai 2017.

[95] Fox, J. Large-Scale Distributed Integrationin Multi-Core Software Development. In: *Embedded Multi-Core Conference*. Juni 2017.

[96] Härdtlein, J. SW Distribution based on context-aware Interoperability. In: *Embedded Multi-Core Conference*. Juni 2017.

[97] Ernst, H., Schmidt, J. und Beneken, G. *Grundkurs Informatik*. Springer Fachmedien Wiesbaden, 2016. ISBN: 978-3-658-14633-7. DOI: https://doi.org/10.1007/978-3-658-14634-4.

[98] Nikolov, V. Scheduling Chains of Activities on Linux-based Multi-core Systems. In: *Embedded Multi-Core Conference*. 2018.

[99] Schwager, C. Central Vehicle Computer: Achieving freedom from interference by temporal software separation. In: *Embedded Multi-Core Conference*. ITK Engineering GmbH. Juni 2018.

[100] Asmus, R. AUTOSAR Adaptive Platform and Classic Platform multi-core improvements. In: *Embedded Multi-Core Conference*. Juni 2017.

[101] Klinger, A. *Embedded-Echtzeit-Linux - Vom Bootloader bis zum Realtime-System*. Techn. Ber. Microconsult GmbH, 2018.

[102] Wirrer, G. und Kumar, V. AUTOSAR MCAL Support for Multi-Core - A pragmatic and performant solution. In: *Embedded Multi-Core Conference*. Infineon AG. Juni 2018.

[103] AG, I. T. *1EDI2002AS EiceDRIVER SIL Datasheet*. Infineon Technologies AG. 81726 Munich, Germany, Juli 2015.

[104] AG, I. T. *1EDI2010AS EiceDRIVER SENSE Datasheet*. Infineon Technologies AG. 81726 Munich, Germany, Juni 2017.

[105] AG, I. T. *TLE987xQX User's Manual V1.3*. Infineon Technologies AG. 81726 Munich, Germany, Juni 2017.

[106] AG, I. T. *TLF35584 Multi Voltage Safety Micro Processor Supply Datasheet*. Infineon Technologies AG. 81726 Munich, Germany, März 2017.

[107] AUTOSAR *AUTOSAR Classic Platform - Layered Software Architecture*. R22-11. AUTOSAR. Bremerstraße 11, 80807 Munich, Germany, Nov. 2022.

[108] Stumpf, F., Pohl, C., Hoettges, D. und Klein, T. Introducing HSM-based secure on-board communication in vehicles - Challenges and Lessons Learned. In: *17th ESCAR Europe*, 2019.

[109] Kim, D., Shin, E., Park, J. S., LEE, K., Gui, K. C. und Scheibert, K. Secure Boot Implementation for Hard Real-Time Powertrain System. In: *SAE Technical Paper*. SAE International, März 2017. DOI: 10.4271/2017-01-1656. URL: https://doi.org/10.4271/2017-01-1656.

[110] Wolf, M. und Gendrullis, T. Design, Implementation, and Evaluation of a Vehicular Hardware Security Module. In: *Information Security and Crypto-*

logy - ICISC 2011. Hrsg. von H. Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 302–318. ISBN: 978-3-642-31912-9.

[111] Bello, L. L., Mariani, R., Mubeen, S. und Saponara, S. Recent Advances and Trends in On-Board Embedded and Networked Automotive Systems. In: *IEEE Transactions on Industrial Informatics* 15(2):1038–1051, 2019. DOI: 10.1109/TII.2018.2879544.

[112] Koppe, U. *Einführung in CANopen*. Techn. Ber. MicroControl GmbH & Co. KG, 2014.

[113] Stingl, A. ARTI (AUTOSAR Run-Time Interface) – the future of AUTOSAR OS-aware debugging and tracing. In: *Embedded Multi-Core Conference*. Juni 2017.

[114] Grave, R. Multi-Core in an AUTOSAR environment. In: *Embedded Multi-Core Conference*. Juni 2017.

[115] Hoxha, B., Abbas, H. und Fainekos, G. Benchmarks for Temporal Logic Requirements for Automotive Systems. In: *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*. Hrsg. von G. Frehse und M. Althoff. Bd. 34. EPiC Series in Computing. EasyChair, 2015, S. 25–30. DOI: 10.29007/xwrs. URL: https://easychair.org/publications/00_Paper/4bfq.

