Adaptives Monitoring für Mehrkernprozessoren in eingebetteten sicherheitskritischen Systemen

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS (Dr.-Ing.)

an der Fakultät für
Elektrotechnik und Informationstechnik
am Karlsruher Institut für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Ing. Falco K. Bapp

geb. in Waiblingen

Tag der mündlichen Prüfung:
30. November 2017

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. Zhihong Wu

Adaptives Monitoring für Mehrkernprozessoren in eingebetteten sicherheitskritischen Systemen

1. Auflage: Januar 2018

„Zwei Dinge sind zu unserer Arbeit nötig:
Unermüdliche Ausdauer und die Bereitschaft, etwas, in das man viel Zeit und Arbeit gesteckt hat, wieder wegzuwerfen.“

Albert Einstein

Zusammenfassung

In vielen Anwendungsdomänen tragen softwarebasierte Systeme maßgeblich zu neuen Trends und Innovationen bei – so auch in den Mobilitätsdomänen Automobilbau, Luftfahrt und Eisenbahnindustrie. Wesentliche Neuerungen können in Software auf neuester Hardware-Technologie entwickelt und in Umlauf gebracht werden. Speziell in den Mobilitätsdomänen sind besondere Anforderungen zu berücksichtigen sobald die Funktionen und Technologien in sicherheitskritischen Anwendungen integriert und eingesetzt werden. Neueste Hardware ist jedoch oftmals nicht für den Einsatz in solchen Anwendungen ausgelegt und kann daher die durch Standards und Normen vorgegebene Anforderungen nicht ohne weiteres erfüllen.

Dies gilt auch für den aktuellen Trend in der Prozessortechnologie, den Mehrkernprozessoren. Die bereits in Multimedia und Unterhaltungsmedien weit verbreiteten Mehrkernprozessoren können nicht uneingeschränkt Einzug in sicherheitskritische Anwendungen halten. Spezielle Methoden zur Absicherung im Sinne der funktionalen Sicherheit werden benötigt, um Mehrkernprozessoren überwachen und somit mindestens ein gleiches Maß an Sicherheit, wie in bereits etablierten Technologien, garantieren zu können.

In der vorliegenden Arbeit werden Methoden vorgestellt, die zur Steigerung der Zuverlässigkeit für Multicoreprozessoren eingesetzt werden können und es erleichtern, diese neuartige, komplexe Technologie in eingebetteten sicherheitskritischen Anwendungen einzusetzen. Anwendungsbereiche stellen beispielsweise Automobile, Flugzeuge, Anwendungen im Bereich der Industrieautomatisierung oder Züge dar.

Obwohl (verteilte) Mehrprozessorsysteme bereits seit einigen Jahren eingesetzt werden, unterscheiden sich die Herausforderungen zur Absicherung durch die Integration in einen Chip erheblich von den bereits

bekannten Herausforderungen bei der Entwicklung von Mehrprozessorsystemen. Der Übergang von verteilten Mehrprozessorsystemen zu hoch integrierten Mehrkernprozessoren bringt nicht nur eine neue Technologie, sondern auch eine immens gesteigerte Komplexität mit sich.

In den folgenden Kapiteln dieser Arbeit werden zunächst aktuelle Arbeiten und die Herausforderungen sowie die einhergehende Komplexität beim Übergang von Mehrprozessor- zu Mehrkern-Systemen vorgestellt. Diese Herausforderungen werden im Kontext der Applikationen als Fehlerbilder sichtbar, die wiederum zu Systemausfällen mit schwerwiegenden Folgen führen können. Diese resultierenden Fehlerbilder und deren Ursprung werden dargestellt. Um mögliche Fehler und daraus resultierende Ausfälle frühzeitig erkennen zu können werden im weiteren Verlauf der Arbeit *neuartige Methoden zur Überwachung und Fehlererkennung in Mehrkernprozessoren* vorgestellt und gegen die eingeführten Fehlerbilder reflektiert. Die Monitoring Mechanismen sind dabei nicht auf einen einzelnen Teil des Mehrkernprozessors oder eine Ebene im Design beschränkt, vielmehr handelt es sich um eine *Hardware/Software Co-Design* Entscheidung, welche der Mechanismen in Hardware und/oder in Software abgebildet und auf welcher Ebene im System diese umgesetzt werden. Das hieraus entstehende *Multi-Level Monitoring mit parametrierbaren und adaptiven Konzepten* deckt alle Ebenen von der Applikation bis zur Hardwareplattform ab.

Doch nicht nur die Überwachung von Mehrkernprozessoren spielt eine entscheidende Rolle, auch die *sichere, deterministische und effiziente Nutzung von Ressourcen* innerhalb des System-On-Chip stellt eine besondere Herausforderung dar. Dieser Nutzung wird ein weiteres Kapitel dieser Arbeit mit einem neuartigen Konzept gewidmet, das eine für die Software transparente Virtualisierung bereitstellt. Die eingeführte *Hardware-Virtualisierung* kann in weiten Bereichen ebenfalls parametriert werden und bietet die Möglichkeit zur Integration eines anwendungsspezifischen Schedulingverfahrens. Die vorgestellten Konzepte werden prototypisch implementiert, bewertet und es wird eine Validierung gegen die Fehlerbilder durchgeführt.

Weiterhin wird basierend auf den aktuellen Trends in der Industrie und Forschung davon ausgegangen, dass zukünftige Anwendungen, speziell durch den steigenden Grad an Automatisierung, strengeren

Anforderungen genügen müssen. Dies bedingt, dass eine einfache Fehlererkennung und die Überführung in einen sicheren Systemzustand den künftigen Anforderungen nicht mehr genügen und ein bestimmter, minimaler Funktionsumfang immer bereitgestellt werden muss. Ein *Konzept für die dynamische Migration von Funktionen für künftige Fail-Operational Systeme* zur Integration in einen Mehrkernprozessor rundet die in dieser Arbeit vorgestellten Konzepte ab.

Speziell die Entwicklung von sicherheitskritischen Anwendungen folgt strikten, durchgängigen und wohldefinierten Prozessen, in welchen die Mechanismen nicht losgelöst voneinander betrachtet werden dürfen. Zur besseren Handhabung der Konzepte und zur Anbindung an bereits bestehende und etablierte Entwicklungsprozesse, werden die Methoden in ein *Bibliothekskonzept* integriert. Dies sichert die einfache Nutzbarkeit und die Übertragbarkeit auf andere Anwendungsfälle und Architekturen. Die so entwickelten Systeme werden durch die vorgestellten Konzepte, die weitgehend parametriert und konfiguriert werden können und sich auf den jeweiligen Anwendungsfall anpassen lassen, unterstützt und reduzieren die Komplexität bei der Entwicklung.

Abstract

In many application domains, software-based systems build the foundation for new trends and innovations – such as in the mobility domains Automotive, Avionics and Railway. Important innovations can be developed in software running on latest hardware technology and hence be introduced into new products. Especially in the mobility domains, strict requirements have to be considered when the applications and technology shall be used and integrated into a safety-critical application. However, latest hardware is often not intended to be used in such a scenario and hence cannot fulfill requirements specified by standards of the related domains without modifications or special treatment.

This extends to the latest trend in processor technologies, the so called multicore processors. This technology is already widely used for several years in multimedia products, but cannot be introduced easily in safety-critical applications. Special methods for protection and validation in the context of functional safety are needed to provide at least the safe confidence and level of safety as well known and established technology.

In this work methods are presented that can be used to increase the level of reliability of multicore processors and ease the use of this new and upcoming complex technology in embedded safety-critical applications. The area of application ranges from automobiles and airplanes to industrial automation and railways.

Even though (distributed) multiprocessor systems are already used for several years, the challenges in the development of a safe system using these integrated multiple cores in a single chip differs fundamentally from known challenges using multiprocessor systems. The transition from multi-processor systems to highly integrated multicore processors does not only introduce a new technology; it also introduces an enormous increase in complexity of the system.

In the following chapters of this thesis, related work and challenges are described as well as the complexity coming along with the transition from multiprocessor to multicore systems. These challenges are becoming visible in the context of the applications as error pattern, which can subsequently lead to system failures with possibly fatal consequences. The resulting error pattern and their sources are introduced. In order to recognize failures and prevent from resulting system failures, new *methods for monitoring and failure recognition in multicore processors* are discussed and validated against the described error pattern. The monitoring mechanisms are not limited to one level in the design, they are rather *Hardware/Software Co-Design* decisions about the realization in hardware and/or software and about their introduction on the different levels of the system. The resulting *Multi-Level monitoring with the capability of parametrization and adaptability* covers all levels from the application down to the hardware platform.

However, monitoring of the behavior of a multicore processor is not the only issue to be covered. *Efficient, deterministic and safe usage of shared resources* within a System-on-Chip is another huge challenge in the development of multicore-based systems. To address this usage, a subsequent chapter of this work describes a new concept, which introduces a software transparent solution of device virtualization. The introduced *Hardware-Virtualization* can also be parametrized in many aspects and provides an opportunity for integrating a use case specific scheduling mechanism. The described approaches are implemented prototypically and validated against the error pattern.

Furthermore, latest trends in industry as well as in research show that future applications will have to fulfill more and more stringent requirements basically due to the increasing level of automation. This implies that a simple failure recognition and transition to a safe state is not enough to cover future requirements, they rather need a minimal functionality, at least degraded, which is provided in any case. Hence, a *concept for the dynamic migration of functions for future fail-operational systems* using multicore processors completes the methods described in this work.

Especially in safety-critical applications, the development has to follow well-defined, strict and holistic processes in which mechanisms are not

considered separately. For better handling of the concepts and possible integration into existing and established processes, the methods are integrated into a *library concept*. This ensures an easy applicability and portability for other use cases and architectures. Developed systems based on the introduced concepts, which are widely parametrizable and adaptable for the use case, support and ease the development and hence reduce the complexity.

Vorwort

In einigen spannenden Jahren als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Institut für Technologie (KIT) ist die vorliegende Arbeit entstanden. In den Jahren hatte ich die Gelegenheit viele wundervolle Menschen kennenzulernen und Kontakte zu knüpfen. Ebenso bekam ich einen wertvollen Einblick in unterschiedliche Forschungsthemen, Lehrveranstaltungen und Kulturen. Ich denke sehr gerne an diese schöne und prägende Zeit zurück und möchte mich hiermit bei allen bedanken, die mich auf diesem Weg begleitet haben.

Zunächst möchte ich mich besonders bei meinem Doktorvater Prof. Jürgen Becker bedanken, der mir im Anschluss an das Studium die Promotion am ITIV ermöglicht hat. Mit einem immer offenen Ohr, einer exzellenten Betreuung sowie kritischen Diskussionen trug er maßgeblich zum Gelingen dieser Arbeit bei. Weiterhin möchte ich mich für das entgegengebrachte Vertrauen bedanken, das es mir erlaubte, Verantwortung in Forschungsprojekten, -anträgen und am Institut zu übernehmen. Die hierdurch gesammelten Erfahrungen und Kontakte trugen sehr zu einer schönen und spannenden Zeit am Institut bei. Hierzu gehören auch interessante Gespräche und der Einblick in weitere vielfältige Forschungsthemen in unterschiedlichem Kontext.

Ebenfalls herzlich bedanken möchte ich mich bei Prof. Wu Zhihong von der Tongji Universität in Shanghai, der sich nicht nur zur Übernahme des Korreferats bereiterklärt hat, sondern mir die Möglichkeit gegeben hat, eine neue Kultur kennenzulernen und besondere Erfahrungen zu sammeln. Weiterhin möchte ich Prof. Wu für die hervorragende Kooperation und den spannenden Austausch mit ihm und seinem Team danken.

Einen weiteren Dank möchte ich an meine einzigartigen Kollegen am ITIV aussprechen, die mich mehrere Jahre begleitet und mit Kritik und viel Spaß jeden Tag bereichert haben. Besonders erwähnen möchte ich an dieser Stelle Timo Sandmann, als langjährigen Kollegen in Forschungsprojekten des ITIV, der mir immer unterstützend zu Seite stand und auch, wenn angebracht, Kritik geübt hat. Auch Oliver Sander, der mich speziell zu Beginn meiner Promotion unterstützt hat und mir mit Rat und Tat am Institut und privat zur Seite stand. Ein weiterer Dank geht auch an die ehemaligen Kollegen und Freunde, die vor mir das Institut verlassen haben, Michael Dreschmann, Joachim Meyer, Lukas Meder und Alexander Klimm. Aber auch allen aktuellen Kollegen vom Erdgeschoss bis in den dritten Stock des ITIV, speziell bei den Organisatoren Lidia, Steffen und Houssem, möchte ich gerne für sehr lustige Grillabende, Kinobesuche oder sonstige Freizeitaktivitäten danken – bei solchen Kollegen wird der Arbeitsalltag nie langweilig.

Weiterhin möchte ich mich noch bei allen bedanken, die durch ihre Anregungen, Korrekturen und Diskussionen auf die eine oder andere Weise zur Erstellung dieser Arbeit beigetragen haben. Bedanken möchte ich mich auch bei den Studenten, die im Rahmen ihrer HIWI Tätigkeit, Studien-, Diplom-, Bachelor oder Masterarbeit in diesem Themengebiet einen Beitrag geleistet haben.

Nicht zuletzt geht ein ganz spezieller Dank an meine Familie, ganz besonders an meine Eltern Klaus und Ursula als auch an meine Geschwister Branca und Jaro. Auch sie unterstützten mich stets auf dem Weg zu dieser Dissertation und halfen mir dabei, die Themen auch für Laien erklären zu lernen und sorgten für den nötigen Ausgleich zur täglichen Forschungsarbeit.

Karlsruhe, im Januar 2018
Falco K. Bapp

Inhaltsverzeichnis

1 Einleitung

Im täglichen Leben werden wir zunehmend mit Elektronik sowie softwarebasierten eingebetteten Systemen konfrontiert. Diese spielen eine zunehmend wichtigere Rolle und begleiten uns durch den gesamten Alltag, sei es in Radioweckern, elektrischen Zahnbürsten, Smartphones oder in Mobilitätssystemen. In vielen Systemen in unterschiedlichen Domänen werden zunehmend Prozessoren integriert (vgl. Abb. 1.1). Mehr und mehr Innovationen werden von softwarebasierten Systemen getrieben, so beispielsweise ca. 90% der Innovationen in der Automobilindustrie. Die eingesetzten Systeme werden an ihrem Performanzlimit betrieben und können nach wie vor nicht alle Anforderungen zufriedenstellend erfüllen.

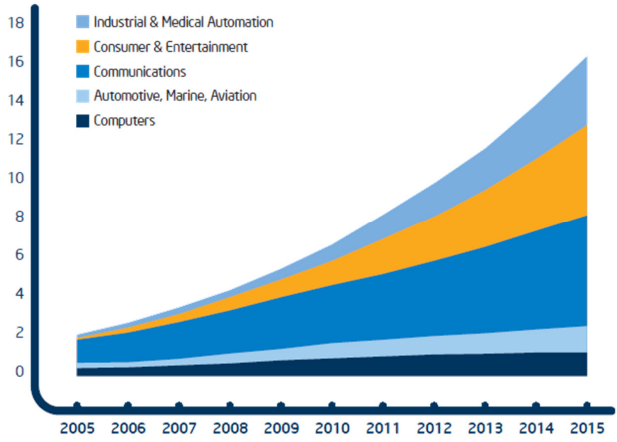


Abbildung 1.1: Weltweite Entwicklung der Prozessorenanzahl in unterschiedlichen Domänen (in Milliarden), Quelle: [1]

Dem gegenüber steht die Halbleiterindustrie, die zunehmend performantere Recheneinheiten mit immer kleiner werdenden Strukturgrößen bereitstellt. Lange Zeit folgten die Entwicklungen dem Mooreschen Gesetz, welches besagt, dass sich die Anzahl an Komponenten in einem Chip zunächst jährlich, später alle zwei Jahre verdoppelt. Doch die Integrationsdichte von Transistoren und die realisierbaren Taktfrequenzen stoßen zunehmend an physikalische und thermische Grenzen. Zur weiteren Steigerung der Rechenleistung von Prozessoren ist eine Steigerung der Taktfrequenz nicht mehr möglich. Mehr und mehr Hersteller gehen zu Prozessoren über, die mehr als nur einen Rechenkern integrieren und mit gleicher oder sogar geringerer Taktfrequenz betrieben werden, wie es auch in der Entwicklung der Prozessoren von Intel zu erkennen ist (Abb. 1.2). Theoretisch wird mit einer Verdopplung der Rechenkerne die Rechenleistung verdoppelt. Dies ist jedoch durch eine Vielzahl von nicht vervielfältigten Komponenten, wie beispielsweise Speichercontroller, nicht der Fall.

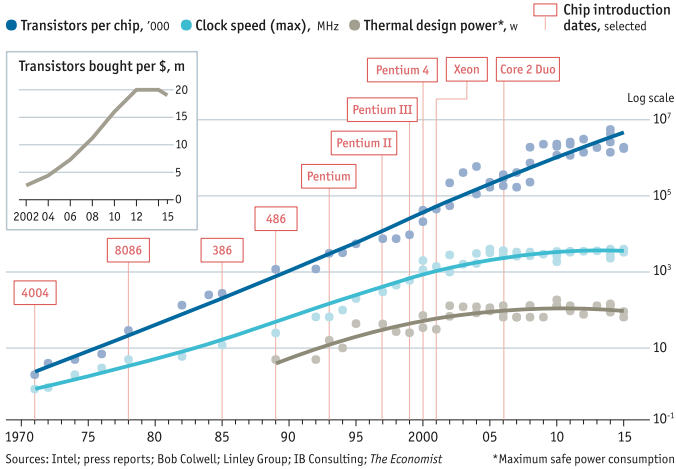


Abbildung 1.2: Prozessorentwicklung von Intel, Quelle: [2]

Doch nicht nur die ausbleibende Vervielfältigung von Komponenten innerhalb eines Mehrkernprozessors stellt sich dabei als Einschränkung heraus. Ebenso wird die Performanzsteigerung durch die Charakteristik

der Applikation eingeschränkt. Dies wurde von Amdahl bereits in den frühen 60er Jahren beschrieben und ist heutzutage als Amdahl's Gesetz bekannt. Amdahl beschreibt dabei einen Performanzzuwachs in Abhängigkeit der verfügbaren Rechenkerne und des parallelisierbaren Anteils der Applikation (Abb. 1.3). Wenn man aktuelle Entwicklungen in der Prozessortechnologie berücksichtigt, so beispielsweise lokale Caches pro Rechenkern, so wirkt Amdahl's Gesetz sogar zu pessimistisch. In realen Szenarien bleibt der Performanzzuwachs jedoch unterhalb der vorhergesagten Kurve.

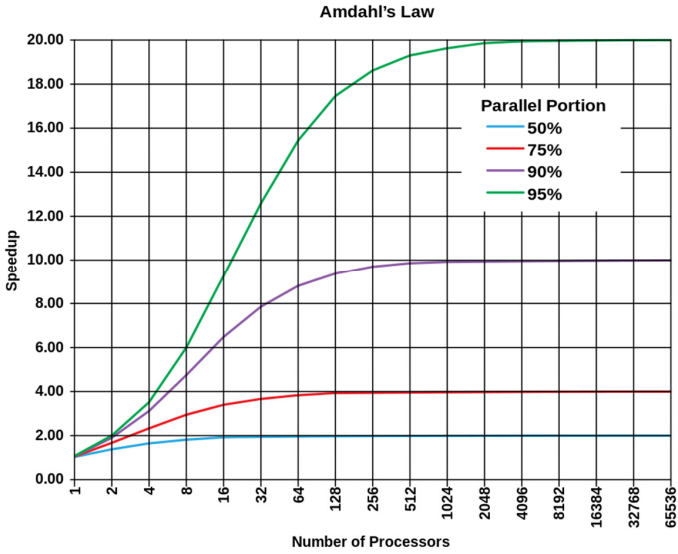


Abbildung 1.3: Amdahl's Gesetz, Quelle: [3]

Begründet werden kann der ausbleibende Zuwachs an Performanz durch die entstehende Kommunikation durch die Parallelisierung der Applikationen und die gemeinsam genutzten und zum Teil nur einfach vorhandenen Komponenten. Dennoch wird die Mehrkernprozessortechnologie als einzige verfügbare Technologie zur Erfüllung der steigenden Performanzanforderungen angesehen.

1.1 Motivation und Umfeld

Die in den vergangenen Jahren zu beobachtenden Fortschritte in der Prozessortechnologie zeigen einen klaren Trend in Richtung Mehrkernprozessoren. Diese, mittlerweile in den Unterhaltungsdomänen weit verbreiteten Architekturen, weisen durch ihre hochgradig parallele Ausführung von Software eine weitere Performanzsteigerung und eine erhöhte Bereitstellung digitaler Rechenleistung auf. Parallel zu dieser Entwicklung steigen die Anforderungen an eingebettete, digitale Rechenleistung in den Mobilitätsdomänen (z.B. Automobilbau, Bahn und Luftfahrt). Jedoch gilt es hier andere und zumeist wesentlich striktere Anforderungen zu erfüllen. Diese resultieren aus der Tatsache, dass die betrachteten Systeme typischerweise eine sicherheitskritische Funktion ausführen, so beispielsweise Steuergeräte in Fahrzeugen (z.B. ABS, ESP), Flugzeugen (z.B. Flugsteuerungen, Triebwerkssteuerungen) oder auch in Eisenbahnen. Unter sicherheitskritisch wird in diesem Zusammenhang verstanden, dass der Ausfall bzw. die Abweichung einer Systemfunktion zu erheblichen wirtschaftlichen oder humanitären Schäden führen. Aus diesem Grund werden für Mobilitätsdomänen Normen und Standards bereitgestellt, die bei der Entwicklung berücksichtigt werden müssen. Die strikten Anforderungen resultieren häufig aus den einschlägigen Standards für die Entwicklung von elektronischen Komponenten und Systemen der jeweiligen Domäne.

Die gestiegene Anforderung an Rechenleistung in den Mobilitätsdomänen kann zum einen auf neue Funktionen, zum anderen auch auf eine Integration von unterschiedlichen Funktionen in ein Steuergerät zurückgeführt werden. In vielen Anwendungsdomänen ist das Streben nach immer mehr Funktionalität zumeist geprägt durch softwarebasierte Systeme zur Steigerung der Produktqualität aber auch der Konkurrenzfähigkeit an der Tagesordnung. Aufgrund der nicht-funktionalen Anforderungen (Space, Weight and Power (SWAP)) ist die Realisierung dessen auf weiteren kleineren Steuergeräten, basierend auf Prozessoren mit geringerer Rechenleistung, nicht möglich. Speziell in der Automobilbranche ist derzeit ein weiterer Trend hin zu immer leistungsfähigeren Domänenleitrechnern zu beobachten [4], in denen alle Funktionalität einer Fahrzeugdomäne (z.B. Chassis, Power-Train, Body) integriert werden. Ähnliche Ansätze sind in der Luftfahrt zu verzeichnen. Funk-

tionen, die klassischerweise über mehrere Steuergeräte verteilt waren und dort jeweils einen eigenen Prozessor zur Verfügung hatten, teilen sich nun eine gemeinsame Hardwareplattform (u.a. Ausführungseinheit und Speicher). Da typischerweise in den Produkten der Mobilitätsdomänen SWAP Anforderungen sehr hoch sind, wird durch die Integration versucht eine Reduktion von Steuergeräten zu erreichen.

Es ist naheliegend den Trend der Mehrkernprozessoren auch in die Mobilitätsdomänen zu übernehmen, um den hohen Anforderungen an Rechenleistung gerecht werden zu können. Aus der Architektur der Mehrkernprozessoren resultieren aber verschiedenste Herausforderungen, die für einen sicheren und zuverlässigen Einsatz bei der Ausführung einer sicherheitskritischen Funktion gemeistert werden müssen. Hierzu gehören u.a. die aus der Parallelität entstehenden möglichen Überschneidungen zeitlicher Natur (zum Beispiel zeitgleicher Zugriff auf Speicher oder Peripherieeinheiten), sowie die gemeinsame Nutzung von Sub-Komponenten in der Prozessorarchitektur (räumlicher Natur). Gerade zur Sicherstellung der korrekten Ausführung einer bestimmten Funktion müssen solche Überschneidungen vermieden bzw. mindestens erkannt werden. Die Rest-Fehlerwahrscheinlichkeit muss unter den in den Normen und Standards geforderten Schranken liegen. Zur Erreichung und Sicherstellung dieser Anforderungen müssen Methoden, Konzepte und Pattern erforscht werden, die eine Fehlererkennung und Vermeidung ermöglichen.

Diese Konstellation von neuartiger, aufkommender Prozessortechnologie mit den Mobilitätsdomänen und den dabei entstehenden Herausforderungen motivieren die in dieser Arbeit entstandenen Methoden und Konzepte, welche (Teil-) Lösungen für den Einsatz von Mehrkernprozessoren in den Mobilitätsdomänen darstellen. Um die korrekte Ausführung auf einem Mehrkernprozessor sicherzustellen, ist eine Überwachung (Monitoring) auf unterschiedlichen Ebenen unerlässlich. Die Umsetzung dieser Überwachung kann vielfältig und aus unterschiedlichen Betrachtungsweisen (Hardware, Software, System) heraus erfolgen. Zur sicheren Nutzung von gemeinsamen Ressourcen in einem Mehrkernprozessor müssen geeignete Methoden angewandt werden, die ein fehlerhaftes Verhalten verhindern. Dies sind betrachtete Herausforderungen, die in dieser Arbeit näher untersucht und denen mit entsprechenden Methoden entgegnet wird.

1.2 Zielsetzung und Abgrenzung der Arbeit

Anknüpfend an die Beobachtungen im Bereich der geforderten digitalen Rechenleistung in sicherheitskritischen Anwendungen in den Mobilitätsdomänen, beschäftigt sich diese Arbeit mit Methoden zur Überwachung von sowie der sicheren Nutzung der Ressourcen in Mehrkernprozessoren. Um einen möglichst geringen Overhead in Software wie auch in Hardware zu erreichen, wird das Augenmerk auf ressourcensparende Konzepte und Methoden gelegt. Da bei der Verwendung von Mehrkernprozessoren eine nicht zu unterschätzende Komplexität (im Sinne der Softwareentwicklung aber auch der Dokumentation) in das Systemdesign eingebracht wird, zielt die Arbeit auf eine möglichst transparente Umsetzung für den Softwareentwickler ab. Insbesondere werden heterogene Mehrkernprozessoren betrachtet und dabei untersucht in wie weit die Heterogenität vorteilhaft genutzt werden kann. Die Lösung aller Herausforderungen beim Einsatz von Mehrkernprozessoren in sicherheitskritischen Anwendungen ist ausdrücklich nicht das Ziel dieser Arbeit.

1.3 Aufbau der Arbeit

Die Arbeit ist, um die Thematik zu beschreiben, wie folgt gegliedert:

In Kapitel 2 werden einige Grundlagen beschrieben, die zum besseren Verständnis der behandelten Themen beitragen. Darüber hinaus werden Begriffe definiert und eingeführt, wie sie im weiteren Verlauf der Arbeit verwendet werden. Zu den Grundlagen gehören, abgesehen von den Begriffen und Definitionen, eine Beschreibung der im Kontext der Arbeit relevanten Domänen und den zugehörigen Standards sowie dem zugehörigen Entwicklungsvorgehen. Gefolgt wird diese Beschreibung von einer kurzen Einführung in aktuelle Prozessortechnologien sowie einer Vorstellung von bekannten Virtualisierungstechniken.

Ergänzt werden die Grundlagen durch Kapitel 3, in welchem der aktuelle Stand der Forschung und Technik aufgearbeitet wird. In Zusammenhang mit der vorliegenden Thematik sind dabei Redundanzkonzepte, der Einsatz von Prozessoren in sicherheitskritischen Anwendungen

sowie Monitoringkonzepte und Virtualisierung im Fokus. Das Kapitel schließt mit einer Abgrenzung der vorliegenden Arbeit zu bekannten Ansätzen des Standes der Technik und identifiziert die Lücken, die mit diesem Beitrag gefüllt werden sollen.

Auf Basis der identifizierten Lücken folgt in Kapitel 4 die Herleitung der Herausforderungen bei der Entwicklung multicorebasierter Systeme. Dabei wird verdeutlicht, welche Unterschiede sich aus der Architektur ergeben und welche spezifischen Aspekte bei der Entwicklung berücksichtigt werden müssen. Auf Basis der noch zu lösenden Herausforderungen wird aufgezeigt, welche Fehlerbilder aus Applikationssicht entstehen und welchen Ursprung diese haben können. Die Fehlerbilder werden im weiteren Verlauf der Arbeit zur Validierung der Methoden verwendet und bilden daher eine wichtige Basis.

Kapitel 5 beschäftigt sich mit Methoden zur Überwachung und Fehlererkennung in Mehrkernprozessoren. Es werden verschiedene Ansätze vorgestellt, die sich von Redundanz über eine Überwachung auf unterschiedlichen Ebenen bis hin zum Monitoring von speziellen Komponenten erstrecken. Die vorgestellten Methoden werden jeweils anhand von prototypischen Implementierungen validiert und abschließend in Bezug zu den Fehlerbildern aus Applikationssicht gesetzt.

Nicht nur die Überwachung des Prozessors bildet eine Lücke im Stand der Technik die adressiert wird, auch die sichere Nutzung von gemeinsam genutzten Ressourcen spielt eine wichtige Rolle. In Kapitel 6 wird aus diesem Grund eine Methode zur sicheren Nutzung von Ressourcen vorgestellt und anhand der Beschreibung ihrer prototypischen Implementierung validiert und wiederum in Bezug zu den Fehlerbildern aus Applikationssicht gesetzt. Für zukünftige Systeme mit noch höheren Anforderungen an die Verfügbarkeit wird außerdem eine Methode zur dynamischen Migration von Funktionen vorgestellt und diskutiert.

Gefolgt wird die Beschreibung der Methoden aus Kapitel 5 und 6 von einer ganzheitlichen Betrachtung. Diese Betrachtung sowie eine Beschreibung der Übertragbarkeit auf andere Architekturen wird in Kapitel 7 erörtert. Hierbei wird dargestellt, welche Ansätze kombiniert werden können und welche Anforderungen an die Plattform gestellt

werden. Das Kapitel schließt mit der Beschreibung der Übertragbarkeit der Methoden und deren Skalierbarkeit.

Abschließend werden in Kapitel 8 Schlussfolgerungen gezogen, die sich aus den durchgeführten Arbeiten ableiten lassen. Ebenfalls werden die Ergebnisse bewertet und ein Ausblick auf mögliche Folgearbeiten skizziert.

2 Grundlagen

Die in diesem Kapitel aufgeführten Grundlagen stellen eine Basis für das Verständnis des Umfelds und der zugrundeliegenden Technologien dar. Hierfür wird eine kurze Übersicht über die diskutierten Anwendungsdomänen und ihre spezifischen Charakteristika gegeben, um das Verständnis und die Motivation der weiteren Absicherung zu stärken. Weiterhin wird die eingesetzte Prozessortechnologie und mögliche Architekturen eingeführt, die im weiteren Verlauf der Arbeit referenziert werden und zur Umsetzung dienen. Gefolgt wird diese Einführung von einer kurzen Erläuterung von Virtualisierung.

2.1 Begriffe und Definitionen

Da sich die Arbeit insbesondere mit dem Einsatz vom Mehrkernprozessoren in sicherheitskritischen Anwendungen befasst, werden der Kontext und die daraus resultierenden Spezifika im Folgenden kurz erläutert. Dazu gehört auch eine Einführung unterschiedlicher Begriffe, die im weiteren Verlauf der Arbeit verwendet werden.

2.1.1 Sicherheit

Eine Anwendung wird dann als sicherheitskritisch bezeichnet, wenn es eine Systemfunktion gibt, bei deren Ausfall eine erhebliche Gefahr für Menschen oder Sachwerte entstehen. Zur Sicherstellung der korrekten Funktion und damit der Reduktion des Restrisikos werden unterschiedliche Maßnahmen benötigt. Der in dieser Arbeit verwendete Begriff der Sicherheit bezieht sich auf den Terminus der Betriebssicherheit/funktionale Sicherheit (engl. Safety, im weiteren Verlauf auch als Synonym

verwendet) und explizit nicht auf den Begriff der Angriffssicherheit (engl. Security). Alle in einem sicherheitskritischen System enthaltenen Komponenten, inkl. Software, werden auch als sicherheitskritisch eingestuft. Die Einstufung der Kritikalität erfolgt anhand von Standards und wird in Kapitel 2.2 kurz erläutert. Ein System gilt auch bei einem gewissen Restrisiko als sicher. Die Grenzwerte und zulässigen Restrisiken werden durch Standards und Normen vorgeschrieben und müssen mindestens erreicht werden (vgl. Kapitel 2.2).

2.1.2 Risiko

Der Begriff **Risiko** beschreibt die zu erwartende Häufigkeit, mit welcher ein Ereignis, das zu Schäden führen kann, eintritt, in Kombination mit dem zu erwartenden Schadensausmaß. Die Häufigkeit setzt sich dabei typischerweise zusammen aus:

- Aufenthaltsdauer im Gefahrenbereich
- Möglichkeit zur Abwendung der Gefahr
- Wahrscheinlichkeit des Eintretens eines unerwünschten Ereignisses.

2.1.3 Fehler

Der Ausfall eines Systems resultiert aus Fehlern. In der deutschen Sprache wird häufig nur der Begriff Fehler verendet, der aber differenzierter betrachtet werden muss. Im Englischen finden die Begriffe *error - fault - failure* Anwendung. Die folgende Konkretisierung des Begriffs „Fehler“ kann entsprechend verwendet werden, um Klarheit über die gewünschte Bedeutung zu erreichen und lehnt sich an [5] und [6] an:

Fehlerursache (engl. Fault) ist die Quelle eines Fehlerzustandes. Fehlerursachen können durch zufällige Ereignisse oder aber durch systematische Fehler in der Entwicklung eintreten. Hierbei ist noch zu unterscheiden, ob die Fehlerursache permanent anhaltend, periodisch oder flüchtig ist. Permanente Fehlerursachen können beispielsweise aus einem Defekt einer Komponente resultieren, periodische hingegen treten

von Zeit zu Zeit wieder auf und resultieren zum Beispiel aus systematischen Fehlern (z.B. Überschreiten von Zeitgrenzen).

Fehlerzustand (engl. Error) ist ein Systemzustand, der nicht dem spezifizierten Zustand genügt. Der Fehlerzustand resultiert aus der Fehlerursache und wird später als Fehlerauswirkung sichtbar.

Fehlerauswirkung (engl. Failure) ist das abweichende Verhalten eines Elements, welches nicht mehr in der Lage ist, die benötigte Funktion auszuführen.

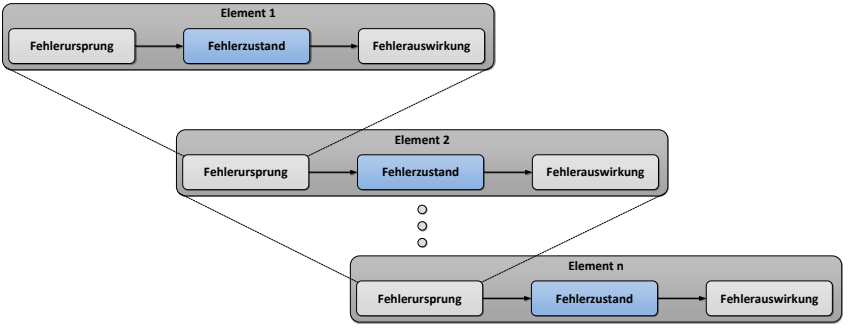


Abbildung 2.1: Darstellung von Fehlerursprung - Fehlerzustand - Fehlerauswirkung und der resultierenden Fehlerkette

Der Zusammenhang von Fehlerursprung, Fehlerzustand und Fehlerauswirkung ist in Abbildung 2.1 dargestellt. Es ist ebenfalls der Zusammenhang von verschiedenen hierarchisch verknüpften Elementen gezeigt und wie sich ein Fehler im ersten Element auf die weiteren Elemente auswirkt.

Es muss jedoch nicht zwangsweise eine hierarchische Verknüpfung von Elementen vorliegen. Fehler können sich ebenso im System entlang der funktionalen Kette ausbreiten. Dies ist exemplarisch in Abbildung 2.2 dargestellt. Der Fehler in *Element 1* kann somit entlang der funktionalen Kette (z.B. durch weitere Berechnungen auf fehlerhaften Daten) über *Element 3* und *Element 4* bis *Element 2* propagieren, auch wenn keine direkte Abhängigkeit von *Element 1* und *Element 2* vorhanden ist. Hierbei spricht man von kaskadierten Fehlern.

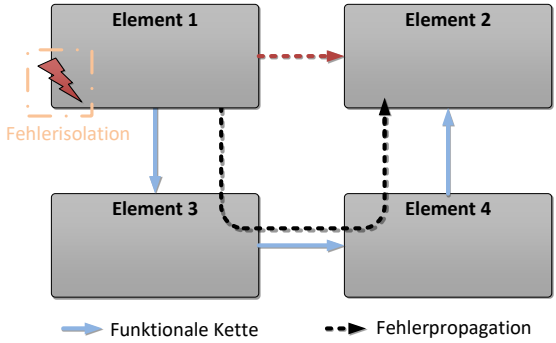


Abbildung 2.2: Darstellung der Fehlerpropagation entlang der funktionalen Zusammenhänge - kaskadierte Fehler

Eine weitere Konstellation, in welcher ein Fehlerursprung zu einem Fehlerzustand in mehreren Elementen führt, sind *Ausfälle aufgrund gemeinsamer Ursachen* (engl. Common Cause Failure - CCF). Beispiele hierfür können eine gemeinsame Takt- oder Spannungsversorgung sein. Die Abhängigkeiten sind in Abbildung 2.3 verdeutlicht. Die möglichen Fehlerquellen müssen von einer externen und unabhängigen Einheit überwacht werden, die von einem Ausfall nicht betroffen ist. Dies kann bedeuten, dass eine redundante Ausführung der Spannungsversorgung benötigt wird und die Signale galvanisch getrennt werden müssen.

2.1.4 Zuverlässigkeit

Bei der Zuverlässigkeit handelt es sich um eine Eigenschaft eines Systems, die angibt, in welchem Zeitintervall die dem System zugewiesene Funktion erfüllt wird. Die Zuverlässigkeit wird benötigt, da kein technisches System frei von Ausfällen ist. In engem Zusammenhang damit steht die Beschreibung der Zeitintervalle zwischen Ausfällen (engl. Mean Time Between Failure MTBF; bei reparierbaren Systemen) bzw. bis zu einem Ausfall (engl. Mean Time To Failure MTTF; irreparable Systeme).

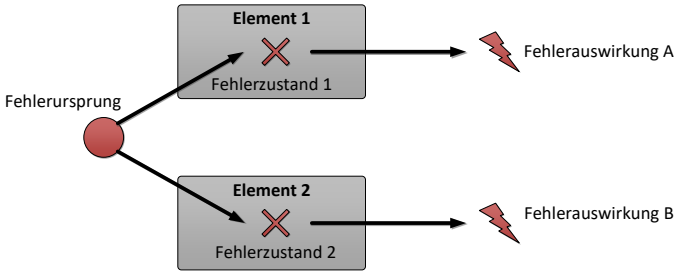


Abbildung 2.3: Darstellung der Abhängigkeiten bei Ausfällen mit gemeinsamen Ursachen (CCF)

2.1.5 Determinismus

Der **Determinismus** beschreibt eine Eigenschaft eines Systems, die besagt, dass sich das System bei definierten Eingaben immer gleich verhält. Dies gilt auch für zukünftige Zeitpunkte. Im hier betrachteten Kontext bedeutet dies, dass ein Prozessor zu jeder Zeit bei gleichen Eingaben die gleichen Ausgaben erzeugt.

2.1.6 Segregation

Segregation, auch Partitionierung genannt, hat zum Ziel, Ressourcen sicher zwischen mehreren Softwarekomponenten zu teilen. Dies ist ein kritischer Aspekt in Multicoreprozessoren, da mehrere Rechenkerne Software ausführen, die sich Ressourcen, wie das zentrale Bussystem oder Speicher, teilen. Dabei wird in zeitliche und räumliche Segregierung unterschieden.

2.2 Standards

Wie bereits in Kapitel 2.1.1 dargestellt, müssen bei der Entwicklung von sicherheitskritischen Anwendungen in der Regel Standards/Normen eingehalten und berücksichtigt werden. Die Normen bilden hierbei

einen Leitfaden für die Entwicklung und setzen damit Anforderungen. In den Normen sind Entwicklungsabläufe und Vorschläge zur Absicherung eines sicherheitskritischen Systems auf einer sehr abstrakten Ebene beschrieben.

Die funktionale Sicherheit wird in der Norm IEC61508 - "Functional Safety of electric/ electronic/ programmable electronic safety-related systems ausführlich beschrieben. Hier werden auch die damit in Zusammenhang stehenden Begriffe Risiko, Grenzrisiko, usw. definiert. Der Standard IEC61508 bietet die Basis für weitere domänenspezifische Normen (siehe Abbildung 2.4), die sich auf unterschiedliche Anwendungsgebiete beziehen.

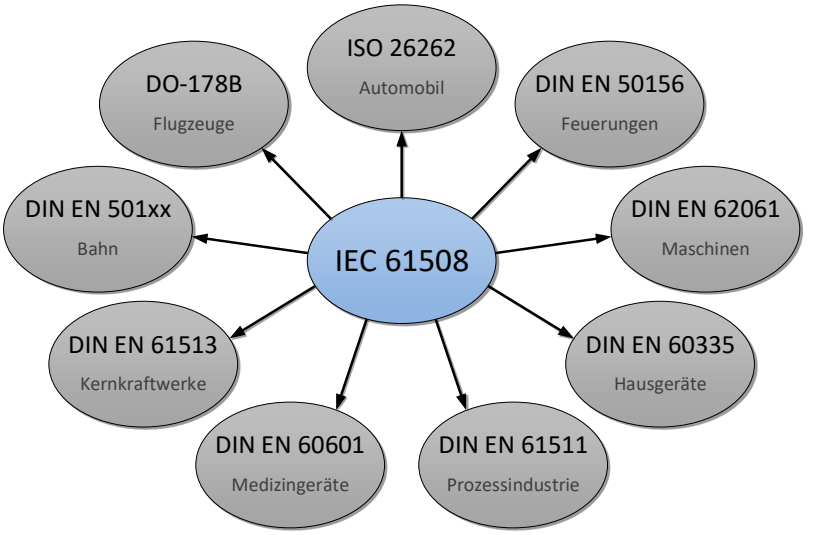


Abbildung 2.4: Übersicht der Normen, eigene Darstellung nach [7]

Die internationale Norm IEC61508 befasst sich mit den Anforderungen an elektrische/elektronische/programmierbare elektronische Systeme (E/E/PES). Diese Anforderungen dienen dazu, klare Grenzwerte und Sicherheitsziele zu definieren, die vom fertigen System eingehalten werden müssen. Hierzu wird ein tolerierbares Risiko (*Restrisiko*) festgelegt. Die Risikominimierung hat dabei höchste Priorität. Es handelt

sich um ein Vorgehen zur Vermeidung oder Beherrschung von gefährlichen Ausfällen eines Systems. Solche Ausfälle können durch systematische Fehler oder durch zufällige Ausfälle der Hardware entstehen. Systematische Fehler resultieren aus menschlichem Versagen, sowohl in der Anwendung als auch bei der Entwicklung (Spezifikationsfehler, Entwurfsfehler, Implementierungsfehler, etc.), wohingegen zufällige Hardwarefehler die begrenzte Zuverlässigkeit eines Hardwarebausteins (*Fehlerrate*) widerspiegeln.

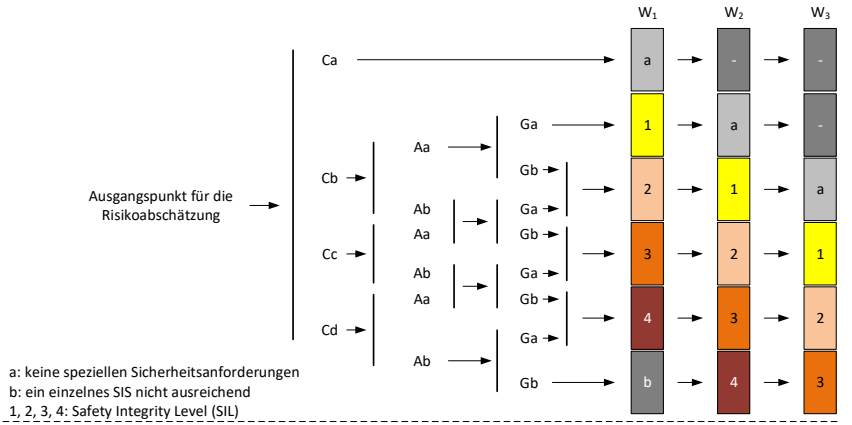
Aus diesen Gründen stellt die IEC61508 Anforderungen an die Vorgehensweise zur Entwicklung solcher Systeme, um den Sicherheitsanforderungen gerecht werden zu können. Hierzu zählen u.a. die Durchführung einer Risikoanalyse und der Entwurf der Hard- bzw. Software nach bestimmten vorgegebenen Prinzipien (vgl. [8]).

Auch die aus der IEC61508 abgeleiteten Normen folgen prinzipiell dieser Vorgehensweise, beinhalten jedoch branchenspezifische Änderungen. So muss vor der Entwicklung geprüft werden, nach welchen bereichsspezifischen Normen und Standards vorgegangen werden muss.

Um eine der Norm entsprechend selbst hergestellte oder beschaffte Komponente zu integrieren, müssen sicherheitskritische Systeme nach der Integration getestet werden. Anforderungen an Herstellungs- oder Beschaffungsprozesse stellt die Norm keine. Bei der Integration soll lediglich nachgewiesen werden, dass die Module korrekt aufeinander wirken und die spezifizierten Funktionen erfüllen. Es sollen hier nur Hardware- und Schnittstellenfunktionen geprüft werden, die Kombination mit der zugehörigen Software erfolgt erst im Sicherheitslebenszyklus der Software. Eine ausführliche Dokumentation über die Tests ist vom Entwickler anzufertigen. Im Anschluss an die Integration kann die Validierung erfolgen, um nachzuweisen, dass das System den spezifizierten Sicherheitsanforderungen entspricht (vgl. [9]).

2.2.1 Safety Integrity Level (SIL)

Um die Einstufung eines Systems fassbar machen zu können, bedient man sich sog. SIL. Diese SIL sind der Hauptaspekt der IEC61508. Hier werden nach bestimmten Kriterien Einstufungen vorgenommen,



Schadensausmaß		Gefahrenabwendung	
C_a	Leichte Verl. einer Person, kleinere schädl. Umwelteinflüsse	G_a	Möglich unter bestimmten Bedingungen
C_b	Schwere Verletzungen oder Tod einer Person	G_b	Kaum möglich
C_c	Tod mehrerer Personen	**Eintrittswahrscheinlichkeit**	
C_d	Tod sehr vieler Personen	W_1	Sehr gering
Aufenthaltsdauer einer Person im gefährlichen Bereich		W_2	Gering
A_a	Selten bis häufig	W_3	Relativ hoch
A_b	Häufig bis dauernd		

Abbildung 2.5: Qualitative Methode zur Ermittlung des SIL, eigene Darstellung nach [10]

die festlegen, dass entsprechende Maßnahmen zur Risikominimierung bestimmt werden müssen.

In Abbildung 2.5 ist eine qualitative Methode zur Ermittlung des SIL gezeigt (vgl. [10]). Zunächst muss das mögliche Ausmaß des Schadens festgelegt werden. Im Anschluss an diese Einstufung kann als weitere Spezifizierung die Aufenthaltsdauer im gefährlichen Bereich identifiziert werden. Nach den beiden letzten Einstufungen, sprich der Möglichkeit der Gefahrenabwendung und der Ermittlung der Eintrittswahrscheinlichkeit, kann das

entsprechende SIL abgelesen werden. Sollten nach durchlaufen dieses Schemas von der Norm selbst keine speziellen Sicherheitsanforderungen vorgeschrieben werden, sollten zumindest Aspekte der Qualitätssicherung und des eigenen Anspruchs greifen. Es wird deutlich, dass anhand von verschiedenen, nicht genau definierten Bedingungen ein Safety Integrity Level ermittelt wird. Die Ermittlung des SIL ist somit kein eindeutiger Prozess. Die Festlegung der relativen Parameter muss vom Entwicklungsteam erfolgen und einheitlich kommuniziert werden. Die IEC61508 gibt vier Safety Integrity Level vor, wobei die höchste (SIL 4) für diejenigen Funktionen bestimmt ist, welche bei Ausfall die verheerendsten Auswirkungen hat. Nach der Bestimmung des SIL können die zulässigen Ausfallraten der jeweiligen Systeme bzw. Komponenten aus Tabellen der IEC61508 entnommen werden (siehe Tabelle 2.1, vgl. [9]).