[116] Kehr, S., Böddeker, B. und Langen, D. Energy-aware Parallelization of AUTOSAR Legacy Applications. In: *Embedded Multi-Core Conference*. Juni 2017.

[117] Yip, E. Pulling the trigger - Practical considerations when applying the timetriggered approach to embedded multi core systems. In: *Embedded Multi-Core Conference*. Juni 2017.

[118] Martin, F. Multi-core software design and verification. In: *Embedded Multi-Core Conference*. Juni 2017.

[119] Barth, T. und Fromm, P. Warp 3 zwischen allen Kernen - Entwicklung einer schnellen und sicheren Multicore-RTE. In: *Tagungsband Embedded Software Engineering Kongress 2016*. 2016.

[120] Barth, T. und Fromm, P. Functional Safety on Multicore Microcontrollers for Industrial Applications. In: *Embedded World 2016 Exhibition & Conference*. 2016.

[121] Lugo, T., Lozano, S., Fernández, J. und Carretero, J. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. In: *IEEE Access* 10:21853–21882, Feb. 2022. DOI: 10.1109/ACCESS.2022.3151891.

[122] Software, G. H. *MULTI: Building Applications for Embedded V850 and RH850*. Build V800-544152. Green Hills Software. 30 West Sola Street in Santa Barbara, California 93101, USA, Okt. 2015.

[123] Suhendra, V., Mitra, T., Roychoudhury, A. und Ting Chen WCET centric data allocation to scratchpad memory. In: *26th IEEE International Real-Time Systems Symposium*. Dez. 2005, 10 pp.–232. DOI: 10.1109/RTSS.2005.45.

[124] Reder, S. und Becker, J. Interference-Aware Memory Allocation for Real-Time Multi-Core Systems. In: *2020 IEEE Real-Time and Embedded Tech-*

nology and Applications Symposium (RTAS). Apr. 2020, S. 148–159. DOI: 10.1109/RTAS48715.2020.00-10.

[125] Rouxel, B., Skalistis, S., Derrien, S. und Puaut, I. Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Hrsg. von S. Quinton. Bd. 133. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 25:1–25:24. ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.25. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10762.

[126] Luppold, A. und Falk, H. Schedulability-Aware SPM Allocation for Preemptive Hard Real-Time Systems with Arbitrary Activation Patterns. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2017, S. 1074–1079. DOI: 10.23919/DATE.2017.7927149.

[127] Falk, H. und Kleinsorge, J. C. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In: *2009 46th ACM/IEEE Design Automation Conference*. 2009, S. 732–737. DOI: 10.1145/1629911.1630101.

[128] Peng, B., Fisher, N. und Bertogna, M. Explicit Preemption Placement for Real-Time Conditional Code. In: *2014 26th Euromicro Conference on Real-Time Systems*. 2014, S. 177–188. DOI: 10.1109/ECRTS.2014.25.

[129] Limited, M. C. *MISRA C:2012 - Guidelines for the use of the C language in critical systems*. MISRA Consortium Limited. 1 St James Court, Norfolk, England, März 2013. ISBN: 978-1-906400-11-8.

[130] Verma, M., Wehmeyer, L. und Marwedel, P. Dynamic overlay of scratchpad memory for energy minimization. In: *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2004, S. 104–109.

[131] Udayakumaran, S., Dominguez, A. und Barua, R. Dynamic allocation for scratch-pad memory using compile-time decisions. In: *ACM Transactions on Embedded Computing Systems (TECS)* 5(2):472–511, 2006.

[132] Yao, G., Pellizzoni, R., Bak, S., Betti, E. und Caccamo, M. Memory-centric Scheduling for Multicore Hard Real-time Systems. In: *Real-Time Syst.* 48(6):681–715, Nov. 2012. ISSN: 0922-6443. DOI: 10.1007/s11241-012-9158-9. URL: http://dx.doi.org/10.1007/s11241-012-9158-9.

[133] Kim, Y., Broman, D., Cai, J. und Shrivastava, A. WCET-aware dynamic code management on scratchpads for Software-Managed Multicores. In: *Real-Time Technology and Applications - Proceedings* 2014:179–188, Okt. 2014. DOI: 10.1109/RTAS.2014.6926001.