Tabelle 2.1: Zulässige Ausfallraten nach IEC61508 [9]

Safety Integrity Level (SIL)	Low demand mode of operation[1]	High demand or continous mode of operation, PFH[2]
1	$\leq 10^{-2}$ bis $< 10^{-1}$	$\leq 10^{-6}$ bis $< 10^{-5}$
2	$\leq 10^{-3}$ bis $< 10^{-2}$	$\leq 10^{-7}$ bis $< 10^{-6}$
3	$\leq 10^{-4}$ bis $< 10^{-3}$	$\leq 10^{-8}$ bis $< 10^{-7}$
4	$\leq 10^{-5}$ bis $< 10^{-4}$	$\leq 10^{-9}$ bis $< 10^{-8}$

[1] Average probalility of failure to perform its designed function on demand
[2] PFH (Probability of one dangerous failure per hour)

In einem System ist es möglich, serielle oder auch parallele Funktionsgruppen oder Elemente zu kombinieren. Hieraus folgt ein evtl. anderes Safety Integrity Level. In einer seriellen Kombination ergibt sich in der Regel das niedrigste SIL der in der Kombination vorhandenen Elemente, wohingegen bei parallelen Kombinationen evtl. ein höheres SIL beansprucht werden kann. Dies wird in Abbildung 2.6 verdeutlicht.

2.2.2 Safe Failure Fraction (SFF)

Bei der SFF handelt es sich um die Zahl der ungefährlichen Fehler zusammen mit der Anzahl der entdeckten gefährlichen Fehler, die in einem System auftreten. Die SFF wird in der IEC61508 [9] einge-

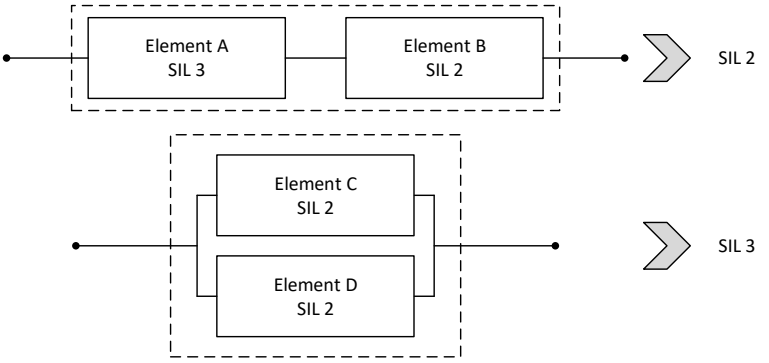


Abbildung 2.6: Bestimmung des resultierenden SIL nach [7]

führt. Die Übersicht aller in einem System möglichen Ausfälle kann aus dem Diagramm in Abbildung 2.7 entnommen werden. Um die SFF bestimmen zu können, müssen das Safety Integrity Level (SIL), die Fehlertoleranz und der Typ des Systems bekannt sein (vgl. [11]). Die Fehlertoleranz gibt an, wie viele Fehler ein System haben kann, bevor es seine Funktion nicht mehr erfüllt. So ist beispielsweise eine Fehlertoleranz von $N = 1$ so definiert, dass ein Fehler auftreten kann, das System jedoch weiter den dafür vorgesehenen Zweck erfüllt oder zumindest in einen sicheren Zustand wechselt (Beispiel hierfür ist Redundanz von Hardwareelementen). Beim Typ eines Systems handelt es sich um die Eigenschaft, die angibt, ob das Ausfallverhalten aller eingesetzten Bauteile ausreichend („Typ A“) oder nicht ausreichend („Typ B“) bekannt ist. Bei Elementen vom Typ A handelt es sich beispielsweise um Druckschalter oder ähnlichem, wohingegen komplexe Bauteile wie Prozessoren zu Typ B gezählt werden. Die SFF bestimmt sich aus der Kombination dieser nun bekannten Werte durch Ablesen aus Tabellen (vgl. Tabelle 2.2). Sie gibt den Anteil an unkritischen Fehlern an, der mindestens erreicht werden muss, um dem entsprechenden SIL gerecht zu werden.

Die Ausfallrate λ besteht aus unterschiedlichen Anteilen von Ausfallraten, die in Abbildung 2.7 dargestellt sind. Die Ausfallrate der ungefährlichen Ausfälle kann mit nachstehender Formel berechnet werden:

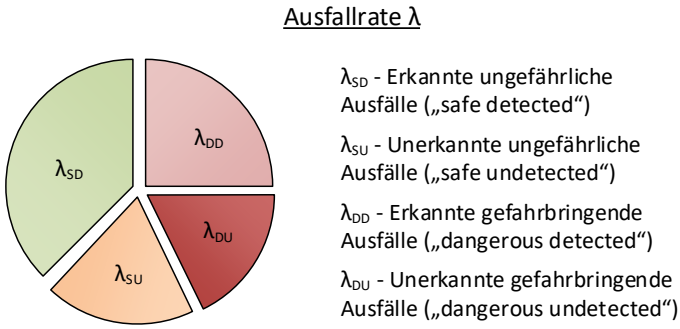


Abbildung 2.7: Zusammensetzung der Ausfallrate λ

$$\lambda_S = \lambda_{SD} + \lambda_{SU} \tag{2.1}$$

Analog können die gefährlichen Ausfälle berechnet werden:

$$\lambda_D = \lambda_{DD} + \lambda_{DU} \tag{2.2}$$

Die SFF kann darauf basierend mittels der Formel (2.3) exakt berechnet werden (vgl. [7]).

$$SFF = \frac{\Sigma\lambda_S + \Sigma\lambda_{DD}}{\Sigma\lambda_S + \Sigma\lambda_{DD} + \Sigma\lambda_{DU}} = \frac{\Sigma\lambda - \Sigma\lambda_D + \Sigma\lambda_{DD}}{\Sigma\lambda} \tag{2.3}$$

Tabelle 2.2: Safe Failure Fraction und zugehöriges SIL nach [7]

Safe Failure Fraction (SFF)		HW-Fehlertoleranz		
Typ A	Typ B	$N = 0$	$N = 1$	$N = 2$
–	0% bis 60%	–	SIL 1	SIL 2
0% bis 60%	60% bis 90%	SIL 1	SIL 2	SIL 3
60% bis 90%	90% bis 99%	SIL 2	SIL 3	SIL 4
$\geq$ 90%	$\geq$ 99%	SIL 3	SIL 4	SIL 4

2.2.3 Funktionale Sicherheit in Personenkraftwagen

Bei der ISO26262 [6] handelt es sich um eine auf Personenkraftwagen (PKW) bezogene spezifische Norm. Diese Norm setzt sich mit der Entwicklung von elektrischen-elektronischen Systemen im PKW auseinander. Es handelt sich hierbei um eine Spezialisierung der bereits beschriebenen IEC61508 auf den Kraftfahrzeugbereich, speziell PKW. Aufgrund von Unsicherheiten bei der Anwendung der IEC61508 im Bezug auf Kraftfahrzeuge ist diese Ableitung und Spezialisierung entstanden. Im Vergleich zur IEC61508 werden hier Modifizierungen vorgestellt, die speziell für die Serienproduktion von Fahrzeugen ausgelegt sind. So wird beispielsweise die Sicherheits-Gesamtvalidierung, welche die IEC61508 für die Inbetriebnahme vorschreibt, durch das Product-Release ersetzt. Durch die große Stückzahl bei der Produktion wäre eine solche Validierung nicht für jedes einzelne Fahrzeug möglich. Darüber hinaus bedient man sich im Automobilsektor nicht ausschließlich der Redundanz, die beispielsweise vermehrt in der Avionik angewendet wird, da hier die Kosten bei der Produktion einiger hunderttausend Fahrzeuge enorm ansteigen würden. Auch die große Anzahl an vernetzten Steuergeräten wird in der IEC61508 nicht adressiert. Es gibt keine Hinweise zum Vorgehen bei Entwicklung und Test solcher Systeme (vgl. [7, 12, 13, 14, 15, 16]). Um jedoch im Bereich der Personenkraftwagen eine Einstufung der Systeme vornehmen zu können, bietet die ISO26262 eine Abwandlung der SIL. Hierbei handelt es sich um die Automotive Safety Integrity Level (ASIL). Im Automobilbereich kann davon ausgegangen werden, dass die vorhandenen Systeme nie SIL 4 erreichen, da in IEC61508 die Einstufung SIL4 Gefährdungen vorbehalten ist, die eine Katastrophe mit vielen Toten zur Folge haben. Die meisten sicherheitskritischen Systeme in diesem Bereich sind zwischen SIL 2 und SIL 3 angesiedelt. Da hier die Zuordnung zu einem der beiden Level sehr schwer fällt, führt die Einführung der ASIL zu einer Zuordnung, bei der ASIL C zwischen SIL 2 und SIL 3 der IEC61508 liegt. Weitere Zuordnungen, Anmerkungen sowie Informationen dazu können Tabelle 2.3 und [7] entnommen werden.

Im Automobilsektor herrscht die Situation, dass nach der Auslieferung ein Fahrzeug nicht mehr unter der Kontrolle des Herstellers steht. Somit können zufällige Hardwareausfälle nicht systematisch erfasst

Tabelle 2.3: Vergleich SIL der IEC61508 mit ASIL der ISO26262 nach [7]

DIN EN 61508	ISO 26262	Anmerkungen
	QM	
SIL 1	ASIL A	
SIL 2	ASIL B	
-	ASIL C	Entwurfsanforderungen in etwa wie SIL 2. Verifikationsanforderungen in etwa SIL 3.
SIL 3	ASIL D	Ein ASIL-D-System ist ein SIL-3-System, nicht aber umgekehrt.
SIL 4		In Automotive keine elektronischen Systeme mit SIL-4-Anforderungen.

Dies ist nur eine Leitlinie. Dennoch müssen die ASIL-x-Anforderungen erfüllt sein, wenn ein als SIL y entwickeltes System in einer ASIL-x-Anwendung eingesetzt werden soll, und umgekehrt.

und analysiert oder erst gar nicht bekannt werden. Aus den vorab genannten Gründen wird in ISO26262 ein stärker qualitativ orientierter Weg gegangen. Es wird verstärkt auf die Vermeidung von Fehlern im Produktentstehungsprozess geachtet. So beispielsweise in der Vermeidung von Fehlern in der Spezifikation, der Konstruktion der Hardware und bei Tests. Für den gesamten Lebenszyklus eines Fahrzeugs (von der ersten Idee bis zur Verschrottung) muss ein Lebenszyklusmodell zugrunde gelegt werden. Im Falle der hier diskutierten Norm liegt das V-Model zugrunde. Hierbei handelt es sich um ein sehr geläufiges Model in diesem Sektor. Auf der linken Seite des „V“ stehen die Erstellung der Anforderungen und das Design, während auf der rechten Seite die Integration, Tests und die Validierung aufgeführt sind (siehe Abbildung 2.8). Es wird deutlich, dass die Entwicklung in einem iterativen Prozess stattfindet. Einzelne Ergebnisse müssen immer wieder gegen die entsprechenden Spezifikationen geprüft werden.

2.2.4 Funktionale Sicherheit in der Avionik

Sicherheitskritischen Systemen in der Avionik muss ebenfalls die IEC61508 zugrunde gelegt werden. Hierbei liegt aber das Augenmerk auf der Aufrechterhaltung der sicherheitskritischen Funktionen im Redundanzprinzip. Bei der Anwendung von Redundanz ist darauf zu achten, dass die redundant eingesetzten Systeme nicht vom glei-

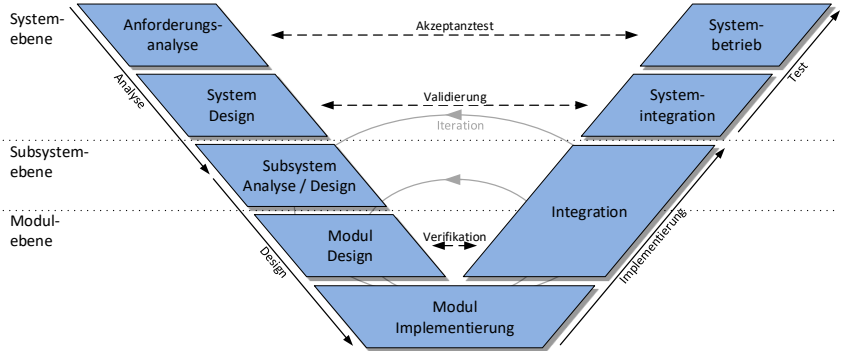


Abbildung 2.8: Darstellung des V-Modells

chen Hersteller kommen, da aufgrund von Fehlern in der Soft- bzw. Hardware sonst die redundanten Systeme gleichzeitig ausfallen könnten. Hierfür werden Produkte unterschiedlicher Hersteller mit einer gewissen Diversifikation verwendet, um die gemeinsame Ausfallwahrscheinlichkeit zu minimieren. Häufig eingesetzte Strategien in der Luftfahrt sind beispielsweise Duo-Duplex-Redundanz, Vierfachrechner, Triplex-Triplex-Redundanz (vgl. Kapitel 3.1 und [17, 18]).

Tabelle 2.4: Design Assurance Level in der Avionik nach [19]

Software Level (DAL)	Aircraft Level Criticality	Meaning
A	Catastrophic	Aircraft destroyed, many fatalities
B	Hazardous	Damage to aircraft, Crew overextended, occupants hurt, some fatal
C	Major	Large reduction in safety margins, occupants injury
D	Minor	Little effect on operation of aircraft and crew workload
E	No effect	No effect on operation of aircraft or crew workload

Um in der Avionik Hardware zu entwickeln, gibt es die Richtlinie DO-254 [20]. Hierbei handelt es sich um einen de-facto Standard, also

um eine von allen Luftfahrtbehörden wie beispielsweise der Federal Aviation Administration (FAA) angewendete Richtlinie (vgl. [21, 22]). Diese Richtlinie beschreibt den Lebenszyklus der Hardwareentwicklung und die zu den jeweiligen Phasen gehörigen Aktivitäten und Objekte. In DO-254 sind Reviews in regelmäßigen Abständen sowohl innerhalb des Projektteams als auch mit den Zertifizierungsbehörden vorgesehen. Vergleichbar mit den SIL der IEC61508 werden in DO-254 verschiedene „Design Assurance Level“ (DAL A - DAL E) definiert, die das Pendant zu den SIL darstellen. Hierbei resultiert bei einem Ausfall eines Systems mit DAL A eine Katastrophe, während DAL E einem System zugeordnet wird, welches bei Ausfall keine Auswirkung auf die Sicherheit eines Flugzeuges hat. Einige beispielhafte Erläuterungen zu den einzelnen Levels können der Tabelle 2.4 entnommen werden. Der Einfluss des DO-254 geht vom Flugzeughersteller (OEM) bis hin zu den Halbleiterherstellern. Sämtliche verwendeten Komponenten müssen über dem DO-254 entsprechende Artefakte und Dokumentationen verfügen (vgl. [22]).

2.2.5 Hardwareentwicklung nach IEC61508

Um im Zuge der Entwicklung eines E/E/PES für eine sicherheitskritische Anwendung ein ASIC zu entwickeln, muss auch hier der Sicherheitslebenszyklus basierend auf dem V-Modell angewendet werden (siehe Abbildung 2.9). Dieser umfasst alle Schritte von der Spezifikation bis hin zur Validierung. Es wird festgelegt, welche Maßnahmen direkt im ASIC realisiert oder der Hardware- bzw. Softwarearchitektur zugewiesen werden. Beispiele für Funktionen, die dem ASIC direkt zugewiesen werden können, sind RAM-Tests, Watchdog oder die Überwachung der Spannungsversorgung (vgl. [7]). Um die im ASIC vorhandenen Funktionen redundant auslegen zu können, müssen spezielle Anforderungen respektiert werden. Hierbei ist beispielsweise zu beachten, dass für jedes Element, das eine Sicherheitsfunktion ausführt, ein separater Block auf dem Substrat vorhanden sein muss. Weitere Anforderungen können direkt aus der Norm entnommen werden.

Um den Anteil an Ausfällen aufgrund von gemeinsamen Ursachen abschätzen zu können, führt die Norm das β_{ASIC}- Modell ein. Der β-Faktor kann mithilfe von Tabellenkalkulationen ermittelt werden und

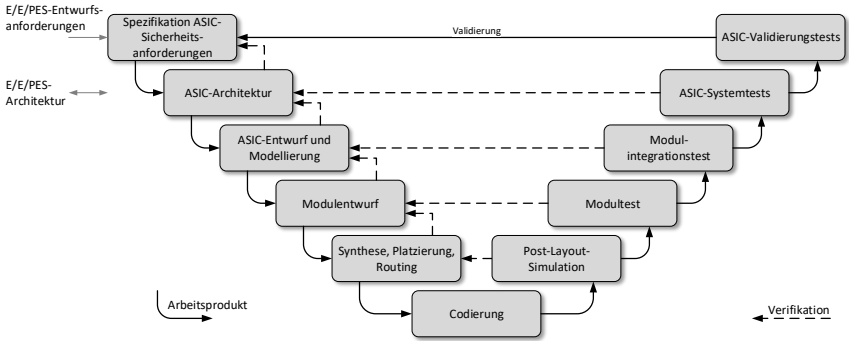


Abbildung 2.9: ASIC Entwicklungszyklus nach IEC61508 in Darstellung analog zu [7]

muss eine bestimmte Vorgabe aus der Norm erfüllen. Dieser Faktor spiegelt die gemeinsamen Fehlerursachen wider. Weiteres kann aus [7] entnommen werden. Gemeinsame Fehlerursachen können Hotspots, gemeinsame Spannungsversorgung oder ähnliches sein. Um Ausfälle zu vermeiden, gibt die Norm verschiedene Empfehlungen an die Entwicklung von ASICs und anwenderprogrammierbaren ICs. Unter anderem zählen hierzu die Verwendung von betriebsbewährten Werkzeugen, Bibliotheken und Produktionsverfahren (mind. zwei Jahre in Projekten eingesetzt), ausführliche Dokumentationen von Simulationsergebnissen und die Verifizierung aller erzielten Ergebnisse. Je nach SIL gibt es unterschiedliche Empfehlungen über anzuwendende Verfahren und Maßnahmen, um systematische Fehler in der Entwicklung zu vermeiden. Als Beispiel aus der Praxis, wird hier auf [7] Kapitel 8.10 verwiesen. Dabei wird deutlich, dass die On-Chip Redundanz für Mikroprozessoren eine notwendige Architektur darstellt. Die Chiphersteller müssen den Erfüllungsgrad des β_{ASIC}- Modells abschätzen und nachweisen.

2.2.6 Softwareentwicklung

Um eine Software für ein sicherheitsrelevantes System zu entwickeln, gibt die IEC61508 ein genaues Vorgehen vor. Hierbei handelt es sich um die Erstellung eines Sicherheitslebenszyklus, der einem V-Modell entspricht. Anhand dieses Vorgehens wird eine Software entwickelt, die nach erfolgtem Validierungstest mit der Hardware integriert werden kann. In erster Linie geht es bei der Spezifikation der Software darum, die Sicherheitsanforderungen festzulegen. Dies geschieht mit Hilfe der zuvor erstellten E/E/PES Sicherheitsspezifikationen und der daraus resultierenden Architektur. Weiterhin sind für die Software verschiedene andere Qualitätsmerkmale wichtig. Hierzu zählen beispielsweise die Testbarkeit, Dokumentation, Eindeutigkeit und die Verständlichkeit. Darüber hinaus müssen die Software- Sicherheitsanforderungen dem entsprechenden SIL angemessen sein. Hierzu zählt unter anderem die Verwendung von rechnergestützten Spezifikationswerkzeugen. Aus der IEC61508 kann aus verschiedenen Tabellen entnommen werden, welche Verfahren bei einem bestimmten SIL anzuwenden sind. Für ein SIL 3 System kann beispielsweise eine Kombination aus Ablauf- und Zustandsübergangsdiagrammen angewandt werden. Diese Art der Diagramme lässt sich mithilfe von rechnergestützten Werkzeugen in UML darstellen. Wird in einer Softwarearchitektur sowohl eine sicherheitskritische als auch eine nicht sicherheitskritische Funktion realisiert, so zählt die komplette Softwarearchitektur als sicherheitskritisch und wird so behandelt. Für jedes Softwareelement muss laut IEC61508 ein Sicherheitshandbuch erstellt werden (vgl. [23]). Hierdurch wird sichergestellt, dass einzelne Softwareelemente wiederverwendet werden können. Es ist darauf zu achten, dass Schnittstellen und Wechselwirkungen zwischen Softwareelementen ausreichend geprüft werden. Um eine Fehlertoleranz in der Software zu erreichen, gibt es verschiedene Techniken. Ein Beispiel hierfür ist die Verwendung von fehlererkennenden/fehlerkorrigierenden Codes bei der Übertragung von kritischen Daten. Für die Software ist ein weiterer entscheidender Aspekt die Testbarkeit. Es ist sicherzustellen, dass die Software gegen die Anforderungen wie z.B. Fehlertoleranz und Integration im System getestet werden kann. Nicht nur die Auswahl der Architektur, sondern auch die Auswahl geeigneter Tools zur Softwareentwicklung und die Programmiersprache sind entsprechend des angestrebten SIL

zu wählen. Um einen Compiler für eine sicherheitskritische Software verwenden zu können, muss dieser zertifiziert oder betriebsbewährt sein. Hieraus entsteht eine geschlossene Kette von Entwicklungswerkzeugen. Um die Komplexität zu begrenzen, bedient man sich der Modularisierung (vgl. [24]). Dies vermeidet gleichzeitig die Fehleranfälligkeit beim Entwurf. Eine weitere Empfehlung der Norm liegt in der defensiven Programmierung. Aufgrund der Anwendung dieser Empfehlung kontrolliert sich die Software selbst auf Datenplausibilität oder ähnliches und wird damit fehlertolerant. Durch die Verringerung der Komplexität leidet meistens die Effizienz. Es können längere Durchlaufzeiten und ein erhöhter Speicherbedarf entstehen. Im Fall von sicherheitskritischer Software hat aber die funktionale Sicherheit Vorrang. Um die Software nach und nach aufbauen zu können, werden zunächst die einzelnen Module den entsprechenden Tests unterzogen. Sind alle Testfälle als bestanden akzeptiert, so kann das Zusammenfügen von Modulen zu Teilsystemen geschehen. Bei Fehlschlagen der Tests, müssen die Implementierung bzw. die Architektur geändert, die Modultests angepasst und erneut getestet werden. Dieses Vorgehen ist bei den Integrationstests der Softwaremodule zu wiederholen. Im Anschluss erfolgt die Integration der Software auf der programmierbaren Hardware, um deren Verträglichkeit zu prüfen. Bei der Validierung wird nun die gesamte Software gegen die Spezifikation der Sicherheitsanforderungen geprüft. Zum vollständigen Lebenszyklus einer sicherheitskritischen Software gehört darüber hinaus noch das Konfigurationsmanagement. Hier wird strikt dokumentiert, was, aus welchem Grund, von wem an der Software verändert wurde. Um in redundanten Systemen eine weitere Sicherheit zu erhalten, bedient man sich, abgesehen vom Sicherheitslebenszyklus der Software, auch des N-Version-Programming. Aufgrund der möglichen, noch in der Software vorhandenen Fehler, wird dieselbe Software von verschiedenen Teams (oder auch von anderen Lieferanten) erstellt.

2.2.7 Ergänzende Regularien

Nachdem die beschriebenen Standards nur gewisse Vorgehensweisen definieren und noch weiteren Spielraum für Interpretationen offenlassen, gibt es ergänzende Dokumente, die zur Klarstellung dienen. Hier

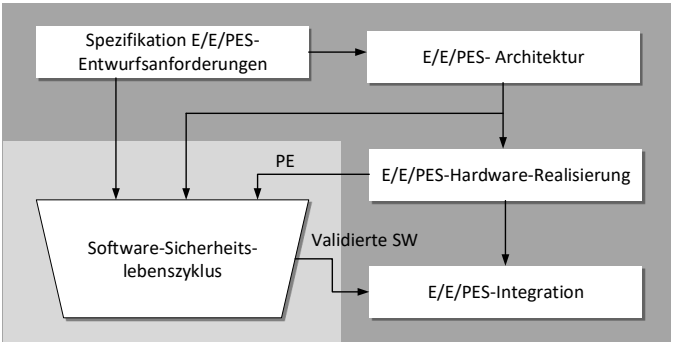


Abbildung 2.10: Übersicht und Integration E/E/PE-System nach [7]

wird nicht ausführlich auf diese ergänzenden Materialien eingegangen, sie werden der Vollständigkeit halber erwähnt.

CAST32 & CAST32A: beim CAST-32A [25] handelt es sich um ein Positionspapier des *Certification Authorities Software Team (CAST)*, zur Klarstellung der Haltung der Zertifizierungsbehörden bezüglich des Einsatzes von Multicoreprozessoren in der Avionik. Dabei werden verschiedene Aspekte, die die funktionale Sicherheit, Rechenleistung und Robustheit von Software auf Mehrkernprozessoren betreffen diskutiert. Zu jedem dieser identifizierten Aspekte stellt das Papier eine Begründung bereit, die erklärt, warum diese Aspekte besonders betrachtet werden müssen.

Certification Memorandum: das Certification Memorandum [26, 27] der EASA stellt eine Interpretationshilfe für die Standards der Luftfahrt dar. Unter anderen werden verschiedene Abschnitte und Aspekte aus der DO-254 klargestellt und präzisiert. Ebenso adressiert dieses Memorandum die für die Zertifizierung relevanten Aspekte. Speziell die Verwendung von Hardware in sicherheitskritischen Anwendungen wird diskutiert. Dies umfasst den Einsatz von Standardhardware (Commercial of the shelf COTS) und deren Einordnung in die Kategorien *Einfach - Komplex - Sehr Komplex*. Entscheidend hierbei ist, dass die Verwendung von Mikrocontrollern, und damit auch Mehrkernprozessoren, immer als sehr komplex einzustufen ist. Dies resultiert in weiteren Anforderungen für die Entwicklung von Systemen basierend

auf solchen Rechnern. Details können direkt dem Certification Memorandum [26, 27] entnommen werden.

2.3 Prozessortechnologie

Die vergangenen Jahre zeigen eine erhebliche Änderung und technologische Weiterentwicklung in der Prozessortechnologie. Die eingesetzten Prozessoren in eingebetteten Systemen werden immer leistungsfähiger. Ausgehend von Einkernprozessoren (Singlecore) wird auch in eingebetteten Systemen der Trend zu Mehrkernprozessoren (Multicore) zunehmend deutlich. Aus diesem Grund und zur besseren Darstellung der Unterschiede und Herausforderungen, die durch den Umstieg auf Mehrkernprozessoren entstehen, werden die Prozessorarchitekturen nachfolgend grob erläutert. Für detailliertere Beschreibungen sowie Erklärungen wird beispielsweise auf [28, 29, 30] verwiesen.

2.3.1 Singlecoreprozessoren

Die nach wie vor am meisten eingesetzten Prozessorarchitekturen in eingebetteten Systemen sind Einkernprozessoren. Bei diesen Architekturen handelt es sich um Prozessoren, die lediglich einen Rechenkern (Prozessorkern) beinhalten. Weiterhin vorhanden sind typischerweise Peripheriekomponenten, Bus-Systeme sowie Speicher. Diese Prozessorarchitekturen reichen von einfachen 8Bit bis hin zu leistungsfähigen 64Bit Prozessoren.

Für den detaillierten Aufbau eines einfachen Rechenkerns kann Abb. 2.11 herangezogen werden. Hierbei sind die wichtigen Bestandteile des Rechenkerns das Steuerwerk (Steuerungsaufgaben, Befehlsregister - BR, Register für Steuerbits - ST, Befehlszähler - BZ), das Speicherwerk (Adresszähler - AR, Speicher - SP, Speicherregister - SR), das Rechenwerk (Arithmetische-logische Einheit - ALU, Akkumulator - AK) und das E/A-Werk (Eingabe/Ausgabe - E/A).

Bei der Architektur unterscheidet man typischerweise zwischen zwei unterschiedlichen Schemata. Bei der Von Neumann-Architektur (vgl.

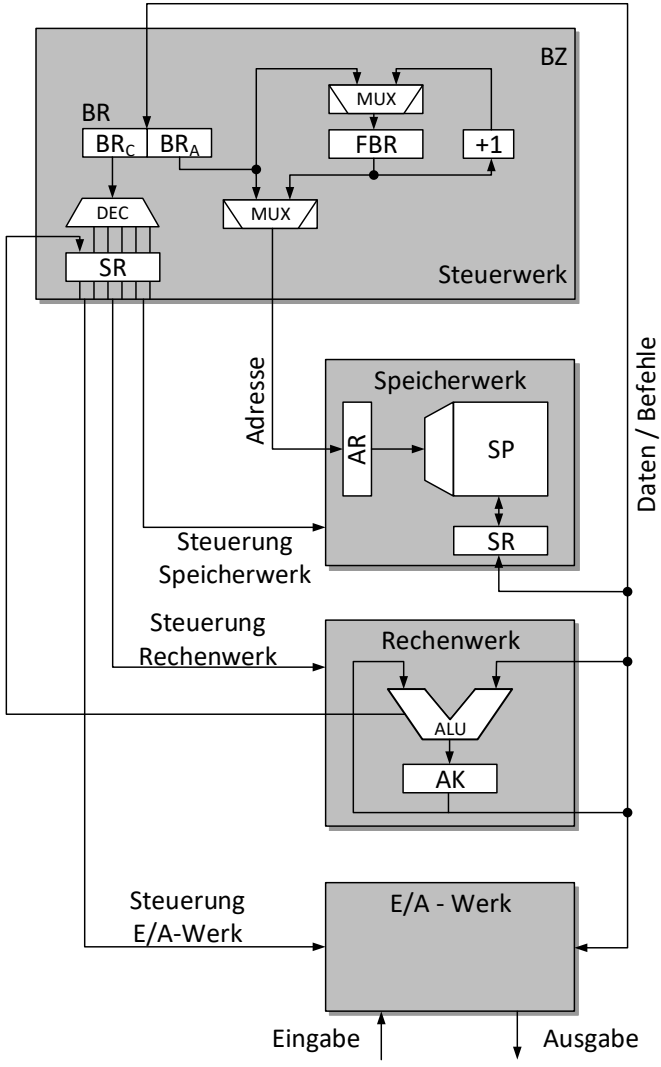


Abbildung 2.11: Aufbau eines einfachen Prozessorkerns

Abb. 2.12, *1.)*) wird nur ein Speicher an den Prozessorkern angebunden, der sowohl Daten als auch Befehle beinhaltet. Dies hat jedoch starke Einschränkungen was die Zugriffe auf den Speicher angeht. Die zweite weit verbreitete Architektur ist die Harvard-Architektur (vgl. Abb. 2.12, *2.)*). Hierbei werden zwei Speicher an den Rechenkern angebunden, ein Speicher für Daten und ein Speicher für Adressen. Dies lässt erheblich bessere Zugriffe auf die Speicher zu und ist daher die heutzutage am weitesten verbreitete Architektur. Beide Architekturen sind in Abbildung 2.12 dargestellt. Der Aufbau zeigt sich beispielsweise in der Cache-Architektur in Singlecoreprozessoren, speziell in der Trennung von Daten- (D-Cache) und Instructioncache (I-Cache).

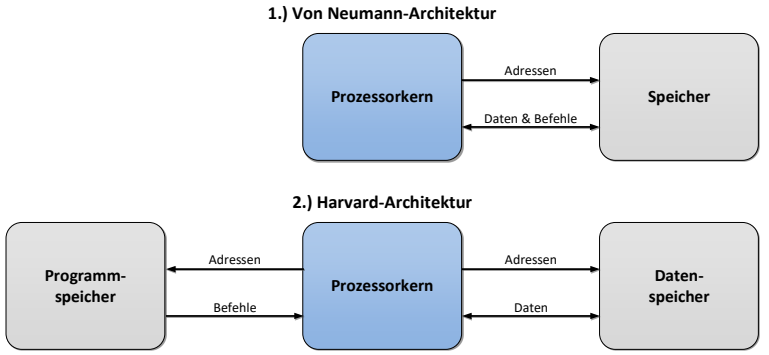


Abbildung 2.12: Referenzmodelle für Computerarchitekturen

Auf Prozessorebene bzw. System-on-Chip (SoC) Level werden weitere Komponenten integriert. Zusätzlich zum eigentlichen Rechenkern werden Peripheriekomponenten (z.B. Kommunikationsschnittstellen - UART, Ethernet, I2C), On-Chip Speicher (z.B. Caches) sowie Kommunikationsstrukturen (z.B. Bussysteme, Cross-Bars) integriert. Je nach Hersteller und Einsatzzweck werden weitere unterschiedliche Ausprägungen und Komponenten verwendet. Eine Schematische Darstellung einer abstrahierten (SoC) Architektur ist in Abbildung 2.13 zu finden.

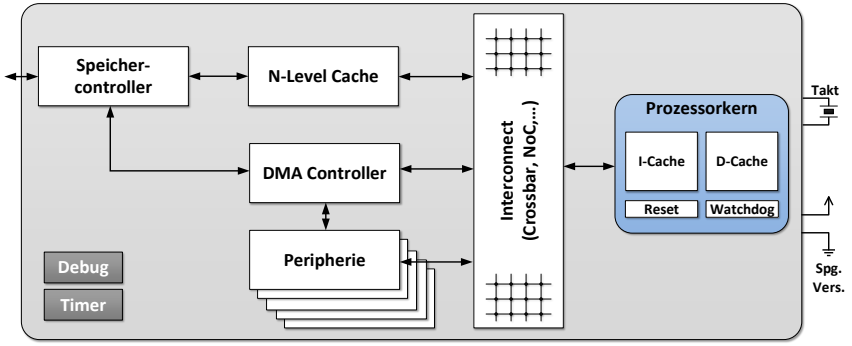


Abbildung 2.13: Abstrahierte und beispielhafte Singlecoreprozessorarchitektur

Zu den Peripheriekomponenten, die typischerweise in einen Prozessor integriert werden und im weiteren Verlauf dieser Arbeit relevant sind, gehören Direct Memory Access (DMA)-Controller und Watchdogs:

- **Direct Memory Access (DMA)-Controller:**

  Bei einem DMA-Controller handelt es sich um eine Hardwarekomponente innerhalb eines System-on-Chip (SoC), die zum Ziel hat, die Ausführungsgeschwindigkeit zu erhöhen. Dies wird erreicht indem die Kommunikation der Rechenkerne mit den E/A-Geräten minimiert wird. Ein DMA-Controller übernimmt die Übertragung von Daten im Speicher an eine andere Position im Speicher oder zu Peripherals. Durch diese Entlastung der Rechenkerne wird die Geschwindigkeit der Ausführung der Rechnerarchitektur beschleunigt. DMA-Controller werden dabei nicht nur in Singlecoreprozessoren eingesetzt, sie sind auch in Multicoreprozessoren zu finden.

- **Watchdog:**

  Um die zeitliche Abfolge und die korrekte Funktionsweise eines Prozessors zu überwachen, werden Watchdogs eingesetzt. Dabei handelt es sich um eine Einheit, die bei nicht erfolgter Rücksetzung einen Reset des Systems auslösen oder zumindest ein Fehl-

verhalten signalisieren kann. Der Watchdog wird von der Software, die auf dem Prozessor ausgeführt wird, zurückgesetzt.

Eine der für diese Arbeit wesentlichen Eigenschaften eines Singlecoreprozessors ist die rein sequentielle Ausführung von Software. Aufgrund der nur einfach verfügbaren Recheneinheit (Prozessorkern) ist diese Limitierung der Ausführung bedingt. Dies wiederum bewirkt einen rein sequentiellen Zugriff auf beispielsweise Peripheriekomponenten oder Speicher. Etwaige Überschneidungen durch zeitgleiche, parallele Zugriffe von unterschiedlichen Ausführungseinheiten sind ausgeschlossen.

Da aufgrund thermischer Eigenschaften und Limitierungen in der Skalierung von Transistoren eine Erhöhung der Rechenleistung nicht mehr möglich ist, werden neue Ansätze entwickelt. In aktuellen Entwicklungen in der Prozessortechnologie werden zur Erhöhung der Rechenleistung weitere Rechenkerne in einen Prozessor integriert. Eine Beschreibung der Eigenschaften und des schematischen Aufbaus folgt in Kapitel 2.3.2.

2.3.2 Multicoreprozessoren

Zur Realisierung von höheren Rechenleistungen für eingebettete Systeme sind Singlecoreprozessoren nicht mehr ausreichend. Technologisch, speziell aus thermischen Gründen, ist eine Erhöhung der Taktfrequenz nicht mehr möglich. Eine Steigerung der Rechenleistung kann somit nur noch durch eine Umsetzung von Parallelität erreicht werden. Dies geschieht durch die Integration weiterer Prozessorkerne in einem Chip. Die resultierenden Multiprozessor System-on-Chip (MPSoC) bieten durch die parallele Ausführung von Software eine entsprechende Steigerung an Rechenleistung. Damit gehen jedoch auch einige Herausforderungen einher, die besonders beachtet werden müssen. Hierzu gehören beispielsweise zeitgleiche Zugriffe auf gemeinsam genutzte Ressourcen, z.B. Peripheriekomponenten oder Speicher. Die echte Parallelität in Mehrkernprozessoren bildet somit den Hauptunterschied zu Einkernprozessoren und den Ausgangspunkt für weitere Anforderungen für deren Einsatz in sicherheitskritischen Anwendungen. In

Kapitel 2.3.2.1 und 2.3.2.2 werden zwei mögliche Realisierungen von Mehrkernprozessoren dargestellt und die Unterschiede diskutiert.

2.3.2.1 Homogene MCP

Ein homogener Mehrkernprozessor enthält nur Kerne vom selben Typ. Dies bedeutet, alle Rechenkerne haben die gleiche Mikroarchitektur und auch den gleichen Instruktionssatz. Zusätzlich zur Integration mehrerer Kerne werden, analog zu Singlecoreprozessoren, zum Beispiel Peripheriekomponenten und Speicher für das SoC eingebracht. Peripheriekomponenten und Speicher werden jedoch in der Regel nicht mehrfach instanziiert. Typischerweise werden diese Komponenten nur einfach im SoC vorgehalten und nicht vervielfältigt. Der schematische Aufbau ist in Abbildung 2.14 dargestellt. Speziell bei Komponenten, die von allen integrierten Prozessorkernen verwendet werden, wie zum Beispiel Speichercontroller, kann ein zeitgleicher Zugriff entstehen, der bei Singlecoreprozessoren aufgrund der Architektur und der sequentiellen Ausführung nicht möglich war. Bei der zeitgleichen Ausführung von Software auf mehreren Kernen in einem MPSoC können parallel mehrere Zugriffe auf eine gemeinsam genutzte Ressource entstehen.

2.3.2.2 Heterogene MCP

Im Gegensatz zu homogenen Mehrkernprozessoren (vgl. Kapitel 2.3.2.1) sind in heterogenen Mehrkernprozessoren verschiedene Arten von Prozessorkernen enthalten. Hierbei kann es sich um Prozessorkerne mit unterschiedlichen Mikroarchitekturen oder auch um spezielle Beschleuniger handeln. Durch einen solchen Aufbau können weitere Prozessorkerne im System vorhanden sein, die ebenfalls zeitgleich auf eine nur einfach vorhandene Komponente im System zugreifen. Ebenfalls verfügbar sind Architekturen, die zusätzlich zu einem Mehrkernprozessor eine rekonfigurierbare Logikarchitektur integrieren. Bei dieser rekonfigurierbaren Logik handelt es sich typischerweise um eine Field Programmable Gate Array (FPGA) Architektur (vgl. [31]), die über eine sehr enge Anbindung zur internen Kommunikationsarchitektur des Mehrkernprozessors verfügt. So geartete Architekturen bieten einen

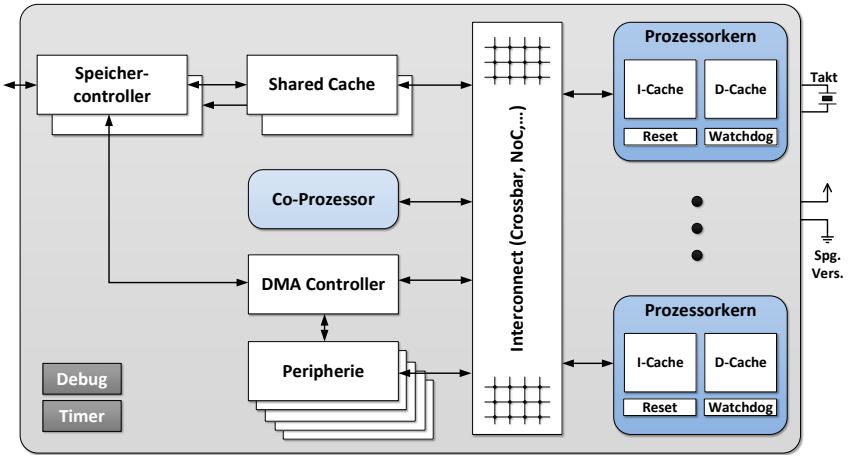


Abbildung 2.14: Abstrahierte und beispielhafte Multicoreprozessorarchitektur

wesentlich erweiterten Entwurfsraum und somit Möglichkeiten zur Integration weiterer Hardwarekomponenten, die in einem Standard Prozessor (als ASIC vorhanden) nicht möglich sind. Beispielsweise können HW-Beschleuniger für spezielle Algorithmen integriert werden, die eine Beschleunigung der Berechnung und eine Auslagerung vom Mehrkernprozessor erlauben. Der abstrahierte, schematische Aufbau mit den potentiell vorhandenen Komponenten und einer rekonfigurierbaren Hardwarearchitektur sind in Abbildung 2.15 dargestellt. Wie abgebildet, besitzt die rekonfigurierbare Logik eine direkte Anbindung zur internen Kommunikationsstruktur und somit ebenfalls zu den verfügbaren Peripheriekomponenten, Speichercontrollern etc.

Die Verfügbarkeit einer vom Nutzer konfigurierbaren HW-Logik bringt zusätzlich eine erhebliche Flexibilität mit sich. Beliebige, ebenfalls parallellaufende Prozesse oder State-Machines können integriert werden und direkt mit den Prozessorkernen interagieren. Die in der rekonfigurierbaren Hardware umgesetzten Komponenten können dabei so implementiert werden, dass sie keine Nachteile bzgl. des Deter-

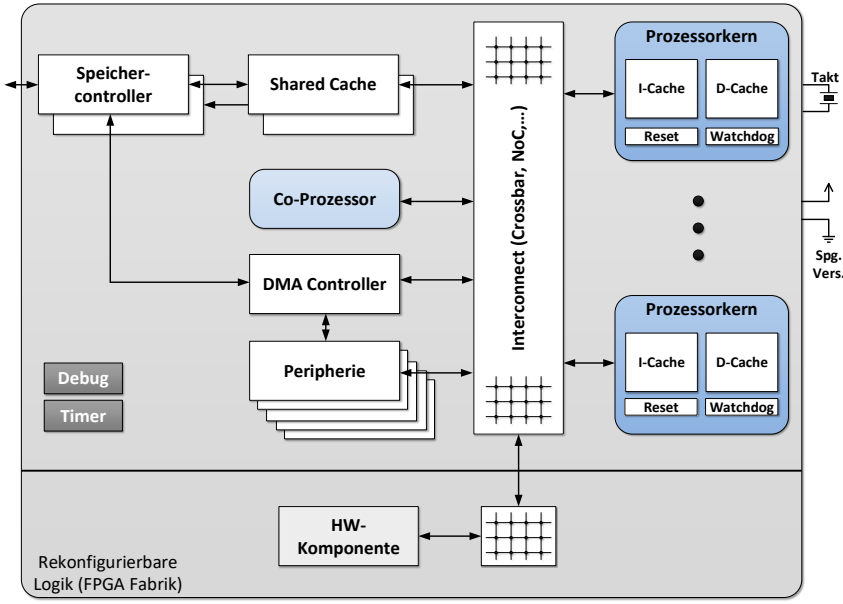


Abbildung 2.15: Abstrahierte und beispielhafte heterogene Multicore-prozessorarchitektur mit integrierter rekonfigurierbarer Logik

minismus darstellen und eine wiederholbare Ausführung garantieren können.

2.4 Virtualisierung

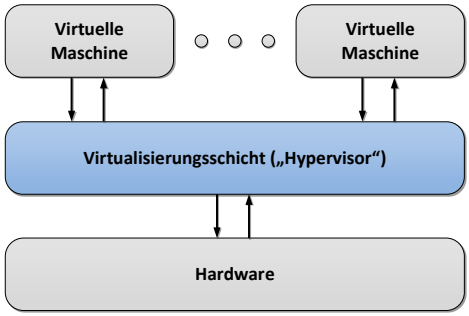


Abbildung 2.16: Darstellung des zusätzlichen Layers bei Virtualisierung

Virtualisierung ist ein sehr breites Feld. Im Kontext dieser Arbeit wird Virtualisierung als System Virtualisierung, im Speziellen als Hardware Virtualisierung, verstanden. Unter Virtualisierung versteht man das Nachbilden einer Hardware- oder Softwarekomponente durch das Einfügen einer zusätzlichen Abstraktionsschicht. Dieses Konzept bietet somit eine Möglichkeit, eine nur einfach im System vorhandene Komponente als vielfach vorhandene virtuelle Komponente darzustellen. Diese virtuellen Komponenten können durch unterschiedliche Nutzer in gleicher Weise wie das physikalische Gerät im nicht-virtualisierten Fall angesteuert werden. Die unterschiedlichen, etablierten Ansätze für Virtualisierung können grob in zwei Kategorien eingeteilt werden - Voll- und Para-Virtualisierung. Beide Konzepte können in Software umgesetzt werden, wenn keine Hardwareunterstützung verfügbar ist. Bei Verfügbarkeit einer Hardwareunterstützung können diese Ansätze jedoch optimiert und somit der Overhead in Software reduziert werden. Unabhängig von der Hardwareunterstützung wird ein neuer Software-Layer, der sog. Virtual Machine Monitor (VMM) oder Hypervisor einge-

führt, der sich um die Überwachung des Systems kümmert. Nachfolgend werden die beiden Konzepte kurz diskutiert, da Virtualisierung im weiteren Verlauf der Arbeit eine Rolle spielt.

2.4.1 Voll-Virtualisierung

Bei der Voll-Virtualisierung handelt es sich um eine Technik, welche es erlaubt, Gast-Betriebssysteme ohne Modifikation auf der Hardware auszuführen. In diesem Fall bildet der Hypervisor eine Abstraktion der darunterliegenden, geteilten Hardware, also eine Simulation der Hardware. Instruktionen des Gast-Betriebssystems werden durch Hardwareunterstützung abgefangen und übersetzt (emuliert durch den Hypervisor), um den gewünschten Effekt auf der virtualisierten Hardware zu erreichen. Auch ohne Hardwareunterstützung kann dies durch dynamische binäre Übersetzung erreicht werden. Der Hypervisor übersetzt alle Befehle des Betriebssystems zur Laufzeit und speichert die Ergebnisse für eine spätere Verwendung zwischen, während User Level Befehle unmodifiziert ausgeführt werden. Der Aufbau ist in Abbildung 2.17 dargestellt.

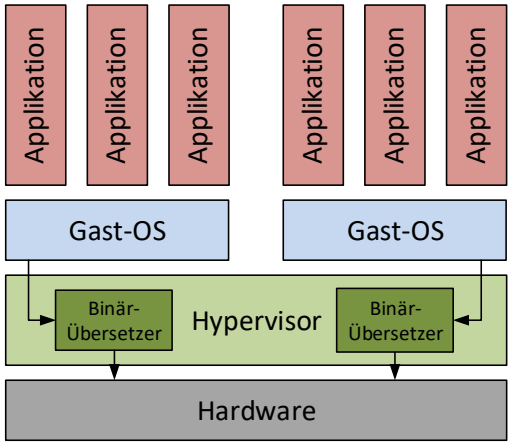


Abbildung 2.17: Aufbau der Voll-Virtualisierung

2.4.2 Para-Virtualisierung

Bei Para-Virtualisierung handelt es sich um ein weiteres softwarebasiertes Virtualisierungskonzept. Hierbei ist der Hauptunterschied zur Voll-Virtualisierung, dass der Code des Betriebssystems, innerhalb einer virtuellen Maschine, explizit modifiziert wird. Dies dient dazu Aufrufe für Befehle, die nicht direkt auf der Hardware ausgeführt werden können, im Hypervisor auszulösen, was in Abbildung 2.18 visualisiert ist. In diesem Fall weiß das Gast-Betriebssystem, dass eine Virtualisierungsschicht existiert. Dieses Konzept kann effizienter als die Voll-Virtualisierung umgesetzt werden, was jedoch auf Kosten der Komplexität bei der Implementierung geht. Speziell zur Modifizierung des Gast-Betriebssystems muss der Source Code verfügbar sein und ist damit nicht immer umsetzbar.

Speziell beim Einsatz von eingebetteten Systemen die den Echtzeitbedingungen genügen müssen, wird häufiger Para-Virtualisierung eingesetzt, vor allem dann, wenn keine Hardwareunterstützung vorhanden ist. Dies ist zumeist bei ressourcenschwächeren Prozessoren, beispielsweise im Automotive-Umfeld, der Fall.

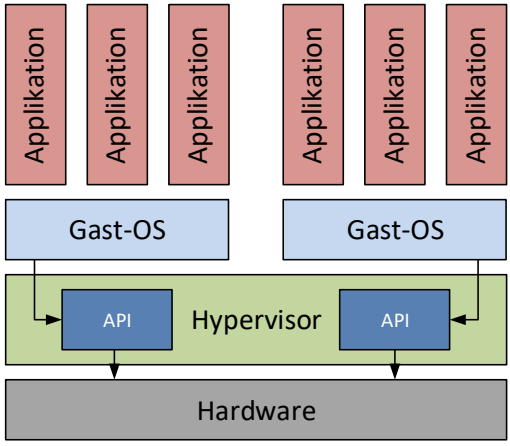


Abbildung 2.18: Aufbau der Para-Virtualisierung

3 Stand der Technik

Schon seit einigen Jahren werden Prozessoren in sicherheitskritischen Systemen eingesetzt. Bislang wurden als Prozessoren hauptsächlich Singlecoreprozessoren aufgrund der Verfügbarkeit verwendet. Zur Absicherung der Funktionalität und zur Steigerung der Zuverlässigkeit werden unterschiedliche Mechanismen zur Überwachung oder Schaffung von Redundanzen eingesetzt. Je nach angestrebter Verfügbarkeit der Systeme werden verschiedene Maßnahmen verwendet. Auch der Trend hin zu virtualisierten Systemen im Bereich eingebetteter Systeme ist zu beobachten. Diese von Datencentern und Desktop Rechnern bekannte Technologie wird vermehrt in kleinen, leistungsschwächeren Prozessoren Anpassungen vorgenommen und evaluiert.

In den folgenden Kapiteln wird der Stand der Technik in den genannten Bereichen zusammengefasst. Dabei liegt der Fokus auf den Themen, die im Zusammenhang mit der vorliegenden Arbeit stehen und für die Erreichung der Ziele relevant sind. Hierfür folgt ein umfassender Überblick über existierende Redundanzkonzepte, Monitoringansätze sowie Virtualisierung im Bereich eingebetteter Systeme.

3.1 Redundanzkonzepte

In aktuellen sicherheitskritischen Anwendungen, welche auf Singlecoreprozessoren basieren, werden bereits Redundanzkonzepte eingesetzt. Viele dieser Architekturen sind u.a. in der Avionik im Einsatz. Diese Redundanzkonzepte sind zum Teil bereits etliche Jahre etabliert.

Eines der verwendeten Konzepte ist die Duo-Duplex-Architektur. Bei der Duo-Duplex-Redundanz handelt es sich um einen Systemaufbau, welcher zwei redundante Systeme verwendet, welche intern

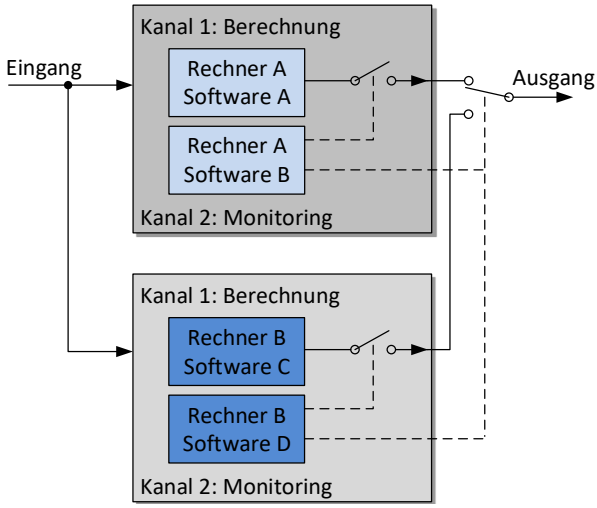


Abbildung 3.1: Darstellung der Duo-Duplex-Architektur nach [18]

eine weitere duale Redundanz aufweisen. Aus Abbildung 3.1 kann der schematische Aufbau entnommen werden. Der in diesem Zusammenhang verwendete Begriff Kanal bezieht sich auf die in der IEC61508 [9] vorhandenen Definition. Hier handelt es sich um ein Element oder eine Gruppe von Elementen, die unabhängig voneinander eine Funktion ausführen (vgl. [9]). Bei der Duo-Duplex-Architektur werden zwei unterschiedliche Softwareversionen (Software A und Software B) auf Rechner A ausgeführt. Während Rechner A mit Software A als operativer Rechner dient, wird Rechner A mit Software B als Monitor eingesetzt. Die Ergebnisse von A und B werden miteinander verglichen. Kommt es zu einer Diskrepanz, so wird das betroffene Teilsystem isoliert und ein Ersatzrechner (Kanal 2) übernimmt dessen Aufgabe. Dieser ist wiederum redundant ausgelegt und basiert auf der Hardware eines anderen Herstellers. Weitere Einzelheiten über den Aufbau können [18] entnommen werden.

In einem Vierfachrechner bedient man sich einem Aufbau aus vier unterschiedlichen Rechnern (vgl. Abbildung 3.2). Jedem Rechner wird ein Voter zur Seite gestellt, der das jeweilige Ergebnis des Rechners verifi-

ziert und mit den Ergebnissen der anderen Rechner vergleicht. Stellt ein Voter eine Abweichung fest, so wird der entsprechende Kanal isoliert und das System kann seine Funktionalität aufrechterhalten. Hier wäre das System selbst bei einem Ausfall von drei Rechnern noch funktionsfähig. Bei Ausfall eines Rechners kann nach wie vor ein Mehrheitsentscheid getroffen werden, falls es zu weiteren Abweichungen kommt. Es muss auch hier beachtet werden, dass gemeinsame Fehlerursachen ausgeschlossen werden. Dabei fällt beispielsweise die Spannungsversorgung auf, die gesondert untersucht werden muss (vgl. [18]).

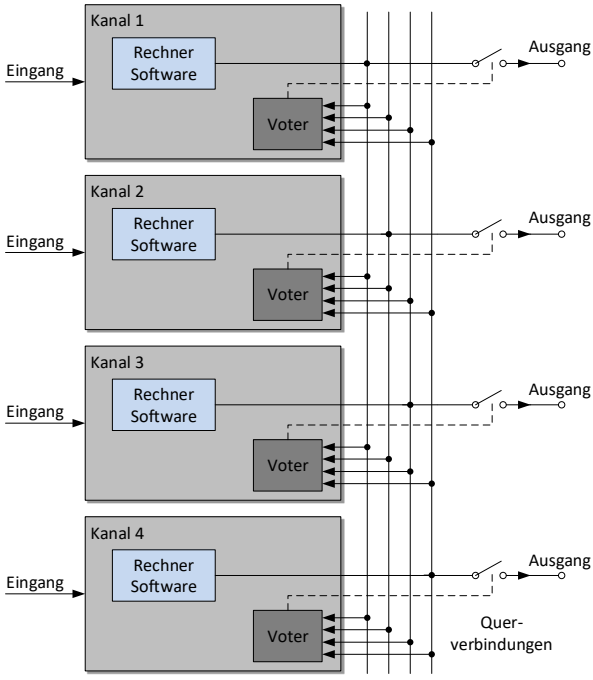


Abbildung 3.2: Darstellung der Quadruplex-Architektur nach [18]

Der Triplex-Triplex-Redundanz-Ansatz kombiniert die beiden bisher vorgestellten Prinzipien. Es werden drei identische Kanäle mit je einer internen Redundanz von drei aufgebaut. In jedem Kanal wertet ein Voter die Ergebnisse der verschiedenen Rechner aus. Dabei ist darauf zu

achten, dass in jedem Kanal wiederum Hard- bzw. Software von unterschiedlichen Herstellern zu verwenden ist. In den jeweiligen Teilkanälen dienen zwei Rechner zur Überwachung eines operativen Rechners. Der Voter vergleicht sowohl die Ergebnisse innerhalb eines Kanals miteinander, als auch die Ergebnisse mit den Ergebnissen der anderen Kanäle. Der Aufbau wird in Abbildung 3.3 verdeutlicht. Aufgrund dieser hohen Redundanz können zwei Kanäle und ein Teilkanal des verbleibenden Kanals ausfallen, ohne dass die Funktion des Systems eingeschränkt wird (vgl. [18]). Gemeinsame Fehlerquellen müssen auch hier untersucht werden.

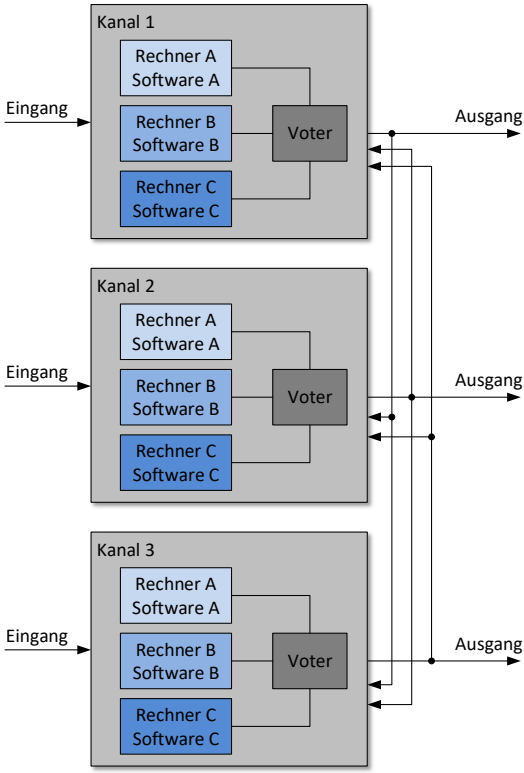


Abbildung 3.3: Darstellung der Triplex-Triplex-Architektur nach [18]

Die vorgestellten Konzepte bedingen alle eine Replikation der Hardware und sind nicht für Multicoreprozessoren ausgeprägt. Speziell die Herausforderungen, die durch die Architektur eines Multicoreprozessors eingebracht werden, werden nicht adressiert. Einzig die Lock-Step Architektur, welche industriell verfügbar ist, besteht aus zwei Rechenkernen, die sehr eng gekoppelt sind und mit einem Taktversatz von 0 - 2 Takten dieselbe Software ausführen und die Ergebnisse mittels einer komplexen Logik vergleichen [32]. Bei einem Rechenkernpaar in Lock-Step Konfiguration handelt es sich um ein „1-out-of-2 (1oo2)" Redundanzschema ggf. mit Diagnose. Hierbei sind die beiden Rechenkerne nicht als Multicoreprozessor zu betrachten, da für die Software und deren Entwickler nur ein funktionaler, logischer Kern sichtbar ist. Es kann mit diesem Aufbau ein Fehler erkannt werden, jedoch kann nicht festgestellt werden, welcher der beiden Kerne den Fehler hervorgerufen hat. Ein Ansatz, der in der Lage ist einen Fehler zu erkennen und zu korrigieren, ist die Durchführung eines Mehrheitsentscheids (z.B. 2-out-of-3 (2oo3) Architektur) über mehrere Rechner hinweg. Hierzu gehört beispielsweise die sog. Triple Modular Redundancy (TMR) [33]. In diesem Ansatz werden drei unabhängige Kanäle ausgewählt, deren Ergebnisse von einem Voter verglichen werden und eine Entscheidung der Korrektheit auf Basis einer Mehrheit getroffen wird. Ein anderer Ansatz diskutiert eine fehlertolerante Struktur für SRAM basierte FPGA Systeme [34, 35].

Auf Software Ebene kann ebenso eine Redundanz eingebracht werden. So wird in [36] eine Prozess-Redundanz vorgeschlagen, welche in Software realisiert und für das Betriebssystem transparent dargestellt wird. Alternativ dazu wird in [37] ein Ansatz zum Coded Processing diskutiert und in [38] ein Ansatz vorgestellt, der energieeffizient Tasks repliziert, um die Zuverlässigkeit zu steigern. Des Weiteren werden Konzepte zum dynamischen Scheduling von Software Komponenten diskutiert, um eine Fehlertoleranz zu erreichen [39]. Ebenso können Redundanzen auf Steuergeräte Ebene betrachtet und mit Softwareunterstützung umgesetzt werden [40]. Ein anderer Ansatz der sich mit kernübergreifender Redundanz bei Hard-Errors[1] befasst wird in [41] diskutiert.

[1] Hierbei handelt es sich um Defekte im Chip des Prozessors, die beispielsweise durch Alterung entstehen und nicht repariert werden können.

3.2 Prozessoren in sicherheitskritischen Anwendungen

Da schon längere Zeit Prozessoren in sicherheitskritischen Anwendungen eingesetzt werden, existieren einige Ansätze sowie spezialisierte Prozessoren. So werden beispielsweise von verschiedenen Herstellern Prozessoren bereitgestellt, die Mechanismen zur Absicherung der Funktionalität mitbringen. In diesem Kontext sind die AURIX Prozessorfamilie [42] von Infineon, die Hercules RM Prozessorfamilie [43] von Texas Instruments oder Prozessoren der MCP5xxx-Serie [44] von NXP zu nennen. Diese beinhalten Maßnahmen, um Fehler erkennen zu können. Hierzu gehören Error Correction Codes (ECC), Lock-Step Prozessoren und Watchdogs [45, 33, 46]. Ebenso werden externe Watchdogs, die beispielsweise über ein Serial Peripheral Interface (SPI) angebunden werden, vorgeschlagen um einen Prozessor zusätzlich überwachen zu können [47, 48]. Diese Prozessoren bieten aber je nach sicherheitskritischer Anwendung nicht mehr ausreichend Rechenleistung. Leistungsfähigere Prozessoren, wie beispielsweise die P4080 Architektur [49] von NXP, bieten keine Mechanismen, die für einen Einsatz in sicherheitskritischen Anwendungen benötigt werden.

Um Fehler in der zeitlichen Ausführung zu erkennen werden Watchdogs in den meisten Prozessoren eingesetzt, so auch in Multicoreprozessoren. Je nach Architektur variiert die Anzahl der Watchdogs, so sind beispielsweise im P4080 [49] jeweils ein unabhängiger Watchdog pro Core, im MCP5746M [44] ein Watchdog für das SoC vorhanden. Diese Watchdogs können Fehler im Timing erkennen, die zum Beispiel aus einer nicht terminierenden Bus-Transaktion oder einer Endlosschleife der Software entstehen. Je nach Architektur können die Maßnahmen, die ergriffen werden, variieren und reichen vom Auslösen eines Interrupts bis hin zum System-Reset. Typischerweise werden die Informationen vom Watchdog direkt verarbeitet und eine Reaktion eingeleitet. Eine übergreifende Einheit, die sich aus Systemsicht um die Fehlerbehandlung kümmert, existiert zumeist nicht.

Zusätzlich zu den beschriebenen Reaktionen eines Watchdogs sind ausgefallenere Mechanismen denkbar. Hierbei wird oftmals in sicherheitskritischen Systemen auf eine Challenge-Response-Abfrage zurück-

gegriffen [45, 32]. Ein einfaches Rücksetzen des Watchdog Timers ist dabei nicht ausreichend. Im MCP5746M von NXP wird beispielsweise die Konfiguration des Watchdogs in dieser Weise vorgenommen. Es muss eine bestimmte Sequenz eingehalten werden, um die entsprechenden Werte setzen zu können. Dieser Mechanismus wird häufig auch für eine Kommunikation mit einer externen Einheit eingesetzt. Für das Rücksetzen des Watchdog Timers von der Software wird dieser Mechanismus jedoch nicht verwendet.

Ein Ansatz, der sich mit der Überwachung eines Prozessors auseinandersetzt und im Automotive-Umfeld häufig eingesetzt wird, ist das E-Gas Konzept [50]. In diesem Konzept werden drei Ebenen der Überwachung vorgestellt. Ebene 1 beinhaltet die eigentliche Funktion, Ebene 2 überwacht Ebene 1 (beispielsweise durch einen Watchdog), während Ebene 3 das gesamte Element (z.B. den Multicoreprozessor) über eine externe Einheit überwacht.

Um eine noch höhere Sicherheit beim Einsatz von Singlecorprozessoren zu erreichen werden verschiedenste Mechanismen und Konzepte angewendet, um ein besseres und detaillierteres Verständnis über den Systemzustand zu erlangen. Diese Mechanismen umfassen Online-Monitoring des Systems sowie den Einsatz von Redundanz um transiente, sporadische und permanente Fehler in digitalen Systemen erkennen und korrigieren zu können [51, 52, 45, 33, 53, 54].

3.3 Monitoringkonzepte

Typische MPSoC Architekturen bringen neue Herausforderungen beim Einsatz in sicherheitskritischen Anwendungen mit sich. Durch immer weiter sinkende Transistorgröße, die steigende Integrationsdichte und durch die niedrigeren Spannungsversorgungen steigt die Soft-Error-Rate in Mehrkernprozessoren [55, 56, 57]. Diese Fehler führen zu einer Abweichung des gewünschten Verhaltens und erfordern somit die Umsetzung von Online-Monitoring. Zusätzlich zu den sog. Soft Errors kommen zeitgleiche Zugriffe auf gemeinsam genutzte Ressourcen innerhalb des Chips. Diese Zugriffe müssen ebenso im Betrieb überwacht werden, um die korrekte Funktion des Systems sicherzustellen.

Zwei entscheidende Charakteristika, die Systeme in sicherheitskritischen Anwendungen erfüllen müssen, sind Determinismus und Vorhersagbarkeit. Standard Mehrkernprozessoren können diese beiden Charakteristika nicht ohne Weiteres erfüllen. Dies resultiert aus sehr leistungsfähigen Cache Architekturen, mehreren Bus Mastern im System und zeitgleichen Zugriffen. Um dem entgegenzuwirken, muss das System mit äußerster Sorgfalt konfiguriert werden. Dies bedeutet im Umkehrschluss, dass nicht die maximal mögliche Performanz des Systems erreicht werden kann. Um auch während des Betriebs sicherstellen zu können, dass die Konfiguration Bestand hat, muss diese Überwacht werden. In vielen Architekturen existieren bereits Möglichkeiten, mit denen ein Online-Monitoring umgesetzt werden könnte, diese sind aber nicht dafür vorgesehen oder erst gar nicht dokumentiert. Einige der Möglichkeiten können nur zu Debugging Zwecken eingesetzt werden [58, 59, 60, 61] und versetzen so das System in einen Zustand, der nicht dem Produktivsystem entspricht. Andere werden zur Optimierung [62, 63, 61] und Evaluation [64] genutzt. Ein weiteres Beispiel ist die Nutzung von Online-Monitoring in Commercial Off The Shelf (COTS) Prozessoren das in [65, 66] vorgestellt wird, aber ausschließlich auf Zugriffe für den Hauptspeicher limitiert ist. Ein weiterer Ansatz zeigt die Verwendung von in den Prozessorkernen integrierten Zählern, die zu einem nicht-intrusiven Monitoring eingesetzt werden können [67, 68], aber nur zur Abschätzung der Worst Case Execution Time (WCET) genutzt werden. Ebenso existieren Ansätze, die ein zusätzliches Monitoring-Subsystem integrieren, um Systemzustände zu evaluieren [69, 70]. Hierbei wird aber nicht von eingebetteten Echtzeit Systemen ausgegangen und es wird eine zusätzliche Infrastruktur benötigt.

Ein weiterer wichtiger Aspekt ist die Überwachung von Komponenten im SoC. An dieser Stelle existieren einige Ansätze, die sich mit der Überwachung von Bussystemen und Network-on-Chip (NoC) beschäftigen [71, 72, 73, 66, 74, 75].

Im Kontext von Online-Monitoring werden des Weiteren Ansätze verfolgt, die es ermöglichen, Plattformservices zu überwachen, so z.B. die Überwachung von virtuellen Maschinen auf einem Multicore in [76] oder für Betriebssysteme [77, 78, 79].

3.4 Virtualisierung

Virtualisierung wird eingesetzt, um Ressourcen wie CPU, I/O Komponenten/Peripherals, Speicher oder Co-Prozessoren zwischen verschiedenen isolierten Partitionen, oftmals als Virtuelle Maschine/Virtual Machine (VM) bezeichnet, zu teilen. Dabei kann die Ausprägung so geartet sein, dass die Virtualisierung für die Applikation und Treiber innerhalb einer virtuellen Maschine mehr oder weniger transparent ausgeprägt ist. Obwohl Virtualisierung hauptsächlich Anwendung in Datencentern findet, gibt es auch in eingebetteten Systemen mehr und mehr den Trend und durch Hardwareunterstützung eine gesteigerte Attraktivität zum Einsatz von Virtualisierung [80, 81]. Durch das Bestreben von Systemintegratoren hin zu immer höher integrierten Systemen[2] mit diversen Funktionen, welche oftmals sehr unterschiedlichen Anforderungen an die Performanz, Reaktionszeit und Zuverlässigkeit besitzen, werden Ansätze aus Datencentern mehr und mehr in eingebettete Systeme übertragen.

Eine Virtualisierungsschicht, die rein in Software oder mit Hardwareunterstützung umgesetzt werden kann, kümmert sich um die Arbitrierung und passendes Scheduling sowie die Zugriffssteuerung auf gemeinsam genutzte Geräte. Hierbei wird den virtuellen Maschinen vorgespielt, dass sie exklusiven Zugriff auf eine Ressource besitzen. Basierend auf diesen Annahmen liegt der Fokus traditioneller Virtualisierungslösungen auf einem guten Mittelwert der Rechenleistung und des Durchsatzes, während Zugriffe strikt eingehalten und eine räumliche Trennung (Spatial Segregation) von Speicherbereichen umgesetzt wird und so die Systemstabilität sichern. Die strikten Zugriffe, sowie die räumliche Trennung von Speicherbereichen, stellen oftmals Grundvoraussetzungen in sicherheitskritischen Anwendungen dar.

Unterschiedliche bekannte Ansätze zur Gerätevirtualisierung sind in Abbildung 3.4 dargestellt und werden im Folgenden diskutiert.

[2] Hochintegrationsszenarien sind beispielsweise die im Automotive Umfeld angestrebten Domänenleitrechner, in denen viele Funktionen einer Fahrzeugdomäne zusammengefasst werden, um Steuergeräte einsparen zu können.

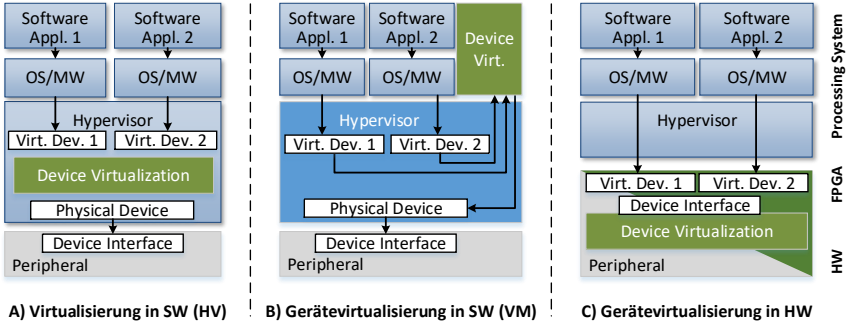


Abbildung 3.4: Übersicht von Virtualisierungstechnologien

Die Maximierung des Durchsatzes und die Minimierung der Latenz von virtualisierten Peripheriegeräten sind die beiden Hauptkriterien für eine gute Performanz der virtualisierten Systeme. Eine mögliche Technik für die Virtualisierung von Peripheriegeräten ist die Emulation von Gerätetreibern, wie sie in [82] beschrieben wird. Bei diesem Ansatz können die Gerätetreiber innerhalb der VM unmodifiziert weiterverwendet werden. Privilegierte Instruktionen, die von diesen Gerätetreibern ausgeführt werden, werden im Hypervisor abgefangen und gesondert verarbeitet. Der hauptsächliche Nachteil dieser Technik sind die Kosten für das Abfangen der Instruktionen, die Emulation selbst, sowie die damit einhergehenden Kontextwechsel. In Abbildung 3.4 A) ist das Konzept für die Gerätevirtualisierung dargestellt. Die eigentliche Handhabung der gemeinsamen Nutzung von Ressourcen findet dabei im Hypervisor statt, was eine erhöhte Komplexität des Hypervisors und damit ein erhöhtes Sicherheitsrisiko darstellt.

Um der Herausforderung bzgl. der Performanz der Geräte Emulation zu begegnen, wird Paravirtualisierung vorgestellt. Bei Paravirtualisierung wird ein Back-End Treiber im Hypervisor vorgehalten oder eine dedizierte VM in Kombination mit angepassten Front-End Treibern innerhalb der virtuellen Maschinen verwendet [83]. In dieser Konstellation wird kein Abfangen im Hypervisor mehr benötigt und somit der Overhead reduziert. Durch die nach wie vor benötigten Kopiervorgänge der Daten und die Kontextwechsel wird der Overhead noch immer auf ca.

20% geschätzt, wenn angenommen wird, dass die VM mit dem Back-End Treiber und die Applikationen auf unterschiedlichen Kernen ausgeführt werden, wie in [84] beschrieben. Zur gemeinsamen Nutzung von Controller Area Network (CAN) Controllern in Intergrated Modular Avionics (IMA) Systemen durch Paravirtualisierung wird in [85] eine zusätzliche Latenz von ca. 200 μs im Vergleich zum nicht-virtualisierten System beschrieben. Durch die Verwendung einer eigenen VM zur Umsetzung der gemeinsamen Nutzung von Geräten kann der Code des Hypervisors und damit die Komplexität reduziert werden.

Eine verbesserte Implementierung beim Einsatz einer dedizierten Treiber/Proxy VM zur gemeinsamen Nutzung von Graphics Processing Unit (GPU)s durch unterschiedliche virtuelle Maschinen für Automotive HumanMachine Interface (HMI) Anwendungen wird in [86] vorgestellt. Der Ansatz ist dabei auf Safety Anforderungen im Automotive Umfeld ausgelegt und integriert effiziente Integrationsstrategien für HMI Systeme im Automobil.

Um den Overhead weiter zu reduzieren, wird in [87] von Raj et al. ein Ansatz zur Selbstvirtualisierung vorgestellt. Durch die Verschiebung der Virtualisierungsschicht in Hardware, nahe an das eigentliche physikalische Gerät, kann der Durchsatz verdoppelt werden. Diese Alternative ist in 3.4 C) dargestellt. Durch direkte Gerätezuweisung kann der Zugriff ohne Aktivität des Hypervisors erfolgen. Dieser Ansatz ist nur möglich, wenn Geräte selbst in Hardware implementiert sind und angepasst werden können (vgl. graue Boxen in Abbildung 3.4) oder aber in einem FPGA (vgl. graue Boxen in 3.4) vorhanden sind.

Darüber hinaus wird in [88] der Vergleich zweier weiterer Ansätze zur gemeinsamen Nutzung von CAN-Controllern von unterschiedlichen virtuellen Maschinen vorgestellt. Das erste Konzept, welches in [89] beschrieben wird, nutzt Paravirtualisierung vergleichbar mit der Darstellung in Abbildung 3.4 B) auf einer Automotive Multicoreplattform (Infineon AURIX [42]). Der zweite Ansatz basiert auf einem selbstvirtualisierten CAN-Controller, der die SR-IOV-Technologie für PCIe eines Intel Core i7 nutzt. Dabei sind CAN-Controller und Gerätevirtualisierung komplett in einem FPGA realisiert, wie es in Abbildung 3.4 C) durch die grünen Boxen dargestellt ist.

In einem weiteren Ansatz von Pham et al. [90] wird ein Hypervisor Konzept für eine hybride ARM-FPGA Architektur vorgestellt, der rekonfigurierbare Funktionen in Hardware unterstützt. Ein weiterer Ansatz für FPGAs wird in [91] vorgeschlagen. Beide Ansätze generieren jedoch einen erheblichen Overhead in Software, welcher die Komplexität der Software steigert und damit die Beherrschbarkeit weiter einschränkt.

Im Bereich kleinerer eingebetteter Controller wird eine Virtualisierung in [92] vorgeschlagen, jedoch ohne heterogene Architekturen zu adressieren und mit Overhead in Software. Des Weiteren wird ein Partitionierender/Isolierender Hypervisor in eingebetteten Systemen in [93] diskutiert. Der Proteus Hypervisor bietet sowohl Vollvirtualisierung als auch Paravirtualisierung für eingebettete Multicoresysteme [94], kommt jedoch auch nicht ohne zusätzliche Software aus.

3.5 Abgrenzung

Basierend auf dem aktuellen Stand der Technik werden mehrere Lücken in der Forschung adressiert. Unter anderem wird betrachtet, in wie weit das Einbringen einer Redundanz in einen Multicorprozessor darstellbar ist. Es wird methodisch untersucht, wie ein solches Konzept aussehen und integriert werden kann und welche Implikationen dies auf die Hardware/Software Architektur hat. Ebenso wird geprüft, welcher Overhead bzw. welche zusätzliche Latenz durch die Einbringung der Redundanz in einem Mehrkernprozessor entsteht. Um auch übergreifende Fehlerquellen erkennen und abfangen zu können, wird beurteilt, welche zusätzlichen Maßnahmen ergriffen werden müssen.

Zur Adressierung der für Multicoreprozessoren spezifischen Fehlerursachen und der nachfolgenden Erkennung von Fehlern, werden Methoden untersucht, welche zum Online-Monitoring für Bus-Master in einer solchen Architektur notwendig sind und verwendet werden können. Hierfür existieren keine parametrierbaren Ansätze, die für eine Vielzahl von Bus-Master-Komponenten Anwendung finden können und speziell die Herausforderungen in einem Mehrkernprozessor adressieren.

Im aktuellen Stand der Technik existieren einige Ansätze zur Virtualisierung für Multicoreprozessoren. Diese Ansätze kommen meist nicht ohne eine Anpassung der Software und damit nicht ohne eine erhöhte Komplexität aus. Andere Ansätze benötigen eine Anpassung der physikalischen Geräte, die virtualisiert werden sollen. Es wird eine parametrierbare Virtualisierung benötigt, die ohne eine Anpassung der Software und somit transparent für den Software-Entwickler ist. Speziell im Bereich heterogener Multicorearchitekturen existieren keine Ansätze, die eine Virtualisierung von ASIC-Geräten ohne deren Anpassung erlaubt. Techniken zur Gerätevirtualisierung, die in der Literatur diskutiert werden, benötigen entweder eine Anpassung der Software oder eine Anpassung der physikalischen Geräte.

Speziell die zusätzlich eingebrachten Fehlerquellen in Multicoreprozessoren (u.a. die gesteigerte Soft-Error-Rate [55, 56, 57], Fehler durch gemeinsam genutzte Ressourcen sowie Fehler durch parallele Zugriffe und daraus resultierenden Verzögerungen in der Ausführung) müssen angegangen werden. Um dies zu erreichen und eine Überwachung sowie sichere Handhabung von Ressourcen zu gewährleisten, werden die folgenden Methoden zur Überwachung und Fehlererkennung sowie zur sicheren Nutzung von Ressourcen und zur Funktionsmigraton vorgeschlagen, die dazu dienen die existierenden Lücken im Stand der Technik zu schließen.

4 Die Multicore Herausforderung

In diesem Kapitel werden die durch die Architektur von Multicoreprozessoren resultierenden Herausforderungen aufgearbeitet und basierend darauf mögliche Fehlerbilder aus Applikationssicht abgeleitet. Es wird verdeutlicht, dass der Übergang von Mehrrechnerarchitekturen zu Mehrkernprozessoren einen erheblichen Unterschied mit sich bringt. Hierfür wird analysiert, welche Änderungen sich ergeben und in welchen Herausforderungen dies resultiert. Die anschließend beschriebenen Fehlerbilder bilden im weiteren Verlauf der Arbeit die Grundlage für die Validierungen.

4.1 Herausforderungen bei der Entwicklung multicorebasierter Systeme

In vielen Anwendungen werden bereits parallele Ausführungseinheiten genutzt, zumeist jedoch auf einer hohen Abstraktionsebene. So zum Beispiel in Automobilen oder Flugzeugen, in denen die Vernetzung von Steuergeräten als verteilte, parallele Ausführung von Funktionen mit einem gemeinsamen Kommunikationskanal dargestellt wird. Als eine Weiterentwicklung dessen können Mehrprozessorsysteme betrachtet werden, die eine höhere Integrationsdichte aufweisen. Bei Mehrprozessorsystemen werden mehrere Prozessoren in ein Steuergerät integriert und über einen Kommunikationskanal auf einer Platine verbunden, während die anderen Komponenten unabhängig bleiben. Sehr abstrakt betrachtet, ähneln sich die beiden Architekturen und bieten auch gewisse Gemeinsamkeiten. Beide Architekturen führen Funktionen in sicherheitskritischen Anwendungen aus und teilen sich jeweils einen Kommunikationskanal. Hinzu kommt jedoch, dass auf Ebene eines

Steuergeräts nur eine Schnittstelle der eigentlichen Electronic Control Unit (ECU) zur Vernetzung im Fahrzeug existiert. Diese Architektur ist bereits sehr nahe an einer Multicorearchitektur.

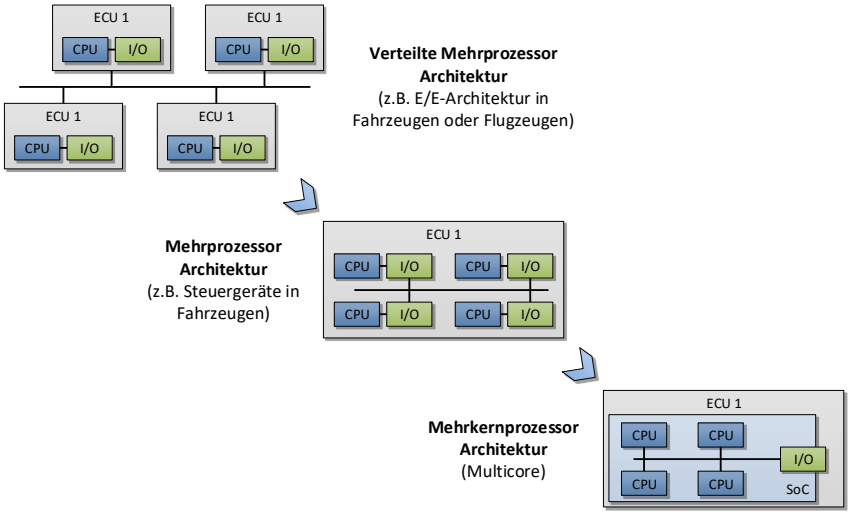


Abbildung 4.1: Von verteilten Mehrprozessorsystemen zu Mehrkernprozessoren

Basierend auf dem Grad der Integration ergeben sich neue Herausforderungen. Durch die sehr hohe Integration von Rechenleistung, Input/-Output (I/O) Komponenten und Speicher innerhalb eines Chips, wird ein Level erreicht, auf dem die Parallelität eine noch größere Rolle spielt.

In jedem Zyklus könnte theoretisch jeder Rechenkern einen Zugriff auf den gemeinsam genutzten Speicher oder eine gemeinsam genutzte Peripherikomponente ausführen. Auf Basis dieser Beobachtungen wird deutlich, dass die Wahrscheinlichkeit für eine Kollision von mehreren Zugriffen bei einem zeitlich parallelen Zugriff erheblich ansteigt.

Solche Kollisionen aufgrund von parallelen Zugriffen können bereits in einem gemeinsamen Verbindungsnetzwerk im Chip oder aber erst an der eigentlichen Ressource auftreten. Je nach Ausprägung können Kollisionen durch eine erhöhte Bandbreite oder durch Arbitrierung an

der Ressource gelöst werden, verursachen damit jedoch eine gewisse Wartezeit für einen der zugreifenden Rechenkerne. Da in vielen Fällen eine Erhöhung des Durchsatzes der Peripheriekomponenten oder der Speichercontroller typischerweise nicht im gleichen Verhältnis wie die Rechenleistung der Prozessorkerne steigt, können damit die Kollisionen in der Regel nicht gelöst werden.

Durch die beschriebenen Kollisionen entsteht eine Beeinflussung von Rechenkernen und damit eine nicht vorhersagbare Wartezeit selbst bei funktional unabhängigen Rechenkernen.

Ergänzend zu den funktional unabhängigen Rechenkernen in einem Multicore sind Konstellationen mit funktionalen Abhängigkeiten in vielen Szenarien erstrebenswert zur Steigerung der Performanz. Auch hier treten die gemeinsam genutzten Ressourcen, wie Speicher, in den Vordergrund. Zusätzlich jedoch wird in einer solchen funktional abhängigen Verteilung der Software eine Synchronisation unabdingbar. Entscheidend ist sowohl die Datenkonsistenz als auch die Vermeidung von Warteschleifen aufgrund nicht verfügbarer Daten.

Durch die beschriebene Dateninkonsistenz und die möglichen Warteschleifen kann ein funktionales Fehlverhalten auftreten.

Weiterhin resultiert durch die gesteigerte Integrationsdichte in einem Multicore eine wesentlich höhere Komplexität im Vergleich zu Singlecoreprozessoren. Typischerweise werden in neuen Generationen eines MPSoC nicht nur mehr Rechenkerne, sondern auch zusätzliche Peripheriekomponenten, Beschleuniger, Test- und Debug-Möglichkeiten oder andere Funktionen integriert. Für die Anwendung in sicherheitskritischen Anwendungen müssen sichere und zuverlässige Konfigurationen erarbeitet werden, was sich aufgrund des wesentlich größeren Konfigurationsraums als eine Herausforderung darstellt.

Durch die Integration von mehr und mehr Funktionen und Komponenten in das MPSoC steigt der Umfang der verfügbaren Dokumentationen. Jedoch werden einige Komponenten und Funktionen innerhalb eines SoC nicht dokumentiert, da diese zu Test- und Entwicklungszwecken bei den Herstellern eingebracht werden. Speziell diese wenig dokumentierten Funktionen müssen bei einer Zertifizierung in einigen Domänen durch Dokumentationen belegt und ein Einfluss auf das Produktivsystem ausgeschlossen werden.

Die für Multicoreprozessoren in sicherheitskritischen Anwendungen relevanten Aspekte und Herausforderungen können folgendermaßen zusammengefasst werden:

- **Trennung in Raum und Zeit (Segregation in Time and Space):**

  Es werden Methoden benötigt, die eine effiziente gemeinsame Nutzung von Ressourcen unter Einhaltung der Trennung realisieren.

- **Synchronisation und effiziente Verteilung der Applikationssoftware:**

  Softwarekomponenten/Funktionen für die Realisierung einer Applikation müssen vor dem Hintergrund einer Kostenfunktion hinsichtlich der Synchronisation und Kommunikation verteilt werden.

- **Effiziente Verteilung von Plattformsoftware:**

  Analog zur Verteilung von Applikationssoftware muss auch Plattformsoftware, wie beispielsweise ein Betriebssystem, optimiert verteilt werden, um Kosten für die Synchronisation und Kommunikation gering zu halten.

- **Analyse von Multicore Plattformen und Software:**

  Zur Abschätzung und Bestimmung von Garantien sowie oberen Schranken, wie beispielsweise der WCET, muss die Hardware/-Software Architektur analysiert werden.

- **Handhabung und Management der Komplexität:**

  Die Beherrschung des Konfigurationsraums eines Multicores ist notwendig, um eine sichere und zuverlässige Konfiguration der Plattform zu erreichen.

Für die beschriebenen Herausforderungen kann nicht immer zur Design-Zeit eine definitive Lösung erarbeitet werden. Oftmals können beispielsweise keine Vorhersagen für Kollisionen während der Entwicklung eines multicorebasierten Systems identifiziert werden. Dies motiviert den Einsatz von Methoden zur Überwachung im Betrieb sowie den Einsatz von Methoden zur sicheren gemeinsamen Nutzung von Ressourcen, um den Fehlerbildern aus Applikationssicht zu begegnen.

4.2 Fehlerbilder aus Applikationssicht

Zur Entwicklung einer neuen Datenverarbeitungseinheit (DVE) für eine sicherheitskritische Anwendung muss zunächst eine genaue Spezifikation des Gesamtsystems erfolgen. Je nach Einsatzzweck und Systemkontext, wird eine Einstufung der DVE im Sinne der Kritikalität im Falle eines Ausfalls vorgenommen. Diese Einstufung erfolgt auf Basis der/des anzuwendenden Standards der Domäne. Unterschiedliche Kritikalitätslevel bedingen andere Anforderungen der Überwachung und Fehler-Präventionsmechanismen. Zunächst erfolgt somit eine Ableitung der Anforderungen auf Basis der durchgeführten Einstufung. Unter anderem gehört die Feststellung der Anforderungen an die Ausfallwahrscheinlichkeiten einzelner Komponenten im System zu einem notwendigen Schritt. Basierend auf den Standards werden tolerierte Ausfallraten identifiziert und heruntergebrochen auf Komponentenebene.

Die Anforderungen dienen im weiteren Verlauf der Entwicklung als Ausgangspunkt und bedingen wiederum den Einsatz verschiedener Mechanismen, die zur Fehlererkennung sowie der darauffolgenden Reaktion im Systementwurf berücksichtigt werden müssen, um die benötigte Zuverlässigkeit zu erreichen und die Ausfallwahrscheinlichkeit unter die in den Standards gesetzten Grenzwerte zu senken. Hierfür ist es ausreichend, Fehler zu erkennen und eine entsprechende Reaktion herbeizuführen. Bei der Reaktion handelt es sich um das Überführen des Systems in einen sicheren Zustand (z.B. Reset-Zustand eines Prozessors mit definierten Ausgangswerten). Um jedoch einen Fehler überhaupt erst erkennen zu können, sind unterschiedliche Mechanismen auf verschiedenen Ebenen des Systems notwendig. Speziell bei Mehrkernprozessoren resultiert eine Vielzahl neuartiger Fehler aus der Prozessorarchitektur. Zur Erkennung dieser Fehler werden im Folgenden verschiedene neue Ansätze für Multicoreprozessoren präsentiert und diskutiert. Es sei jedoch erwähnt, dass keiner der Ansätze für sich genommen alle Anforderungen zum Einsatz von Mehrkernprozessoren erfüllen kann, sondern eine Kombination mit weiteren Methoden (z.B. durch weitere externe Beschaltung) notwendig ist.

Zusätzlich zu Methoden der Fehlererkennung spielt in Mehrkernprozessoren die sichere gemeinsame Nutzung von Ressourcen eine wich-

tige Rolle, um hierbei Fehler von vorn herein zu vermeiden. Zugriffe auf Ressourcen innerhalb eines MCP müssen auch bei parallelem Eintreffen von Anfragen einer deterministischen Abfolge genügen. Darüber hinaus muss für jeden Zugriffsberechtigten die Möglichkeit bestehen, die Komponente zuverlässig ansteuern zu können und dabei nicht durch andere Zugriffe verdrängt zu werden.

Die im weiteren Verlauf dieser Arbeit beschriebenen Methoden sind somit als Sammlung von Konzepten und Architekturmustern (Pattern) zu verstehen, die einen sicheren Einsatz von Mehrkernprozessoren unterstützen. Eine konkrete Kombination von Mechanismen muss auf den Anwendungsfall angepasst und anschließend evaluiert werden. Die Konzepte eignen sich jeweils für den Einsatz in einer Vielzahl von Prozessortypen und können je nach Anwendungsfall erweitert oder reduziert umgesetzt werden. Dies bedeutet, dass im jeweiligen Anwendungsfall an unterschiedlichen Parametern eine Einpassung bzw. Integration im Gesamtsystemkontext vorgenommen werden kann, mit dem, je nach Zielfunktion, ein Trade-Off zwischen Monitoring-Detailgrad, Mechanismen zur gemeinsamen Nutzung von Komponenten und Overhead (im Sinne von Performanzeinbußen) sowie Ressourcenbedarf gefunden werden kann.

Um den sicheren Einsatz eines MCP unterstützen zu können, müssen alle Komponenten entlang der funktionalen Kette von Fehlererkennung über Fehlerisolation bis hin zu Fehlerkorrektur bzw. der Reaktion auf einen Fehler berücksichtigt werden. Als Beispiel ist in Abbildung 4.2 ein Fehlerszenario auf einem Mehrkernprozessor dargestellt. Hierbei entsteht ein ungewünschter Zugriff auf ein Peripheral zum falschen Zeitpunkt durch einen sporadischen Fehler im Prozessorkern (vgl. Abbildung 4.2 „Fehlerursprung“). Dies führt im Umkehrschluss dazu, dass ein weiterer Prozessorkern nicht wie gewünscht auf das Peripheral zugreifen kann, da dieses bereits genutzt wird. Der Fehler ist somit im System propagiert und beeinflusst nicht nur die Ausführung auf einem Prozessorkern, sondern beeinflusst das gesamte SoC.

Nachfolgend werden Klassen von Fehlern vorgestellt, die in einem Mehrkernprozessor auftreten können, gefolgt von Methoden zur Fehlererkennung, Fehlerisolation und Fehlerkorrektur. Die entwickelten Methoden und Mechanismen zur Fehlererkennung sowie eine Methode

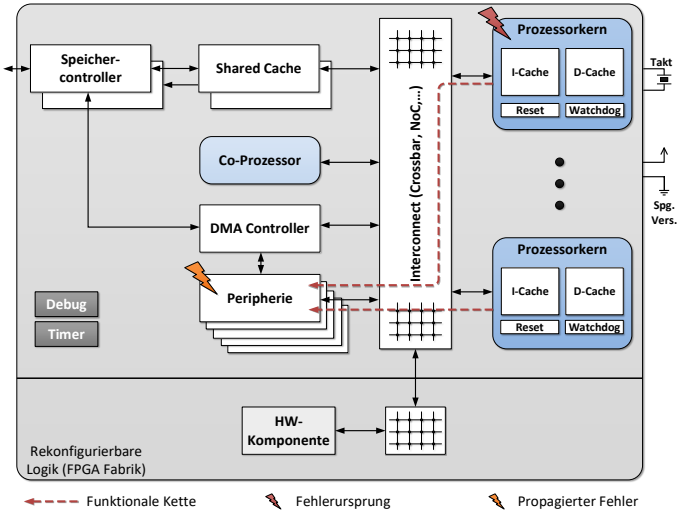


Abbildung 4.2: Beispielhafte Fehlerpropagation in einem Mehrkernprozessor

zur sicheren gemeinsamen Nutzung von Ressourcen decken die gesamte Fehlerkette ab.

Im Allgemeinen können Fehler in einem Multicoreprozessor in drei Varianten, unabhängig vom Fehlerursprung, sichtbar werden:

- Indeterministische/zu lange Ausführungszeit
- Fehlerhafte Daten/Berechnung
- Fehlende Ausführung von Funktionen oder Funktionsteilen.

Die Fehlerbilder an sich können auch in Singlecoreprozessoren auftreten, die Auswirkungen sind in Multicoreprozessoren jedoch weitreichender. Ebenso sind die Ursachen, die ein solches Fehlerbild hervorrufen, bei Multicore vielfältiger. Aus diesem Grund wird eine gesonderte Überwachung benötigt. Eine Übersicht der Fehlerbilder aus Applikationssicht sowie mögliche Fehlerquellen sind in Abbildung 4.3 dargestellt.