[134] Tabish, R., Mancuso, R., Wasly, S., Alhammad, A., Phatak, S. S., Pellizzoni, R. und Caccamo, M. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2016, S. 1–11. DOI: 10.1109/RTAS.2016.7461321.

[135] Bertogna, M., Xhani, O., Marinoni, M., Esposito, F. und Buttazzo, G. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In:

2011 23rd Euromicro Conference on Real-Time Systems. 2011, S. 217–227. DOI: 10.1109/ECRTS.2011.28.

[136] Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M. und Pellizzoni, R. Real-time Cache Management Framework for Multi-core Architectures. In: *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. RTAS '13. Washington, DC, USA: IEEE Computer Society, 2013, S. 45–54. ISBN: 978-1-4799-0186-9. DOI: 10.1109/RTAS.2013.6531078. URL: http://dx.doi.org/10.1109/RTAS.2013.6531078.

[137] Yao, G., Pellizzoni, R., Bak, S., Yun, H. und Caccamo, M. Global Real-Time Memory-Centric Scheduling for Multicore Systems. In: *IEEE Transactions on Computers* 65(9):2739–2751, Sep. 2016. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2500572.

[138] Gracioli, G. und Fröhlich, A. A. On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures. In: *ACM SIGOPS Operating Systems Review* 49(2):2–16, Jan. 2016. ISSN: 0163-5980. DOI: 10.1145/2883591.2883594. URL: https://doi.org/10.1145/2883591.2883594.

[139] Kim, Y., Papamichael, M., Mutlu, O. und Harchol-Balter, M. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In: *43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2010, S. 65–76. DOI: 10.1109/MICRO.2010.51.

[140] Green, M., Rodrigues-Lima, L., Zankl, A., Irazoqui, G., Heyszl, J. und Eisenbarth, T. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In: *26th USENIX Security Symposium*. Vancouver, BC: USENIX Association, 2017, S. 1075–1091. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/green.

[141] Metzlaff, S., Guliashvili, I., Uhrig, S. und Ungerer, T. A Dynamic Instruction Scratchpad Memory for Embedded Processors Managed by Hardware. In: *International Conference on Architecture of Computing Systems*. Springer. 2011, S. 122–134.

[142] Song, Y., Alavoine, O. und Lin, B. A Self-Aware Resource Management Framework for Heterogeneous Multicore SoCs with Diverse QoS Targets. In: *ACM Transactions on Architecture and Code Optimization* 16(2), Apr. 2019. ISSN: 1544-3566. DOI: 10.1145/3319804. URL: https://doi.org/10.1145/3319804.

[143] Fukuda, T. Multicore Design Studies with Powertrain/ADAS SW and Future Challenges of Highly Self-Driving Vehicles. In: *Embedded Multi-Core Conference*. HITACHI. Juni 2018.

[144] Akram, N., Zhang, Y., Ali, S. und Amjad, H. M. Efficient Task Allocation for Real-Time Partitioned Scheduling on Multi-Core Systems. In: *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. Jan. 2019, S. 492–499. DOI: 10.1109/IBCAST.2019.8667139.

[145] Meyer, J., Houben, I. und Münzenberger, R. Kommunikationsoverhead bei Multi-Core Systemen früh beherrschen. In: *Tagungsband Embedded Software Engineering Kongress 2014*. Dez. 2014.

[146] Kachris, C., Nikiforos, G., Papaefstathiou, V., Yang, X., Kavadias, S. und Katevenis, M. Low-latency explicit communication and synchronization in scalable multi-core clusters. In: *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*. Sep. 2010, S. 1–4. DOI: 10.1109/CLUSTERWKSP.2010.5613092.

[147] Gerdes, M., Kluge, F., Ungerer, T., Rochange, C. und Sainrat, P. Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In: *2012 Design, Automation and Test in Europe Conference*. 2012, S. 671–676. DOI: 10.1109/DATE.2012.6176555.

[148] Mader, R. LET Implementation in Classic AUTOSAR - How Logical Execution Time can act as an enable for deterministic parallel execution behavior. In: *Embedded Multi-Core Conference*. Continental. Juni 2018.