Eine verlängerte Ausführungszeit sowie die Berechnung der Worst Case Execution Time (WCET) wird durch parallele/zeitgleiche Zugriffe

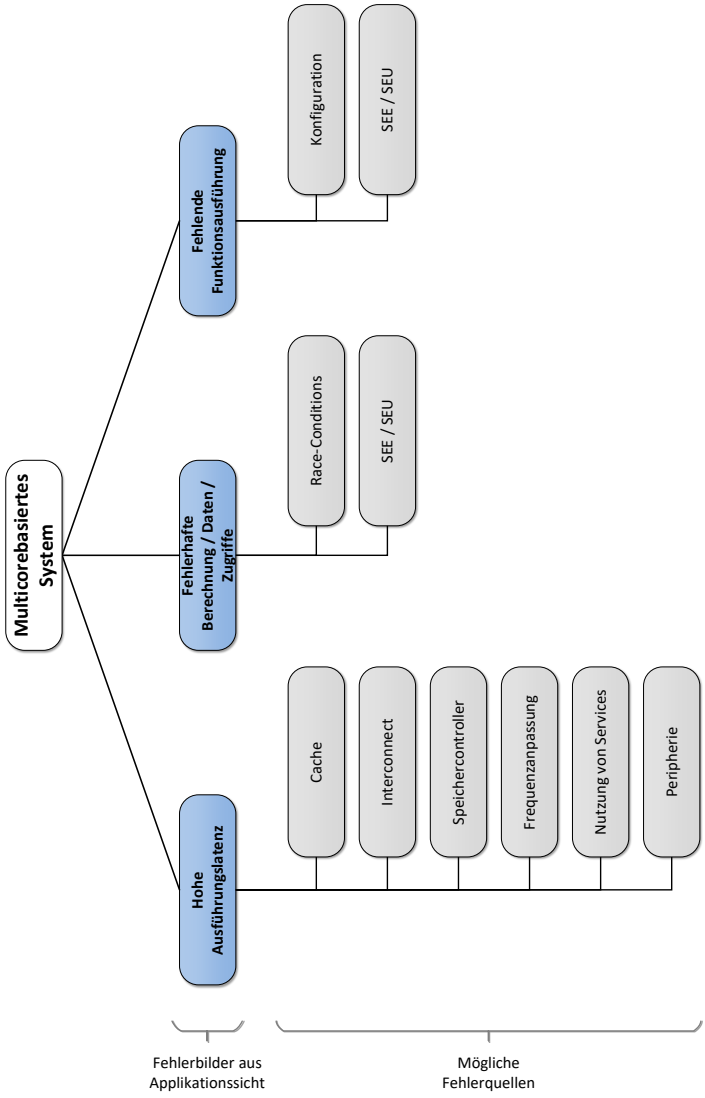


Abbildung 4.3: Fehlerbilder aus Applikationssicht und mögliche Fehlerquellen

auf Ressourcen beeinflusst bzw. erschwert. Beispielsweise entsteht ein solches Verhalten durch gemeinsam genutzte Caches, da ein Datum von einem konkurrierenden Rechenkern verdrängt wurde. Bei der Verwendung von effizienten (und dadurch zumeist indeterministischen) Cache Strategien kann ein nicht vorhersagbares Zeitverhalten eines Rechenkerns entstehen, was dazu führt, dass eine Deadline nicht eingehalten wird. Ein ähnliches Verhalten ist durch parallele oder übermäßige Kommunikation über ein Interconnect zu beobachten, bei dem die Latenzen nicht deterministisch vorhergesagt werden können. Ergänzend werden Verzögerungen durch Zugriffe auf geteilte Ressourcen (z.B. Speichercontroller, Peripherals, Plattform Services) oder durch eine Blockierung (z.B. durch Priorisierung) ermöglicht.

Fehlerhafte Daten bzw. Fehler in der Berechnung können aus der nicht erwünschten Verwendung veralteter Daten resultieren. Hierbei kann das Datum zum Zeitpunkt des Auslesens bereits wieder veraltet sein, wenn parallel zum Auslesen durch einen Rechenkern ein anderer Rechenkern das Datum überschreibt. Diese sog. *Race-Conditions* treten dann auf, wenn ein Rechenkern das Datum im Speicher überschreiben möchte, aber aufgrund einer niedrigeren Priorität noch keinen Zugriff erhält, während ein anderer Rechenkern mit höherer Priorität das Datum auslesen kann. Des Weiteren kann ein fehlerhaftes Datum bzw. ein Fehler in der Berechnung durch sog. Bit-Flips[1] hervorgerufen werden. Diese können durch Strahlung sog. Single Event Effect (SEE)/Single Event Upset (SEU)[2] entstehen und können sowohl Änderungen in den Befehlen als auch in Daten herbeiführen.

Als weiteres Fehlerbild kann die fehlende Ausführung von Funktionsteilen auftreten. Hierbei werden Teile oder sogar ganze Funktionsteile nicht ausgeführt. Aufgrund der hohen Konfigurierbarkeit und den daraus entstehenden Permutationen kann es durch einen Bit-Flip in den Konfigurationsregistern zu einer fehlerhaften Zuweisung von Interrupts kommen. Diese werden dann fälschlicherweise an einen anderen Rechenkern geleitet als ursprünglich im Design vorgesehen. Da für diesen Fall keine explizite Interrupt Service Routine (ISR) vorge-

[1] Inversion eines Bits von seinem ursprünglichen Wert. Beispielsweise im Speicher oder in Registern.

[2] Durch Strahlung auftretende Effekte innerhalb einer integrierten Schaltung.

sehen ist, wird der eigentliche Funktionsteil, sprich die vorgesehene ISR, nicht ausgeführt. Eine fehlerhafte Zuweisung von Interrupts zur Designzeit des Systems würde hingegen während des Verifikations-/Validierungsprozesses erkannt werden. Ergänzend können durch Bit-Flips in Befehlen Funktionsteile übersprungen bzw. ausgelassen werden.

Die dargestellten Fehlerbilder dienen im weiteren Verlauf der Einordnung der Monitoring Konzepte, die unterschiedliche Fehlerbilder adressieren. Ebenso wird die Abdeckung der dargestellten möglichen Fehlerbilder in der gesamtheitlichen Betrachtung der Monitoringansätze verdeutlicht.

5 Parametrierbare und adaptive Monitoring-Methoden zur Fehlererkennung

Einige im Mehrkernprozessor enthaltenen Komponenten bieten ein erhöhtes Fehlerpotential aufgrund der gemeinsamen Nutzung durch mehrere Rechenkerne sowie andere Master-Komponenten am gemeinsamen Interconnect. Zur Erkennung von Fehlern, die bei der Nutzung dieser Komponenten auftreten, müssen im Prozessor an unterschiedlichen Stellen Monitore eingebracht werden. Diese Monitore müssen zum einen den Fehler erkennen und zum anderen diese Erkenntnis an eine übergeordnete Einheit melden, um die weiteren Reaktionen auf einer höheren Ebene auszulösen. Die eingebrachten Monitore sollen hierbei möglichst wenig Overhead bedingen und sich, soweit möglich, bereits vorhandene Komponenten zu Nutze machen.

Fehler können jedoch nicht nur erkannt, sondern auch korrigiert bzw. abgeschwächt/mitigiert werden. So kann beispielsweise der Einsatz von Redundanz, welche einen größeren Overhead erzeugt, einen Fehler erkennen und abschwächen, so dass das System weiterhin voll funktionstüchtig bleibt. Je nach Einsatzzweck ist ein solches Architekturmuster jedoch notwendig. Hierbei spielt der Zielkonflikt - *Ausfallsicherheit vs. Overhead* - eine entscheidende Rolle.

Im Folgenden werden zwei Methoden zur Erkennung von Fehlern in kritischen Komponenten eines Mehrkernprozessors sowie eine Methode zur Umsetzung von Redundanz (zur Aufrechterhaltung der Systemfunktion) auf Prozessorlevel vorgestellt und diskutiert.

5.1 Redundanz in Mehrkernprozessoren

Es gibt vielfältige Möglichkeiten, um auf einen aufgetretenen Fehler zu reagieren. Diese Reaktionen sind wiederum vom Anwendungsfall und Einsatzzweck abhängig. Beispielsweise kann beim Fehler in einer Berechnung, resultierend aus einem Bit-Flip, ein Zwischenergebnis verworfen und im nächsten Zyklus mit einem korrekten Ergebnis gerechnet werden. Einige Anwendungsfälle lassen ein solches Vorgehen zu, z.B. bei Regelung eines trägen mechanischen Systems, welches im nächsten Regelzyklus noch keine kritische Position eingenommen hat. Es gibt jedoch Anwendungsfälle, in denen jedes Ergebnis zur Ansteuerung korrekt vorliegen muss. Hierfür kann man sich einer redundanten Berechnung bedienen und anschließend die Ergebnisse vergleichen. Beim Einsatz eines Multicoreprozessors kann die redundante Berechnung in einen MPSoC verschoben werden. Ein mehrfaches Vorhalten der Hardware ist dafür nicht mehr notwendig. Die zur Umsetzung einer redundanten Berechnung notwendigen Ressourcen innerhalb eines Mehrkernprozessors müssen jedoch vorgehalten werden. Dies beinhaltet, je nach redundanter Auslegung, die Bereitstellung jeweils einer Berechnungseinheit und einer auswertenden Instanz (Voter). Dieser Voter kann wiederum auf einem Rechenkern in Software oder alternativ sehr effizient in Hardware umgesetzt werden. Basierend auf den Anforderungen an die Sicherheit, wird das Redundanzschema sehr grob- oder feingranular aufgebaut. Bei einem grobgranularen Aufbau werden beispielsweise nur Endergebnisse verglichen, während bei einem feingranularen Aufbau beliebig viele Zwischenergebnisse verglichen werden. Oftmals ist es ausreichend die Endergebnisse, die zur Ansteuerung oder Weiterverarbeitung genutzt werden sollen, zu vergleichen. Zur Steigerung der Rückverfolgbarkeit des Fehlerursprungs bietet es sich jedoch an, Zwischenergebnisse auszuwerten und den jeweiligen Status zu speichern.

5.1.1 Konzept

Zur Realisierung eines TMR-Konzepts, bietet sich die Möglichkeit, dies auf nur einem Chip mit mehreren Rechenkernen umzusetzen. Dieses

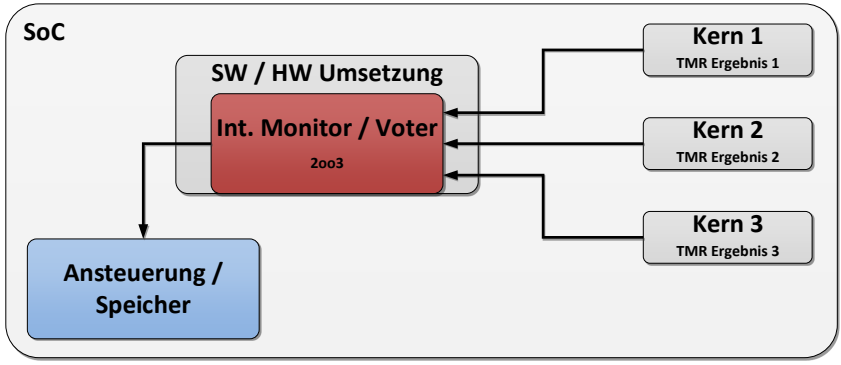


Abbildung 5.1: Konzept des TMR on Chip

Konzept basiert auf der Diplomarbeit [95]. Die Idee, beispielsweise drei Prozessorkerne für TMR zu verwenden und einen weiteren als Voter, erspart die Vervielfachung der Hardware (Prozessoren oder ganze ECUs), bringt aber neue mögliche Fehlerquellen mit sich. Als zusätzliche Fehlerquellen müssen vor allem gemeinsame Ressourcen, wie die Spannungsversorgung, der gemeinsame Takt oder auch der gemeinsam genutzte Speicher (Common Cause Failure (CCF)), betrachtet werden. Um in einem MPSoC, auf dem das TMR-Konzept umgesetzt ist, die einzelnen Prozessoren zu überwachen, werden weitere Mechanismen benötigt. Hier kann man sich einer Challenge-Response-Abfrage bedienen, welche eine zyklische Abfrage einer CPU realisiert. Im einfachsten Fall ist dies über einen Watchdog oder bei aufwendigeren Systemen mit einer weiteren CPU und mehr Funktionalität in Software realisiert. Zur Umsetzung wird von einem überwachenden Prozessor des Multicores zyklisch eine Anfrage an die TMR-Prozessoren gestellt, die diese innerhalb einer Deadline beantworten müssen. Wird die Anfrage nicht rechtzeitig beantwortet, so müssen Konsequenzen, wie zum Beispiel die Isolation oder der Restart des entsprechenden Prozessors, gezogen werden. Dieser interne Monitor hat die Eigenschaft, eine sehr hohe Kommunikationsbandbreite zu den TMR-Prozessoren zu haben, was sehr kurze Anfrage-Antwort-Zeiten ermöglicht, die von einem externen überwachenden Element nicht gehalten werden können.

Ein Nachteil des internen Prozessors sind die gleichen gemeinsamen Fehlerquellen wie bei den TMR-CPUs. Um auch solche Fehlerquellen überwachen zu können, wird ein externes Überwachungselement benötigt. Hierbei handelt es sich um einen externen Monitor, der die gemeinsamen Fehlerquellen überwacht und den internen Monitor kontrolliert. Die Funktion des internen Monitors wird auch hier über eine Challenge-Response-Abfrage realisiert. Der externe Monitor stellt zyklisch eine Anfrage an den internen Monitor, der innerhalb einer gewissen Zeitspanne antworten muss. Kommt keine Reaktion oder eine verspätete Antwort, müssen auch hier Konsequenzen gezogen werden. Die Challenge-Response-Abfrage des externen Monitors an den internen ist aufgrund der Busanbindung und der Latenzzeiten der hierzu nötigen Peripherieeinheiten, z.B. ein I²C-Interface, nicht in solch kurzen Zeitabständen möglich, wie die Abfrage der TMR-Prozessoren vom internen Monitor aus, welche über den direkten Prozessorinterconnect gestellt wird. Um mögliche Fehler erkennen zu können wird ein System implementiert, welches sowohl das TMR-Konzept beinhaltet als auch einen einfachen hierarchischen Aufbau der Monitore. Die Übersicht des Systems ist in Abbildung 5.1 dargestellt. Hier liefern die Kerne 1 - 3 jeweils ein Ergebnis der Berechnung, welche in den Vergleich mit einbezogen werden.

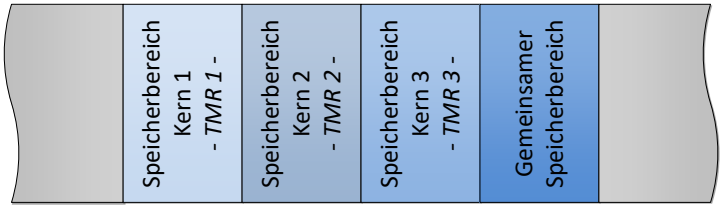


Abbildung 5.2: Schematische Darstellung der Partitionierung des Speichers im TMR Konzept

Zur Sicherstellung der Konsistenz der Daten und der Umsetzung einer räumlichen Trennung wird eine Partitionierung des Speichers vorgenommen. Dies bedeutet, dass jedem der Rechenkerne, die eine Berechnung durchführen, ein dedizierter Speicherbereich zugewiesen wird. Diese Aufteilung ist schematisch in Abbildung 5.2 dargestellt. Durch Speicherschutzmechanismen kann überwacht und sichergestellt

werden, dass kein anderer Kern einen Zugang zu diesem Bereich besitzt und somit nicht fehlerhafterweise die Daten eines anderen Rechenkerns überschreibt. Zum Austausch der Ergebnisse mit dem Voter kann jedoch ein gemeinsam genutzter Speicherbereich verwendet werden, in den die Rechenkerne nach Beendigung der eigentlichen Berechnung die Ergebnisse kopieren und so dem Voter zur Verfügung stellen.

Diese Ergebnisse werden somit dem Voter übergeben, welcher einen Mehrheitsentscheid durchführt. Je nach zur Verfügung stehender Hardware kann der Voter in Hardware oder in Software auf einem weiteren Rechenkern implementiert werden. Dies ist in Abbildung 5.1 dargestellt. Beide Varianten bieten Vor- und Nachteile. Eine Implementierung in Hardware kann völlig transparent ablaufen und muss lediglich ausgelöst werden, wenn die Ergebnisse zur Verfügung stehen. Jedoch kann eine Erweiterung um eine Challenge-Response-Abfrage in einer Hardware-Implementierung erheblich aufwendiger gestaltet sein als in Software. Umgekehrt benötigt die Umsetzung in Software einen weiteren Rechenkern, auf welchem das Software Voting durchgeführt wird. Die Flexibilität zur Integration einer Challenge-Response-Abfrage ist dafür wesentlich höher.

Der schematische Ablauf ist in Form eines Aktivitätsdiagramms in Abbildung 5.3 dargestellt. Der für den Voter dargestellte Ablauf kann sowohl in Hardware als auch in Software realisiert sein. Zunächst wird das TMR-System vom Voter initialisiert und die Kerne, welche die Berechnung durchführen, synchronisiert. Die Berechnung auf den Kernen 1 - 3 wird somit ausgelöst. Jeder der Kerne, welche die funktionalen Berechnungen durchführen, muss eine Anfrage (Challenge) des Voters innerhalb einer definierten Zeitspanne beantworten. Bleibt die Antwort auf die Anfrage aus, wird das Ergebnis des entsprechenden Kerns beim Voting nicht berücksichtigt. Nach Beendigung der Berechnung wird dem Voter von den Kernen signalisiert, dass die Ergebnisse zur Verfügung stehen. Die Ergebnisse können dabei direkt zum Voter übertragen oder in einem gemeinsam genutzten Speicherbereich abgelegt werden. Zur Übertragung der Ergebnisse von einem privaten Speicherbereich eines Kerns in den gemeinsam genutzten Speicher wird ein DMA Transfer verwendet.[1]

[1]Da hierbei ein Zugriff auf eine gemeinsam genutzte Ressource entsteht, muss diese zusätzlich überwacht werden, da auch damit zu rechnen ist, dass die Berechnungen

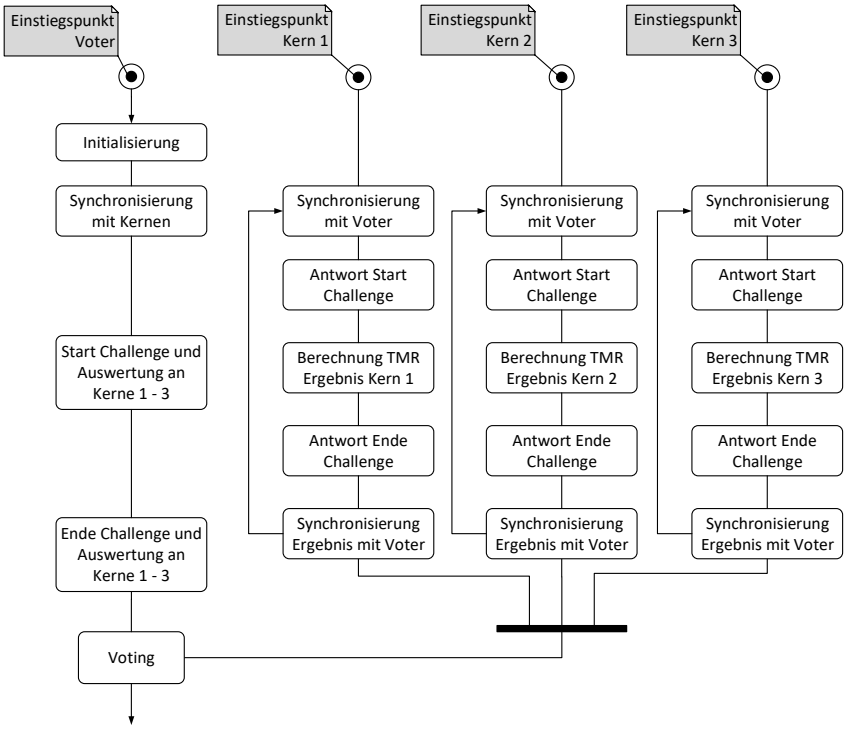


Abbildung 5.3: Aktivitätsdiagramm des TMR on Chip Konzepts

Der Voter wertet anschließend die Ergebnisse aus und übergibt das Mehrheitsergebnis an einen entsprechenden Ausgang (z.B. Peripheral, Kommunikationsschnittstelle) oder legt das Resultat für weitere Berechnungen im Speicher ab.[2] Das Ergebnis des TMR-Votings (E_{TMR}) kann als logische Operation bzw. Verknüpfung der drei Teilergebnisse (E_1, E_2, E_3) dargestellt werden:

$$E_{TMR} = (E_1 \wedge E_2) \vee (E_1 \wedge E_3) \vee (E_2 \wedge E_3) \tag{5.1}$$

Die in Gleichung (5.1) verwendeten Literale repräsentieren Ergebnisvektoren, die je nach Implementierung unterschiedliche Datenformate besitzen können.

Basierend auf dem vorgestellten Konzept kann je nach benötigter Zuverlässigkeit eine Erweiterung durchgeführt werden. Eine Skalierung zu einem n-out-of-m Aufbau kann durch Berechnungen auf weiteren Rechenkernen und der Skalierung des Voters einfach umgesetzt werden. Die Berechnung folgt damit der Vorschrift (5.2).

$$E_{n-oo-m} = (E_1 \wedge E_2 \ldots \wedge E_n) \ldots \vee (E_2 \wedge E_3 \ldots \wedge E_{n+1}),\ mit\ n < m \tag{5.2}$$

Ebenso kann eine Erweiterung um eine Berechnung auf einer diversitären Architektur (z.B. in Hardware in einer rekonfigurierbaren Architektur eines MPSoC) im selben Chip oder auch auf einer externen Einheit von Nöten sein. Je nach benötigter Berechnungsrate kann eine externe Einheit jedoch potentiell nicht effizient genug angebunden werden, weshalb eine interne Realisierung bevorzugt wird.

Durch die Verwendung einer diversitären Hardwarearchitektur werden zusätzlich systematische Fehler in der Architektur und den verwendeten Werkzeugen (z.B. Code-Generatoren, Compiler) verhindert. Ebenso

der Kerne zeitgleich abgeschlossen werden. Ein Konzept für die Überwachung von DMA Zugriffen ist in Kapitel 5.3 beschrieben.

[2] Beim Zugriff auf ein gemeinsam genutztes Peripheral muss ausgeschlossen werden, dass sich hierbei Überschneidungen mit anderen Zugriffen auf dieses ergeben. Ein Konzept zur Vermeidung und Erkennung von Überschneidungen von Peripherals ist in Kapitel 6.1 beschrieben.

können systematische Fehler in der Implementierung der Software durch unterschiedliche Entwicklerteams erkannt werden. Dies hat jedoch zur Folge, dass verschiedene Teams die Anforderungen in gleicher Weise verstehen und eine weitere Hardwareplattform vorgehalten werden muss.

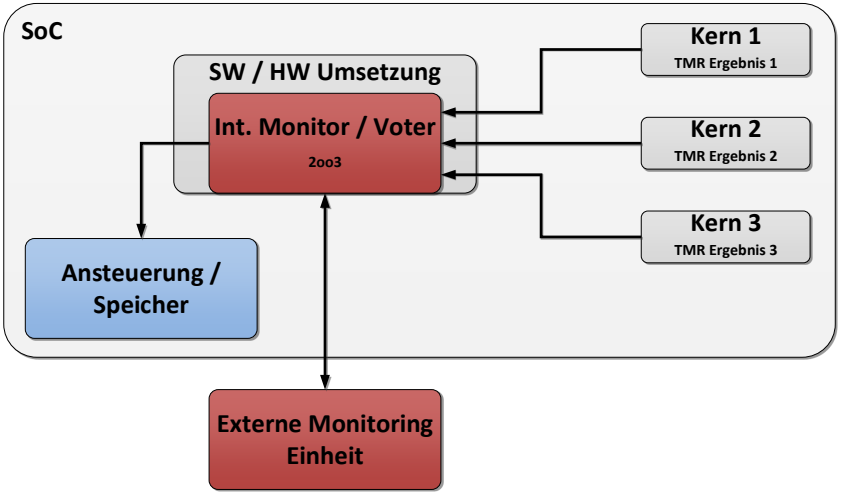


Abbildung 5.4: Erweiterung des TMR on Chip um einen externen Monitor

Durch eine Umsetzung innerhalb eines einzelnen MPSoC können jedoch zusätzliche Fehlerquellen auftreten, die zu einem Ausfall des gesamten TMR-Systems führen. Hierzu gehören beispielsweise die gemeinsame Spannungsversorgung sowie der Systemtakt. Diese zusätzlichen Fehlerquellen müssen gesondert durch eine externe Einheit überwacht werden. Dieser externe Monitor überwacht die grundlegenden Basisvoraussetzungen, die benötigt werden, um das MPSoC zu betreiben, wie etwa Spannungsversorgung und Takt. Des Weiteren kann aber auch eine Überwachung des Programmablaufs des MPSoC durchgeführt werden. Hierzu dient eine Challenge-Response-Abfrage, welche vom externen Monitor gestellt und vom internen Monitor beantwortet werden muss. Durch diese Kommunikation ist es möglich, die

Informationen über den Systemzustand des MPSoC an eine externe Einheit weiterzugeben. Diese besitzt die Möglichkeit, einen externen Reset (vgl. R_{ES} Abbildung 5.14 in Kapitel 5.2) des Multicores durchzuführen. Ebenso ist das Auslesen von spezifischen Werten des MPSoC über den externen Watchdog denkbar. Hierzu gehören beispielsweise Systemparameter wie die Chip Temperatur, die z.B. über eine JTAG Schnittstelle ausgelesen werden kann. In diesem Fall muss gesondert untersucht werden, in wie weit dieses Auslesen eine Beeinflussung für den Programmablauf auf dem Multicore darstellt (intrusiver vs. nicht-intrusiver Zugriff). In Abbildung 5.4 ist die Erweiterung des On-Chip Konzepts dargestellt. Hierbei kommuniziert der externe Monitor mit dem internen Monitor/Voter[3].

Die hier beschriebene Methode wurde auf dem *Midwest Symposium on Circuits and Systems (MWSCAS)* vorgestellt und in [San+14a] veröffentlicht.

5.1.2 Prototypische Umsetzung und Validierung

Zur Evaluation des in Kapitel 5.1.1 vorgestellten Redundanzkonzepts wird dies prototypisch realisiert, um den Ansatz auf realer Hardware validieren zu können. Es wird zur Umsetzung ein frei verfügbares Multicoresystem genutzt und in einem FPGA implementiert. Da der Basisaufbau auf der Diplomarbeit [95] basiert, finden sich einige technisch detailliertere Beschreibungen der HW/SW-Architektur in der Ausarbeitung.

5.1.2.1 Hardwareaufbau des Multiprozessor System-on-Chip (MPSoC)

Als Zielarchitektur wird für eine erhöhte Flexibilität ein FPGA mit Mehrkernprozessor herangezogen. Durch die Verwendung eines frei verfügbaren und quelloffenen Prozessors, bieten sich Möglichkeiten für tiefe Einblicke in die Architektur sowie Möglichkeiten zu Anpassungen der

[3] Die Beschreibung und der Aufbau des internen/externen Monitors (*Hierarchischer Watchdog*) wird in Kapitel 5.2 näher beschrieben.

Hardware, falls nötig. Im Gegensatz dazu kann in COTS Prozessoren kein Einfluss auf die Hardware genommen werden. Verwendet wird zur Realisierung des Redundanzansatzes ein Virtex 5 FPGA [96] von Xilinx mit einem SPARC V8 [97] Mehrkernprozessor. Die Umsetzung im FPGA erfolgt dabei als Quadcore-Prozessor mit einem zentralen Kommunikationsbus, low- und high-speed Peripherals sowie einer Möglichkeit zur Anbindung von externem DDR-Speicher. Der schematische Aufbau des Leon 3 Quadcore-Systems inklusive der zugehörigen Komponenten ist in Abbildung 5.5 dargestellt. Ebenso wird die Verbindung zur externen Monitoring Einheit verdeutlicht.

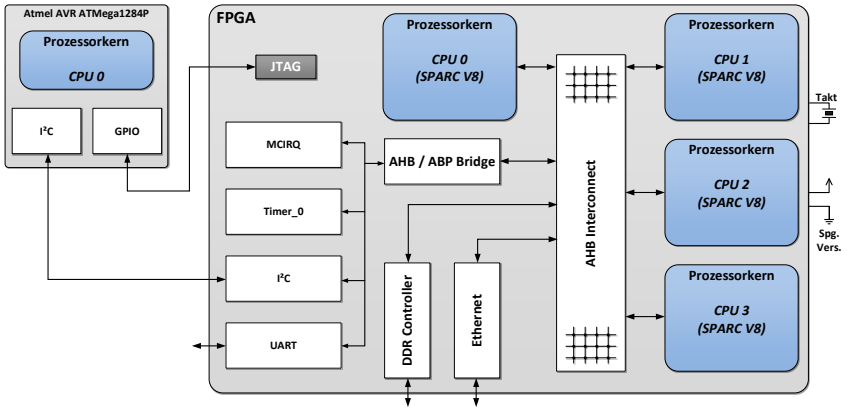


Abbildung 5.5: Realisierung des TMR Konzepts

Der Aufbau des MPSoC bedient sich der von Cobham Gaisler AB bereitgestellten Komponenten und der vorgeschlagenen Vernetzungstopologie (vgl. [98]). Der Kern des Aufbaus sind die 4 SPARC V8 Prozessorkerne, die an ein gemeinsames Advanced High-Performance Bus (AHB) System angebunden sind. Zusätzlich sind mit dem AHB der DDR Controller sowie das Ethernet Interface verbunden. Über eine Bridge, welche die Protokollumsetzung von AHB zu Advanced Peripheral Bus (APB) übernimmt, ist der Peripheral Bus für low-speed Peripherals erreichbar. An diesem APB Bus sind eine Inter-Integrated Circuit (I^2C)- und eine Universal Asynchronous Receifer Transmitter (UART)-Schnittstelle sowie ein Multicore Interrupt Controller (MCIRQ) und ein Timer

zu finden. Der DDR-Speicher wird für den Datenaustausch der Kerne gemeinsam verwendet. Dieses Minimal-Set an HW-Komponenten dient als Repräsentant für einen Multicoreprozessor und wird somit zur Validierung des TMR-Ansatzes verwendet.

Über die I²C-Schnittstelle wird ein einfacher externer 8-Bit Microcontroller angebunden, der als externe Monitoring Einheit verwendet wird und die Challenge-Response-Abfrage mit dem internen Monitor/Voter durchführt. Zusätzlich wird eine Verbindung zum JTAG-Anschluss des FPGA hergestellt. Über die JTAG-Schnittstelle können weitere Informationen des FPGA ausgelesen werden, so zum Beispiel die Versorgungsspannung und die Chip-Temperatur.

5.1.2.2 Softwarearchitektur

Die Softwarearchitektur bildet den Kern des Konzepts. Um das TMR-Konzept realisieren zu können, muss auf drei der vier Kerne dieselbe Funktion ausgeführt werden und deren Ergebnisse anschließend verglichen werden. Zur Vermeidung eines gegenseitigen Einflusses der Rechenkerne, werden die Programm- und Datenspeicher im DDR-Speicher manuell voneinander separiert, bzw. eine Partitionierung des Speichers durchgeführt. Eine Memory Management Unit (MMU) wird hier nicht eingesetzt, sondern es erfolgt eine manuelle Partitionierung, was aber keine Einschränkung darstellt. Entscheidend für das Konzept ist es, eine Möglichkeit zu schaffen, mit der die Berechnungen der einzelnen Kerne synchronisiert werden können. Des Weiteren wird ein Mechanismus benötigt, der einen atomaren, geschützten Zugriff auf Speicherstellen zulässt, während derer kein anderer Kern auf die Speicherstelle zugreifen kann. Hierfür wird ein *Mutex* implementiert.

Für die Umsetzung wird auf ein Betriebssystem verzichtet, um keine weitere Komplexität in die Softwarearchitektur einzubringen und die atomaren Zugriffe zu ermöglichen. Die Softwarearchitektur folgt somit einem Aufbau wie in Abbildung 5.6 dargestellt.

Auf jedem der vier Rechenkerne wird somit eine *Bare Metal* Applikation ausgeführt, die jeweils zyklisch durchlaufen wird. Prozessorkern *CPU 0* kommt dabei eine besondere Rolle zuteil. Applikation 0, welche auf Prozessorkern *CPU 0* ausgeführt wird, beinhaltet die Challenge-

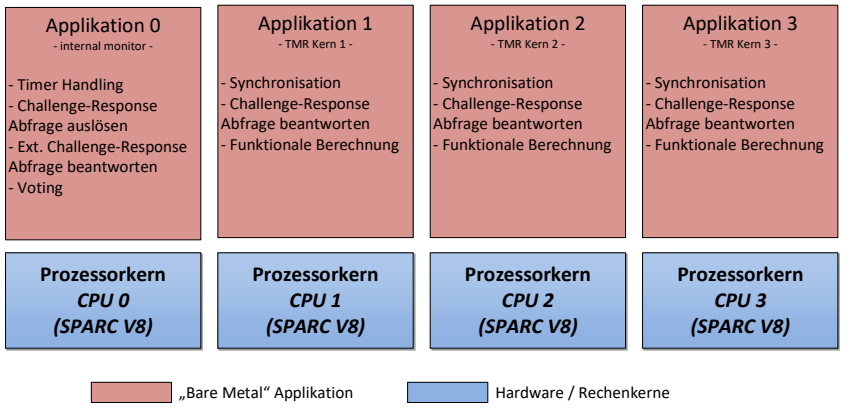
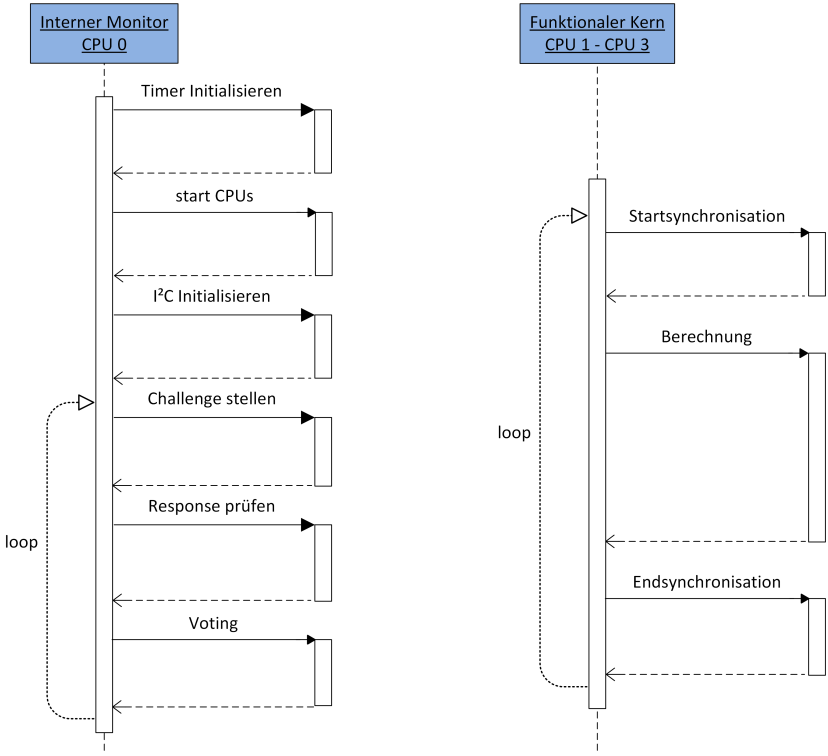


Abbildung 5.6: Softwarearchitektur des TMR Konzepts

Response-Abfrage der internen Rechenkerne sowie die Beantwortung der externen Abfrage. Zusätzlich wird die SoC Initialisierung und das TMR-Voting auf *CPU 0* durchgeführt. Die Rechenkerne *CPU 1 - CPU 3* beantworten die gestellten Anfragen und führen die eigentliche funktionale Berechnung durch und erzeugen somit die Ergebnisse, die beim Voting verglichen werden. Für die Beantwortung der gestellten Anfragen wird von *CPU 0* ein Timer gestartet, der zur Prüfung der Deadlines herangezogen wird. Diese Abfrage tritt mehrfach in einem Zyklus der Kerne *CPU 1 - CPU 3* auf, was eine feingranulare Zeitauflösung bewirkt. Der Softwareablauf auf *CPU 0* kann Abbildung 5.7(a) entnommen werden.

Speziell die Synchronisation der Rechenkerne *CPU 1 - CPU 3* zur Sicherstellung der Konsistenz der Daten sowie die Synchronisation mit *CPU 0* für das Voting stellen eine große Herausforderung dar. Die Synchronisation der funktionalen Kerne[4] untereinander ist entscheidend um sicherzustellen, dass diese auf den gleichen Eingangsdaten arbeiten. Sollte hier ein Versatz entstehen, würden zwangsweise die Ergebnisse, über welche der Mehrheitsentscheid gebildet wird, auseinanderlaufen. Zur Umset-

[4]Damit werden im Folgenden die Rechenkerne bezeichnet, welche die eigentlich funktionale Berechnung durchführen und damit die gewünschte Funktion erfüllen.



(a) Interner Monitor (b) Funktionaler Kern

Abbildung 5.7: Sequenzdiagramme der Software des TMR Konzepts.

zung wird eine Synchronisationsbarriere zu Beginn der zyklischen Funktion integriert, welche zusätzlich einen Time-Out integriert. *CPU 1 - CPU 3* zeigen damit ihre Bereitschaft und Verfügbarkeit für die nächste Rechenoperation. Der Time-Out wird dabei benötigt, um im Falle des Ausbleibens einer Rückmeldung mit den verbliebenen zwei Kernen eine Berechnung durchführen zu können. Beide verbliebene Rechenkerne melden das Ausbleiben eines Rechenkerns sowie die Nummer der CPU an *CPU 0* zurück, um dieses Ergebnis im Voting aussparen zu können. Eine Verzögerung eines Rechenkerns kann dabei beispielsweise beim Zugriff auf den gemeinsam genutzten Speicher entstehen, was eine zu große Verzögerung zur Folge hat. Im nächsten Zyklus kann der Rechenkern durch Bereitstellung der vorangegangenen Ergebnisse wieder in die Berechnung einsteigen und entsprechend beim Voting berücksichtigt werden. Tritt das Ausbleiben eines Kerns häufiger auf, so werden in *CPU 0* nach Erreichen einer maximalen Anzahl Maßnahmen ergriffen. Der Ablauf der Software, welcher auf den funktionalen Kernen umgesetzt wird, kann Abbildung 5.7(b) entnommen werden.

Parallel zur Software auf den Rechenkernen der Multicorearchitektur wird ein externer Monitor aufgesetzt, der die Funktionalität des internen Monitors auf Korrektheit überprüft. Hierzu wird auf einem Singlecore Microcontroller (Atmel ATMega1284P) eine Challenge-Response-Abfrage realisiert, die den internen Monitor/Voter überwacht. Hierzu gehört sowohl die Überprüfung der korrekten Antwort als auch das Timing. Der Ablauf, der für die Challenge-Response-Abfrage umgesetzt wird, ist im Sequenzdiagramm in Abbildung 5.8 verdeutlicht.

Ergänzend zur Challenge-Response-Abfrage wird eine JTAG State-Machine[5] implementiert, die über den JTAG-Port des FPGA Informationen über Spannungsversorgung und Chip-Temperatur ausliest. Die physikalische Verbindung zum JTAG-Port des FPGA wird über GPIO-Ports des Microcontrollers realisiert. Bei Abweichungen kann der externe Monitor den Reset des SoC auslösen und einen Neustart herbeiführen oder das SoC im Reset-Zustand halten. Da die Anbindung des externen Monitors per I²C realisiert ist, wird eine höhere zeitliche Granularität verwendet als beim internen Monitor.

[5] Die implementierte State-Machine orientiert sich dabei am Standard IEEE 1149.1 [99]

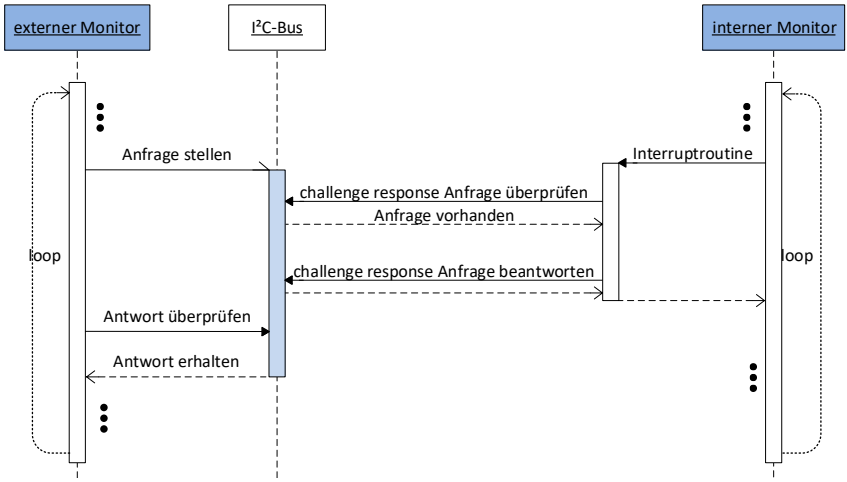


Abbildung 5.8: Sequenzdiagramm der Challenge-Response-Abfrage des externen zum internen Monitor

Zur Überprüfung der Funktionalität und zur Evaluation wird ein Mechanismus zur Fehlerinjektion integriert. Hierbei werden Fehler sowohl in Daten als auch im Timing bewusst herbeigeführt. Durch fehlerhafte Daten kann der Voting Mechanismus überprüft und auf Robustheit untersucht werden. Verzögerungen im Timing werden hingegen vom internen Monitor bei der Challenge-Response-Abfrage erkannt. Damit kann das Auslassen von Ergebnissen beim Voting überprüft werden. Ebenfalls werden Fehler im internen Monitor injiziert, welche vom externen Monitor erkannt werden können.

Eine Übertragbarkeit auf andere Architekturen, die beispielsweise als Application Specific Integrated Circuit (ASIC) vorliegen, ist mit kleineren Anpassungen der Software möglich, da keine Änderungen der Hardware benötigt werden. Durch die Verwendung eines FPGA-basierten Ansatzes entstehen somit keine Einschränkungen und dieser kann als Grundlage für weitere Implementierungen als Ausgangspunkt dienen.

5.1.3 Bezug zu den Fehlerbildern

Bei Verwendung eines Redundanz Ansatzes, wie dem hier beschriebenen TMR on Chip, liegt der Fokus auf der Erkennung und Korrektur von Fehlern in der Berechnung und der daraus resultierenden fehlerhaften weiteren Verarbeitung (Fehlerbild: *Fehlerhafte Berechnung/Daten/-Zugriffe*). In Bezug auf die in Kapitel 4.2 dargestellten Fehlerbilder stellt das vorgestellte Konzept eine Möglichkeit zum Umgang mit SEE/SEU dar. Die aus SEE/SEU resultierenden fehlerhaften Berechnungen werden durch die eingebrachte Redundanz vollständig erkannt und im Falle eines Einzelfehlers korrigiert. Im Gegensatz zu verfügbaren Konzepten wie Lock-Step (LS) kann bei einem *2-o-o-3* Ansatz ein Einzelfehler im laufenden Betrieb korrigiert werden. Dies bedingt keinen Neustart eines oder mehrerer Kerne oder ähnliche Maßnahmen. Analog zum Lock-Step-Konzept müssen gemeinsame Fehlerquellen, die einen Ausfall des gesamten MPSoC bedingen, gesondert überwacht werden. Abhängig von der Anwendung, die auf dem Multicore ausgeführt wird, kann eine solche Fehlerkorrektur im laufenden Betrieb notwendig sein, um die Berechnungen innerhalb der geforderten Zeitschranken abzuschließen.

Weiterhin ist durch die Integration eines Challenge-Response-Verfahrens eine zeitliche Überwachung sowie eine Abfrage der korrekten Funktionsweise der Kerne, welche die funktionale Berechnung durchführen, möglich. Fehlerhaftes Zeitverhalten kann somit frühzeitig erkannt und ausgewertet werden. Durch die korrekte Rückmeldung auf die Anfrage des internen Monitors/Voters, wird eine korrekte Ausführung der entsprechenden Funktionsteile nachgewiesen und damit dem Fehlerbild *fehlende Funktionsausführung* entgegengewirkt.

5.2 Multi-Level Watchdog

Durch die Integration mehrerer Applikationen in einen Mehrkernprozessor ergeben sich zusätzliche Anforderungen hinsichtlich Segregation und Synchronisation. Werden beispielsweise zwei Applikationen unterschiedlicher Fahrzeugdomänen[6] in einem MCP integriert, so kann eine Reaktion auf Fehler je nach Anwendung unterschiedlich ausfallen. Zu den möglichen Reaktionen auf einen solchen Fehler gehört beispielsweise der Reset-Zustand bzw. der Neustart. Da in einem Mehrkernprozessor ein Reset des gesamten Prozessors beide darauf ausgeführten Applikationen betreffen würde, ergibt sich eine Notwendigkeit weitere Mechanismen vorzusehen, die dazu dienen, Fehler auf einzelnen Kernen zu erkennen und getrennt, also pro Kern, reagieren zu können. Durch das Überführen eines Kerns in einen Reset-Zustand, kann der fehlerhafte Kern isoliert und somit eine Propagation des Fehlers im gesamten Multicore verhindert werden.

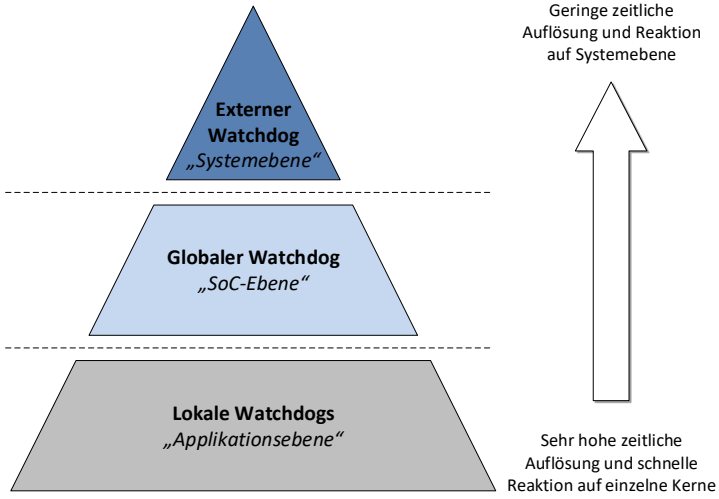


Abbildung 5.9: Hierarchischer Aufbau der beteiligten Watchdogs

[6] Beispielsweise eine Kombination der Fahrzeugdomänen Infotainment, Body, Chassis und Power-Train

5.2.1 Konzept

Speziell zur zeitlichen Überwachung der Softwareausführung in einem Multicore wird eine Methode benötigt, die es erlaubt, sowohl feingranular (auf Rechenkern-Ebene) als auch grobgranular (auf SoC-Ebene) Timing-Informationen zu sammeln und Maßnahmen einzuleiten. Die Hierarchiepyramide mit den zugehörigen Ebenen der Überwachung ist in Abbildung 5.9 zu finden.

Zur Erkennung von feingranularem Timing-Verhalten wird ein Lokaler Watchdog pro Rechenkern verwendet. Dieser ist für die Überwachung des korrekten Timing-Verhaltens des Kerns zuständig. Hierbei wird überwacht, ob die Ausführung der Software auf diesem Kern entsprechend der Anforderungen ausgeführt wird. Dazu können mehrere Referenzzeitpunkte herangezogen werden, zu denen ein Rücksetzen des Watchdog Timers ausgelöst wird. Da diese Information sich ausschließlich auf einen der Rechenkerne bezieht, entspricht diese nicht dem Zustand des Gesamtsystems. Daher wird diese Information von jedem lokalen Watchdog zyklisch an eine übergeordnete Einheit (Globaler Watchdog) kommuniziert, welche auf Basis aller verfügbaren Informationen den Zustand des Gesamtsystems ableitet und diese wiederum an eine externe Einheit (Externer Watchdog) kommuniziert. Der schematische Aufbau des hierarchischen Watchdogs ist in Abbildung 5.10 dargestellt. Es wird das Zusammenspiel der lokalen Watchdogs mit dem globalen Watchdog und die zeitliche Abfolge verdeutlicht.