[149] Lalo, E. Challenges of migrating automotive event-triggered (ET) systems to LET paradigm. In: *Embedded Multi-Core Conference*. Vector Informatik Germany. Juni 2018.

[150] Xu, R., Friese, M. J., Hasseln, H. von und Nowotka, D. Data Access Time Estimation in Automotive LET Scheduling with Multi-core CPU. In: *15th Junior Researcher Workshop on Real-Time Computing 2022*. 2022, S. 11. URL: https://rtns2022.inria.fr/files/2022/06/proceedings_jrwrtc2022_final.pdf.

[151] Kirsch, C. M. und Sokolova, A. The Logical Execution Time Paradigm. In: *Advances in Real-Time Systems*. Springer, 2012, S. 103–120.

[152] Kluge, F., Schoeberl, M. und Ungerer, T. Support for the Logical Execution Time Model on a Time-predictable Multicore Processor. In: *SIGBED Rev.* 13(4):61–66, Nov. 2016. ISSN: 1551-3688. DOI: 10.1145/3015037.3015047. URL: http://doi.acm.org/10.1145/3015037.3015047.

[153] Kluge, F., Gerdes, M. und Ungerer, T. An Operating System for Safety-Critical Applications on Manycore Processors. In: *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*:238–245, 2014.

[154] Carle, T., Papagiannopoulou, D., Moreshet, T., Marongiu, A., Herlihy, M. und Bahar, R. I. Thrifty-malloc: A HW/SW Codesign for the Dynamic Management of Hardware Transactional Memory in Embedded Multicore Systems. In: *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. 2016, S. 1–10. DOI: 10.1145/2968455.2968513.

[155] Fetzer, C. und Felber, P. Transactional memory for dependable embedded systems. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2011, S. 223–227. DOI: 10.1109/DSNW.2011.5958817.

[156] Martin, M. M. K., Hill, M. D. und Sorin, D. J. Why On-Chip Cache Coherence is Here to Stay. In: *Communications of the ACM* 55(7):78–89, Ju-

li 2012. ISSN: 0001-0782. DOI: 10.1145/2209249.2209269. URL: https://doi.org/10.1145/2209249.2209269.

[157] Dietrich, C., Wägemann, P., Ulbrich, P. und Lohmann, D. SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems (Outstanding Paper). In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2017, S. 37–48. DOI: 10.1109/RTAS.2017.37.

[158] Dietrich, C. und Wägemann, P. SysWCET: Ende-zu-Ende-Antwortzeiten für OSEK-Systeme. In: *Tagungsband Embedded Software Engineering Kongress 2017*. Dez. 2017. ISBN: 978-3-8343-3426-8.

[159] Sheikh, S. Z. und Pasha, M. A. Energy-Efficient Multicore Scheduling for Hard Real-Time Systems: A Survey. In: *ACM Transactions on Embedded Computing Systems* 17(6):94:1–94:26, Dez. 2018. ISSN: 1539-9087. DOI: 10.1145/3291387. URL: http://doi.acm.org/10.1145/3291387.

[160] Krapf, R. *Elementare Grundlagen der Hochschulmathematik*. Hrsg. von R. Krapf. Springer Fachmedien Wiesbaden, 2020. ISBN: 978-3-658-29953-8.

[161] The Multicore Association, I. *Software-Hardware Interface for Multi-Many-Core (SHIM) Specification*. V2.00. The Multicore Association, Inc. The Multicore Association, Inc. PO Box 4854 El Dorado Hills, CA 95762, Jan. 2019.

[162] BV, T. *TASKING VX-toolset for TriCore User Guide*. MA160-800 (v6.2). TASKING BV. Spoetnik 50, 3824 MG Amersfoort, Netherlands, Dez. 2016.

[163] Zamorano, J. und Puente, J. A. de la Memory Isolation in Many-Core Embedded Systems. In: *Highperformance and Real-time Embedded System (HiRES)*, 2014.

[164] Majéric, F., Gonzalvo, B. und Bossuet, L. JTAG Fault Injection Attack. In: *IEEE Embedded Systems Letters* 10(3):65–68, Sep. 2018. ISSN: 1943-0671. DOI: 10.1109/LES.2017.2771206.