Pro Rechenkern ist somit ein lokaler Watchdog vorhanden, welcher den zeitlichen Ablauf des zugehörigen Rechenkerns kontrolliert und überwacht. Zu definierten Zeitpunkten muss von jedem Rechenkern ein Rücksetzen des lokalen Watchdog Timers ausgelöst werden. Parallel dazu wird vom globalen Watchdog eine Abfrage der lokalen Watchdogs ausgelöst. Jeder lokale Watchdog ist mit einer Möglichkeit zum Setzen eines Interrupts und einem Reset des jeweiligen Rechenkerns ausgestattet. Der globale Watchdog ermöglicht einen globalen Reset. Der globale Reset ermöglicht es, eine Reset-Anfrage für einen einzelnen Kern im zugehörigen lokalen Watchdog anzufordern, welcher diesen dann zurücksetzt. Die lokalen Watchdogs werden vom globalen Watchdog zyklisch in einer definierten Reihenfolge abgefragt. In Abbildung 5.10 ist dies als Tokens dargestellt. Ebenso besitzt der globale Watchdog

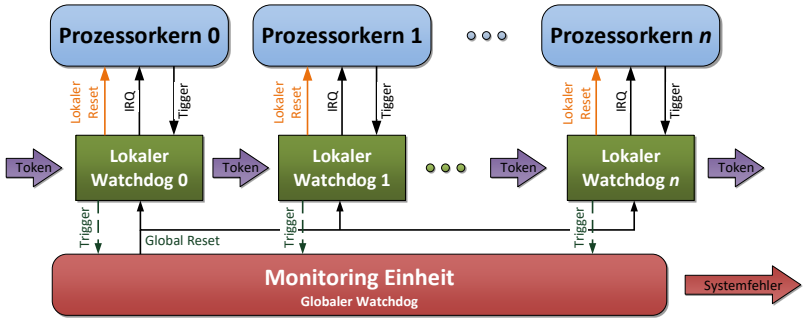


Abbildung 5.10: Konzept und logischer Aufbau des hierarchischen Watchdogs

eine Möglichkeit zur Signalisierung eines Systemfehlers. Diese Signalisierung wird innerhalb des Chips weiterverarbeitet oder nach außen an eine externe Einheit kommuniziert, die anschließend Maßnahmen treffen kann. Der Ablauf, welcher im globalen Watchdog realisiert ist, wird in Abbildung 5.11 als Unified Modelling Language (UML) Aktivitätsdiagramm dargestellt. Der globale Watchdog startet mit der Abfrage des ersten lokalen Watchdogs. Hierbei wird eine Anfrage (Challenge) an den lokalen Watchdog (LWD) gestellt und ein interner Timer gestartet. Innerhalb einer definierten Zeitspanne muss der lokale Watchdog diese Anfrage beantworten. Kommt keine Antwort in der gesetzten Zeitspanne, wird sowohl der lokale Watchdog wie auch der zugehörige Kern zurückgesetzt und neu gestartet. Kommt die Rückmeldung innerhalb der gesetzten Zeitspanne, wird der Timer zurückgesetzt. In beiden Fällen wird anschließend mit dem nächsten lokalen Watchdog in der Kette fortgefahren. Der globale Watchdog legt alle gesammelten Informationen ab und zieht so Schlüsse zum Systemzustand. Tritt ein Fehlverhalten eines Kernes vermehrt auf, so kann dieser permanent in einen sicheren Zustand (z.B. Reset-Zustand) überführt werden. Die Historie wird vom globalen Watchdog gesichert und ausgewertet.

Auf der Seite der lokalen Watchdogs, wird eine ähnliche Prozedur durchgeführt, die in Abbildung 5.12 dargestellt ist. Hierbei wird jedoch die Anfrage nicht an einen anderen Watchdog gesendet, sondern direkt an den zugehörigen Rechenkern. Die Reaktion, die ausgelöst wird, wenn

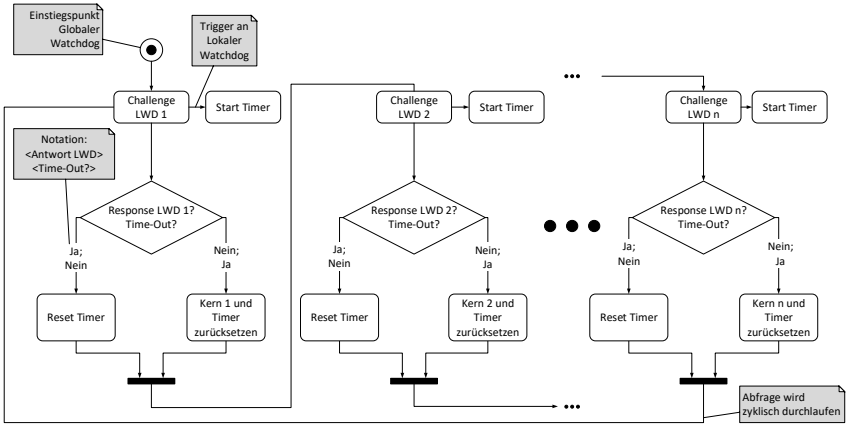


Abbildung 5.11: Aktivitätsdiagramm des globalen Watchdogs

das Zurücksetzen des Timer vor Ablauf der Deadline ausbleibt, ist zweistufig. Bei Ablauf der ersten Deadline wird im Rechenkern vom lokalen Watchdog ein Interrupt ausgelöst, um zu überprüfen, ob die Interrupt Routine noch richtig ausgeführt wird und den Watchdog zurücksetzen kann. Ist auch dies nicht erfolgreich, wird der Rechenkern zurückgesetzt, sobald keine Rückmeldung innerhalb der gewünschten Zeitspanne (bei Erreichen der zweiten Deadline) ankommt oder die Anfrage falsch beantwortet wurde. Tritt dieser Fall ein oder wird ein Interrupt ausgelöst, so wird der globale Watchdog informiert.

Überträgt man diesen schematischen, logischen Aufbau auf ein MPSoC, erhält man die in Abbildung 5.13 dargestellte Architektur. Eine Umsetzung des Konzepts kann komplett in Hardware erfolgen, um zusätzliche Komplexität der Software zu vermeiden. Lediglich die ISR muss in Software realisiert werden. Eine entscheidende Grundvoraussetzung ist es, eine Möglichkeit zu schaffen, über welche einzelne Kerne zurückgesetzt werden können, ohne dass die Ausführung und der Ablauf der verbleibenden Kerne gestört werden. Hierzu gehört auch die Initialisierung von gemeinsam genutzten Komponenten, die nicht ausgeführt werden darf, da sonst sämtliche Rechenkerne, welche diese Einheit benutzen, in ihrer Ausführung gestört werden. Eine *Minimal-Initialisierung* muss

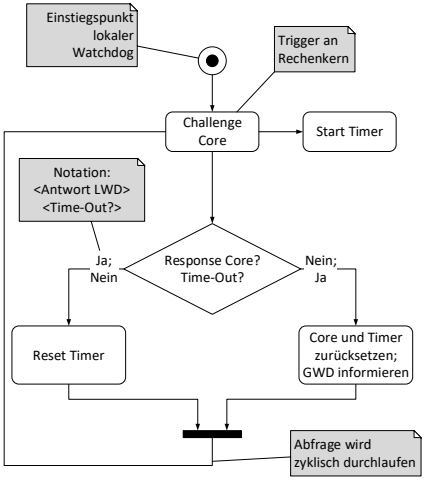


Abbildung 5.12: Aktivitätsdiagramm der lokalen Watchdogs

jedoch durchgeführt werden, um zu vermeiden, dass noch alte Daten/ Informationen des fehlerhaften und anschließend neu gestarteten Kerns vorhanden sind. Beim Starten des Rechenkerns wird zusätzlich ein Selbsttest (Build-In Self Test (BIST)) durchgeführt, welcher eine Abfrage durch den lokalen Watchdog beinhaltet. Des Weiteren werden Mechanismen benötigt, welche eine Re-Synchronisation des neu gestarteten Kerns zulassen.

Abbildung 5.13 verdeutlicht, dass der globale Watchdog in Hardware, hier in einer rekonfigurierbaren Architektur, realisiert ist. Durch die direkte Anbindung an das interne Interconnect kann eine Kommunikation mit geringer Latenz mit den lokalen Watchdogs garantiert werden. Ebenso ermöglicht dieser Aufbau einen Zugriff auf die Register der Rechenkerne sowie eine Verbindung nach außen, über welche die Meldung eines Systemfehlers und weitere Statusmeldungen übertragen werden können. Die Möglichkeit zum Zurücksetzen eines einzelnen Kerns wird direkt in diesem umgesetzt.

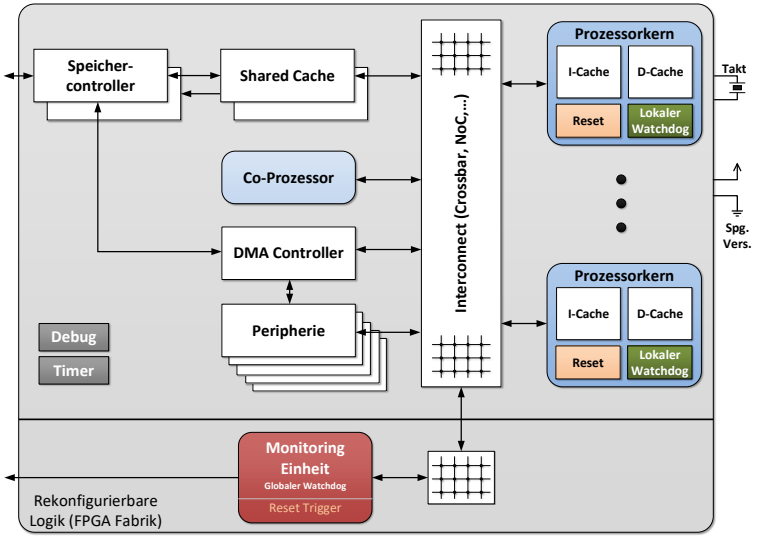


Abbildung 5.13: Hierarchischer Watchdog in einem MPSoC

Bei der Verwendung von Mehrkernprozessoren, ist eine Konstellation, in der Software (z.B. Betriebssystem) über mehrere Kerne verteilt ausgeführt wird, denkbar. Diese Variante der Software-Architektur bedingt, dass das Reset-Handling anders gehandhabt werden muss, als wenn ausschließlich eine Applikation pro Kern ausgeführt wird. Wenn beispielsweise eine Partition über zwei Kerne verteilt verarbeitet wird, muss der globale Watchdog darüber in Kenntnis gesetzt sein, um bei einem Fehler in einem Kern auch den anderen zurücksetzen zu können und die gesamte Partition neu zu starten.

Der Reset eines Kerns kann als Teilmenge des Partitions-Reset dargestellt werden. Dieser wiederum als Teilmenge des internen System-Reset, welcher wiederum eine Teilmenge des externen System-Reset bildet. Diese Hierarchie kann mathematisch mit Hilfe von Mengen dargestellt werden.

Die folgenden Mengen werden dazu herangezogen, welche in Abbildung 5.14 dargestellt sind:

- Reset der Rechenkerne: R_{Cn} mit $n = \{1, ..., AnzahlKerne\}$
- Reset einer Partition: R_{Pm} mit $m = \{1, ..., AnzahlPartitionen\}$
- Interner Reset des SoC: R_{IS}
- Externer Reset des SoC: R_{ES}

Dabei gelten die folgenden Beziehungen:

$$R_{Cn} \subseteq R_{Pm} \subseteq R_{IS} \subseteq R_{ES} \tag{5.3}$$

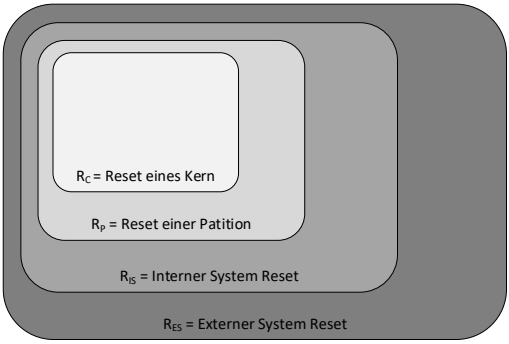


Abbildung 5.14: Übersicht der unterschiedlichen Reset als Mengendarstellung

Sobald ein Reset auf der höchsten Ebene (externer System-Reset) durchgeführt wird, werden sämtliche interne Reset deaktiviert und das komplette MPSoC zurückgesetzt bzw. neugestartet. Im Falle der Verwendung von Virtualisierung (vgl. Kapitel 2.4) kann der Reset einer Partition auch über einen Hypervisor ausgelöst werden. In diesem Fall hat der Hypervisor Kenntnis über die pro Partition verwendeten Kerne und kann damit den Reset der Rechenkerne auslösen.

5.2.2 Prototypische Umsetzung und Validierung

Die Realisierung des Multi-Level Watchdog Konzepts erfolgt auf Basis der in Kapitel 5.1.2 beschriebenen Hardware-Architektur. Ergänzend

wird das SoC im Wesentlichen durch zwei HW-Anpassungen erweitert. Es wird zum einen ein globaler Watchdog mit Reset Controller benötigt, welcher von jedem lokalen Watchdog[7] an den Kernen die Informationen sammelt und jeden Kern separat zurücksetzen kann und eine Möglichkeit bietet, einzelne Kerne neu zu starten. Der Controller wird ergänzend im FPGA als HW-Komponente realisiert. Dieser wird als IP-Core im FPGA implementiert und erhält dabei eine Verbindung zum zentralen AHB-Interconnect und zum Registerinterface der einzelnen Kerne. Zum anderen werden die Rechenkerne selbst um eine Möglichkeit zum Zurücksetzen erweitert. Es wird eine Möglichkeit geschaffen, welche es ermöglicht, die Rechenkerne und deren Kontext zurückzusetzen.

Um eine korrekte Rücksetzung umsetzen zu können, müssen einige Komponenten der Rechenkerne zurückgesetzt bzw. in ihren Ausgangszustand zurückversetzt werden. Hierzu gehören unter anderem die Pipeline sowie die Caches, um vermeiden zu können, dass veraltete Daten bei einem Neustart verarbeitet werden. Diese Maßnahmen werden durch die Erweiterung an den Leon3 Kernen umgesetzt. Der Reset selbst wird vom Reset Controller ausgelöst.

Der Reset Controller bietet zusätzlich eine Schnittstelle über die der interne Monitor die einzelnen Reset der Kerne anfordern kann. Zusätzlich können Informationen der einzelnen lokalen Watchdogs abgefragt werden. Diese Schnittstelle ist als AHB-Schnittstelle realisiert und kann somit einfach angesprochen werden. Die erweiterte Hardware-Architektur ist in Abbildung 5.15 dargestellt. Die Validierung des Ansatzes erfolgt über einfache Fehlerinjektionen in den Prozessorkernen. Diese wird in Form von Warteschleifen realisiert, in welcher der lokale Watchdog nicht innerhalb der vorgegebenen Deadline zurückgesetzt wird. Dies wird vom globalen Watchdog aufgenommen und gespeichert. Zusätzlich kann über die AHB-Schnittstelle diese Information vom internen Monitor (vgl. *CPU 0* Abbildung 5.15) ausgelesen werden. Ergänzend kann bei Erkennung von beispielsweise Berechnungsfehlern (wie in Kapitel 5.1 beschrieben) ein Reset vom internen Monitor/Voter ausgelöst werden, der vom Reset-Mechanismus am Rechenkern umgesetzt wird.

[7] Als lokale Watchdogs werden die in der SPARC-Architektur vorhandenen Watchdogs verwendet.

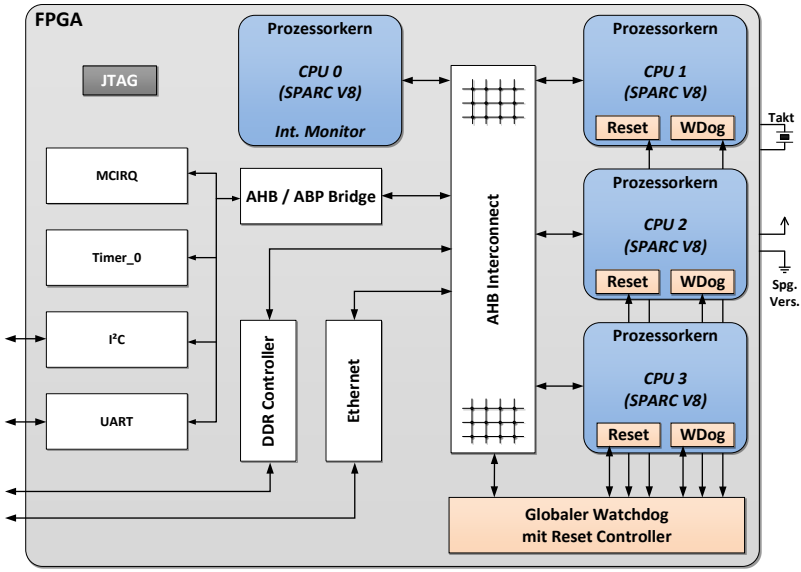


Abbildung 5.15: Erweiterung der Hardwarearchitektur um den hierarchischen Watchdog

5.2.3 Bezug zu den Fehlerbildern

Die Anwendung und Integration eines hierarchischen Watchdogs dient der zeitlichen Überwachung der Rechenkerne. In Bezug auf die Fehlerbilder, die in Kapitel 4.2 beschrieben sind, bedeutet dies hauptsächlich einen Beitrag zum Fehlerbild *Hohe Ausführungslatenz*. Es können zeitliche Fehler in der Ausführung auf fein- und grobgranularem Level erkannt und darauf reagiert werden. Zudem werden die Informationen in einer höheren Instanz gesammelt, um eine Übersicht auf System-Ebene zu erhalten. Ebenso dient die Möglichkeit des Reset der einzelnen Kerne, Partitionen und des SoC zur Fehlerisolation. Ein Kern, der in einen Resetzustand versetzt wird, hat zumindest keine weiteren negativen Auswirkungen auf andere gemeinsam genutzte Ressourcen. Jedoch muss der globale Watchdog dafür sorgen, dass keine Beeinträchtigungen der anderen Rechenkerne durch fehlende Daten entstehen. Zusätzlich kann der hierarchische Watchdog dem Fehlerbild *„Fehlende Funktionsausführung“* entgegnen. Ausbleibendes Rücksetzen des Watchdogs lässt Rückschlüsse auf eine fehlende Ausführung von Funktionsteilen zu. Dies betrifft nicht nur die eigentliche Funktionsausführung, das Konzept kann ebenso auf Interrupts ausgeweitet werden. Beim Auslösen eines Interrupts wird ein Timer gestartet, welcher von der ISR zurückgesetzt wird. Ein ausbleibendes Rücksetzen lässt darauf schließen, dass die ISR nicht korrekt innerhalb einer bestimmten Zeitspanne ausgeführt wird. Speziell bei Echtzeitsystemen, die eine zyklische Ausführung besitzen, werden Interrupts verwendet. Die Interrupts besitzen dabei eine definierte Zeitspanne, welche benötigt wird, um die ISR zu starten. Diese wohldefinierte Zeitspanne kann für einen Interrupt-Watchdog verwendet werden. Diese wird mit einem gewissen Sicherheitsaufschlag als obere Schranke definiert und dient somit als Basis für den Interrupt-Watchdog.

5.3 Monitoring für Master-Komponenten im Multicoreprozessor

Als besonders kritisch können Bus-Master Komponenten innerhalb eines Multicoreprozessors angesehen werden. Diese haben durch ihre Master Eigenschaft mehr Möglichkeiten Zugriffe auf Komponenten durchzuführen, auch wenn diese nicht dem gewünschten Fall entsprechen. Zusätzlich kann durch Master-Komponenten die Auslastung des Interconnect besonders beeinflusst werden. Dies gilt für die meisten Master-Komponenten und wird im Folgenden anhand eines DMA-Controllers diskutiert, da dieser zusätzlich direkte Zugriffe auf den Speicher ausführen kann. Der Ansatz der Überwachung, der vorgestellt wird, ist jedoch auch für andere Bus-Master übertragbar. Zur Sicherstellung der Übersichtlichkeit wird im Folgenden der Fokus auf einen DMA-Controller gelegt.

Eine besonders kritische Komponente in einem Mehrkernprozessor stellt der Speichercontroller dar. Aufgrund des Zugriffs sowohl von Rechenkernen als auch von DMA-Controllern, kann eine Auslastung am Limit des Speichercontrollers schnell zu einem Fehler im System führen. Hierbei ist zu beachten, dass der DMA-Controller seinerseits auch von mehreren Einheiten genutzt werden kann. Ebenso wie der Speichercontroller, wird der DMA-Controller von den Rechenkernen, aber auch von weiteren Bus-Mastern im Prozessor, alloziert. Hierzu gehören beispielsweise Peripherals, die einen hohen Datendurchsatz besitzen - z.B. Ethernet oder PCI(e). Eine Überauslastung des DMA-Controllers kann darauffolgend zu einem undefinierbaren Verhalten am Speichercontroller und somit zu möglichen Fehlern des Systems führen. Die potentielle Überauslastung des Speichercontrollers spiegelt sich vor allem in höheren Latenzen sowie möglichen Race-Conditions wieder. Eine schematische Darstellung des Konzepts ist in Abbildung 5.16 abgebildet.

Die Problematik entsteht, wenn über Peripherals unerwartet häufige Anfragen an den DMA-Controller gestellt werden. Dies erfolgt zumeist ohne Kenntnis bei den Rechenkernen, die somit verzögert werden. Auf Basis dieser Problematik wird ein Konzept vorgestellt, welches eine Möglichkeit zur Überwachung eines DMA-Controllers darstellt

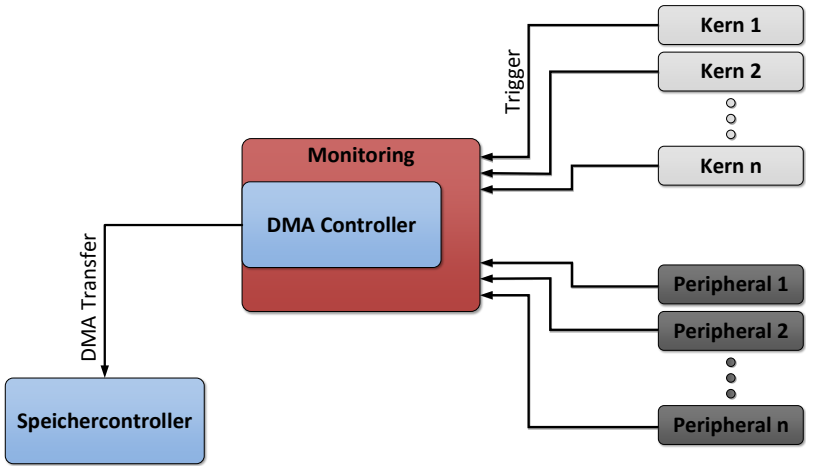


Abbildung 5.16: Schematische Darstellung des DMA-Monitoring

und weitere Möglichkeiten zur Integration von Mechanismen erlaubt. Ebenso wird auf die Übertragbarkeit des Ansatzes für weitere Komponenten im SoC eingegangen. Das Konzept basiert auf der Masterarbeit [Bir13].

5.3.1 Konzept

Die Auslastung des DMA-Controllers (vgl. Formel (5.5)) ist ein entscheidender Indikator für das Zugriffsverhalten auf den Speichercontroller. Basierend auf diesem Parameter kann im laufenden Betrieb die verbleibende Kapazität geschätzt und der Systemzustand evaluiert werden. Speziell beim Überschreiten von maximal zulässigen Grenzwerten der Auslastung und der daraus resultierenden Zugriffe auf den Speichercontroller, müssen entsprechende Gegenmaßnahmen in Betracht gezogen werden. In einem solchen Szenario wird, soweit möglich, die Quelle der Zugriffe identifiziert und der Fehler somit eingegrenzt und im Ursprung behandelt. Eine Überauslastung des DMA-Controllers kann zu einer Verzögerung wichtiger anderer Systemfunktionen führen, die hieraus

resultierend ihre Funktion nicht mehr in der geforderten Zeitschranke erfüllen können.

Nicht nur die obere Schranke der maximal zulässigen Auslastung kann bei einer Überwachung von Interesse sein. Auch eine Unterschreitung einer minimalen Auslastung kann Rückschlüsse auf eine ausbleibende Anfrage einer Komponente zulassen. Betrachtet man eine zyklisch aufgerufene Funktion, die bei jedem Aufruf einen DMA-Transfer auslöst, so fällt eine ausbleibende Anfrage durch die Überwachung auf. Es können hierdurch weitere Mechanismen greifen, die speziell die Komponente, deren Zugriff ausbleibt, untersuchen.

Die Auslastung des DMA-Controllers kann dabei durch die zeitliche Länge des angeforderten Transfers bzw. durch die Aufrufhäufigkeit beeinflusst werden. Selbst zeitlich kurz andauernde Anfragen, die häufig angefordert werden, können einen zusätzlichen Aufwand durch Kontextwechsel hervorrufen. Dieses Verhalten wird in Abbildung 5.17 verdeutlicht. Hierbei ist im oberen Zeitdiagramm die Ausführung eines einzelnen zeitlich lange andauernden Transfers gezeigt und im unteren die Ausführung mehrerer „kurzer“ Transfers.

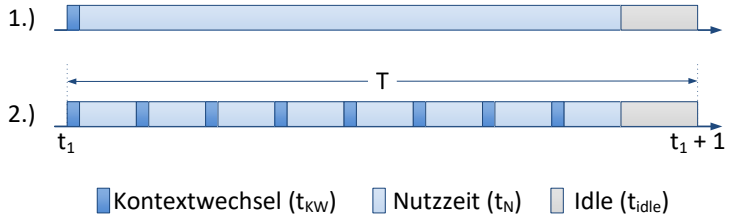


Abbildung 5.17: Darstellung der zeitlichen Abläufe auf dem DMA-Controller

Beide sind jeweils für eine Periode T und den Zeitraum t_1 bis $t_1 + 1$ aufgezeigt. Das Verhältnis (vgl. (5.4)) von *benötigter Zeit für Kontextwechsel* (t_{KW}) zu *Nutzzeit* (t_N) des DMA-Controllers beschreibt die erreichte *Effizienz* η_{DMA}.

In einer Periode T mit n als laufende Nummer des Aufrufs innerhalb von T gilt:

$$\eta_{DMA} = \frac{t_{KW}}{t_N} = \frac{\sum t_{KWn}}{\sum t_{Nn}} \tag{5.4}$$

Es muss also differenziert betrachtet werden, wie die Nutzung des DMA-Controllers ausgelegt werden kann. Für den hier betrachteten Fall der Überwachung ist diese Auslegung insofern interessant, da sich die Grenzen für die Auslastung identifizieren lassen.

Betrachtet man nun im Zeitintervall T den Anteil der *Kontextwechsel plus Nutzzeit* im Verhältnis zur *gesamten Periode T* erhält man die prozentuale Auslastung ϱ_{DMA} des DMA-Controllers:

$$\varrho_{DMA} = 100 * \frac{\sum t_{KWn} + \sum t_{Nn}}{T} \tag{5.5}$$

Über (5.5) kann zwar die gesamte Auslastung des DMA-Controllers bestimmt werden, jedoch nicht die Quellen der Aufrufe. Zur Bestimmung der Aufrufhäufigkeit einzelner Kanäle (Kanal $\widehat{=}$ eine Ressource, die den Transfer auslöst) wird ein weiterer Mechanismus eingesetzt. Das Überwachen der Aufrufe der einzelnen Kanäle kann dazu genutzt werden eine fehlerhafte Ressource zu identifizieren, die übermäßig häufig auf den DMA-Controller zugreift. Weiterhin kann das Monitoring verwendet werden um Budgets einzuhalten und zu überwachen, die den einzelnen Kanälen zugeordnet sind. Abhängig von der konkreten Implementierung des DMA-Controllers wird diese Überwachung nicht unterstützt. Es wird ein Zähler/Timer benötigt, der pro Periode die Aufrufe durch die einzelnen Kanäle aufzeichnet und auswertet, sowie entsprechend Maßnahmen ergreifen kann.

Bei Architekturen mit einem rekonfigurierbaren Anteil (beschrieben in Kapitel 2.3.2.2) können solche Zähler/Timer sehr effizient in Hardware realisiert werden. Hierfür wird ein DMA-Controller entsprechend erweitert. Alternativ kann bei DMA-Controllern, wie in Kapitel 2.3.1 dargestellt, diese Funktionalität nicht in Hardware integriert werden. Hierfür ist eine softwarebasierte Lösung von Nöten. Diese erlaubt eine höhere Flexibilität, die jedoch zu Lasten der Effizienz geht. Nach jedem Kontextwechsel wird zunächst eine Routine ausgeführt, welche die Kanalzähler iteriert und somit

die Aufrufhäufigkeit durch einen einzelnen Kanal aufzeichnet. Bei der Routine handelt es sich um eine in der Ausführungszeit sehr kurze Routine (t_{ct}), die in Abbildung 5.18 im Gesamtablauf dargestellt ist.

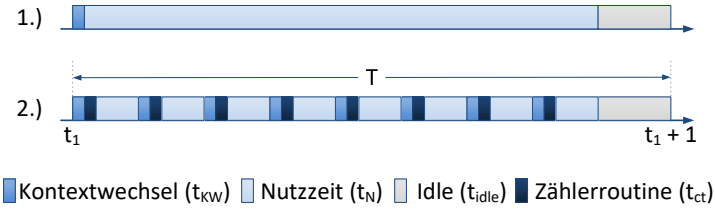


Abbildung 5.18: Darstellung der zeitlichen Abläufe auf dem DMA-Controller inkl. Zählerroutine

Die Berechnung der Effizienz des DMA-Controllers verändert sich, basierend auf (5.6), zu:

$$\eta_{DMA,ct} = \frac{t_{KW} + t_{ctn}}{t_N} = \frac{\sum (t_{KWn} + t_{ctn})}{\sum t_{Nn}} \tag{5.6}$$

Dieser Overhead kann durch eine Implementierung in Hardware, welche vollständig parallel abläuft, verhindert werden, ist jedoch nicht in allen Architekturen umsetzbar.

Bei Verwendung eines softwarebasierten Ansatzes lassen sich noch weitere Aspekte umsetzen, die der funktionalen Sicherheit zuträglich sind. Eine Möglichkeit stellt die Umsetzung eines Voters für eine 2oo3 Architektur (2-out-of-3, z.B. Triple Modular Redundancy, vgl. Kapitel 3.1) dar. Hierbei hat der DMA-Controller die Möglichkeit auf den Speicher über den Speichercontroller zuzugreifen. Die Berechnungsergebnisse der Rechenkerne können so aus dem Speicher geholt und verglichen werden. Das Mehrheitsergebnis wird anschließend wiederum in den Speicher geschrieben oder direkt an ein Peripheral weitergereicht. Die Umsetzung des Voters im DMA-Controller verhindert einen Overhead auf den eigentlichen Rechenkernen und implementiert gleichzeitig die Umsetzung auf einer diversitären Architektur. Der schematische Aufbau ist in Abbildung 5.19 dargestellt.

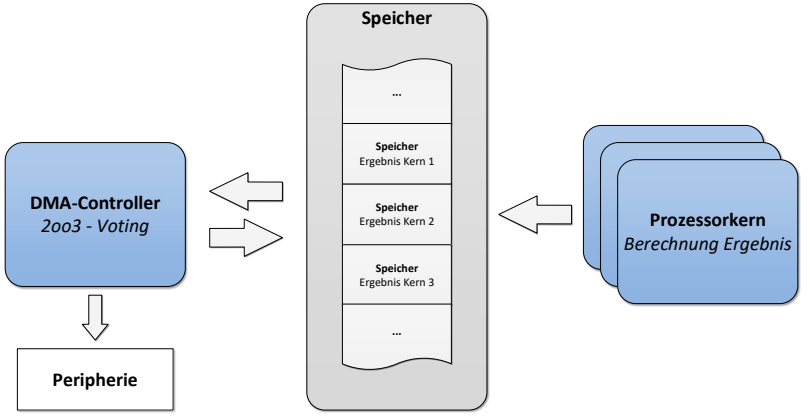


Abbildung 5.19: Darstellung des schematischen Aufbaus des DMA 2oo3 Ansatzes

Die schematische, logische, zeitliche Abfolge des Mehrheitsvergleichs ist in Abbildung 5.20 dargestellt. Der zeitliche Versatz in der Übertragung der Ergebnisse der einzelnen Rechenkerne dient der Steigerung der Übersichtlichkeit, kann aber im realen System auch parallel erfolgen. Die Berechnung der sicherheitskritischen Funktion wird auf den Prozessorkernen (vgl. Abb. 5.19) durchgeführt. Die (Zwischen-) Ergebnisse werden im Speicher in einem bestimmten Bereich abgelegt und ein Flag gesetzt, welches signalisiert, dass das Ergebnis zur Verfügung steht. Die Berechnungen könnten beispielsweise eine zyklische Regelschleife oder Sensordatenverarbeitung sein. Der DMA-Controller holt sich in periodischen Abständen die Ergebnisse aus dem Speicher. Die Ergebnisse der drei Rechenkerne werden anschließend auf dem DMA-Controller verglichen und auf eine Mehrheit überprüft. Das mehrheitlich vorhandene Ergebnis wird, basierend auf der Entscheidung und des Szenarios, wieder für weitere Berechnungen in den Speicher geschrieben oder direkt einem Peripheral zur Kommunikation, auf beispielsweise einem Bus-System (z.B. CAN), übergeben.

Die hier beschriebene Methode wurde auf dem SAE World Congress vorgestellt und in [Bap+15] veröffentlicht.

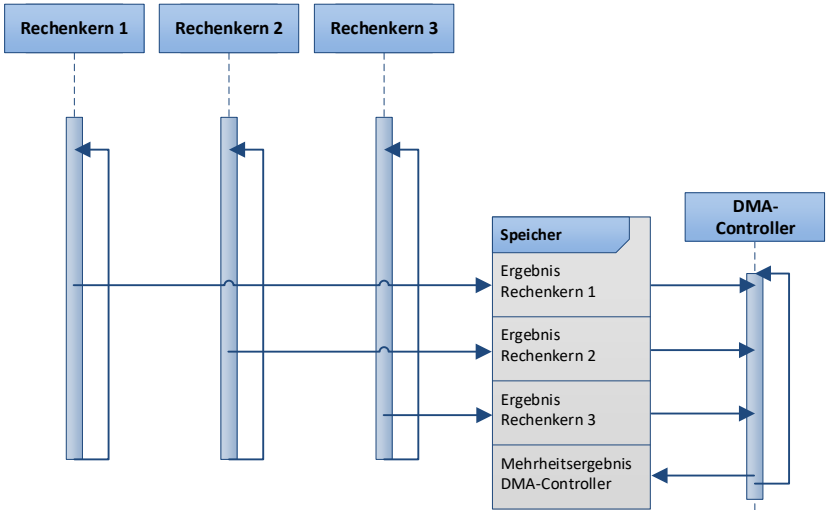


Abbildung 5.20: Darstellung des logischen zeitlichen Ablaufs des DMA 2oo3 Ansatzes

5.3.2 Prototypische Umsetzung und Validierung

Um den Ansatz des Monitoring-Wrappers zu validieren wird eine prototypische Implementierung des DMA-Monitoring aufgesetzt. In diesem Kontext wird eine Freescale[8] Multicorearchitektur verwendet. Zum Einsatz kommt ein Freescale i.MX 6Quad [100] mit 4 ARM Cortex A9 Rechenkernen. Dem Aufbau und der prototypischen Implementierung liegt die Masterarbeit [Bir13] zu Grunde. Dieser Ausarbeitung können technisch detailliertere Ausführungen entnommen werden. Ein Überblick wird im Folgenden gegeben.

5.3.2.1 Hardwareaufbau

Für die prototypische Umsetzung wird ein Commercial Off The Shelf (COTS)-Prozessor in Form eines ASIC verwendet, der keine Ände-

[8]Mittlerweile NXP Semiconductors N.V.

rungen an der Hardware zulässt. Das Konzept lässt sich somit rein in Software umsetzen. Nachfolgend wird die Prozessorarchitektur kurz vorgestellt. Für detailliertere Informationen sei auf [100] verwiesen.

Beim i.MX 6Quad handelt es sich um einen Vierkernprozessor basierend auf der ARM Cortex A9 [101] Architektur. Im SoC sind noch weitere Videoverarbeitungseinheiten, Peripherals, Security-Erweiterungen und Debug Einheiten vorhanden. Für die Implementierung des beschriebenen Ansatzes wird der Fokus auf die Cortex A9 Rechenkerne und den DMA-Controller[9] gelegt. Speziell der DMA-Controller ist der Kern der prototypischen Umsetzung. Die Architektur des SoC ist in Abbildung 5.21 analog der Darstellung aus [100] veranschaulicht und die Kernkomponenten für diesen Ansatz hervorgehoben.

Aus Abbildung 5.21 wird deutlich, dass der SDMA-Controller sowohl mit dem AXI-Interconnect, dem Speichercontroller sowie den gemeinsam genutzten Peripherals verbunden ist. Durch diesen Aufbau erhält der SDMA einen Zugriff auf das innere des SoC sowie über die gemeinsam genutzten Peripherals zur Außenwelt. Der SDMA-Controller selbst beinhaltet einen kleinen RISC Prozessor, welcher vom Nutzer programmiert werden kann. Diese Programmierbarkeit des SDMA sowie der Aufbau des SoC werden im Folgenden verwendet um das Konzept umzusetzen.

5.3.2.2 Softwarearchitektur

Für die Implementierung werden Funktionen in Software sowohl auf den Applikationskernen als auch dem SDMA benötigt. Hierbei muss eine Kommunikationsmöglichkeit zwischen den beiden Teilen ermöglicht werden um gemeinsame Daten auszutauschen. Hierfür wird ein Austausch über gemeinsam genutzte Speicherbereiche eingerichtet, auf die sowohl SDMA als auch Applikationskerne Zugriff haben. Eine Übersicht der Softwarearchitektur und der benötigten Funktionen, welche auf den Applikationskernen bzw. dem SDMA ausgeführt werden, ist in Abbildung 5.22 dargestellt. Bei den in dieser Darstellung als *Script*

[9] In der Freescale i.MX 6Quad Architektur als *SDMA* benannt. Wenn im weiteren Verlauf dieses Kapitels von DMA die Rede ist, wird Bezug auf den Smart Direct Memory Access (SDMA) Controller der i.MX 6Quad Architektur genommen.

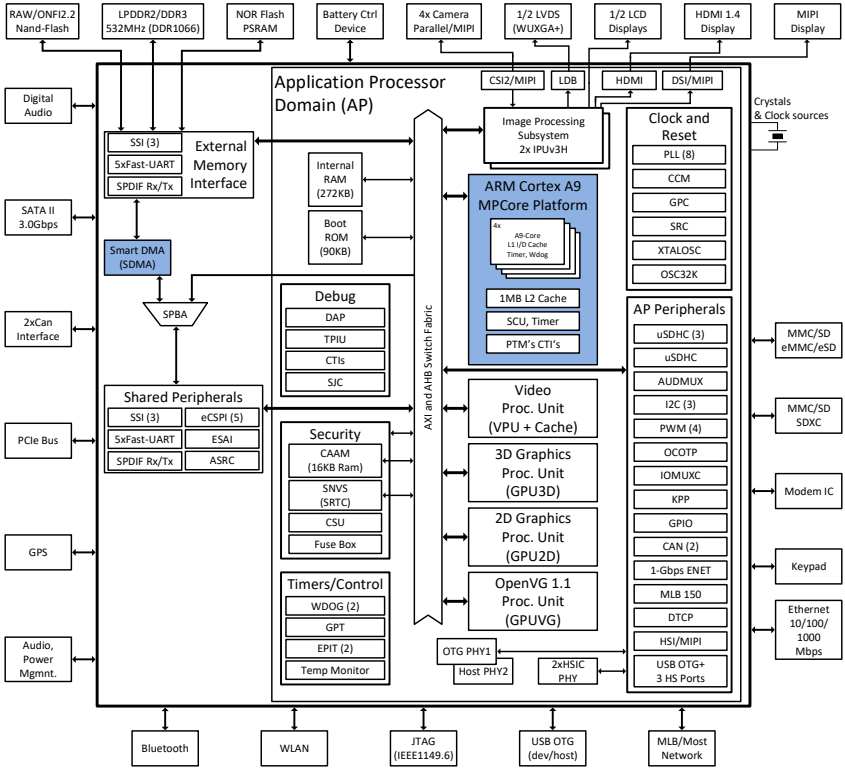


Abbildung 5.21: Architektur des Freescale i.MX 6Quad analog der Darstellung aus [100]

bezeichneten Funktionen handelt es sich um Software, welche auf dem SDMA ausgeführt wird.

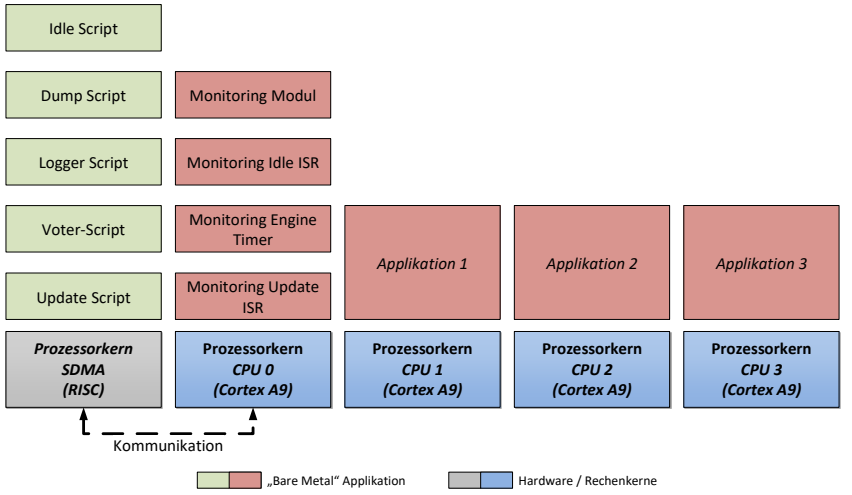


Abbildung 5.22: Softwarearchitektur des DMA-Monitoring

Auf Seiten der Applikationskerne wird eine API definiert. Diese umfasst u.a. Funktionen zum Initialisieren, Starten und Auslesen des Monitorings auf dem SDMA. Ebenso wird ein Applikationskern für Messungen und die Evaluation des Konzepts eingesetzt, dieser könnte aber in einem Produktivsystem für weitere Applikationen, z.B. als interner Monitor der Informationen des gesamten SoC sammelt und auswertet, genutzt werden. Für die Umsetzung spielen die Scripte auf dem SDMA eine entscheidende Rolle und werden daher im Folgenden kurz beschrieben:

- **Idle Script:** Beim Idle Script handelt es sich um die Funktion, welche immer ausgeführt wird, wenn keine DMA-Anforderung vorhanden ist. Das Script wird mit der niedrigsten Priorität ausgeführt, um zu gewährleisten, dass jede Anfrage für einen DMA-Transfer vorgezogen und somit ausgeführt wird. Zusätzlich wird das Idle Script als Watchdog eingesetzt. Ein integrierter Zähler muss vor dem Überlauf zurückgesetzt werden, anderenfalls wird ein Interrupt an die Applikationskerne gesetzt. Der Zähler wird

ebenfalls für die Messung der Auslastung des SDMA-Controllers genutzt.

- **Update Script:** Um die Messergebnisse des Monitoring in den Hauptspeicher zu kopieren und somit für die anderen Kerne, speziell für den internen Monitor, verfügbar zu machen, wird das Update Script umgesetzt. Die Ergebnisse werden vom Update Script in den Hauptspeicher kopiert. Des Weiteren setzt dieses Script die SDMA internen Daten zurück. Die Hauptaufgabe des Script ist somit die Kommunikation mit dem Monitoring Modul auf Seiten der Applikationskerne.
- **Logger Script:** Über das Logger Script werden die Aufrufe der einzelnen DMA-Kanäle aufgezeichnet. Zur Umsetzung wird dieses Script vor jeder Ausführung eines DMA-Kanals ausgeführt. Über einen Zähler werden die Aufrufe pro Kanal inkrementiert und im internen Speicher des SDMA abgelegt.
- **Dump Script:** Da der SDMA-Controller direkt mit den gemeinsamen Peripherals verbunden ist, bietet sich die Möglichkeit, Informationen bzw. Daten direkt über diese Peripherals an eine externe Einheit weiterzugeben. Hierbei kann als externe Einheit beispielsweise ein externer Monitor oder ein Aktor herangezogen werden. Die Kommunikation wird mittels des Dump Script realisiert.
- **Voter Script:** Zur Demonstration und zur Übertragung des TMR-Konzepts (vgl. Kapitel 5.1) auf eine andere Architektur, wird der SDMA-Controller als Voter für einen Mehrheitsvergleich genutzt. Der Vergleich der Ergebnisse (Voting) wird auf dem SDMA-Controller im Voting Script realisiert. Das Mehrheitsergebnis wird dann über die Peripherals extern verfügbar gemacht. Die Berechnung wird auf den Applikationskernen durchgeführt und die Ergebnisse im Hauptspeicher abgelegt. Der SDMA-Controller liest die Ergebnisse und führt im Voter Script den Vergleich durch.

Unter Verwendung der vorgestellten Scripte und den Auswertungen auf den Applikationskernen werden verschiedene Messungen durchgeführt und im Folgenden vorgestellt.

5.3.2.3 Messungen

Die Auslastung des SDMA-Controllers (ϱ_{SDMA}) wird mit Hilfe des Idle Scripts ermittelt. Das Idle Script wird immer ausgeführt, wenn keine Anfragen für DMA-Transfers vorhanden sind. Ein integrierter Zähler wird während der Ausführung des Scripts inkrementiert und der Zählerwert im Speicher abgelegt. Über den Zählerwert (Cnt) und die Taktfrequenz des SDMA-Controllers (f_{SDMA}) kann die Auslastung bzw. die Leerlaufzeit des Controllers nach folgender Formel berechnet werden:

$$\varrho_{SDMA} = 1 - \frac{Cnt}{\Delta T * f_{SDMA}} \tag{5.7}$$

Insgesamt wird die Berechnung über eine Kombination von Funktionen auf dem SDMA-Controller und Funktionen auf einem Applikationskern durchgeführt. Zunächst wird über den *Monitoring Engine Timer* auf dem Applikationskern das Update Script auf dem SDMA aufgerufen, um die Werte für das Update Script zurückzusetzen. Die ausgeführte Sequenz ist in Abbildung 5.23 als Sequenzdiagramm mit Funktionen auf Applikationskern und SDMA-Controller dargestellt.

Zur Validierung und Evaluation der SDMA-Auslastung wird ein Script umgesetzt, welches parametrierbar eine Last des Controllers erzeugt. Das Script wird jede Millisekunde ausgeführt. Basierend auf dem Last-Parameter wird eine längere Laufzeit erzeugt, was sich im Zählerstand niederschlägt. Ein Vergleich der gemessenen zur errechneten Auslastung wird in Tabelle 5.1 vorgenommen.

Die Ergebnisse zeigen eine Abweichung der gemessenen Auslastung von der errechneten von 1-2%. Diese Variation resultiert aus Kontextwechseln, die beim Wechseln der Scripte auftreten. Die Kontextwechsel sind in Abbildung 5.24 zur Verdeutlichung dargestellt.

Auf Basis dieser Darstellung wird deutlich, dass zunächst zu Beginn der Ausführung sowie vor der Ausführung des Idle Scripts ein Kontextwechsel stattfinden muss. Dieser Overhead ist statisch und durch die Architektur bedingt. Die Messung der Auslastung des SDMA-Controllers dient im weiteren Verlauf als Grundlage für weitere

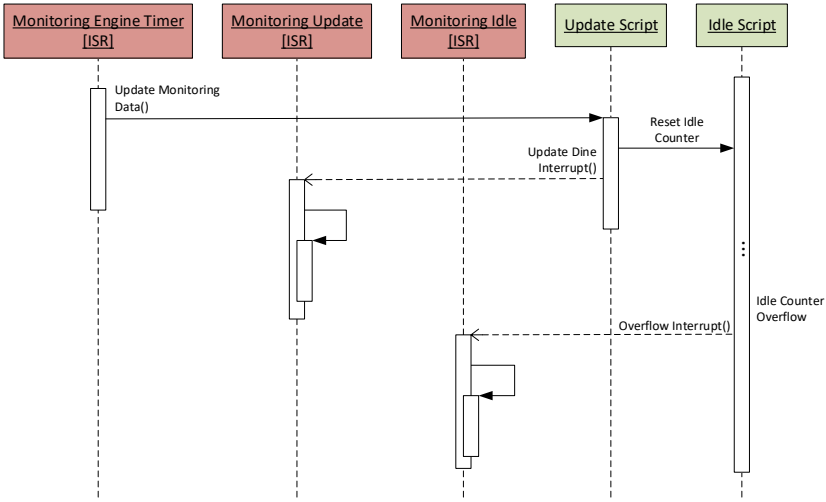


Abbildung 5.23: Sequenzdiagramm zur Messung der Auslastung des SDMA-Controllers

Tabelle 5.1: Messergebnisse der SDMA-Auslastung

#	Berechnete Auslastung (%)	Test Script Länge (cycles)	Gemessene Auslastung (%)
1	0	0	2
2	10	660000	11
3	25	16500000	26
4	50	33000000	51
5	75	49500000	76

Messungen, die zeigen, wie sich der SDMA-Controller unter höheren Lasten sowie mit variierender Datenlänge verhält. Hierfür werden künstliche Daten erzeugt, die in ihrer Länge variiert werden können. Als Beispielapplikation wird eine redundante Berechnung anhand eines TMR-Schemas ausgeführt. Die drei Ergebnisse können entsprechend in ihrer Länge angepasst werden. Basierend auf einer festgelegten Länge der Daten wird die SDMA-Auslastung bei Verwendung der TMR-Applikation gemessen. Diese Messung wird anschließend mit variierender Länge wiederholt. In Abbildung 5.25 werden die Ergebnisse dargestellt. Die horizontale Achse zeigt dabei die Datenmenge, während auf der primären vertikalen Achse SDMA-Last und auf der sekundären vertikalen Achse die Ausführungszeit der TMR-Applikation abgebildet sind.