[165] GmbH, L. *TriCore Debugger and Trace - Release 02.2022*. Release 02.2022. Lauterbach GmbH. Altlaufstr. 40, 85635 Höhenkirchen-Siegertsbrunn, Germany, Feb. 2022.

[166] AG, I. T. *TriBoard Manual TC3X9 - User Manual*. V2.1. Infineon Technologies AG. 81726 Munich, Germany, Nov. 2017.

[167] Stingl, A. UsingTrace for Timing Measurementson a Multi-ECU Systemincluding OS, Runnables and Network. In: *Embedded Multi-Core Conference*. Juni 2017.

[168] Friese, M. J., Grünfelder, S., Paulitsch, M. und Weiss, A. *Tracing von eingebetteten Multicore-Systemen*. Techn. Ber. HANSER Automotive, 2020.

[169] Gliwa, P. A systematic approach for timing requirements. In: *Embedded Multi-Core Conference*. GLIWA GmbH embedded systems. Juni 2018.

[170] Wichelmann, J., Moghimi, A., Eisenbarth, T. und Sunar, B. MicroWalk: A Framework for Finding Side Channels in Binaries. In: *arXiv preprint arXiv:1808.05575*, 2018.

[171] AG, I. T. *TriCore TC1.6.2 Core Architecture Manual*. Infineon Technologies AG. 81726 Munich, Germany, Jan. 2020.

[172] AG, I. T. *iLLD - Infineon Low Level Driver 1.0.1.16.0 - Release Notes*. 1.0.1.16.0. Infineon Technologies AG. 81726 Munich, Germany, März 2023.

[173] Poovey, J. A., Conte, T. M., Levy, M. und Gal-On, S. A Benchmark Characterization of the EEMBC Benchmark Suite. In: *IEEE Micro* 29(5):18–29, 2009. DOI: 10.1109/MM.2009.74.

[174] AG, I. T. *MC-ISAR-AS440-TC3xx-BASIC-2.0.0 - MCAL Documentation*. 2.0.0. Infineon Technologies AG. 81726 Munich, Germany, Apr. 2021.

[175] AG, I. T. *MC-ISAR-TC3xx-DemoApp - MCAL Documentation*. v18.0. Infineon Technologies AG. 81726 Munich, Germany, Apr. 2021.

[176] Buchmann, J., Kraemer, J., Alkadri, N. A., Bindel, N., Bansarkhari, R. E., Goepfert, F. und Wunderer, T. *Bewertung gitterbasierter kryptografischer Verfahren*. Techn. Ber. Bundesamt für Sicherheit in der Informationstechnik, 2018. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Gitterbasierte_Verfahren/Gitterbasierte_Verfahren.pdf.

[177] Informationstechnik, B. für Sicherheit in der *Entwicklungsstand Quantencomputer - Deutsche Zusammenfassung*. Techn. Ber. BSI-Projektnummer: 477 (Version 2.0). Bundesamt für Sicherheit in der Informationstechnik, 2023. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Quantentechnologien-und-Post-Quanten-Kryptografie/Entwicklungsstand-Quantencomputer/entwicklungsstand-quantencomputer_node.html.

[178] Kannwischer, M. J., Schwabe, P., Stebila, D. und Wiggers, T. Improving Software Quality in Cryptography Standardization Projects. In: *2022 IEEE European Symposium on Security and Privacy Workshops*. 2022, S. 19–30. DOI: 10.1109/EuroSPW55150.2022.00010.

[179] Kannwischer, M. J., Rijneveld, J., Schwabe, P. und Stoffelen, K. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. In: *Cryptology ePrint Archive, Paper 2019/844*, 2019. URL: https://eprint.iacr.org/2019/844.

[180] Informationstechnik, B. für Sicherheit in der *Kryptografie quantensicher gestalten - Grundlagen, Entwicklungen, Empfehlungen*. Techn. Ber. Bundesamt für Sicherheit in der Informationstechnik, 2021. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Broschueren/Kryptografie-quantensicher-gestalten.html.

[181] Richter, M., Bertram, M., Seidensticker, J. und Tschache, A. A Mathematical Perspective on Post-Quantum Cryptography. In: *Mathematics* 10(15), 2022. ISSN: 2227-7390. DOI: 10.3390/math10152579. URL: https://www.mdpi.com/2227-7390/10/15/2579.

[182] Murvay, P.-S., Matei, A., Solomon, C. und Groza, B. Development of an AUTOSAR Compliant Cryptographic Library on State-of-the-Art Automotive Grade Controllers. In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. 2016, S. 117–126. DOI: 10.1109/ARES.2016.60.

[183] Song, G., Jang, K., Eum, S., Sim, M. und Seo, H. NTT and Inverse NTT Quantum Circuits in CRYSTALS-Kyber for Post-Quantum Security Evaluation. In: *Applied Sciences* 13(18), 2023. ISSN: 2076-3417. DOI: 10.3390/app131810373. URL: https://www.mdpi.com/2076-3417/13/18/10373.

[184] Winkler, D., Sepulveda, D., Cupelli, M., Olexa, R. und Sepulveda, J. Quantum secure high performance automotive systems. In: *19th ESCAR Europe*. 2021. DOI: 10.13154/294-8352.

[185] Jungklass, P. und Barg, R. Firmware Security Module - Sicherheit über die gesamte Lebensdauer? In: *Tagungsband Embedded Software Engineering Kongress 2023*. 2023. ISBN: 978-3-8343-6314-5.

[186] Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sørensen, R. B., Wägemann, P. und Wegener, S. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Hrsg. von M. Schoeberl. Bd. 55. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 2:1–2:10. ISBN: 978-3-95977-025-5. DOI: 10.4230/OASIcs.WCET.2016.2. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6895.

[187] Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.-P. und De Michiel, M. PapaBench: a Free Real-Time Benchmark. In: *6th International Workshop on Worst-Case Execution Time Analysis*. Hrsg. von F. Mueller. Bd. 4. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2006, S. 1–6. ISBN: 978-3-939897-03-3. DOI: 10.4230/OASIcs.WCET.2006.678. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2006.678.

[188] Kluge, F., Rochange, C. und Ungerer, T. EMSBench: Benchmark and Testbed for Reactive Real-Time Systems. In: *Leibniz Transactions on Embedded Systems* 4(2):02–1–02:23, 2017. ISSN: 2199-2002. DOI: 10.4230/LITES-v004-i002-a002. URL: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v004-i002-a002.

[189] Sommer, L., Stock, F., Solis-Vasquez, L. und Koch, A. DAPHNE - An Automotive Benchmark Suite for Parallel Programming Models on Embedded Heterogeneous Platforms. In: *2019 International Conference on Embedded Software (EMSOFT)*. 2019, S. 1–2.

[190] Ikibas, U. Challenges for the runtime performance of artificial intelligence based systems in future automotive controller networks. In: *Embedded Multi-Core Conference*. Continental. Juni 2018.

[191] Jungklass, P., Manthe, M. und Csejka, D. J. Post-Quantum-Kryptographie auf eingebetteten Steuergeräten. In: *Tagungsband Embedded Software Engineering Kongress 2024*. 2024. ISBN: 978-3-8343-6328-2.

[192] AG, I. T. *BIFACES Base Projects for AURIX 2nd Generation Microcontrollers - Release Notes*. 1.0.1.16.0. Infineon Technologies AG. 81726 Munich, Germany, März 2023.

[193] Peleska, J., Honisch, A., Lapschies, F., Löding, H., Schmid, H., Smuda, P., Vorobev, E. und Zahlten, C. A Real-World Benchmark Model for Testing Concurrent Real-Time Systems in the Automotive Domain. In: *Testing Software and Systems*. Hrsg. von B. Wolff und F. Zaïdi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 146–161. ISBN: 978-3-642-24580-0.

[194] GmbH, E. *RTA-OS TriCore/Tasking Port Guide*. 5.0.6. ETAS GmbH. 70469 Stuttgart, Germany, 2016.

[195] Fricke, T., Uzlu, C., Mallwitz, R., Ries, J., Wussow, J., Brockschmidt, J., Kurrat, M., Engel, B., Jungklass, P. und Grieger, F. NetProsum2030: A Contribution to the Solution for Distributed Energy Supply in 2030. In: *PCIM Europe*. 2020. ISBN: 978-3-8007-5245-4.

[196] Elvers, C. und Jungklass, P. Agile Software-Entwicklung im automobilen Umfeld. In: *Tagungsband Embedded Software Engineering Kongress 2021*. 2021. ISBN: 978-3-8343-6291-9.