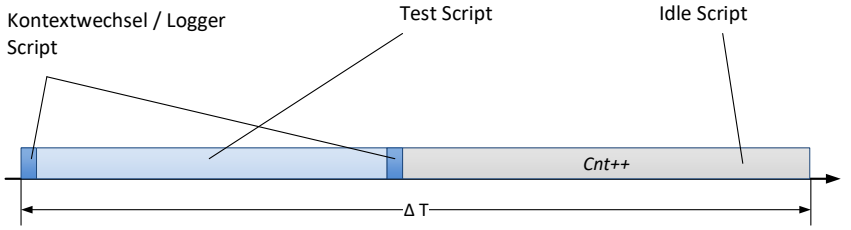


Abbildung 5.24: Zeitliche Abfolge der Messung der SDMA-Auslastung und Darstellung der Kontextwechsel

Bei Anlegen einer Tangente in den Ursprung wird deutlich, dass es ab einer Länge von ca. *2500 words* eine bemerkbare Abweichung gibt. Um die Messung zu validieren ist ebenfalls in Abbildung 5.25 die gemessene Ausführungszeit des Scripts, gemessen vom Applikationskern, abgebildet.

Es wird deutlich, dass auch bei dieser Messung eine Abweichung von der Tangente zu beobachten ist. Mit den dargestellten Messungen auf Basis des vorgestellten Aufbaus, zeigt sich, wie präzise diese Messungen durchgeführt werden können. Zur weiteren Untersuchung dieser Nicht-Linearität wird eine weitere Messung mit größerer Datenlänge durchgeführt.

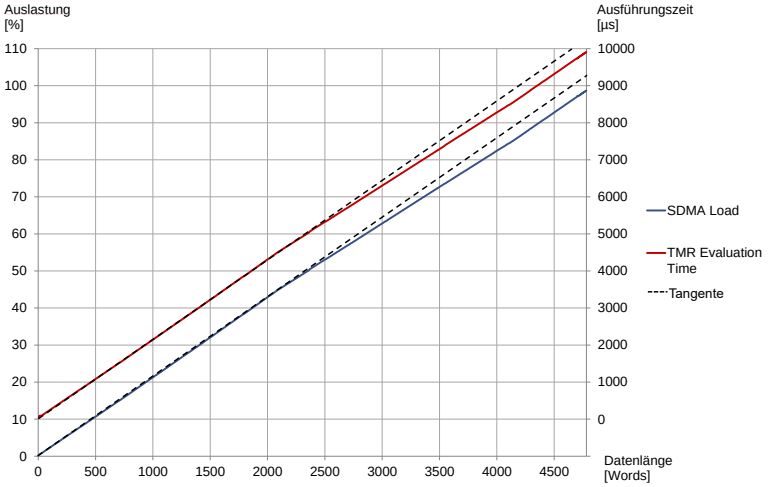


Abbildung 5.25: SDMA-Auslastung und Ausführungszeit bei variierender Datenlänge

Bei genauerer Betrachtung des Graphen wird deutlich, dass zunächst eine Abweichung und im weiteren Verlauf eine erneute Annäherung an die Tangente beobachtet werden kann. Ein vergrößerter Ausschnitt dessen ist in Abbildung 5.26 dargestellt.

Zur genaueren Untersuchung der Abweichung wird die Differenz der Messwerte zur Tangente gebildet. Aus der Darstellung dieser Differenz kann abgelesen werden, dass äquidistante Maxima im Abstand von 4096 Words vorhanden sind. Diese Differenz ist in Abbildung 5.27 dargestellt.

Um systematische Fehler ausschließen zu können, werden die folgenden Parameter variiert:

- die Startadresse im Speicher
- die Zeitintervalle des Monitorings
- die Anzahl an Messungen
- die inkrementellen Schritte der Erhöhung der Datenlänge
- sowie die Speicherposition der Daten

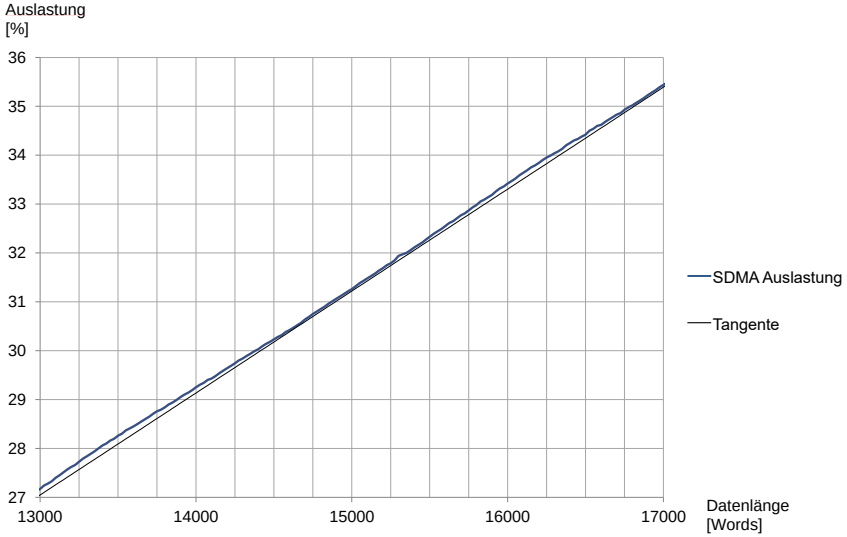


Abbildung 5.26: Vergrößerung der Abweichung der Messergebnisse der Lastmessung bei Variation der Datenlänge

Da nur bei einer Änderung der Speicherposition eine Änderung des Verhaltens auftritt, liegt die Vermutung nahe, dass dieses Verhalten vom Speichercontroller oder vom Speicher selbst hervorgerufen wird und nicht vom Monitoring-Konzept resultiert.

Auf Basis der durchgeführten Messungen zeigt sich nicht nur, dass die Auslastung des SDMA-Controllers überwacht werden kann, sondern auch detaillierte Informationen über das SoC und dessen Architektur

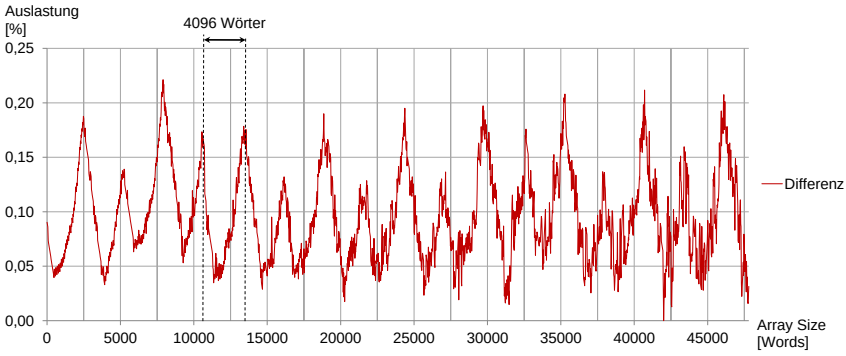


Abbildung 5.27: Differenz der Messwerte zur Tangente bei der Messung der Auslastung des SDMA-Controllers

generiert werden können. Ebenso kann das Verhalten des SoC bei hoher DMA-Auslastung evaluiert werden, um notwendige Grenzwerte zu identifizieren und um Fehler zu vermeiden.

Weitere Messungen, Evaluationen der Plattform und deren Beschreibung können [Bap+15] sowie [Bir13] entnommen werden. Durch die Umsetzung des Konzepts in Software, ist eine Portierung auf andere Architekturen mit Anpassungen möglich. Je nach verwendetem DMA-Controller kann die Umsetzung eingeschränkt sein.

5.3.3 Bezug zu den Fehlerbildern

Durch die Anwendung des DMA-Monitorings können Rückschlüsse auf den Systemzustand und die Verwendung/Auslastung von Ressourcen gemacht werden. Speziell der Zustand des Speichercontrollers und die daraus resultierende Latenz können identifiziert werden und dienen im Weiteren der Steuerung des Gesamtsystems. Diese Steuerung bezieht sich auf die Reaktion durch Zugriffsbeschränkungen (z.B. durch Budgetierung), um in einen stabilen und deterministischen Zustand zu gelangen. Dabei wirkt diese Methode gegen das Fehlerbild *„Hohe Ausführungslatenz“*, hervorgerufen durch eine zu hohe Auslastung des Speichercontrollers oder die fehlerhafte Nutzung von Services (hier des

Direct Memory Access), und unterstützt eine frühzeitige Erkennung eines Fehlverhaltens. Die Monitoring-Ergebnisse (*Aufrufhäufigkeit und Auslastung*) können dann auf Systemlevel mit weiteren Informationen fusioniert werden, um einen Überblick über das Gesamtsystem zu erhalten. Des Weiteren können die Ergebnisse des Mehrheitsentscheids dem Fehlerbild *„Fehlerhafte Berechnung/Daten/Zugriffe“* entgegenwirken und Fehler in der Berechnung eines Kerns korrigieren. Durch das 2oo3 Voting kann selbst bei einer fehlerhaften Berechnung, also einem abweichenden Ergebnis, ein korrektes Ausgangsdatum garantiert werden. Zur Erkennung und Korrektur von mehr als einer fehlerhaften Berechnung ist ein 2oo3 Ansatz nicht mehr ausreichend. Jedoch kann das Voting im DMA-Controller auf einen *n*-out-of-*m* Ansatz erweitert werden, solange genügend Rechenkerne zur Verfügung stehen, welche die Berechnungsergebnisse liefern.

6 Methoden zur sicheren Ressourcennutzung und Funktionsmigration

Die in Kapitel 5 beschriebenen Methoden bilden Möglichkeiten zur Erkennung und zur Korrektur von Fehlern. Dabei werden unter anderem Fehler erkannt, welche durch die gemeinsame Nutzung von Ressourcen entstehen können. Um jedoch die Fehlerhäufigkeit möglichst gering zu halten, werden Methoden zur gemeinsamen Nutzung von Ressourcen benötigt.

Hierzu werden Anpassungen der geteilten Ressource, der Software und/oder der Hardware benötigt, die ggf. eine Möglichkeit zur Überwachung bereits integrieren.

Um den steigenden Anforderungen an die Verfügbarkeit von eingebetteten Systemen in sicherheitskritischen Anwendungen gerecht zu werden, werden zukünftig Fail-Operational Systeme benötigt. Im Gegensatz zu aktuellen Fail-Safe Systemen wird dabei mindestens eine verringerte Funktionalität im Fehlerfall aufrechterhalten. Einige Konzepte hierfür sind bereits aus der Luftfahrt bekannt. Diese bedingen alle eine Replikation von Hardware, was teuer ist und aus Platzgründen nicht immer realisiert werden kann.

Nachstehend wird hierfür eine Methode zur dynamischen Migration einer degradierten Funktion in einem heterogenen MPSoC vorgestellt und diskutiert.

6.1 Hardware-Virtualisierung

Im Gegensatz zu den bisher vorgestellten Methoden zur Überwachung (Monitoring) eines Multicores wird im Folgenden eine Methode zur Nutzung von geteilten Komponenten vorgestellt. Eine Möglichkeit zur sicheren gemeinsamen Nutzung von Ressourcen innerhalb von Prozessoren stellt die Virtualisierung dar. Hierbei wird verschiedenen (zumeist) virtuellen Maschinen oder Partitionen jeweils die Verfügbarkeit einer Komponente vorgetäuscht. Typischerweise werden die Zugriffe aus den einzelnen Partitionen im Hypervisor weiterverarbeitet. Dabei können Prioritäten oder Scheduling Mechanismen zum Einsatz kommen, die einen deterministischen Zugriff sicherstellen. Die in Kapitel 2.4 dargestellten Konzepte können hierfür verwendet werden, bringen aber jeweils einen beträchtlichen Overhead in Software mit sich, der zumeist nicht transparent für den Softwareentwickler ist. Oftmals müssen die Applikationssoftware und die Basissoftware modifiziert werden, um eine effiziente Implementierung zu erhalten.

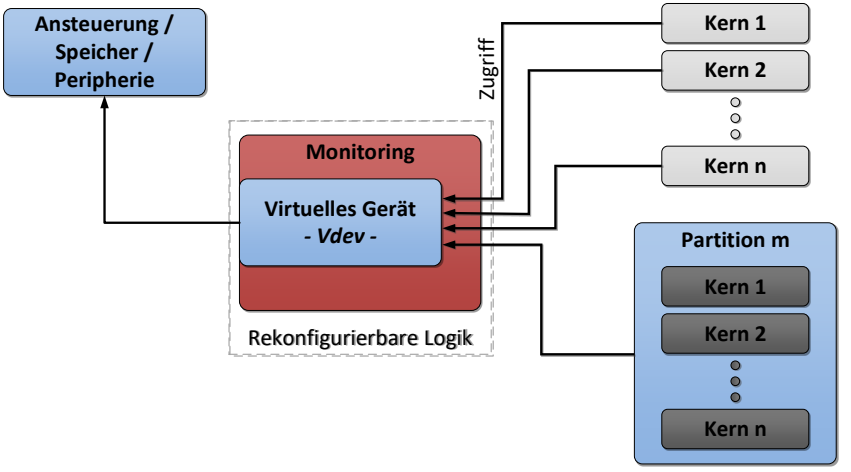


Abbildung 6.1: Konzept der HW-Virtualisierung

Im Kontext von heterogenen Multicorearchitekturen, die einen rekonfigurierbaren Logikanteil besitzen, kann dieser Overhead der Soft-

ware entfallen. Zusätzlich ist in einer Vielzahl von Prozessoren für eingebettete Systeme keine Hardwareunterstützung für Virtualisierung vorhanden, was eine Software Lösung oftmals sehr ineffizient macht. Das Konzept der Hardware-Virtualisierung wird in Abbildung 6.1 dargestellt und in den folgenden Kapiteln erläutert und diskutiert. Das vorgestellte Konzept basiert auf der Masterarbeit [Sto15].

6.1.1 Konzept

Um Virtualisierung auf einer heterogenen Architektur mit Hardwareunterstützung bzw. rein in Hardware umzusetzen, kann man sich der rekonfigurierbaren Logik bedienen. In Abbildung 6.2 ist die schematische Architektur sowie die Kommunikation von Rechenkernen bis zur Peripherieeinheit dargestellt.

Dabei wird deutlich, dass die typischerweise in Software realisierte Virtualisierungsschicht (vgl. Kapitel 2.4, Hypervisor) in die rekonfigurierbare Logik verschoben wird. Durch die direkte Anbindung an das zentrale Interconnect, an welches die Rechenkerne sowie die Peripherals angebunden sind, kann auch die Virtualisierungsschicht sehr effizient angebunden werden. Für jeden Rechenkern wird in der Virtualisierungsschicht ein virtuelles Gerät zur Verfügung gestellt, auf welches der jeweilige Rechenkern/die jeweilige Partition[1] exklusiven Zugriff besitzen. Der Rechenkern/die Partition interagiert mit dem gleichen Interface, das bei der physikalischen Komponente vorhanden ist. Für die Zugriffe wird somit keine Änderung der Software notwendig. Es handelt sich beim Zugriff lediglich um einen anderen Adressbereich, der angesprochen wird. Bei der Ausführung der Software und der beabsichtigten Berechnung entsteht kein zusätzlicher Overhead in Form von benötigter Rechenzeit.

Der in Abbildung 6.2 dargestellte HW-Virtualisierungsblock stellt dabei die Menge der instanziierten virtuellen Geräte sowie die zugehörige kombinatorische Logik und weitere Steuerkomponenten dar. Als Schnittstelle zu den Rechenkernen dient eine Standardschnittstelle,

[1] Bei Verteilung einer Applikation oder eines Betriebssystems über mehr als einen Kern wird hier von einer Partitionierung und damit von einer Partition gesprochen.

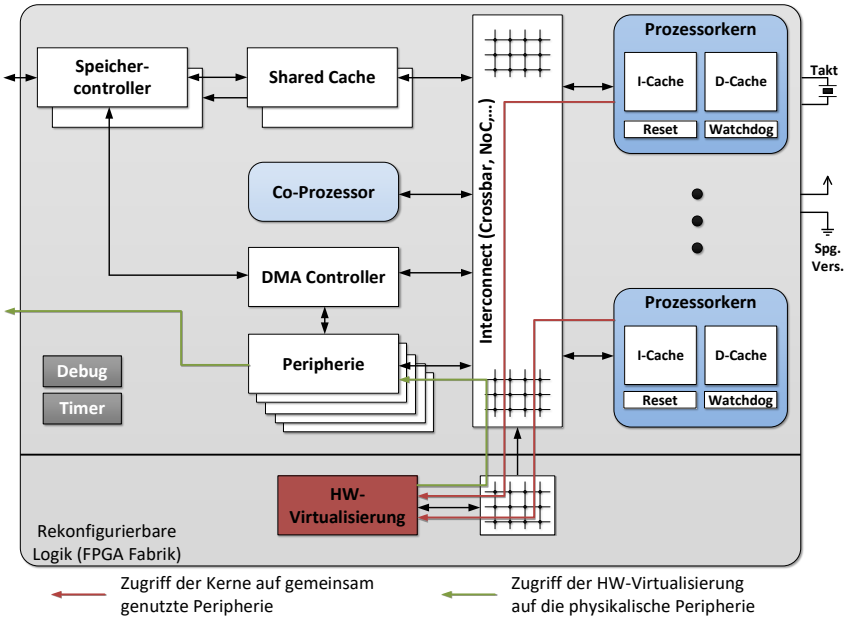


Abbildung 6.2: Schematische Architektur der Hardware-Virtualisierung und der Kommunikation in einem MPSoC

welche je nach verwendeter Architektur flexibel ausgewählt werden kann.

In Abbildung 6.3 ist der innere Aufbau der HW-Virtualisierungsschicht dargestellt. Basierend auf der Anzahl der Rechenkerne (n_{Kerne}) oder Partitionen (n_{part}), welche sich ein physikalisches Gerät teilen, wird die zugehörige Anzahl an virtuellen Geräten (virtual Devices n_{Vdev}) instantiiert ($n_{Vdev} = n_{Kerne}$ bzw. $n_{Vdev} = n_{Part}$).

Zu jedem virtuellen Gerät gehört ein Interface (*Virt.-Interface_n*), welches für den Datenaustausch vom Rechenkern/der Partition mit der HW-Virtualisierung verwendet wird. Diese virtuellen Geräte sind zusätzlich mit dem Virtualisierungsmanager (VMNG) verbunden, welcher den Zugriff auf das physikalische Gerät regelt und durchführt. Der Virtualisierungsmanager besitzt ein Interface für jedes virtuelle Gerät (*I-*

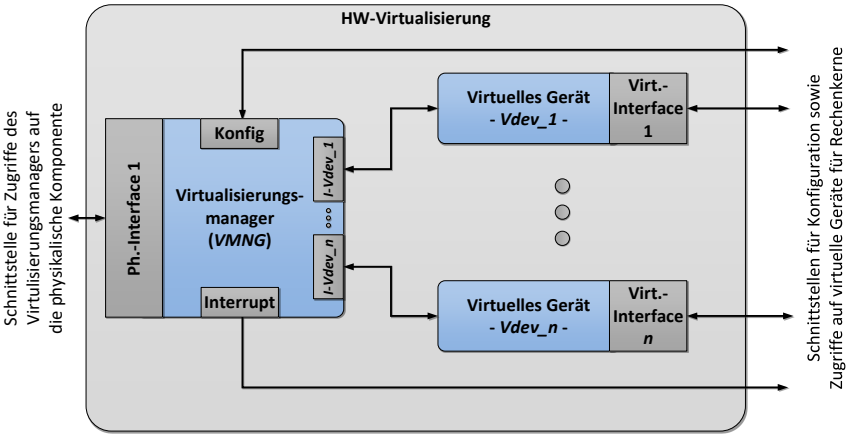


Abbildung 6.3: Aufbau der Virtualisierungsschicht in der rekonfigurierbaren Logik

Vdev_n) sowie ein Interface zur Kommunikation mit dem physikalischen Gerät (*Ph.-Interface_1*). Für die Konfiguration des Virtualisierungsmanagers steht eine weitere Schnittstelle (*Konfig*) zur Verfügung. Diese kann ebenfalls von den Rechenkernen beim Start des Systems angesprochen werden. Über die Konfigurationsschnittstelle kann im Virtualisierungsmanager die Anzahl der virtuellen Geräte konfiguriert werden, um nicht benötigte Schnittstellen zu deaktivieren. Es kann je nach Systemkonstellation von Vorteil sein, mehr Schnittstellen für virtuelle Geräte vorzuhalten, um auf dynamische Änderungen der Partitionen reagieren zu können. Typischerweise sind solche dynamischen Änderungen jedoch in sicherheitskritischen Anwendungen nicht vorhanden, können aber in sonstigen Applikationen denkbar sein.

Der aus dem vorgeschlagenen Aufbau resultierende Overhead schlägt sich sowohl in Form von verwendeten Logikressourcen als auch in Latenz nieder. Der Verbrauch (*Utilization U*) der Logikzellen (U_{Logik}) kann folgendermaßen beschrieben werden:

$$U_{Logik} = U_{VMNG} + n * U_{Vdev} + n * U_{Int} \tag{6.1}$$

In (6.1) wird dabei der Verbrauch an Logikzellen für die Verbindung der einzelnen Komponenten in U_{Int} zusammengefasst. Die verwendeten Logikressourcen zur Verbindung des Virtualisierungsmanagers mit dem physikalischen Gerät sind in U_{VMNG} enthalten, da diese Verbindung einen statischen Verbrauch darstellt. Da der Anstieg des Verbrauchs von Logikzellen zur Verbindung der virtuellen Geräte mit dem Virtualisierungsmanager bei einem Anstieg der Anzahl virtueller Geräte nur sehr gering ist, kann (6.1) folgendermaßen vereinfacht werden:

$$U_{Logik} \approx U_{VMNG} + n * U_{Vdev} + U_{Int} \tag{6.2}$$

Aus (6.2) folgt somit, dass bei einer Änderung der Anzahl der virtuellen Geräte lediglich U_{Vdev} eine Änderung des Ressourcenverbrauchs bewirkt, U_{VMNG} und U_{Int} können als statisch angenommen werden. Die Zunahme des Logikressourcenverbrauchs geschieht somit als lineare Funktion in Abhängigkeit der Anzahl der virtuellen Geräte n_{Vdev} und ist mit der Anzahl verfügbarer Logikressourcen nach oben beschränkt.

Zum bisher diskutierten Overhead kommen noch die Kosten für die Kommunikation, sprich die zusätzliche Latenz, hinzu. Zu betrachten ist hierbei die Worst Case Access Time (WCAT), welche benötigt wird, um von einem Rechenkern auf ein physikalisches Gerät zuzugreifen. Während im nicht virtualisierten Fall ein Zugriff „direkt“, also ohne Umleitung in die rekonfigurierbare Logik geschieht, bringt die HW-Virtualisierung eine zusätzliche Latenz für dieses Re-Routing ein. Die Latenz L der nicht virtualisierten Lösung (L_{NV}) ist somit immer geringer als die Latenz der virtualisierten Lösung (L_{HWV}):

$$L_{NV} < L_{HWV} \tag{6.3}$$

Die Latenz für den Zugriff im nicht virtualisierten Fall setzt sich hauptsächlich aus der Latenz, die über das Interconnect entsteht, zusammen. Im Falle der Virtualisierung in Software, z.B. Nutzung eines Hypervisors, entsteht die zusätzliche Latenz in der verwendeten Softwareschicht. Im Falle der Hardware Virtualisierung wird in der Software keine Latenz eingebracht, dafür jedoch eine gewisse Verzögerung durch die in der rekonfigurierbaren Logik umgesetzten Anteile, also die Latenz

eines virtuellen Geräts (L_{Vdev}) und der Latenz des Virtualisierungsmanagers (L_{VMNG}). Diese Latenz kann als Summe der beiden Teilaspekte beschrieben werden:

$$L_{HWV} = L_{Vdev} + L_{VMNG} \tag{6.4}$$

Bei der Berechnung der Latenz wird nur die Latenz eines virtuellen Geräts einbezogen, da sich diese Berechnung für einen Zugriff eines Kerns auf ein physikalisches Gerät bezieht, unabhängig davon, wie viele virtuelle Geräte instanziiert sind. Des Weiteren wird die Latenz eines virtuellen Geräts (L_{Vdev}) als konstant angenommen, da keine dynamischen Effekte auftreten. Im Gegensatz dazu kann die Latenz im VMNG variieren. Zur Verdeutlichung ist der interne Aufbau des VMNG in Abbildung 6.4 schematisch dargestellt.

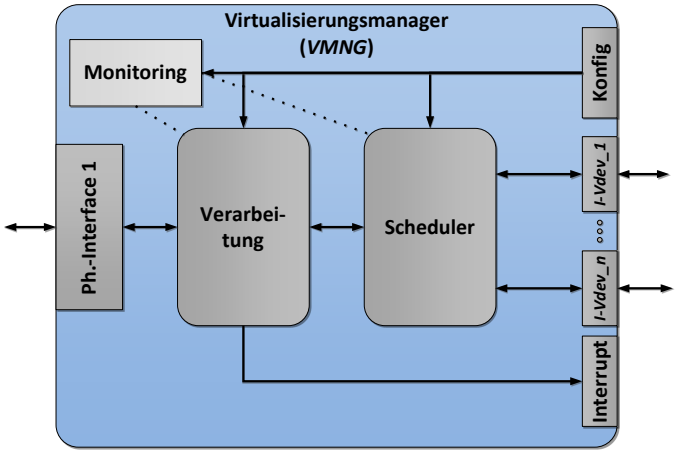


Abbildung 6.4: Schematischer Aufbau des Virtualisierungsmanagers (VMNG)

Speziell im abgebildeten Scheduler können je nach Schedulingverfahren unterschiedliche Latenzen entstehen. Da im Falle sicherheitskritischer Anwendungen die Zugriffe auf Peripherie etc. möglichst deterministisch ablaufen sollen, bietet sich die Integration eines statischen Sche-

duling mit einer eindeutigen Latenz an. Hierbei kann beispielsweise auf das *Round Robin* Schedulingverfahren zurückgegriffen werden, in welchem eine Lastgleichverteilung realisiert wird. Bei der Verwendung eines solchen Schedulingverfahrens kann die WCAT genau bestimmt und vorhergesagt werden und so zu einem deterministischen Systemverhalten beitragen. Die Latenz ist jedoch abhängig von der Anzahl der instanziierten virtuellen Geräte. Bei der Verwendung von *Round Robin* wird pro zusätzlichem virtuellen Gerät ein weiterer Slot eingebracht, der die Latenz erhöht. Alternativ dazu kann ein prioritätsgesteuertes Scheduling zum Einsatz kommen. Dabei ist nicht exakt vorherzusagen, welche Latenz für einen Zugriff entsteht, so lange nicht klar ist, welche anderen Zugriffe mit ggf. höherer Priorität parallel durchgeführt werden. Die Latenz im VMNG (L_{VMNG}) berechnet sich folglich aus der Latenz des Interface zum physikalischen Gerät (L_{Ph-Int}) und des Interface für die Kerne zum virtuellen Gerät (L_{V-Int}) sowie der Verarbeitungs- ($L_{Verarb.}$) und Scheduling-Latenz (L_{Sched}) zu:

$$L_{VMNG} = L_{V-Int} + L_{Ph-Int} + L_{Sched} + L_{Verarb.} \tag{6.5}$$

Nicht nur die Integration eines statischen Schedulingverfahrens kann in sicherheitskritischen Anwendungen von Vorteil sein, auch die Verwendung eines Budgetierungsansatzes ist integrierbar. Durch Budgetzuweisungen können Garantien für die Zugriffe einzelner Rechenkerne auf das physikalische Gerät gegeben werden. Diese werden in der HW-Virtualisierung im Scheduler (vgl. Abbildung 6.4) mit integriert. Das Budget bezieht sich in diesem Fall auf die Anzahl an Zugriffen über eine bestimmte Zeitspanne (z_t über die Periode T). Wie die Verteilung dargestellt wird, bieten verschiedene Parameter die Möglichkeit zur Anpassung an die jeweilige Anwendung. Es kann beispielsweise eine symmetrische oder asymmetrische Verteilung konfiguriert werden. Vorteil einer solchen Budgetierung ist die gute Möglichkeit zur Überwachung und Kontrolle der Zugriffszahlen. Beim Überschreiten des Budgets eines Rechenkerns kann dieser für weitere Zugriffe gesperrt und es kann gefolgert werden, dass es sich um ein fehlerhaftes Verhalten des entsprechenden Kerns handelt. Vor Überschreiten des Budgets wird dem entsprechenden Rechenkern bereits per Interrupt signalisiert, dass sich das Budget dem Grenzwert nähert. Je nach Zustand des Rechen-

kerns kann dieser Maßnahmen ergreifen, z.B. nur noch sicherheitskritische Nachrichten an das physikalische Gerät senden. Ebenso wird die Annäherung an die Budgetgrenze bzw. das Überschreiten der Grenze an eine übergreifende Monitoringeinheit kommuniziert. Diese kann weitere Maßnahmen auf MPSoC Ebene ergreifen und den Gesamtzustand des Systems evaluieren. Erreicht einer der Kerne seine Budgetgrenze während die anderen Rechenkerne noch ein großes Restbudget besitzen, kann eine dynamische Verschiebung des Budgets durchgeführt werden. Dies kann basierend auf der maximalen Zugriffshäufigkeit in Relation zur verbleibenden Zeit der vorgegebenen Zeitspanne geschehen. Über dieses Verhältnis wird weiterhin abgeschätzt, ob ein Kern sein Budget aufbrauchen kann oder ob Zugriffe garantiert nicht benutzt werden. Ist dies der Fall, besteht die Möglichkeit, die übrigen Zugriffe dem Rechenkern zu übertragen (transferiert), welcher sein Zugriffslimit erreicht hat. Die Berechnung der maximal verbleibenden Zugriffe kann folgendermaßen durchgeführt werden:

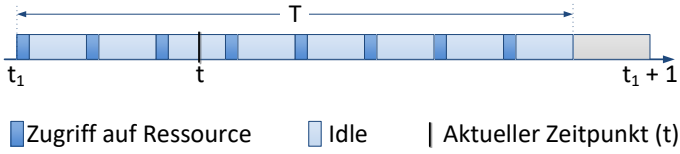


Abbildung 6.5: Darstellung der Zugriffe eines Kerns auf die physikalische Ressource innerhalb einer Periode

Maximale Zugriffsfrequenz:

$$f_{trig\_max} = \frac{1}{T_{trig\_max}} \tag{6.6}$$

Maximal mögliche Anzahl Zugriffe innerhalb der betrachteten Periode:

$$z_{max,T} = \frac{T}{T_{trig\_max}} \tag{6.7}$$

Maximal verbleibe Anzahl Zugriffe innerhalb der betrachteten Periode:

$$z_{max,t} = \frac{T - t}{T_{trig\_max}} \tag{6.8}$$

Verbleibendes Budget zum Zeitpunkt t innerhalb der Periode T, mit z_t als Anzahl der tatsächlichen Zugriffe bis zum Zeitpunkt t:

$$z_{verb,T} = z_{ges} - z_t \tag{6.9}$$

Ist nun

$$z_{verb,T} > z_{max,t} \tag{6.10}$$

dann kann die Differenz

$$z_{trans} = z_{verb,T} - z_{max,t} \tag{6.11}$$

als maximale Anzahl transferierbarer Zugriffe des Budgets eines Kerns zum anderen betrachtet werden. Die Überwachung bzw. Aufzeichnung der tatsächlich durchgeführten Zugriffe eines Kerns auf das physikalische Gerät über die HW-Virtualisierungsschicht wird mit Hilfe von Zählern umgesetzt.

Bisher wurden ausschließlich Zugriffe der Rechenkerne auf das physikalische Gerät betrachtet. Je nach Komponente die virtualisiert werden soll, kann eine Kommunikation in beide Richtungen, sprich von den Rechenkernen zur physikalischen Komponente und zurück, gefordert sein. Dabei kann es sich sowohl um eine einfache Bestätigung der Ausführung einer Anfrage sowie um einen Datenaustausch handeln. Geht man beispielsweise von der Virtualisierung eines Kommunikationsinterfaces aus, wird klar, dass über diese Schnittstelle auch der Empfang von Daten berücksichtigt werden muss. Für diese Kommunikationsrichtung können zwei unterschiedliche Konzepte integriert werden. Zum einen können empfangene Daten an alle Kerne signalisiert werden[2], z.B. über die Verwendung von Interrupts, bei dem jeder Rechenkern dann für sich selbst auswerten muss, ob die Nachrichten für die weitere Bearbeitung notwendig sind oder ob diese verworfen werden können. Als zweite

[2]Dies entspräche einem Broadcast an alle Teilnehmer respektive Kerne oder Partitionen.

Möglichkeit kann ein passender Filter im VMNG integriert werden, welcher Nachrichten nur an die Empfänger weiterleitet, die mit den Daten in der weiteren Verarbeitung umgehen müssen. Hierfür muss der VMNG Kenntnis über die Zusammenhänge *Absender* $\leftrightarrow$ *Empfänger* haben. Die zweite Möglichkeit bietet dabei den Vorteil, dass die Auswertung und Filterung der Nachrichten bereits in der Hardware geschieht und somit eine Reduktion des Interruptaufkommens folgt. Gleichzeitig muss die Auswertung der Nachrichten nicht in Software umgesetzt werden und erspart so Rechenzeit auf den Kernen und kompensiert ggf. die durch die Hardwarelösung eingebrachte Latenz. Diese Filterung und Verteilung der Nachrichten geschieht im Verarbeitungsblock, welcher in Abbildung 6.4 dargestellt ist.

Um die Zugriffe der virtuellen Geräte auf das physikalische Gerät überwachen zu können, wird zusätzlich ein Monitoring integriert. Dieser Monitor kann sowohl die Kommunikationsrichtung *Rechenkern* $\rightarrow$ *physikalisches Gerät* als auch die Richtung *physikalisches Gerät* $\rightarrow$ *Rechenkern* überwachen. Speziell die Zugriffe und das dem zugrundeliegende Scheduling sowie eine potentielle Budgetierung werden überwacht und im Fehlerfall signalisiert. Bei fehlerhaften Zugriffen oder abweichendem Scheduling kann direkt im Virtualisierungsmanager reagiert und die Fehlerursache identifiziert sowie die Information über einen Fehler an eine weitere Instanz kommuniziert werden. Ebenso wird diese Monitoringeinheit für die Überwachung der Budgetierung verwendet. Die verbleibenden Budgets der einzelnen Kerne/Partitionen auf das virtualisierte Gerät können erfasst werden und tragen somit ebenso zur Evaluierung des Systemzustands bei.

Zur Parametrierung der HW-Virtualisierung können verschiedene Gesichtspunkte herangezogen werden. Die folgenden Parameter sind dabei zu beachten:

- Anzahl der virtuellen Geräte, n_{Vdev}
- Benachrichtigungsschema (Broadcast, Filterung), ν_{HWV}
- Schedulingverfahren, σ_{HWV}
- Budgetierungsschema, β_{HWV}

Zur Festlegung der Parameter kann eine Kostenfunktion zu Lasten des Logikressourcenverbrauchs und der Latenz erstellt werden:

$$K_{Latenz} = f(\sigma_{HWV}\ ,\ \beta_{HWV}\ ,\ n_{Vdev}) \quad (6.12)$$

$$K_{Logik} = f(\sigma_{HWV}\ ,\ \nu_{HWV}\ ,\ n_{Vdev}) \quad (6.13)$$

Es wird deutlich, dass je nach Systemauslegung ein Trade-Off zwischen dem Nutzen der Hardware-Virtualisierung und den entstehenden Kosten besteht. Bei einer Virtualisierung in Software tritt, wie auch bei einer Realisierung in Hardware, eine vergrößerte Latenz auf, welche mit der HW-Virtualisierung verglichen werden kann. Da lediglich in der Hardwarelösung Logikressourcen benötigt werden, sind diese schwer der Softwarelösung gegenüberzustellen, jedoch ist dieser Verbrauch an Logikressourcen vergleichbar mit der in Software zusätzlich eingebrachten Komplexität in der Softwarelösung. Bei der Nutzung ist die Konfiguration sowohl in einem Hypervisor bei Verwendung einer Softwarelösung durchzuführen als auch beim Virtualisierungsmanager bei Verwendung der Hardwarelösung.

Die hier beschriebene Methode wurde auf dem Applied Reconfigurable Computing Symposium vorgestellt und in [Bap+16] veröffentlicht.

6.1.2 Prototypische Umsetzung und Validierung

Auf Basis des in Kapitel 6.1.1 beschriebenen Konzepts, wird eine prototypische Implementierung aufgesetzt, um eine Evaluation und eine Abschätzung des Overheads auf realer Hardware durchführen zu können. Für die Umsetzung wird auf eine heterogene Multicorearchitektur zurückgegriffen. Der Basisaufbau basiert auf der Masterarbeit [Sto15], in deren Ausarbeitung weitere technische Details nachgeschlagen werden können.

6.1.2.1 Hardwareaufbau

Zur Umsetzung des Konzepts wird eine heterogene Multicorearchitektur herangezogen, welche sowohl einen Multicore als auch Peripherals in Form eines ASIC beinhaltet und einen FPGA-Anteil enthält.

Bei der Architektur handelt es sich um die Zynq-7000 [102] Architektur von Xilinx. Im SoC enthalten ist ein ARM Cortex A9 [101] Dualcore als sog. Processing System (PS) sowie unterschiedliche Peripherals. Für die Evaluation des Ansatzes wird auf eine CAN-Controller Virtualisierung zurückgegriffen. Die Virtualisierung wird im FPGA-Teil des Zynq-7000, hier Programmable Logic (PL) genannt, implementiert. In Abbildung 6.6 ist die Integration der HW-Virtualisierung in der PL dargestellt. Die bereits in Kapitel 6.1.1 beschriebenen Komponenten sind für die Realisierung in Hardware abgebildet.

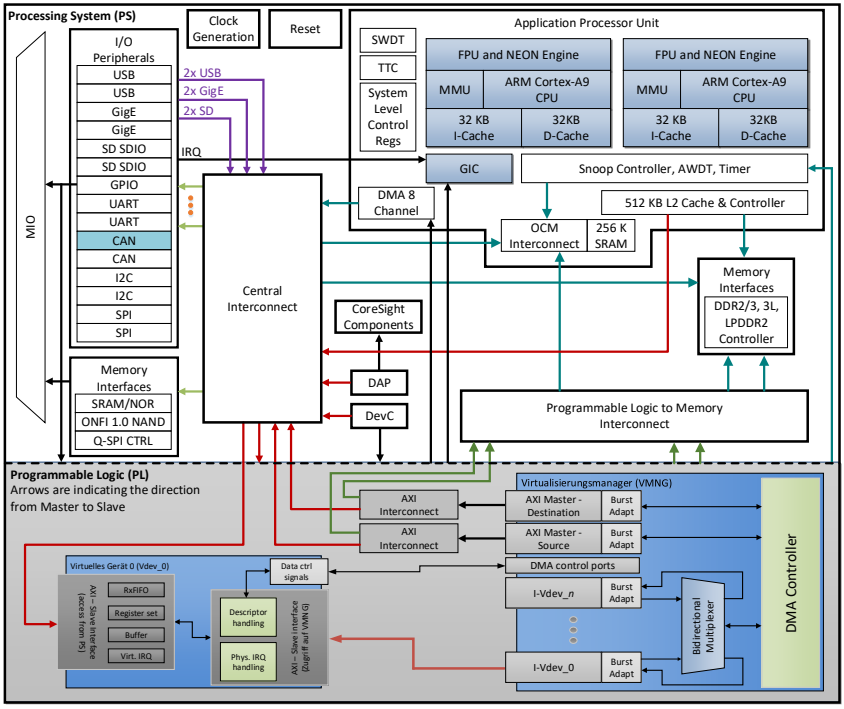


Abbildung 6.6: Integration der HW-Virtualisierungslösung innerhalb der Zynq-7000 Architektur von Xilinx. Die Darstellung der Zynq-7000 Architektur erfolgt in Anlehnung an [102].

Beim integrierten CAN-Controller handelt es sich um einen Standardcontroller, der zu ISO11898-1[103], CAN 2.0A und CAN2.0B kompatibel ist und von Xilinx, welcher als fester IP-Block im SoC integriert wird. Da es für die spätere Evaluation wichtig ist, werden zwei der möglichen Betriebsmodi des CAN-Controllers genannt:

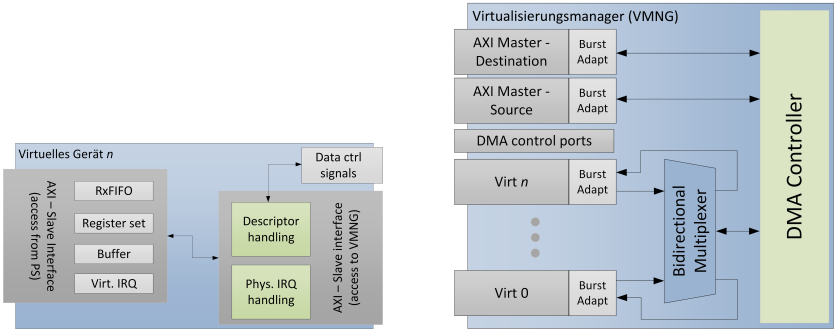
- **Normal Mode:** Hierbei handelt es sich um den im normalen Betrieb eingesetzten Modus, welcher ein Senden und Empfangen von Nachrichten erlaubt.
- **Loopback Mode:** In diesem Modus werden die Sende- (*Rx*) und Empfangsleitung (*Tx*) miteinander verbunden. Dies resultiert in einem direkten Empfang einer gesendeten Nachricht.

Um eine Nachricht per CAN zu versenden, muss lediglich das Datum in den *TxFIFO* des Controllers geschrieben werden. Weitere Eingriffe sind von Seiten der Software nicht nötig.

Die virtuellen Geräte (Vdef) und der Virtualisierungsmanager (VMNG) stellen die Kernkomponenten des Konzepts dar. Die Verbindungen der Komponenten und die Einbettung in das Zynq-7000 SoC sind in Abbildung 6.6 dargestellt. Die virtuellen Interfaces sind über eine AXI Masterschnittstelle mit der PS verbunden. Zusätzlich werden die virtuellen Geräte mit dem Virtualisierungsmanager verbunden. Hierfür werden ebenfalls AXI Interfaces bereitgestellt und zusätzlich wird das Interruptsignal des CAN-Controllers mit dem virtuellen Interface verbunden.

Die virtuellen Geräte bilden die Schnittstelle zum Processing System (PS). Die Adressen, die für die Register benötigt werden, sind im Adressbereich des SoC abgebildet. Über die Schnittstelle zum Virtualisierungsmanager werden Descriptoren und Daten für den Transfer zwischen den beiden Komponenten übertragen.

Um den CAN-Controller zu virtualisieren, muss nicht der komplette CAN-Controller in der Programmable Logic (PL) repliziert werden. Es müssen lediglich wenige Teile des Controllers abgebildet werden. Hierzu gehören zum Beispiel die FIFO-Buffer, die zum Senden und Empfangen verwendet werden. Diese FIFOs werden in Breite, Tiefe und Anzahl konfigurierbar umgesetzt, um eine größere Flexibilität für andere Interfaces als CAN zu erreichen. Die Nachrichten, die im RxFIFO des physikalischen Geräts eintreffen, werden an alle virtuellen Geräte



(a) Virtuelles Gerät (b) Virtualisierungsmanager (VMNG)

Abbildung 6.7: Detaillierte Darstellung der virtuellen Geräte und des Virtualisierungsmanagers.

weitergeleitet. Die Sende-FIFOs (TxFIFO) werden nur gepuffert, aber nicht repliziert. Um Nachrichten vom Processing System (PS) zu senden, werden die virtuellen Geräte adressiert. Die zu sendende Nachricht wird in den Puffer der virtuellen Geräte geschrieben und eine Übertragung vom Puffer zum physikalischen Gerät angefordert. Diese Übertragung übernimmt der im Virtualisierungsmanager integrierte DMA-Controller. Beim Eintreffen einer neuen Nachricht am physikalischen Gerät wird ein *New Message Interrupt* ausgelöst. Dieser Interrupt wird zum Virtualisierungsmanager (VMNG) umgeleitet. Der VMNG überprüft anschließend das Interrupt Status Register des physikalischen Geräts. Ist eine neue Nachricht eingegangen, wird diese mittels integriertem DMA an die virtuellen Geräte verteilt und der Interrupt wird zurückgesetzt. Sobald die Nachricht in die RxFIFOs der virtuellen Geräte übertragen ist, wird ein Interrupt an das Processing System (PS) gesetzt. Durch diesen Aufbau ist es möglich, ohne Software Interaktion die virtuellen Geräte „zwischen“ das Processing System und das physikalische Gerät zu schalten. Die virtuellen Geräte sind somit aus Software-Sicht nicht zu sehen und damit transparent für den Software-Entwickler.

Im Detail bestehen die beiden Kernkomponenten aus weiteren Modulen. Der detaillierte Aufbau des Virtualisierungsmanagers (VMNG) ist in Abbildung 6.7(a) dargestellt, sowie der Aufbau eines virtuellen Geräts in Abbildung 6.7(b).

Durch den modularen Aufbau wird eine Flexibilität erreicht. Dies beinhaltet die Parametrierbarkeit der Anzahl der virtuellen Geräte und der damit verbundenen Anzahl der Schnittstellen im VMNG. Je nach Anwendungsfall können mehrere virtuelle Geräte instanziiert werden. Für die hier beschriebene Evaluation wird ein prioritätsbasiertes Scheduling umgesetzt. Weitere Schedulingverfahren können im VMNG realisiert werden.

Zur Umsetzung des Konzepts werden die in Tabelle 6.1 gezeigten Logikressourcen für die Komponenten benötigt. Da es hier um die Validierung des Ansatzes geht, steht eine Optimierung des Logikverbrauchs nicht im Vordergrund.

Tabelle 6.1: Benötigte Logikressourcen pro Komponente

Ressourcentyp:	LUT	FLOP-LATCH	DMEM
Verf. in Zynq-7020	53200 (100%)	106400 (100%)	17400 (100%)
VMNG (inkl. DMA)	12516 (~23%)	4279 (~4%)	–
DMA	3912 (~7%)	1953 (<1%)	–
Virtuelles Gerät	1893 (~3%)	1493 (<1%)	192 (~1%)

Während für eine Erhöhung der virtuellen Geräte die Logikressourcen für diese vervielfältigt werden, wird nur ein Virtualisierungsmanager für bis zu acht virtuelle Geräte mit einem konstanten Logikressourcenverbrauch benötigt.

6.1.2.2 Softwarearchitektur und Messungen

Für das vorgestellte Konzept ist keine funktionale Applikation in Software für die Evaluation vorgesehen, jedoch wird ein gewisser Anteil für die Durchführung von Messungen benötigt, der hier vorgestellt wird. Um zu evaluieren, welcher zeitliche Overhead (Latenzen) durch das

Konzept in das Systemdesign eingebracht wird, werden die zusätzlich eingebrachten Verbindungen und Zugriffe vermessen.

Um die Software einfacher und besser verständlich zu halten, wird eine Bare-Metal Applikation aufgesetzt. Diese Applikation wird auf einem der ARM Cortex A9 Kerne ausgeführt, die mit einer Frequenz von $ARM_{clk} = 666MHz$ getaktet sind. Für die Messungen wird ein integrierter 32 Bit Timer verwendet, der sehr schnelle Zugriffe (ca. 6 Prozessortaktzyklen) zulässt und mit einer Frequenz $Timer_{clk} = ARM_{clk}/2 = 333MHz$ getaktet wird. Der FPGA hingegen wird mit $FPGA_{clk} = 100MHz$ getaktet. Der virtualisierte CAN-Controller, *CAN 0*, wird mit $500kBaud$ betrieben. Zur Messung wird der im nachfolgenden Listing 6.1 dargestellte Ablauf verwendet.

```
//Check TxFIFO is full #1
    while ((Xil_In32(BaseAddr + 0x1C) & 0x400) != 0x0);
//Write TxFIFO #2
    Xil_Out32(BaseAddr + 0x30, TxFrame[0]);
    Xil_Out32(BaseAddr + 0x34, TxFrame[1]);
    Xil_Out32(BaseAddr + 0x38, TxFrame[2]);
    Xil_Out32(BaseAddr + 0x3C, TxFrame[3]);
//Check if RxFIFO is empty #3
    while ((Xil_In32(BaseAddr + 0x1C) & 0x80) == FALSE);
//Read RxFIFO #4
    RxFrame[0] = Xil_In32(BaseAddr + 0x50);
    RxFrame[1] = Xil_In32(BaseAddr + 0x54);
    RxFrame[2] = Xil_In32(BaseAddr + 0x58);
    RxFrame[3] = Xil_In32(BaseAddr + 0x5C);
//Clear pending interrupts #5
    Xil_Out32(BaseAddr + 0x24, (Xil_In32(BaseAddr + 0x1C)
        & 0x80));
```

Listing 6.1: Softwarefunktion zur Realisierung einer Sende-Empfangsschleife des CAN-Controllers

Die Applikation wird in fünf Teile (vgl. Listing 6.1 #1 - #5) unterteilt, die im Weiteren vorgestellt und vermessen werden. Daraus resultieren die folgenden Messszenarien:

- **Register lesen:**

Da Register die Basis der meisten Zugriffe der folgenden Messungen bilden, wird zunächst ermittelt, wie lange ein Zugriff auf ein solches Register dauert. Um eine Vergleichbarkeit zu erzeugen, wird das Interrupt Service Routine (ISR) Register sowohl vom physikalischen als auch vom virtuellen Gerät für die Messung herangezogen. Der Zugriff auf das Register des physikalischen Geräts beträgt im Mittel *49 Taktzyklen bei 333MHz*, der Zugriff auf die Register des virtuellen Geräts beträgt hingegen *64 Taktzyklen bei 333MHz*. Bei beiden Ergebnissen wurde eine Varianz von +/-2 Taktzyklen festgestellt.

- **TxFIFO check (vgl. Listing 6.1 #1):**

  Um eine Nachricht per CAN verschicken zu können, wird zunächst der Füllstand des TxFIFO ausgelesen. Hierzu wird das Statusregister des CAN-Controllers ausgelesen. Die Messergebnisse sind sowohl für das virtuelle Gerät (Virt0) als auch für das physikalische Gerät (CAN0) in Abbildung 6.8 dargestellt.

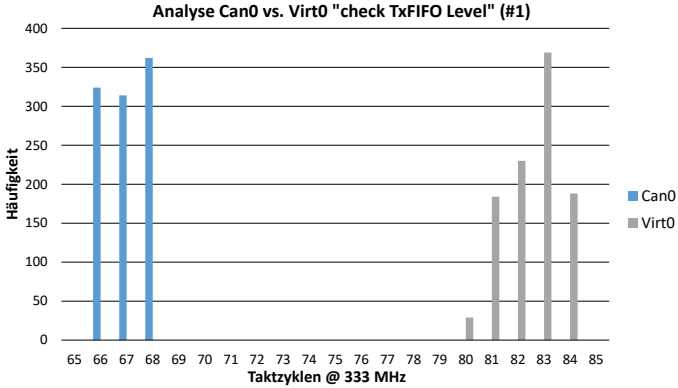


Abbildung 6.8: Messung der Zugriffszeit für das Auslesen des TxFIFO Füllstandes (#1 TxFIFO full check)

- **Daten in TxFIFO schreiben (vgl. Listing 6.1 #2):**

  Der nächste Schritt zum Senden einer Nachricht ist das Schreiben der Daten in den TxFIFO. Der CAN-Frame wird in Form von vier 32Bit Übertragungen in den FIFO geschrieben. Die Übertra-

gung wird als Block umgesetzt und gemessen. Die Ergebnisse der Messung für CAN0und Virt0 sind in Abbildung 6.9 dargestellt.

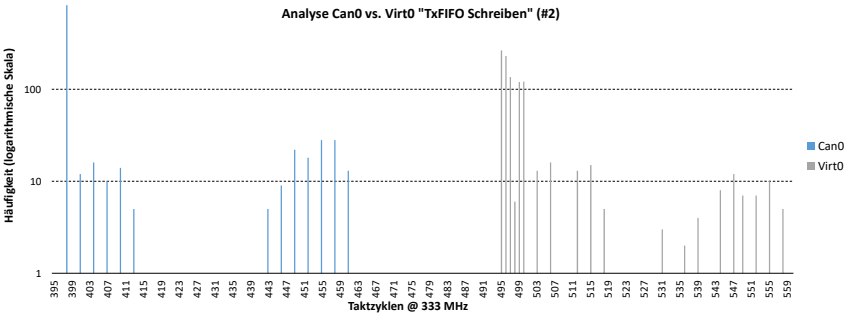


Abbildung 6.9: Messung der Zugriffszeit für das Schreiben des TxFIFO (#2 Write TxFIFO)

- **Neue Nachricht im RxFIFO (vgl. Listing 6.1 #3):**

  Zur Messung der Verzögerung die entsteht, wenn eine neue Nachricht im RxFIFO eintrifft, wird eine zyklische Abfrage (Polling) des zugehörigen Bit im Register implementiert. So lange das Bit nicht gesetzt wird, bleibt der Prozessor in einer Schleife. Dieser Verbleib in der Schleife wird gemessen. Für das physikalische Gerät bleibt der Prozessor 42911 Taktzyklen bei 333MHz und für das virtuelle Gerät 43378 Taktzyklen in der Schleife. Der entstehende Overhead liegt bei 1%. Dieser Wert ist von der im CAN-Controller definierten Baud-Rate abhängig.

- **Daten aus dem RxFIFO lesen (vgl. Listing 6.1 #4):**

  Nachdem das Bit zur Signalisierung einer neuen Nachricht gesetzt wurde, wird die Nachricht aus dem RxFIFO gelesen. Dies geschieht analog zum Beschreiben des TxFIFO mittels vier Übertragungen mit jeweils 32Bit. Die Ergebnisse sind in Abbildung 6.10 dargestellt.

- **Rücksetzen des „New Message“ Bit (vgl. Listing 6.1 #5):**

  Nach dem Auslesen der Nachricht aus dem RxFIFO des CAN-Controllers, muss das „New Message“ Bit im ISR-Register zurück-

gesetzt werden. Dies wird durch Setzen eines weiteren Bit im Register durchgeführt.

- **Sende-Empfangsschleife einer CAN Nachricht:**

  Um die einzelnen vorgestellten Messungen zu verifizieren, wird eine weitere Messung über die gesamte Sende-Empfangsschleife durchgeführt. Diese beinhaltet alle beschriebenen Schritte. Das Ergebnis ist in Abbildung 6.11 zu finden.

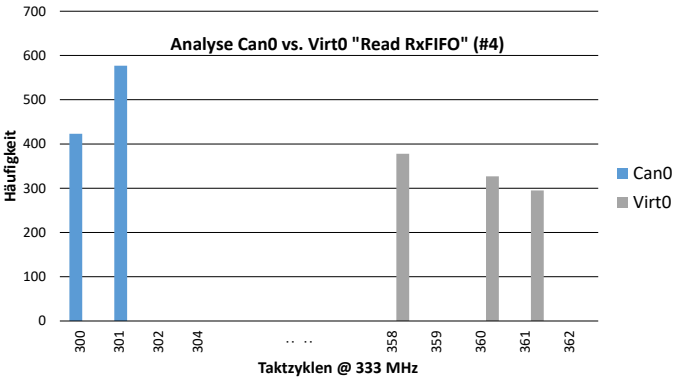


Abbildung 6.10: Messung der Zugriffszeit für das Lesen des RxFIFO (#4 Read RxFIFO)

Tabelle 6.2: Evaluation der Latenz der Hardwareimplementierung

	Taktzyklen@100MHz	μs	Varianz
Nachricht senden	30	0,30	0
Nachricht empfangen	125	1,25	1

Zur weiteren Analyse des Konzeptes wird eine zusätzliche Messung der Latenz der Hardwareimplementierung direkt in Hardware durch einen zusätzlichen Timer umgesetzt. Hierfür wird ein AXI-Timer [104] von Xilinx in die Programmable Logic (PL) integriert, der ausgelöst wird, sobald eine Übermittlungsanfrage an den virtuellen Interfaces vorhanden ist. Wird die zu sendende Nachricht an das physikalische Gerät übermittelt, wird der Timer gestoppt. Dies stellt die Durchlauf-

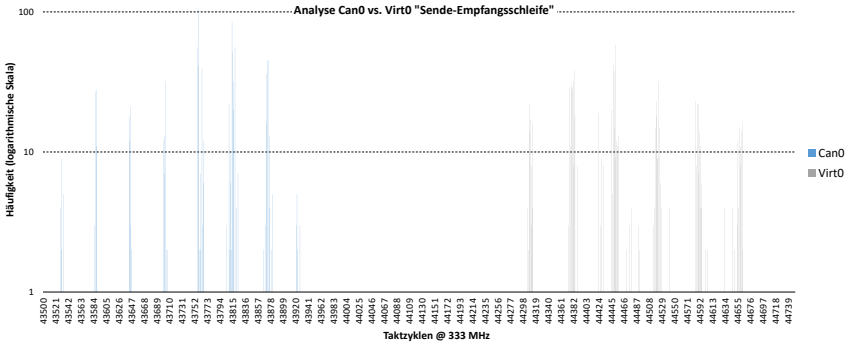


Abbildung 6.11: Messung der gesamten Sende-Empfangsschleife einer CAN-Nachricht

zeit und damit die Latenz der Hardwareimplementierung dar. Beim Empfang einer Nachricht wird der Timer durch den New Message Interrupt gestartet und beim Interrupt, der vom Processing System (PS) ausgelöst wird, wieder gestoppt. Die Ergebnisse sind in Tabelle 6.2 dargestellt.

Zusammenfassend wird die eingebrachte Latenz und der erzeugte Overhead in Tabelle 6.3 dargestellt. Dort sind die Mittelwerte der Zugriffszeiten für das physikalische Gerät ($\bar{t}_{phy\_zugriff}$) und das virtuelle Gerät ($\bar{t}_{virt\_zugriff}$) gegenübergestellt, aus welchen der Overhead berechnet wird. Insgesamt ist die eingebrachte Latenz für einen CAN-Controller als unkritisch einzustufen und für reale Anwendungen einsetzbar.

Tabelle 6.3: Evaluation der Latenz der Hardwarevirtualisierung

Funktion	$\bar{t}_{phy\_zugriff}$ [Taktzyklen]	$\bar{t}_{virt\_zugriff}$ [Taktzyklen]	Overhead [%]
Register lesen	49	64	~30
#1 TxFIFO check	67	83	~23
#2 Schreibe TxFIFO	405	501	~23
#3 „New Messange“ Flag	42911	43378	~1
#4 RxFIFO lesen	301	361	~20
#5 „New Message“ Flag reset	88	117	~33
Sende-Empfangsschleife	43772	44469	~1

6.1.3 Bezug zu den Fehlerbildern

Durch den Einsatz der Hardware-Virtualisierung wird eine Möglichkeit geschaffen, über welche sich die Nutzung von gemeinsam verwendeten Ressourcen optimieren und transparent, aus Sicht der Software, realisieren lässt. Mit Hilfe der Integration eines Scheduling und einer Budgetierung kann dem Fehlerbild einer *hohen Ausführungslatenz* entgegengewirkt werden. Hierbei werden speziell *Peripherie* und die *Nutzung von Services* so ausgelegt, dass ein deterministischer Zugriff und damit eine gut zu bestimmende WCAT entstehen. Die definierten Zugriffe helfen nicht vorhersagbare Zugriffszeiten zu verhindern und gewisse Garantien zu geben. Zusätzlich können durch die Verwendung eines Monitorings Fehler im Zugriffsverhalten erkannt werden, die dem Fehlerbild *Fehlerhafte Berechnung/Daten/Zugriffe*, entstanden durch beispielsweise *Single Event Effect (SEE)/Single Event Upset (SEU)*, entsprechen. Durch die mögliche Parametrierung lässt sich das Konzept auf eine beliebige Anzahl zugreifender Kerne/Partitionen, verschiedene Schedulingverfahren und Budgetierungen erweitern. Je nach verfügbaren Logikressourcen können entsprechende Konstellationen erzeugt werden.

6.2 Dynamische Migration von Funktionen

In zukünftigen Systemen ist zu erwarten, dass die Anforderungen an die Verfügbarkeit weiter steigen werden. Es werden zunehmend Systeme diskutiert, die eine Fail-Operational Architektur[3] beinhalten. Diese Fail-Operational Architekturen sind bereits in der Luftfahrt bekannt und im Einsatz, werden aber nur durch eine Vervielfältigung der Hardware erreicht (vgl. Kapitel 3.1). Mit den aktuellen Entwicklungen im Bereich eingebetteter Prozessoren wird es zunehmend denkbar, eine solche Architektur in ein MPSoC zu integrieren. Hieraus entstehen Ansätze, die einen geringeren Overhead mit sich bringen als die aktuell verwendeten Ansätze aus der Luftfahrt.

Um ein Fail-Operational System bereitzustellen, muss dieses in einem Fehlerfall nicht mehr die vollständige Funktion realisieren, es reicht oftmals aus, eine degradierte bzw. reduzierte Funktion bereitzustellen. Die degradierte Funktion beinhaltet dann den sicherheitskritischen Anteil der Applikation.

Basierend auf der Erkennung eines Fehlers muss das System auf eine Rückfallebene zurückgreifen können. Diese kann dynamisch zur Laufzeit im Fehlerfall aktiviert werden und die Funktion übernehmen. Hierfür kann das Rückfall-System (Backup System) in Stand-by vorgehalten werden oder im Normalbetrieb eine andere Funktion ausführen. Im Falle eines Fehlers muss das Backup System innerhalb einer gewissen Zeitspanne zur Verfügung stehen und den Kontext der ursprünglichen Funktion übernehmen.

Im Kontext von heterogenen Multicorearchitekturen mit rekonfigurierbarer Logik (RL) kann diese Einheit in Form eines Hardware Moduls im FPGA vorgehalten oder in Software realisiert werden. Ein Konzept zur Umsetzung einer dynamischen Migration von Funktionen zur Aufrechterhaltung der sicherheitskritischen Anwendung wird in den folgenden Kapiteln diskutiert. Das vorgestellte Konzept basiert im Wesentlichen auf der Simplex-Architektur aus [105] und den Masterarbeiten [Dör17] und [Bal17].

[3] Mit Fail-Operational werden Systeme bezeichnet, die auch im Fehlerfall mindestens eine degradierte Funktion aufrecht erhalten und nicht wie Fail-Safe Systemen in einen sicheren Zustand übergehen.

6.2.1 Konzept

In Anlehnung an das in [105] vorgestellte Simplex Konzept, das sich nicht auf eine Mehrkernarchitektur bezieht, wird eine Methode zur dynamischen Umschaltung von Kanälen in einem Safety System vorgestellt. Hierbei wird ein Systemanteil, der die Applikation ausführt, und ein Safety Systemanteil, der das Backup System bildet, definiert. Der Applikationsanteil führt im Normalbetrieb die eigentliche Funktion aus. Tritt ein Fehler auf, so wird das Backup Safety System gestartet und übernimmt die Funktion bis zu einem gewissen Umfang. Die Funktion, welche noch ausgeführt wird muss dabei nicht dem vollen Umfang entsprechen, in vielen Fällen ist eine degradierte Funktionalität in sicherheitskritischen Systemen zulässig.

Durch die verfügbaren Ressourcen und die vorhandenen Möglichkeiten in einem Mehrkernprozessor bieten sich Möglichkeiten, eine dynamische Migration von Funktionen auf ein Stand-by System innerhalb kürzester Zeit umzusetzen. Basierend auf der Integration innerhalb eines SoC können wesentlich geringere Reaktionszeiten realisiert und eine Zustandsübergabe effizient ermöglicht werden.

Der schematische Aufbau sowie die beteiligten Komponenten sind in Abbildung 6.12 dargestellt.

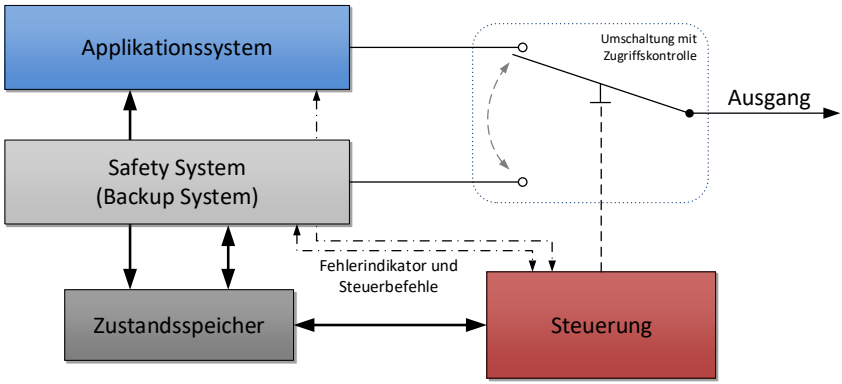


Abbildung 6.12: Darstellung des Konzepts der dynamischen Funktionsmigration als Erweiterung der Simplex-Architektur

Die in Abbildung 6.12 dargestellten Komponenten erfüllen jeweils den folgenden Zweck:

- **Applikationssystem:**

  Das Applikationssystem führt die eigentliche Applikation im Normalbetrieb aus. Dieses System hat im Normalbetrieb vollen Zugriff auf den Ausgang. Während des Normalbetriebs legt das Applikationssystem den aktuellen Zustand im Zustandsspeicher ab. Es besitzt einen Ausgang, der im Fehlerfall ein Signal an die Steuerung übermittelt, sowie einen Eingang der Steuerung zu Überwachungszwecken und für Steuerbefehle.

- **Safety/Backup System:**

  Das Safety System bildet das Backup System, das im Fehlerfall gestartet wird und eine reduzierte Funktion des Applikationssystems ausführt. Der Start des Backup Systems wird von der Steuerung ausgelöst. Es besitzt einen Ausgang, der im Fehlerfall ein Signal an die Steuerung übermittelt, sowie einen Eingang der Steuerung zu Überwachungszwecken und für Steuerbefehle.

- **Steuerung:**

  Die Steuerung übernimmt die Kontrolle des Systems. Hier werden Informationen zu Fehlern im Applikationssystem sowie im Backup System gesammelt und eine Umschaltung des Ausgangs vorgenommen. Zusätzlich erhält die Steuerung Zugriff auf den Zustandsspeicher zur Überwachung.

- **Zustandsspeicher:**

  Im Zustandsspeicher legen sowohl Applikationssystem als auch Backup System ihren aktuellen Zustand ab. Dieser wird bei einem Wechsel von Applikationssystem zu Backup System und zurück benötigt, um den aktuellen Zwischenstand auslesen zu können.

- **Umschaltung mit Zugriffskontrolle:**

  Zur Ansteuerung des Ausgangs muss genau ein System ausgewählt sein. Die Umschaltung realisiert die Verschaltung und besitzt eine interne Zugriffskontrolle, welche die Zugriffe überwacht. Ausgelöst wird ein Umschaltvorgang von der Steuerung.

Das *Applikationssystem (APPS)*, das die eigentliche Funktion ausführt, wird dabei beispielsweise durch eine geeignete Methode Überwacht, dies können Watchdogs, Lock-Step Kerne, zählerbasierte Ansätze oder sonstige Mechanismen zur Fehlererkennung sein. Beim Eintreten eines Fehlers wird dieser geeignet an die *Steuerung* signalisiert. Parallel dazu, sowie in regelmäßigen Abständen, speichert das Applikationssystem seinen Zustand und den aktuellen Status im *Zustandsspeicher*. Die *Steuerung* wertet den Fehler aus, prüft, ob ein valider Status im *Zustandsspeicher* hinterlegt ist und löst den Start des *Backup System (BS)* aus. Ergänzend wird der Ausgang vom *Applikationssystem* getrennt. Sobald das *BS* verfügbar ist, wird der Ausgang mit dem *BS* verbunden. Die Umschaltung des Ausgangs erfolgt dabei durch eine Änderung der Zugriffsberechtigungen.

Das Safety System besitzt nicht die gleiche Rechenkapazität wie das APPS und führt daher nur eine reduzierte Funktion aus. Diese Funktion ist so ausgelegt, dass die mindestens erforderlichen Anteile ausgeführt werden, um einen sicheren Betrieb des Gesamtsystems sicherzustellen.

Ist die Umschaltung erfolgt, so wird von der Steuerung das APPS zurückgesetzt und neu gestartet. Nach dem Neustart wird das APPS Selbsttests unterzogen, um festzustellen, ob es sich um einen permanenten oder einen vorübergehenden Fehler handelt. Wird kein permanenter Fehler festgestellt, so wird von der Steuerung das APPS wieder in Betrieb genommen und der Kontext des BS übernommen. Das APPS kann dabei zunächst parallel arbeiten und erst später wieder produktiv eingesetzt werden, wenn weiterhin keine Fehler auftreten. Die Ergebnisse von APPS und BS werden von der Steuerung verglichen. Sind auch nach einigen Durchläufen keine Fehler aufgetreten, wird der Ausgang von der Steuerung wieder umgeschaltet und das APPS übernimmt wieder dessen Ansteuerung. Das BS wird wieder in einen Ruhemodus versetzt.

Um nicht zu viel Overhead zu generieren, ist es sinnvoll, das BS während des Normalbetriebs entweder für eine andere Best Effort Anwendung zu nutzen oder abzuschalten.

Für das BS muss eine entsprechend sichere Architektur vorgehalten werden, da im Falle eines Fehlers im APPS das BS die einzige verbliebene Einheit darstellt, welche die Funktion aufrechterhält. Je nach Kriti-

kalität bieten sich hierfür Lock-Step Architekturen oder dreifach redundante Prozessoren (TMR) an, um mindestens das gleiche Zuverlässigkeitslevel zu erreichen wie in einem Stand-Alone System.

Betrachtet man den durch den Aufbau und die Umschaltung eingebrachten Overhead, so tragen unterschiedliche Laufzeiten zu einer Verzögerung bei. Hierbei entsteht zunächst eine Verzögerung durch die Fehlererkennung (t_{FE}) und anschließend bei der Verarbeitung der Fehlermeldung in der Steuerung (t_{VS}). Die Steuerung löst dann die Umschaltung und den Start des Backup Systems aus. Dafür wird die Zeitspanne zum Start des BS t_{Start_BS} benötigt. Diese beinhaltet die Zeitspanne, die das BS zum Starten benötigt und um Einsatzbereit zu sein. Anschließend folgt die Signalisierung an die Steuerung, dass das BS verfügbar ist, um die Zugriffssteuerung umzustellen (t_{ZS}).

In Summe resultiert die folgende Latenz t_{Mig}:

$$t_{Mig} = t_{FE} + t_{VS} + t_{Start\_BS} + t_{ZS} \tag{6.14}$$

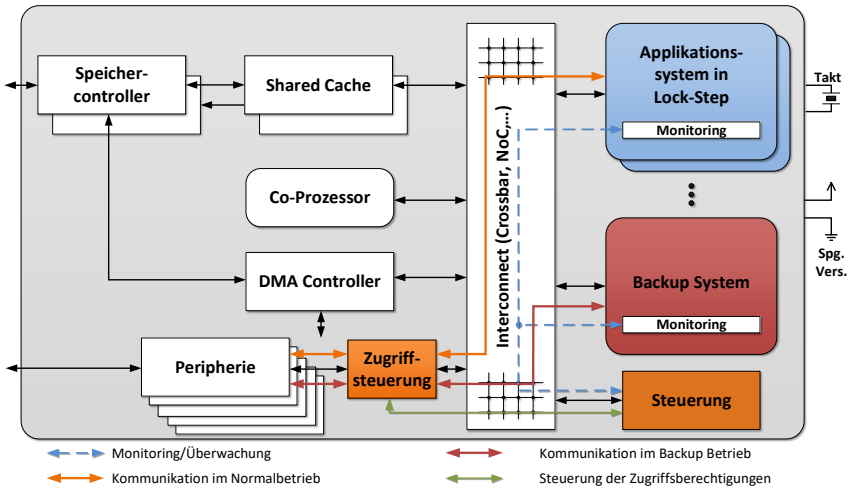


Abbildung 6.13: Darstellung des Konzepts der dynamischen Funktionsmigration auf Basis eines MPSoC

Betrachtet man nun den schematischen Aufbau in Kontext eines MPSoC, so kann die in Abbildung 6.13 dargestellte Architektur angenommen werden. Dabei ist zu beachten, dass das Backup System sowohl in der rekonfigurierbaren Logik als auch im ASIC Teil des MPSoC abgebildet werden kann. Gleiches gilt für die Steuerung. Je nach Architektur, kann es bereits im ASIC Möglichkeiten geben, die Steuerung umzusetzen, in anderen Architekturen hingegen wird eine rekonfigurierbare Logik benötigt, um eine solche Steuerung realisieren zu können.

Ist das System wie in Abbildung 6.13 aufgebaut, so kann die Signalisierung eines Lock-Step Fehlers des Applikationssystems genutzt werden, um die dynamische Migration zu starten. Dieses Signal wird von der Steuerung überwacht.

Um nicht permanent die Ressourcen für das Backup System zu belegen bzw. vorhalten zu müssen, kann dieses in eine rekonfigurierbare Logik verschoben werden und im Falle eines Fehlers konfiguriert werden. Dies kann durch den Einsatz partiell dynamischer Rekonfiguration (vgl. [106, 107, 108, 109]) ermöglicht werden. Hierbei kann während des Normalbetriebs eine andere Komponente im FPGA konfiguriert sein, welche durch das BS im Fehlerfall ersetzt wird. Die im Normalbetrieb vorhandene Komponente darf dabei keine sicherheitskritische Funktion ausüben bzw. Daten für eine solche liefern.

Verdeutlicht man sich den schematischen Aufbau unter Einsatz einer rekonfigurierbaren Einheit und partiell dynamischer Rekonfiguration, so erhält man die in Abbildung 6.14 dargestellte Architektur.

Die in (6.14) vorgestellte Berechnung für die Latenz muss in diesem Fall um die Zeitspanne, die zur Konfiguration der Hardware im FPGA benötigt wird, ergänzt werden. Hieraus folgt:

$$t_{Mig\_rekonf} = t_{FE} + t_{VS} + t_{Start\_BS} + t_{ZS} + t_{rekonf} \tag{6.15}$$

Durch die erhöhte Latenz kann es je nach Anwendungsfall zu einer Latenz kommen, die für den Anwendungsfall nicht mehr ausreichend schnell ist. In der vorliegenden Arbeit ist die Optimierung der Latenz jedoch nicht im Vordergrund, da diese in jedem Anwendungsfall variiert und nicht direkt vom Safety Konzept abhängig ist.

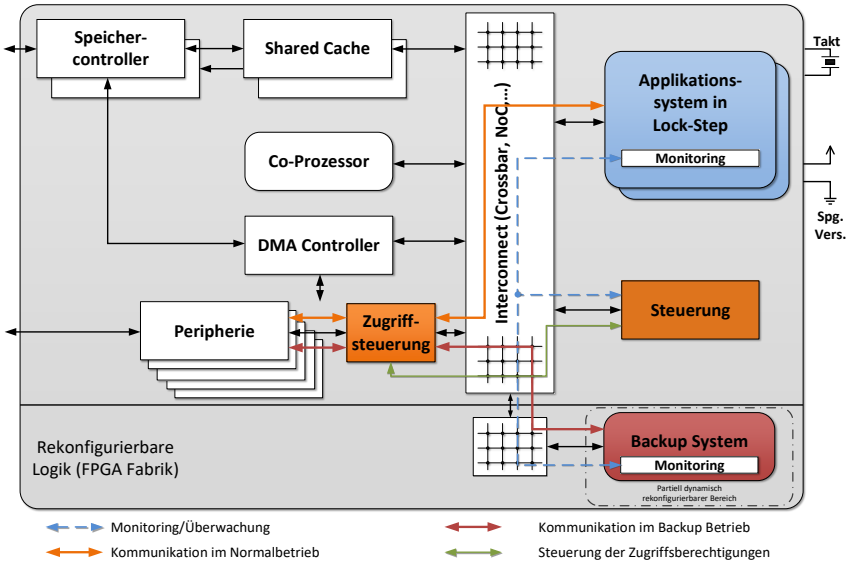


Abbildung 6.14: Schematische Darstellung des Konzepts der dynamischen Funktionsmigration auf Basis eines MPSoC mit rekonfigurierbarer Logik

6.2.2 Prototypische Umsetzung und Validierung

Auf Basis des in Kapitel 6.2.1 vorgestellten Konzepts wird eine prototypische Implementierung angestrebt, die verwendet wird, um das Konzept zu evaluieren. Zur Feststellung des Overheads wird auf eine verfügbare MPSoC Architektur zurückgegriffen. Der Aufbau setzt auf den Masterarbeiten [Dör17] und [Bal17] auf, in deren Ausarbeitungen weitere technische Details nachgeschlagen werden können und weitere Messwerte zu finden sind.

6.2.2.1 Hardwareaufbau

Zur Realisierung der prototypischen Umsetzung des Konzepts wird eine *Xilinx Zynq Ultrascale+* Architektur (vgl. [110]) eingesetzt, die sowohl

eine Real-Time Processing Unit (RPU) in Form eines ARM-Cortex R5 Dual Core Prozessors, eine Application Processing Unit (APU) in Form eines ARM-Cortex A53 Quad Core Prozessors und einen FPGA-Anteil enthält. Das auf die hier relevanten Komponenten reduzierte Blockdiagramm ist in Abbildung 6.15 dargestellt. Die beiden Prozessoren der RPU können dabei sowohl als Zweikernprozessor als auch in Lock-Step Mode betrieben werden. Der vorhandene Lock-Step Mode wird für die weitere Implementierung als Basis-Mechanismus eingesetzt, um Fehler in den Prozessoren erkennen zu können und spielt somit eine wichtige Rolle. Ebenso wird der von Xilinx bereitgestellte Fehlerinjektionsmechanismus genutzt um das System auf seine Funktionalität zu prüfen.

Der Start der Rechenkerne sowie die Überwachung des SoC wird im Xilinx Zynq Ultrascale+ von der Platform Management Unit (PMU) verwaltet. Bei dieser PMU handelt es sich um einen weiteren, vom Nutzer programmierbaren Prozessor, der verwendet werden kann, um den Kontext zu speichern bzw. eine Umschaltung vom APPS auf das BS durchzuführen. Es bietet sich an, die in Kapitel 6.2.1 erläuterte Steuerung auf der PMU zu realisieren und dazu deren Firmware anzupassen[4]. Alternativ wäre auch eine Realisierung der Ablaufsteuerung in der Programmierbaren Logik möglich.

Die Berechnung der eigentlichen Applikation wird auf der RPU implementiert, bei der eine Lock-Step Konfiguration zur Fehlererkennung verwendet wird. Erkennt die Lock-Step Logik einen Fehler, wird dieser an die PMU weitergegeben und das Backup System kann gestartet werden.

Zur Umsetzung der Zugriffssteuerung wird ebenfalls eine bereits im MPSoC enthaltene Komponente genutzt, die Xilinx Peripheral Protection Unit (XPPU). Bei aktivierter XPPU werden unerlaubte Zugriffe auf ein Modul nicht an dieses weitergeleitet und es erfolgt eine Benachrichtigung an die PMU.

Um das Backup System abzubilden wird in der rekonfigurierbaren Logik ein weiterer Prozessor instanziiert, der eine diversitäre Architektur besitzt. Genutzt wird hierfür eine Xilinx Microblaze Architektur. Um diesen Prozessor nicht permanent vorhalten zu müssen, wird dieser

[4]Die konkrete Umsetzung sowie eine Analyse der PMU Firmware kann der Masterarbeit [Dör17] entnommen werden

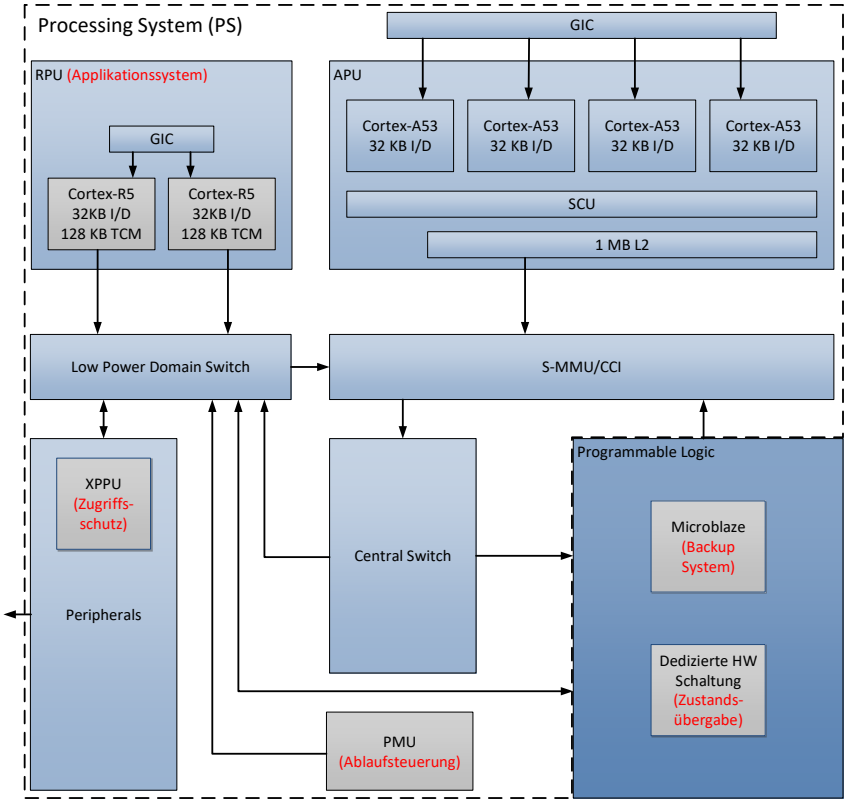


Abbildung 6.15: Blockdiagramm der Xilinx Zynq Ultrascale+ Architektur reduziert auf die in diesem Kontext relevanten Komponenten

in einen partiell rekonfigurierbaren Bereich integriert und kann bei Bedarf konfiguriert werden. Während im System kein Fehler auftritt, kann in diesem Bereich eine andere Applikation oder ein anderer Co-Prozessor konfiguriert werden, der für die eigentliche sicherheitskritische Applikation nicht benötigt wird[5].

Als weitere verbleibende Komponente wird der Zustandsspeicher und die Übergabe realisiert. Diese werden in Form einer dedizierten Schaltung und mit Hilfe von Block-RAMs in der programmierbaren Logik implementiert. Diese Variante der Implementierung reduziert die Zugriffszeit sowie die Zugriffssteuerung und erlaubt eine spezifische Implementierung für die Umschaltung der Block-RAMs.

6.2.2.2 Evaluierung

Für die Anwendung eines Mechanismus, wie in Kapitel 6.2.1 beschrieben, ist es besonders wichtig, die Umschaltzeiten und entstehenden Latenzen zu kennen. Ebenso ist es wichtig zu wissen, wie groß der Zeitraum ist, in dem die Funktionalität kurzfristig nicht bereitgestellt werden kann, um abzuschätzen, ob dies zulässig ist. Je nach Anwendungsfall kann eine gewisse Totzeit akzeptabel sein, da beispielsweise das mechanische System, das gesteuert werden soll träge ist und somit keine merkliche Auswirkung auftritt.

Im Weiteren werden einige ausgewählte Messergebnisse präsentiert, welche die Funktionalität validieren und die Einschränkungen aufzeigen.

Entscheidend ist für das vorgestellte Konzept ein kompletter Zyklus von Fehlerinjektion über die Übernahme durch das Backup System, den Reset des Applikationssystems (APPS) bis zur Rückgabe der Funktionalität an das APPS. Speziell die folgenden Zeitspannen sind zur Charakterisierung des Konzepts und der Implementierung von Interesse:

- Intervall von Fehlerinjektion bis zum Eintritt in den Interrupt Handler.

[5]Die konkrete Umsetzung der partiell dynamischen Rekonfiguration des Microblaze Prozessors kann der Masterarbeit [Bal17] entnommen werden

- Intervall ab Eintritt in den Interrupt Handler bis zur Signalisierung der Bereitschaft des Backup Systems, inkl. der Konfiguration des Zugriffsschutzes.
- Initiierung des RPU Resets.
- Intervall vom Reset der RPU bis zur Signalisierung der Bereitschaft des Applikationssystems.
- Intervall zur Rekonfiguration der Zugriffssteuerung zur Übergabe an das Applikationssystem.

Insbesondere die beiden ersten Intervalle stellen einen kritischen Systemzustand dar. In diesen beiden Zeitspannen ist davon auszugehen, dass das Applikationssystem (APPS) fehlerhaft ist und das Backup System noch nicht bereit ist. Ab dem dritten Intervall, also der Signalisierung der Bereitschaft des Backup Systems, kann mindestens eine degradierte Funktion bereitgestellt werden. Die Rückgabe zum Applikationssystem ist in diesem Fall optional und auf eine sichere Weise erreichbar. Während das Backup System funktioniert, wird das APPS auf Fehler überprüft und neu gestartet. Sind keine permanenten Fehler zu erkennen, kann eine Übergabe von BS zum APPS erfolgen, bedingt jedoch wiederum eine geringe Ausfallzeit.

In Tabelle 6.4 sind die Ergebnisse für die Messung der einzelnen Intervalle gelistet.

Tabelle 6.4: Benötigte Intervalldauer für die dynamische Migration von Funktionen bei Auftreten eines Lock-Step Fehlers

Messintervall	Dauer in PMU Takten bei 33,33MHz
Injektion bis IRQ Handler	$375,01 \pm 0,07$
IRQ Handler bis BS verfügbar	$453,19 \pm 0,81$
Initiierung des RPU Reset	$1.095,32 \pm 0,89$
Reset RPU bis zur Bereitschaft	$5.528,00 \pm 7,41$
Rekonf. der Zugriffssteuerung	$509,67 \pm 4,36$

Der Gesamte Zyklus nimmt somit ca. $238,86\mu s$ in Anspruch, von denen ca. $198,72\mu s$ vom Backup System abgedeckt sind. Nicht abgedeckt sind somit $24,85\mu s$ bei der Übergabe vom APPS an das Backup System bzw.

$15,29\mu s$ bei der Rückgabe. Entscheidend sind somit die $24,85\mu s$, zu Beginn des Zyklus. Dieser Wert stellt jedoch eine sehr kurze Zeitspanne dar, die in sehr vielen Anwendungsfällen als akzeptables Intervall angesehen werden kann[6].

Abgesehen von der Latenz werden durch die Implementierung in Hardware Logikressourcen verwendet. Ein Überblick über die Verwendeten Ressourcen ist in Tabelle 6.5 zu finden.

Tabelle 6.5: Benötigte Hardwareressourcen für die dynamische Migration von Funktionen

Ressourcentyp	verwendet	verfügbar	
CLB	910	34.260	2,66%
LUT	4.432	274.080	1,62%
Flip-Flops	4.452	548.160	0,81%
BRAM	12	912	1,32%
DSP Slices	3	2.520	0,12%

6.2.3 Bewertung

Mit der Einbringung einer dynamischen Migration von Funktionen eines Applikationssystem (APPS) auf ein Backup System (BS) werden neue Möglichkeiten aufgezeigt, die es erlauben, selbst im Fehlerfall ein gewisses Maß an Funktionalität zu gewährleisten. Die einer Migration zu Grunde liegenden Überwachungsmechanismen werden benötigt, um Fehler zu erkennen und diese zu signalisieren. Stellt man diese Migration nun in Relation zu den in Kapitel 4.2 vorgestellten Fehlerbilder, so wird deutlich, dass dieses Konzept nicht direkt eines der Fehlerbilder adressiert. In diesem Fall wird selbst bei Auftreten eines Fehlerbildes eine Möglichkeit geschaffen das System weiterhin, wenn auch mit ggf. reduzierter Funktionalität, zu betreiben. Die Überwachungsmechanismen, die eingesetzt werden um einen Fehler zu erkennen, können

[6]Ein ABS System in einem PKW besitzt beispielsweise eine Regelfrequenz von $\leq 20Hz$ nach [111].

dabei sehr unterschiedlich ausfallen und sich der bereits in Kapitel 5.1, 5.2 und 5.3 vorgestellten Konzepte bedienen.

Weiterhin wird mittels der dynamischen partiellen Rekonfiguration der Overhead geringgehalten. Die Ressourcen müssen ausschließlich im Fehlerfall vorgehalten werden und können im Normalbetrieb für andere Aufgaben und Funktionen eingesetzt werden. Im Normalbetrieb können beispielsweise weitere Überwachungsmechanismen integriert werden, die im Fehlerfall nicht mehr benötigt und so ersetzt werden können.

Für die Migration spielt dabei der Fehlerursprung zunächst keine Rolle und kann daher für alle Fehlerbilder eingesetzt werden.

Durch die Integration innerhalb eines Multiprozessor System-on-Chip (MPSoC) wird zwar Overhead erzeugt, jedoch eine Vervielfältigung der Hardware minimiert. Im Vergleich zu Redundanzansätzen, wie der Quadruplex-Architektur oder der Triplex-Triplex-Architektur (vgl. Kapitel 3.1) wird eine Reduzierung des Hardwareaufwands erreicht. Dieser Aufbau wird erst durch heterogene Multicorearchitekturen ermöglicht. Durch eine separate Spannungsversorgung der Application Processing Unit (APU) bzw. Real-Time Processing Unit (RPU) und des FPGA kann so auch eine Unabhängigkeit der Systemkomponenten erreicht werden. Eine Untersuchung der gemeinsamen Fehlerquellen muss je nach eingesetztem MPSoC durchgeführt werden. Im hier vorliegenden Fall des Xilinx Zynq Ultrascale+ wird beispielsweise die RPU in der *Low-Power Domain* und der FPGA und die APU in der *Full-Power Domain* betrieben.[7]

[7] Weitere Details zur Spannungsversorgung, der Taktverteilung und Isolation der unterschiedlichen Komponenten im Xilinx Zynq Ultrascale+ können [110] entnommen werden.

7 Ganzheitliche Betrachtung und Einordnung

Die in den vorherigen Kapiteln losgelöst beschriebenen Methoden bzw. Ansätze werden in diesem Kapitel zueinander in Bezug gesetzt und gesamtheitlich betrachtet.

Dieses Kapitel gibt einen Überblick über die in dieser Arbeit vorgestellten Methoden zur Überwachung und Fehlererkennung in multicorebasierten Systemen und wie diese miteinander kombiniert werden können. Es soll herausgearbeitet werden, inwiefern Kombinationen der Ansätze möglich sind bzw. benötigt werden, um die Fehlerbilder und die Herausforderungen in multicorebasierten Systemen (vgl. Kapitel 4) zu adressieren. Es wird dargestellt in welchen Kombinationen welche Abdeckung erreicht werden kann.

Die Anforderungen an die Ansätze, welche in Kapitel 5 und 6 vorgestellt wurden, können im Kontext eines Anwendungsfalls sehr unterschiedlich ausfallen und müssen daher Berücksichtigt werden. Diese Anpassungen können in Form von Parametern in der Umsetzung der Ansätze einfließen. Die beschriebenen Komponenten bieten entsprechende Möglichkeiten zur Anpassung an den Anwendungsfall.

Zur Verdeutlichung wird herausgearbeitet, inwieweit die Ansätze auf andere Architekturen übertragbar sind und im Kontext des Anwendungsfalls skaliert und parametriert werden können. Ebenso wird gezeigt, welche Einschränkungen vorhanden sind.

7.1 Ganzheitliche Betrachtung der vorgestellten Methoden

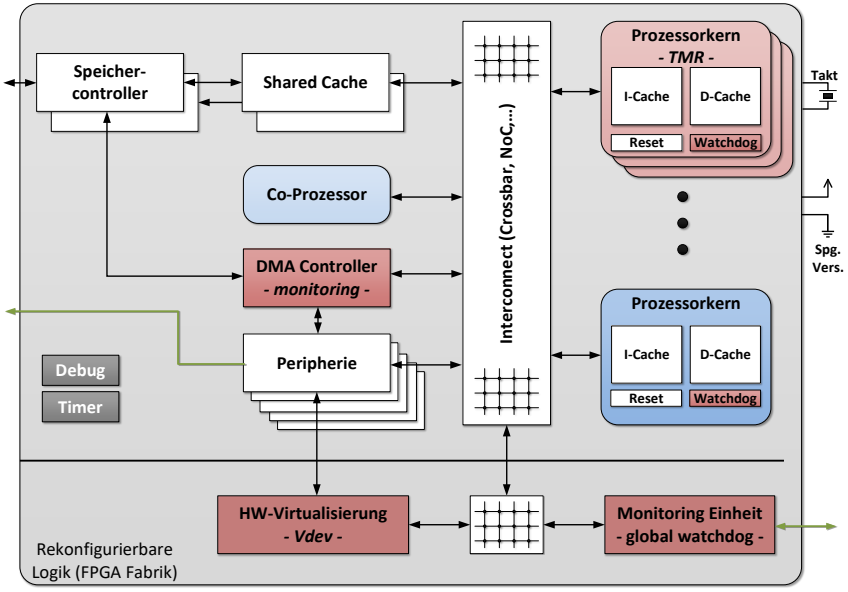


Abbildung 7.1: Gesamtübersicht der kombinierten Monitoringansätze

Die in Kapitel 5.1, 5.2, 5.3 und 6.1 vorgestellten Konzepte können sowohl unabhängig voneinander eingesetzt als auch in einem MPSoC in Kombination integriert werden. In Abbildung 7.1 ist eine schematische Integration der einzelnen Konzepte in ein MPSoC dargestellt. Insgesamt kann durch die Kombination der verschiedenen Ansätze eine sehr gute Überwachungsabdeckung des MPSoC erreicht und der Gesamtzustand evaluiert werden. Alle einzelnen Monitore übertragen die Informationen an die interne *Monitoring Einheit*, welche eine Auswertung auf MPSoC-Level durchführt. Der ermittelte Zustand wird von der *Monitoring Einheit* an eine externe Einheit kommuniziert. Diese dient der Überwachung von Fehlerquellen (CCF), die für den Ausfall des gesamten

MPSoC verantwortlich sein können, z.B. Taktversorgung und Versorgungsspannung.

Ergänzend wird durch die Erkennung von Fehlern dazu beigetragen, die in Kapitel 4.1 beschriebenen Herausforderungen bewältigen zu können. Auch wenn Fehler damit nicht kategorisch vermeidbar sind, wird durch eine Erkennung zu einem sicheren Betrieb beigetragen.

Stellt man nun die Kombination der Monitoringansätze und die möglichen Fehlerbilder aus Applikationssicht (vgl. Kapitel 4.2) gegenüber, so wird deutlich, dass die vorgestellten Ansätze alle drei Hauptkategorien - *Hohe Ausführungslatenz, Fehlerhafte Berechnung/Daten/Zugriffe, Fehlende Funktionsausführung* - adressieren. Dabei wird eine mindestens zweifache Überdeckung der Hauptkategorien erreicht. Die Hauptkategorien können wie folgt abgedeckt werden:

- **Hohe Ausführungslatenz:**

  Hierarchischer Watchdog (Kap. 5.2), DMA-Monitoring (Kap. 5.3), HW-Virtualisierung (Kap. 6.1)
- **Fehlerhafte Berechnung/Daten/Zugriffe:**

  TMR On-Chip (Kap. 5.1), DMA-Monitoring (Kap. 5.3), HW-Virtualisierung (Kap. 6.1)
- **Fehlende Funktionsausführung:**

  TMR On-Chip (Kap. 5.1), Hierarchischer Watchdog (Kap. 5.2)

Die resultierenden Monitoring Informationen ergänzen sich zum Gesamtsystemzustand. In Abhängigkeit vom Safety-Ziel auf Systemlevel kann eine Kombination von zwei oder mehr der vorgestellten Konzepte eingesetzt und trotzdem eine Abdeckung erreicht werden. Mögliche Kombinationen sind in Tabelle 7.1 mit einem Kreuz (x) dargestellt. Dabei stellt eine Zeile jeweils die Kombinationsmöglichkeiten dar. Als Beispiel kann eine Abdeckung durch Kombination des TMR On-Chip Ansatzes mit dem hierarchischen Watchdog oder mit DMA-Monitoring oder mit HW-Virtualisierung erreicht werden.