[197] Jungklass, P., Stoeber-Schmidt, C.-P., Barg, R., Hansen, H. und Siebert, M. Post-Quantum Cryptography on Embedded ECUs. In: *2024 JSAE Annual Congress (Spring)*. JSAE, 2024, S. 1–6.

[198] Sinell, F., Tschirley, R. und Jungklass, P. Evaluation of Post-Quantum Cryptography Signature Scheme on an Automotive Multicore Microcontroller for Real-Time Application. In: *Advances in Information and Communication*. Springer Nature Switzerland, 2025, S. 454–466. ISBN: 978-3-031-85363-0.

[199] Jungklass, P., Stoeber-Schmidt, C.-P., Siebert, M., Rummel, J. und Nigoro, T. Firmware Security Module. In: *2025 JSAE Annual Congress (Spring)*. JSAE, 2025.

[200] Elvers, C., Jungklass, P., Stöber-Schmidt, C.-P., Rummel, J. und Nigoro, T. Agile Software Development in the Automotive Environment - Scrum and Automotive SPICE, Contradiction in Terms? In: *2025 JSAE Annual Congress (Spring)*. JSAE, 2025.

[201] Stöber-Schmidt, C.-P., Nigoro, T., Siebert, M., Jungklass, P. und Rummel, J. Penetration Testing of Automotive Systems - Efficient Security Analysis of Vehicular E/E Systems. In: *2025 JSAE Annual Congress (Spring)*. JSAE, 2025.

[202] Stöber-Schmidt, C.-P., Jungklass, P. und Siebert, M. Post-Quantum Cryptography on Embedded ECUs. In: *International VDI Conference - Cyber Security for Vehicles*. Juni 2024.

[203] Jungklass, P. Post-Quantum Cryptography for Embedded Systems. In: *3rd Charter of Trust Meetup Braunschweig*. Aug. 2024.

[204] Jungklass, P. Firmware Security Module. In: *4th Charter of Trust Meetup Braunschweig*. Nov. 2024.

[205] Jungklass, P. Verfahren zur Zwischenkernkommunikation in einem Mehrkernprozessor. Pat. 10 2018 123 563.1. 2020.

[206] Jungklass, P. Verfahren und Vorrichtung zur statischen Speicherverwaltungsoptimierung bei integrierten Mehrkernprozessoren. Pat. 10 2019 128 206.3. 2022.

[207] Jungklass, P. Verfahren zur Herstellung der Cachekohärenz in Mehrkernprozessoren. Pat. 10 2019 118 757.5. 2023.

[208] Körür, S. *Auswahl und Implementierung einer Embedded-Systems-Kommunikationsschnittstelle für einen Mikroprozessor vom Typ Infineon AURIX TC299*. Ostfalia - Hochschule für angewandte Wissenschaften. Bachelor-Thesis. 2018.

[209] Rettemeyer, T. *Untersuchung von Speicherkorrekturverfahren im Bezug auf Performance, Speicherbedarf und Fehlerkorrekturrate im AURIX TriCore 1G*. Ostfalia - Hochschule für angewandte Wissenschaften. Projektarbeit. 2020.

[210] Körür, S. *Entwicklung einer echtzeitfähigen Ethernet-Kommunikation für eine embedded Steuergeräteplattform in der Prototypentwicklung*. Ostfalia - Hochschule für angewandte Wissenschaften. Master-Thesis. 2021.

[211] Rettemeyer, T. *Entwicklung eines universellen Sicherheitskonzepts für eine embedded Steuergeräteplattform für die Prototypenentwicklung*. Ostfalia - Hochschule für angewandte Wissenschaften. Master-Thesis. 2021.

[212] Rogalski, M. *Entwicklung eines Demonstrators zum Vergleich repräsentativer automotive Lastszenarien für embedded Mikrocontroller*. Ostfalia - Hochschule für angewandte Wissenschaften. Bachelor-Thesis. Apr. 2021.

[213] Poeppel, L. *Embedded Programmierung eines Infineon AURIX (Tricore): Implementierung und Laufzeitanalyse von RSA und SipHash-2-4*. Hochschule für Technik und Wirtschaft Berlin. Bachelor-Thesis. 2022.