Tabelle 7.1: Kombinationsmöglichkeiten der Monitoringansätze bei denen alle drei Hauptkategorien der Fehlerbilder abgedeckt werden

	TMR On-Chip	Hier. Wdog	DMA-Mon.	HW-Virt
TMR On-Chip		x	x	x
Hier. Wdog	x		x	x
DMA-Mon.	x	x		
HW-Virt	x	x		

Die Kombinationen (C_i), die zur Erreichung der Abdeckung anwendbar sind, können als logische Verknüpfung interpretiert werden, die sich nicht ausschließen:

$$C_1 = \text{TMR On-Chip} \wedge \text{Hier.Wdog} \tag{7.1}$$
$$C_2 = \text{TMR On-Chip} \wedge \text{DMA-Mon.} \tag{7.2}$$
$$C_3 = \text{TMR On-Chip} \wedge \text{HW-Virt} \tag{7.3}$$
$$C_4 = \text{Hier.Wdog} \wedge \text{DMA-Mon.} \tag{7.4}$$
$$C_5 = \text{Hier.Wdog} \wedge \text{HW-Virt} \tag{7.5}$$

Aus diesen Konstellationen entsteht eine Menge von Kombinationsmöglichkeiten M_C:

$$M_C = \{C_1, C_2, C_3, C_4, C_5\} \tag{7.6}$$

Hieraus resultiert die Mächtigkeit ($|M_C|$) der Menge der Kombinationsmöglichkeiten (C_i):

$$|M_C| = 5 \tag{7.7}$$

Mit all diesen Kombinationen kann eine Abdeckung der Fehlerbilder aus Applikationssicht, welche in Kapitel 4.2 dargestellt sind, erreicht werden. Die Auswahl der Konstellation wird auf Basis der zur Verfügung stehenden Hardware-Architektur durchgeführt. Die vorliegende Hardware-Architektur bedingt somit Einschränkungen der Realisierbarkeit von verschiedenen Ansätzen.

Betrachtet man die verschiedenen Ansätze, wird deutlich, dass sich diese auf unterschiedlichen Ebenen im System befinden. Zusätzlich können die Ansätze mit Methoden der Überwachung in Software kombiniert werden, wie beispielsweise der in Kapitel 5.1.1 und 5.2.1 in Form einer Challenge-Response-Abfrage vorgestellte Ansatz. Eine Challenge-Response-Abfrage sowie weitere Mechanismen, die in Software realisiert werden, umfassen dann alle Ebenen von der Applikation bis zur Hardware. Eine Darstellung dessen ist in Abbildung 7.2 zu finden. Insgesamt kann die Entwicklung und Einbindung der Überwachung als HS/SW Co-Design Ansatz betrachtet werden.

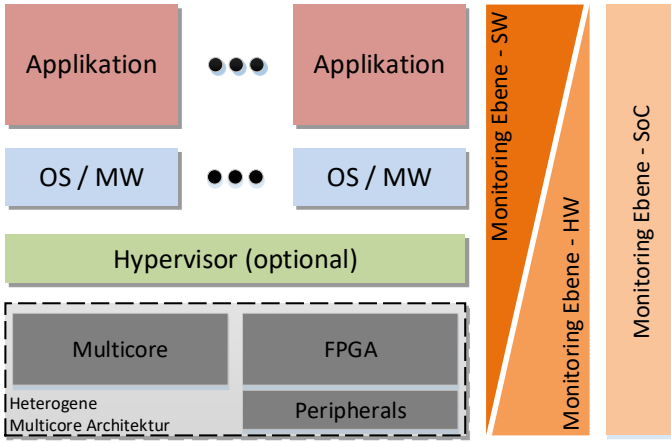


Abbildung 7.2: Monitoringansatz über mehrere Ebenen

Die *Monitoring Ebene - SoC* kann dabei als globale Monitoring Einheit betrachtet werden, welche sämtliche Informationen der unterschiedlichen Ebenen sammelt und auswertet und somit dem Zustand des SoC evaluiert.

Da speziell im Falle eines Mehrkernprozessors die Komplexität der Software[1] durch die eingebrachten Abhängigkeiten besonders hoch ist, wird eine Umsetzung in Hardware bevorzugt. Darüber hinaus wird durch

[1]Hier ist der gesamte Software Stack gemeint und beinhaltet die Applikation, das Betriebssystem sowie weitere optionale Treiberschichten oder Hypervisor.

eine Umsetzung in Hardware und die daraus folgende Transparenz eine Unabhängigkeit geschaffen, die im Kontext der Standards für Sicherheitskritische Systeme (vgl. Kapitel 2.2) einen entscheidenden Aspekt bildet. Wird die Überwachung in Software auf demselben Prozessorkern durchgeführt, so wird ein Fehler in diesem, Auswirkungen sowohl auf die Applikation als auch auf die Überwachung haben. Wird die Überwachung hingegen in Hardware umgesetzt, ist dies zunächst unabhängig von der Applikation und die Überwachung kann im Fehlerfall des Prozessorkerns weiterhin fehlerfrei ausgeführt werden. Durch eine separate Spannungs- sowie Taktversorgung kann eine weitere Trennung erreicht und somit die Unabhängigkeit der Systemteile unterstützt werden. Die kann beispielsweise in heterogenen Architekturen mit rekonfigurierbarer Logik erreicht werden, wenn der FPGA-Anteil separat versorgt wird.

Basierend auf den vorgestellten Konzepten kann die Erkennung von Fehlern im System verbessert werden. Dies trägt dazu bei die zulässigen Ausfallraten, die durch die Standards vorgegeben werden (vgl. z.B. Tabelle 2.1 in Kapitel 2.2.1), zu erreichen. Werden die vorgestellten Konzepte im Kontext von sicherheitskritischen Anwendungen mit SIL-Einstufung eingesetzt, so kann die in Kapitel 2.2.2 vorgestellte Safe Failure Fraction (SFF) verbessert werden. Speziell λ_{DD} in Formel (2.3) wird durch die vorgestellten Mechanismen verbessert und trägt somit zu einer messbaren Verbesserung und zum Erreichen der durch die Standards vorgegeben Ausfallraten bei. Die in die Berechnung der SFF einfließenden Summe der erkannten gefahrbringenden Ausfälle wird durch die Konzepte minimiert. Es wird mindestens ein sicherer Zustand erreicht (entspricht $N = 1$ in Tabelle 2.2), im Falle des vorgestellten Redundanzansatzes und der dynamischen Funktionsmigration wird hingegen die Funktion aufrechterhalten, was $N = 2$ in Tabelle 2.2 entspricht. Wird davon ausgegangen, dass die weiteren Werte für λ und λ_D in Formel (2.3) gleich bleiben und durch die vorgestellten Mechanismen λ_{DD} verbessert wird, so wirkt sich dies direkt auf die SFF aus und trägt somit zur Erfüllung der Anforderung aus den Standards bei.

7.2 Übertragbarkeit und Skalierbarkeit

Basierend auf der vorliegenden Plattform[2], die für das zu entwickelnde System eingesetzt werden soll, kann sich der Einsatz der vorgestellten Methoden unterscheiden. Grundsätzlich sind alle vorgestellten Konzepte in einer heterogenen Multicoreplattform mit rekonfigurierbarer Logik (RL) anwendbar. Speziell durch die Bereitstellung der rekonfigurierbaren Logik entsteht ein wesentlich größerer Entwurfsraum mit einer gesteigerten Flexibilität.

Jedoch sind nicht alle vorgestellten Konzepte auf eine heterogene Multicorearchitektur mit rekonfigurierbarer Logik (RL) angewiesen. Die nachfolgende Tabelle 7.2 gibt einen Überblick über die mögliche Integration der Konzepte in einer heterogenen Commercial Off The Shelf (COTS) Multicorearchitektur mit RL sowie deren Integration in eine Architektur ohne RL, an welcher keine Änderungen der Hardware mehr möglich sind.

Durch die Berücksichtigung der unterschiedlichen verfügbaren Architekturen und dem aufkommenden Trend, wird die Übertragbarkeit weitgehend sichergestellt. In vielen Anwendungsfällen kann die Nutzung einer Architektur mit rekonfigurierbarer Logik vorteilhaft sein, um weitere Co-Prozessoren, Schnittstellen o.ä. integrieren zu können.

Die vorgestellten Ansätze sind weitestgehend skalierbar, was in diesem Zusammenhang einschließt, dass diese mit Blick auf die Zielplattform parametriert werden können. Der vorgestellte Redundanzansatz kann beliebig zu einer *n-out-of-m* Architektur ausgeweitet werden, wenn dies im Systemkontext notwendig ist. Es können weiterhin Ergebnisse einer weiteren, durch Hardware Redundanz abgebildeten, Einheit berücksichtigt und in das Voting mit einbezogen werden. Vergleicht man dies mit der in Kapitel 3.1 vorgestellten Quadruplex-Architektur, kann bereits jeder Kanal, welcher auf einem Mehrkernprozessor basiert, eine interne Redundanz und damit gesteigerte Zuverlässigkeit bereitstellen. Dies geschieht ähnlich zur vorgestellten Triplex-Triplex-Architektur, die jedoch pro Kanal mehrere Rechner benötigt. Im Umkehrschluss ermöglicht dies weitere redundante Kanäle zu minimieren und die Gesamtkosten des Systems im Sinne von Space, Weight and Power

[2]Dies beinhaltet Software und Hardware.

Tabelle 7.2: Integrationsmöglichkeiten der Ansätze in COTS Multicorearchitekturen mit und ohne rekonfigurierbare Logik (RL)

	mit RL	ohne RL	Bemerkungen
TMR On-Chip	*x*	*x*	Umsetzung kann rein in Software erfolgen.
Hier. Wdog	*x*	(*x*)	Die Integration in ein COTS MPSoC ohne rekonf. Logik ist davon abhängig, ob im Prozessor Möglichkeiten zum Reset einzelner Rechenkerne vorhanden sind.
DMA-Mon.	*x*	(*x*)	Die Integration in ein COTS MPSoC ohne rekonf. Logik ist davon abhängig, ob ein frei programmierbarer DMA Controller integriert ist.
HW-Virt	*x*		Umsetzung erfolgt in rekonf. Hardware.
Fkt.-Mig.	*x*		Umsetzung erfolgt in rekonf. Hardware.

(SWAP) zu optimieren. Eine Untersuchung auf gemeinsame Fehlerquellen (Common Cause Failure (CCF)) muss jedoch immer gesondert durchgeführt und entsprechend betrachtet werden.

Analog ist der vorgestellte hierarchische Watchdog je nach Anwendungsfall parametrierbar und skalierbar. Eine erhöhte Anzahl an Rechenkernen kann im globalen Watchdog problemlos integriert werden. Jeder der Rechenkerne muss dafür einen eigenen lokalen Watchdog mitbringen, der die Informationen zum globalen Watchdog weiterreicht. Der globale Watchdog kann fast eine beliebige Anzahl an Rechenkernen unterstützen, bis die Umlaufzeit der Abfragen größer als die kleinste Rückmeldezeit der lokalen Watchdogs ist.

Das hier am Beispiel eines Direct Memory Access-Controllers vorgestellte Überwachungskonzept ist auf andere Bus-Master Komponenten übertragbar. In vielen Fällen sind kleine Reduced Instruction Set Computer (RISC) Prozessoren in solchen Komponenten vorhanden, auf deren Software Einfluss genommen werden kann. Beispiele hierfür sind, wie vorgestellt, DMA-Controller, Hardware Security Module (HSM) oder Plattform Management Einheiten. In Architekturen mit rekonfigurierbarer Logik (RL) kann der Ansatz frei im FPGA implementiert werden und so auf weitere Co-Prozessoren o.ä. angewendet werden.

Sollen mehr Partitionen oder Rechenkerne auf eine gemeinsam genutzte Ressource zugreifen, so bietet die vorgestellte Hardware Virtualisierung die Möglichkeit die Anzahl der virtuellen Geräte zu skalieren und entsprechend der Zahl der zugreifenden Einheiten anzupassen. Der Virtualisierungsmanager (VMNG) erlaubt es, mehr virtuelle Geräte anzubinden, ohne einen weiteren VMNG zu instanziieren. Ebenso kann das Schedulingverfahren, das die Zugriffe auf das physikalische Gerät steuert, parametriert werden. Dabei können unterschiedliche Schedulingverfahren integriert werden, die je nach Anwendungsfall die Anforderungen voll erfüllen.

7.3 Bibliothekskonzept und Einbettung in den Design Flow

Zur Beherrschung der Komplexität und zur Bereitstellung von wiederverwendbaren Pattern, ist ein weiteres Ziel dieser Arbeit, die vorgestellten Konzepte auch für Ingenieure bereitzustellen, die keine Erfahrung mit sicherheitskritischen Anwendungen besitzen. Um dieses Ziel erreichen zu können, wird ein Bibliothekskonzept vorgestellt, welches die erarbeiteten Methoden enthält und somit die Entwicklung sowohl von Software als auch von Hardware erleichtert und eine Möglichkeit für Erweiterungen mitbringt.

Weiterhin wird zur systematischen Anwendung vorgestellt, wie das Bibliothekskonzept im Entwicklungsprozess eingebunden werden kann und welche Informationen benötigt bzw. bereitgestellt werden und welche Einschränkungen sich ergeben.

7.3.1 Aufbau der Safety-Modul Bibliothek

Um eine einfache Wiederverwertung der vorgestellten Konzepte zu ermöglichen, wurde bei der Erstellung darauf geachtet, dass die Konzepte modular, parametrierbar und adaptiv aufgebaut sind. So werden beispielsweise die Hardwareimplementierungen als parametrierbare IP Cores bereitgestellt. Doch nicht nur die Implementierungen in der rekonfigurierbaren Hardware werden in dieser Bibliothek berücksichtigt, auch die Software wird in Form von Treibern und Funktionsmodulen mit eingebunden.

Das es sich bei den vorgestellten Kombinationen der Mechanismen um einen HW/SW Co-Design Ansatz handelt, werden sowohl der Software als auch der Hardware Anteil der Bibliothek in Abbildung 7.3 dargestellt. Die umgesetzten Mechanismen können mit in Kombination mit bestehenden Softwarebibliotheken betrachtet werden. Beispielsweise kann die in Kapitel 5.1 vorgestellte Challenge-Response Architektur, der DMA-Monitoring Ansatz aus Kapitel 5.3 oder der Voting Algorithmus für den Redundanzansatz aus Kapitel 5.1 als Safety Software Modul in die Entwicklung mit einbezogen werden.

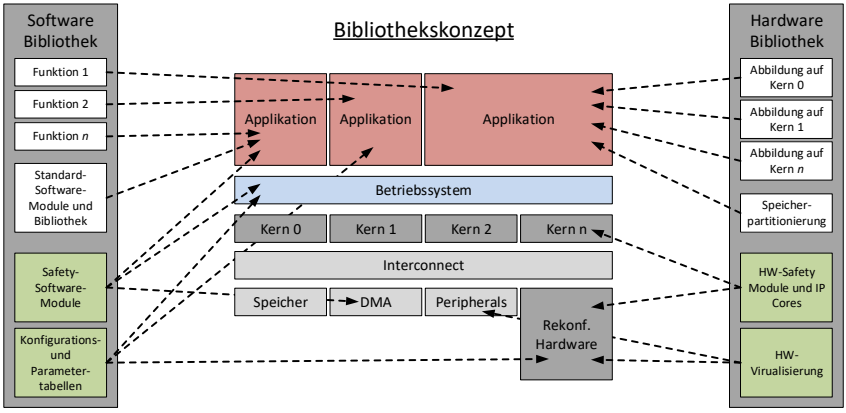


Abbildung 7.3: Bibliothekskonzept für die Entwicklung sicherheitskritischer Multicore Systeme

Parallel zu den Softwarekomponenten, die eine konkrete Funktionalität umsetzen, werden Konfigurationselemente benötigt, welche Einfluss auf Applikation, Betriebssystem und die Hardware-Module der Safety Mechanismen haben. Speziell die Konfiguration und Parametrierung der Ansätze spielt eine entscheidende Rolle, um die korrekte Funktionalität sicherzustellen. Die Konfigurationselemente dienen der Erstellung zulässiger Konfigurationen. Dies kann jedoch nicht unabhängig von der Applikation geschehen und benötigt auch von dieser Informationen. Hierzu gehören beispielsweise Informationen über den Aufbau des Speicherabbilds bzw. der Speicherpartitionierung.

In der Hardware Bibliothek werden die Module in Form von IP Cores gesammelt und können entsprechend der einzubindenden Ansätze ausgewählt, konfiguriert und parametriert werden. Diese werden dann basierend auf der eingesetzten Architektur in die rekonfigurierbare Logik oder auf vorhandene Komponenten abgebildet. Zusätzlich können die IP Cores mit bereits vorhandenen Informationen, beispielsweise über die Abbildung von Softwarefunktionen auf Rechenkerne oder über die Speicherpartitionierung, angereichert werden. In der Bibliothek sind die in Kapitel 6.1 beschriebenen Komponenten der Hard-

ware Virtualisierung und die globale Monitoring Einheit aus Kapitel 5.2 vorhanden.

Beide Bibliotheken können mit weiteren Konzepten erweitert werden. Die Auswahl der Safety Mechanismen hängt stark von der Architektur ab. Die Auswahl und das Vorgehen bei der Auswahl wird im folgenden Kapitel beschrieben.

7.3.2 Einbettung der Bibliothek in den Entwicklungsprozess

Da im Bereich sicherheitskritischer Anwendungen bei der Entwicklung zu meist auf das V-Modell (vgl. Kapitel 2.2.3) zurückgegriffen wird und die auch in den Standards verankert ist, wurde dieses Vorgehen in der vorliegenden Arbeit berücksichtigt. Der Einstieg der Entwicklung ist die Festlegung der Zielfunktion und der Spezifikation. Basierend auf der Zielfunktion und der Spezifikation kann das Sicherheitslevel und das Sicherheitsziel der Funktion eingestuft werden. Diese Einstufung bedingt gewisse Anforderungen an die Safety Mechanismen, die im System integriert werden müssen. Aus der Zielfunktion und der Einstufung des Sicherheitslevels lassen sich die Anforderungen für das System Design ableiten.

Aus den Anforderungen der Applikation[3] und den Safety Anforderungen hinsichtlich Überwachung und Fehlererkennung, können benötigte Mechanismen aus der vorgestellten Bibliothek der Safety Mechanismen ausgewählt und im weiteren Verlauf der Entwicklung berücksichtigt werden. Die Auswahl der Safety Mechanismen kann anhand der in Kapitel 7.1 vorgestellten Tabelle 7.1 auf Basis der Safety Anforderungen an die Überwachung und die zu beachtenden Fehlerbilder getroffen werden. Als Eingangsgröße wird daher davon ausgegangen, dass bereits eine Ableitung der Anforderungen hinsichtlich der zu beachtenden Fehlerbilder aus Applikationssicht durchgeführt wurde und die Anforderungen entsprechend vorliegen.

Zur Auswahl werden jedoch weitere Anforderungen bzw. Einschränkungen durch die geplante HW-Zielplattform bedingt. Diese Einschrän-

[3] Beispielsweise an Rechenleistung oder Speicherbedarf

kungen reduzieren den Entwurfsraum und die Möglichkeiten bei der Auswahl der Mechanismen. Dabei kann durch die Eigenschaften der HW-Zielplattform über die in Kapitel 7.2 vorgestellte Tabelle 7.2 festgestellt werden, welche Mechanismen innerhalb des Entwurfsraums liegen.

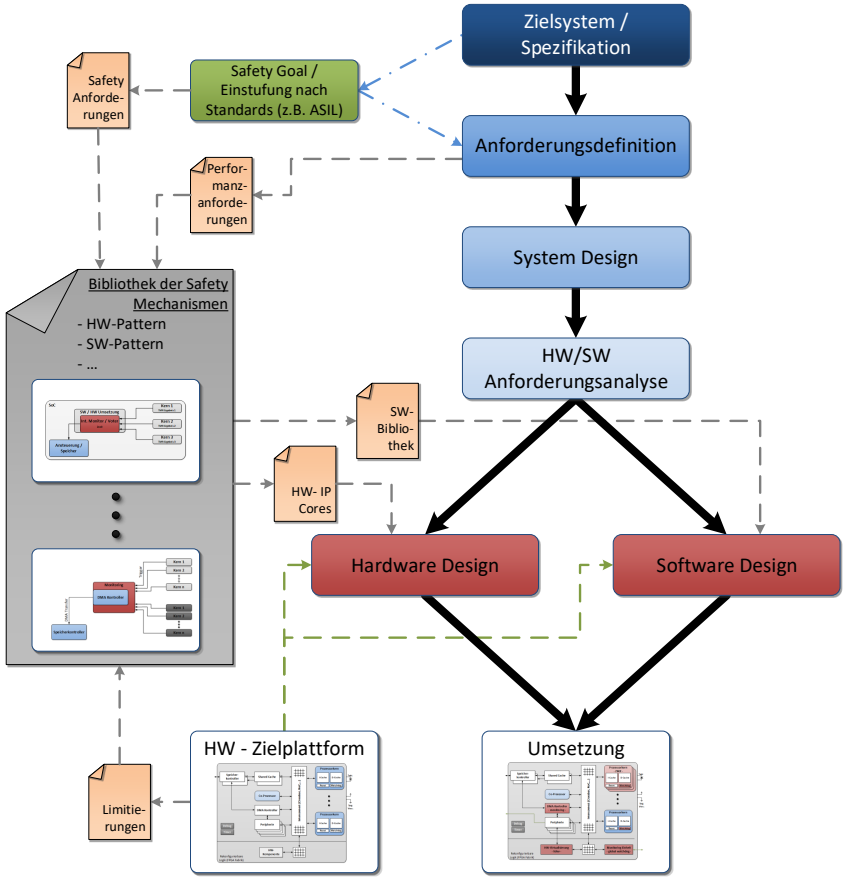


Abbildung 7.4: Einbettung des Bibliothekskonzepts in den Entwicklungsprozess

Dies definiert die Eingangsartefakte der Bibliothek zur Auswahl der Mechanismen. Als Rückfluss in den Entwicklungsprozess bietet die Bibliothek die Möglichkeit, sowohl im Software als auch im Hardware Design, bereits existierende Komponenten wieder zu verwenden. Hierzu gehört auch die Bereitstellung von Konfigurationen, die für die Parametrierung der Mechanismen benötigt werden. Diese werden in die eigentliche Entwicklung der Applikation integriert und können somit auf die Zielplattform abgebildet werden.

Die Einbindung der Bibliothek in den Entwicklungsprozess und dessen Ablauf ist in Abbildung 7.4 dargestellt und zeigt die ausgetauschten Artefakte.

Basierend auf den eingehenden Anforderungen und der Beschreibung der HW-Zielplattform können die benötigten Mechanismen ausgewählt werden. Kann die gewünschte Abdeckung mit mehreren Mechanismen bzw. deren Kombination erreicht werden, so kann eine optimierte Lösung, beispielsweise hinsichtlich der Kosten oder der Performanz, in den folgenden Prozessschritten beispielsweise durch eine Entwurfsraum Exploration (Design Space Exploration) erzielt werden. Für die Auswahl der Mechanismen ist aber nach wie vor ein gutes Detailwissen der Zielfunktion und der Systemarchitektur notwendig, eine automatisierte Auswahl wurde nicht umgesetzt.

In Abbildung 7.5 werden die durchlaufenen Schritte innerhalb der Bibliothek entlang des Entwicklungsprozesses detailliert dargestellt. Es wird verdeutlicht, dass bei einer Verfügbarkeit von mehreren Kombinationen bzw. Mechanismen eine Schleife zur Bewertung der Kosten durchlaufen wird. Ist nur ein Mechanismus vorhanden, kann dieser auch Bewertet und eine Kostenabschätzung durchgeführt werden, eine weitere Auswahlmöglichkeit ist jedoch nicht vorhanden. Sollte keine Kombination bzw. kein Mechanismus die Anforderungen erfüllen können, wird der Durchlauf abgebrochen und es muss auf herkömmliche Weise ein alternativer Mechanismus entwickelt werden. Es kann in diesem Fall nicht auf die Bibliothek zurückgegriffen werden.

Durch dieses systematische und methodische Vorgehen und die definierten Ein- und Ausgangsgrößen der Bibliothek können wiederholbare Ergebnisse erzielt und nachvollziehbar entwickelt werden.

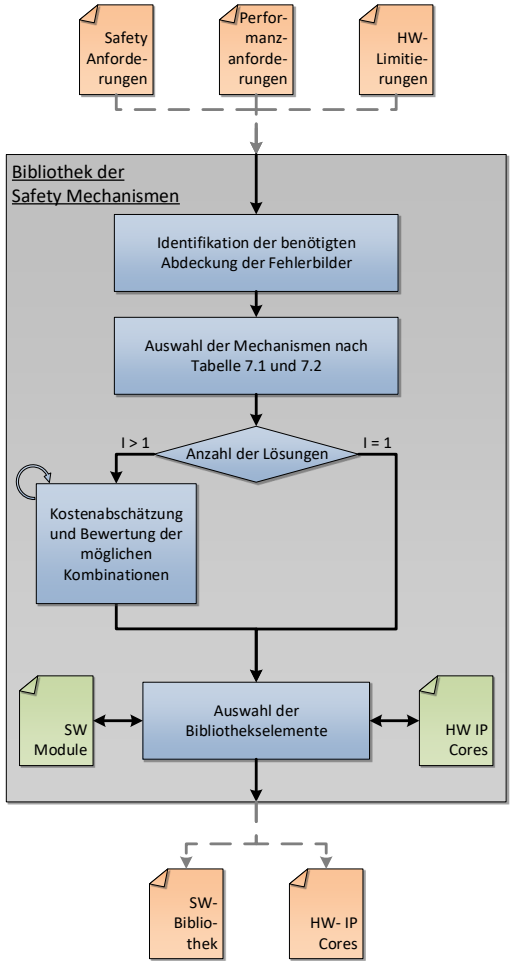


Abbildung 7.5: Ablauf der Auswahl der Mechanismen innerhalb der Bibliothek

8 Schlussfolgerung und Ausblick

Die nachfolgenden Abschnitte fassen die in dieser Arbeit vorgestellten Konzepte und Ergebnisse zusammen und geben einen Ausblick über anschließende, darauf aufbauende mögliche Arbeiten und Erweiterungen.

8.1 Zusammenfassung und Ergebnisse der Arbeit

Durch die in dieser Arbeit erforschten und erarbeiteten Methoden zur Überwachung und Fehlererkennung sowie der sicheren Nutzung von gemeinsam genutzten Ressourcen, wurde eine Sammlung an Methoden und Entwurfsmustern geschaffen, die einen Einsatz von heterogenen Mehrkernprozessoren mit rekonfigurierbarer Logik in sicherheitskritischen Anwendungen wesentlich vereinfachen. Ein Einsatz dieser Technologie in sicherheitskritischen Anwendungen war seither, wenn überhaupt, nur mit sehr großen Einschränkungen möglich.

Basierend auf den systematisch abgeleiteten Herausforderungen bei der Entwicklung von multicorebasierten Systemen im sicherheitskritischen Umfeld und den daraus entstehenden spezifischen Fehlerbildern, wird in dieser Arbeit ein Baukastensystem mit Methoden zur Überwachung und Fehlererkennung bereitgestellt, das die Entwicklung und den Einstieg vereinfacht. Durch die zugrundeliegende Modularität und Parametrierbarkeit kann dieses Baukastensystem auf einfache Weise erweitert und mit anderen Bibliotheken verbunden werden. Durch die Möglichkeit zur Anbindung an bekannte Entwicklungsprozesse wird eine Integration in bekannte und etablierte Entwicklungsschritte vereinfacht.

Selbst die Vielfalt an verfügbaren Mehrkernprozessoren kann durch die beschriebenen Konzepte und deren prototypische Umsetzung auf unterschiedlichen Plattformen beherrscht werden. Die Bibliothekselemente können je nach Zielarchitektur wiederverwendet werden und bedürfen keiner großen Anpassung.

Durch die Bereitstellung von adaptiven, parametrierbaren Multi-Level Monitoring-Mechanismen sowohl in Hardware als auch in Software, wird eine Abdeckung der Überwachung über alle Ebenen des Systems von der Applikation über das Betriebssystem bis hin zur Hardware erreicht. Das in einer typischen Entwicklung von sicherheitskritischen eingebetteten Systemen vorhandene HW/SW Co-Design kann auf diese Weise bestens unterstützt werden.

Die vorgestellte Möglichkeit zur Abbildung einer sicheren Ressourcennutzung in eine rekonfigurierbare Logik bringt eine bisher nicht verfügbare Möglichkeit, gemeinsam genutzte Ressourcen bei sehr geringer zusätzlicher Latenz sicher zu nutzen ohne eine erhöhte Komplexität in Software zu bedingen. Diese Variante kann nicht nur innerhalb eines FPGA implementiert werden, auch eine Integration in künftige Hardware Architekturen in Form eines ASIC ist möglich.

Speziell im industriellen Umfeld spielt Flexibilität und Übertragbarkeit eine entscheidende Rolle, um die Dauer von Entwicklungszyklen reduzieren zu können. Durch die vorgestellten Architekturmuster, die auf eine breite Palette von Architekturen übertragbar sind, können bei der Entwicklung bereits bekannte Muster auf eine neue HW-Zielplattform in einfachster Weise übertragen werden. So kann in industriellen Anwendungen die Zeitspanne bis zur Platzierung eines Produkts am Markt reduziert werden.

Für zukünftige Anwendungen im sicherheitskritischen Umfeld werden immer weiter steigende Anforderungen aufkommen, die sich durch den steigenden Grad an Automatisierung ergeben. Durch eine dynamische Migration von Funktionen, die ohne eine permanente Bereitstellung von Ressourcen auskommt, wird eine Beispielarchitektur geschaffen, Systeme ohne Vervielfältigung der Hardware im Fail-Operational Kontext zu entwickeln.

Zusammenfassend stellt die vorliegende Arbeit eine Basis für den Einsatz und die Beherrschung von heterogenen Multicorearchitekturen

mit rekonfigurierbarer Logik in sicherheitskritischen Anwendungen bereit und gibt einen grundlegenden Überblick über die Herausforderungen und Spezifika von Mehrkernprozessoren.

8.2 Ausblick

Da der Hauptfokus der Arbeit auf der konzeptionellen Erarbeitung von Mechanismen und Methoden lag, wurden Kompromisse beim Ressoucenverbrauch und dem Energieverbrauch eingegangen. In vielen Anwendungsfällen kann jedoch gerade der Energieverbrauch und der Ressourcenaufwand eine wichtige zusätzliche Rolle spielen, weswegen eine Optimierung und Analyse in diese Richtung eine mögliche Erweiterung dieser Arbeit darstellt.

Auf Basis eines konkreten Anwendungsfalles kann zusätzlich eine detaillierte Berechnung von Ausfallraten durchgeführt werden. In dieser Arbeit wurde darauf verzichtet, da für eine Berechnung Informationen der HW-Hersteller und der Anwendung benötigt werden, die in der Regel nur in Serienprodukten zur Verfügung gestellt und im Falle einer Zertifizierung nachgewiesen werden müssen.

Zur weiteren Umsetzung und Untersuchung von Fail-Operational Systemen, kann im konkreten Anwendungsfall eine detaillierte Untersuchung von Abhängigkeiten der Systemkomponenten durchgeführt werden. In den beschriebenen Konzepten wurde dies nicht nachgeprüft, es wurden lediglich die Informationen aus verfügbaren Quellen zu Rate gezogen.

Bei aktuellen und neuen Designs wird mehr und mehr eine modellbasierte Entwicklung verfolgt. Beim Einsatz eines modellbasierten Top-Down Entwicklungsprozesses ist es von besonderem Vorteil die Hardware frühzeitig in den Entwicklungsprozess einbeziehen zu können. Hierfür werden Modelle auf unterschiedlichen Ebenen benötigt, die Safety Mechanismen einschließen. Diese Modelle können für die Allokation und Scheduling von Software verwendet werden. Daher kann eine Modellierung der Mechanismen auf Basis eines gängigen Standardformats durchgeführt werden, sobald dies vorhanden ist. Die Modelle können ebenso in einer Design Space Exploration als Eingangsarte-

fakte frühzeitig im Entwicklungsprozess einbezogen und berücksichtigt werden.

Verzeichnisse

Abbildungsverzeichnis

Tabellenverzeichnis

Formelverzeichnis

Abkürzungsverzeichnis

ADAS	Advanced Driver Assistance Systems
AHB	Advanced High-Performance Bus
ALU	Arithmetic Logic Unit
APB	Advanced Peripheral Bus
API	Application Programming Interface
APPS	Applikationssystem
APU	Application Processing Unit
ARM	Advanced RISC Machines
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface Bus
BIST	Build-In Self Test
BS	Backup System
CAN	Controller Area Network
CCF	Common Cause Failure
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
DAL	Design Assurance Level
DDR	Double Data Rate
DMA	Direct Memory Access
DVE	Datenverarbeitungseinheit
E/A, I/O, IO	Eingabe/Ausgabe (Input/Output)
E/E	Electric/Electronic

E/E/PES	Elektrische/Elektronische/Programmierbare Elektronische Systeme
ECC	Error Correction Code
ECU	Electronic Control Unit
FIFO	First In First Out
FMEDA	Failure Modes, Effects and Diagnostic Analysis
FPGA	Field Programmable Gate Array
FTA	Fault Tree Analysis
GWD	Globaler Watchdog
GPIO	General Purpose Input Output
GPU	Graphics Processing Unit
HMI	HumanMachine Interface
HSM	Hardware Security Module
HW	Hardware
I/O	Input/Output
I²C	Inter-Integrated Circuit
IC	Integrated Circuit
IMA	Intergrated Modular Avionics
IP	Intellectual Property
IRQ	Interrupt
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LS	Lock-Step
LWD	Lokaler Watchdog
MCIRQ	Multicore Interrupt Controller
MCP	Multicore Processor
MMU	Memory Management Unit
MPSoC	Multiprozessor System-on-Chip

MPU	Memory Protection Unit
MTBF	Mean Time Between Failure
MTBR	Mean Time Between Repair
MTTF	Mean Time To Failure
MW	Middleware
NoC	Network-on-Chip
OS	Operating System
PCI(e)	Peripheral Component Interconnect (express)
PFH	Probability of one dangerous failure per hour
PKW	Personenkraftwagen
PL	Programmable Logic
PMU	Platform Management Unit
PS	Processing System
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RL	Rekonfigurierbare Logik
RPU	Real-Time Processing Unit
SDMA	Smart Direct Memory Access
SEE	Single Event Effect
SEU	Single Event Upset
SFF	Safe Failure Fraction
SIL	Safety Integrity Level
SoC	System-on-Chip
SPARC	Scalable Processor ARChitecture
SPI	Serial Peripheral Interface
SR-IOV	Single Root I/O Virtualisierung
SRAM	Static random-access memory
SW	Software

SWAP	Space, Weight and Power
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receifer Transmitter
UML	Unified Modelling Language
VM	Virtuelle Maschine/Virtual Machine
VMM	Virtual Machine Monitor
VMNG	Virtualisierungsmanager
WCAT	Worst Case Access Time
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
XPPU	Xilinx Peripheral Protection Unit

Literatur- und Quellennachweise

[1] „Intel Developer Conference“. In: Jan. 2009.

[2] T. Economist. *Technology Quarterly after Moore's Law - Double, double, toil and trouble*. URL: http://www.economist.com/technology-quarterly/2016-03-12/after-moores-law.

[3] M. Wolfe. *Compilers and More: Is Amdahl's Law Still Relevant?* Jan. 2015. URL: https://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/.

[4] P. Leteinturier, S. Brewerton und K. Scheibert. „MultiCore Benefits & Challenges for Automotive Applications“. In: *SAE World Congress & Exhibition*. SAE International, Apr. 2008. DOI: 10.4271/2008-01-0989. URL: http://dx.doi.org/10.4271/2008-01-0989.

[5] B. Parhami. „Defect, Fault, Error,..., or Failure?“ In: *IEEE Transactions on Reliability* 46.4 (Dez. 1997), S. 450–451. ISSN: 0018-9529. DOI: 10.1109/TR.1997.693776.

[6] International Standardization Organization. *ISO 26262 Road vehicles - Functional Safety*. 1. Aufl. Nov. 2011.

[7] P. Löw, R. Pabst und E. Petry. *Funktionale Sicherheit in der Praxis : Anwendung von DIN EN 61508 und ISO DIS 26262 bei der Entwicklung von Serienprodukten*. 1. Heidelberg: dpunkt-Verl., 2010. ISBN: 978-3-89864-570-6.

[8] V. Lefftz, J. Bertrand, H. Casse, C. Clienti, P. Coussy, L. Maillet-Contoz, P. Mercier, P. Moreau, L. Pierre und E. Vaumorin. „A design flow for critical embedded systems“. In: *International Symposium on Industrial Embedded System (SIES)*. Juli 2010, S. 229–233. DOI: 10.1109/SIES.2010.5551393.

[9] International Electrotechnical Commision. *IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems*. 2. Aufl. Apr. 2010. ISBN: 978-2-88910-524-3.

[10] S. AG. *Funktionale Sicherheit in der Prozessinstrumentierung mit Einstufung SIL Fragen, Beispiele, Hintergründe*. Apr. 2007. URL: http://docplayer.org/7120238-Siemens-ag-2007-funktionale-sicherheit-in-der-prozessinstrumentierung-mit-einstufung-sil-fragen-beispiele-hintergruende-broschuere-april-2007.html.

[11] H. Gall. „Functional safety IEC 61508/IEC 61511 the impact to certification and the user“. In: *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*. IEEE. 2008, S. 1027–1031.

[12] S. H. Jeon, J. H. Cho, Y. Jung, S. Park und T. M. Han. „Automotive hardware development according to ISO 26262“. In: *13th International Conference on Advanced Communication Technology (ICACT2011)*. Feb. 2011, S. 588–592.

[13] Z. Q. Zhai und B. H. Zhou. „Functional Safety Management in Microcontroller Design and Development Process: the Case of Safety-Critical Vehicle Systems“. In: *Advances in Civil Engineering, CEBM 2011*. Bd. 255. Advanced Materials Research. Trans Tech Publications, Aug. 2011, S. 2179–2182. DOI: 10.4028/www.scientific.net/AMR.255-260.2179.

[14] N. Navet, A. Monot, B. Bavoux und F. Simonot-Lion. „Multi-source and multicore automotive ECUs - OS protection mechanisms and scheduling“. In: *2010 IEEE International Symposium on*

Industrial Electronics. Juli 2010, S. 3734–3741. DOI: 10.1109/ISIE.2010.5637677.

[15] S. Benz. „Eine Entwicklungsmethodik für sicherheitsrelevante Elektroniksysteme im Automobil. Karlsruhe, Universität Karlsruhe (TH)“. Diss. Dissertation, April, 2004.

[16] R. Mariani, P. Fuhrmann und B. Vittorelli. „Fault-robust microcontrollers for automotive applications“. In: *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*. IEEE. 2006, 6–pp.

[17] L. M. Kinnan. „Use of multicore processors in avionics systems and its potential impact on implementation and certification“. In: *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*. Okt. 2009. DOI: 10.1109/DASC.2009.5347560.

[18] H. Flühr. „Avionik und Flugsicherungstechnik–Einführung in Kommunikationstechnik, Navigation“. In: *Surveillance, Springer-Verlag, Berlin* (2010).

[19] R. L. Eveleens. „Integrated modular avionics development guidance and certification considerations“. In: *Mission Systems Engineering* 2 (2006), S. 1120–1132.

[20] I. RTCA. *DO-254 Design Assurance Guidance For Airborne Electronic Hardware*. Apr. 2000.

[21] R. Berger, L. Burcin, D. Hutcheson, J. Koehler, M. Lassa, M. Milliser, D. Moser, D. Stanley, R. Zeger, B. Blalock et al. „The rad6000mc system-on-chip microcontroller for spacecraft avionics and instrument control“. In: *Aerospace Conference, 2008 IEEE*. IEEE. 2008, S. 1–14.

[22] L. Gagea und I. Rajkovic. „Designing devices for avionics applications and the DO-254 guideline“. In: *Semiconductor Conference, 2008. CAS 2008. International*. Bd. 2. IEEE. 2008, S. 377–380.

[23] A. Kornecki und J. Zalewski. „Software certification for safety-critical systems: A status report“. In: *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*. IEEE. 2008, S. 665–672.

[24] J. Rushby und P. S. Miner. „Modular certification“. In: (2002).

[25] Certification Authorities Software Team (CAST). *Position Paper CAST-32A - Multi-core Processors*. Rev. 0. Nov. 2016.

[26] European Aviation Safety Agency (EASA). *Certification memorandum - SWCEH-001 Development Assurance of Airborne Electronic Hardware*. Rev. 01. Aug. 2011.

[27] European Aviation Safety Agency (EASA). *Certification memorandum - SWCEH-002 Software Aspects of Certification*. Rev. 01. März 2012.

[28] T. Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer-Verlag, 2010.

[29] W. K. Giloi. *Rechnerarchitektur*. Springer-Verlag, 2013.

[30] H. Malz. „Von Neumann—Rechnerarchitektur“. In: *Rechnerarchitektur*. Springer, 2004, S. 43–142.

[31] N. Instruments. *Einführung in die FPGA-Technologie: Die 5 größten Vorteile*. Mai 2012. URL: http://www.ni.com/white-paper/6984/de/.

[32] R. Mariani und P. Fuhrmann. „Comparing fail-safe microcontroller architectures in light of IEC 61508“. In: *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*. IEEE. 2007, S. 123–131.

[33] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri und S. Pezzini. „Fault-tolerant Platforms for Automotive Safety-critical Applications“. In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '03. San Jose, California, USA: ACM,

2003, S. 170–177. ISBN: 1-58113-676-5. DOI: 10.1145/951710.951734. URL: http://doi.acm.org/10.1145/951710.951734.

[34] F. L. Kastensmidt, L. Sterpone, L. Carro und M. S. Reorda. „On the optimal design of triple modular redundancy logic for SRAM-based FPGAs“. In: *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*. IEEE Computer Society. 2005, S. 1290–1295.

[35] L. Sterpone, M. Aguirre, J. Tombs und H. Guzmán-Miranda. „On the design of tunable fault tolerant circuits on sram-based fpgas for safety critical applications“. In: *Proceedings of the conference on Design, automation and test in Europe*. ACM. 2008, S. 336–341.

[36] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi und D. A. Connors. „PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures“. In: *IEEE Transactions on Dependable and Secure Computing* 6.2 (Apr. 2009), S. 135–148. ISSN: 1545-5971. DOI: 10.1109/TDSC.2008.62.

[37] J. Braun und J. Mottok. „Fail-safe and fail-operational systems safeguarded with coded processing“. In: *Eurocon 2013*. Juli 2013, S. 1878–1885. DOI: 10.1109/EUROCON.2013.6625234.

[38] M. A. Haque, H. Aydin und D. Zhu. „Energy-aware task replication to manage reliability for periodic real-time applications on multicore platforms“. In: *2013 International Green Computing Conference Proceedings*. Juni 2013, S. 1–11. DOI: 10.1109/IGCC.2013.6604518.

[39] M. H. Mottaghi und H. R. Zarandi. „DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors“. In: *Microprocessors and Microsystems* 38.1 (2014), S. 88–97. ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2013.11.013. URL: http://www.sciencedirect.com/science/article/pii/S0141933113001993.

[40] A. Kohn, M. Käßmeyer, R. Schneider, A. Roger, C. Stellwag und A. Herkersdorf. „Fail-operational in safety-related automotive multi-core systems“. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. Juni 2015, S. 1–4. DOI: 10.1109/SIES.2015.7185051.

[41] M. D. Powell, A. Biswas, S. Gupta und S. S. Mukherjee. „Architectural Core Salvaging in a Multi-core Processor for Hard-error Tolerance“. In: *SIGARCH Comput. Archit. News* 37.3 (Juni 2009), S. 93–104. ISSN: 0163-5964. DOI: 10.1145/1555815.1555769. URL: http://doi.acm.org/10.1145/1555815.1555769.

[42] I. T. AG. *AURIX™ – TC297T/TC298T/TC299T Product Brief*. Aug. 2016.

[43] T. Instruments. *RM44Lx20 16- and 32-Bit RISC Flash Microcontroller*. Nov. 2016.

[44] Freescale. *MPC5746M Microcontroller Reference Manual*. Dez. 2012.

[45] R. Mariani und P. Fuhrmann. „Comparing fail-safe microcontroller architectures in light of IEC 61508“. In: *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*. Sep. 2007, S. 123–131. DOI: 10.1109/DFT.2007.63.

[46] M. Baleani, L. Mangeruca, M. Peri und S. Pezzini. „Multi Processor Micro-Controllers for Automotive Safety-Critical Applications“. In: *{IFAC} Proceedings Volumes* 37.22 (2004). {IFAC} Symposium on Advances in Automotive Control 2004, Salerno, Italy, 19-23 April 2004, S. 41–46. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)30319-1. URL: http://www.sciencedirect.com/science/article/pii/S1474667017303191.

[47] I. T. AG. *Cost-Optimized Safety Computing Platform - XC2300 and CIC61508*. Jan. 2012.

[48] J. Eltze. *Flexible CPU Architecture to Handle Single-and Multi-core Applications in Embedded Automotive Systems*. Techn. Ber. SAE Technical Paper, 2009.

[49] Freescale. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual*. 1. Aufl. Jan. 2012.

[50] iav automotive engineering. *E-Gas Monitoring Concepts*. Juli 2013. URL: http://www.iav.com/publikationen/technische-veroeffentlichungen/e-gas-monitoring-concepts.

[51] J. Sosnowski. „Transient fault tolerance in digital systems“. In: *IEEE Micro* 14.1 (Feb. 1994), S. 24–35. ISSN: 0272-1732. DOI: 10.1109/40.259897.

[52] A. Bondavalli, S. Chiaradonna, F. D. Giandomenico und F. Grandoni. „Threshold-based mechanisms to discriminate transient from intermittent faults“. In: *IEEE Transactions on Computers* 49.3 (März 2000), S. 230–245. ISSN: 0018-9340. DOI: 10.1109/12.841127.

[53] R. Schneider, M. Kalhammer, D. Eberhard und S. Brewerton. „Basic Single-Microcontroller Monitoring Concept for Safety Critical Systems“. In: *SAE World Congress & Exhibition*. SAE International, Apr. 2007. DOI: 10.4271/2007-01-1488. URL: http://dx.doi.org/10.4271/2007-01-1488.

[54] S. Brewerton, R. Schneider und D. Eberhard. „Implementation of a Basic Single-Microcontroller Monitoring Concept for Safety Critical Systems on a Dual-Core Microcontroller“. In: *SAE World Congress & Exhibition*. SAE International, Apr. 2007. DOI: 10.4271/2007-01-1486. URL: http://dx.doi.org/10.4271/2007-01-1486.

[55] A. Dixit und A. Wood. „The impact of new technology on soft error rates“. In: *2011 International Reliability Physics Symposium*. Apr. 2011, 5B.4.1–5B.4.7. DOI: 10.1109/IRPS.2011.5784522.

[56] R. Baumann. „The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction“. In: *Digest. International Electron Devices Meeting,* Dez. 2002, S. 329–332. DOI: 10.1109/IEDM.2002.1175845.

[57] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas und X. Vera. „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.

[58] C. Ciordas, T. Basten, A. Rădulescu, K. Goossens und J. V. Meerbergen. „An Event-based Monitoring Service for Networks on Chip“. In: *ACM Trans. Des. Autom. Electron. Syst.* 10.4 (Okt. 2005), S. 702–723. ISSN: 1084-4309. DOI: 10.1145/1109118.1109126. URL: http://doi.acm.org/10.1145/1109118.1109126.

[59] B. Vermeulen und K. Goossens. „A Network-on-Chip monitoring infrastructure for communication-centric debug of embedded multi-processor SoCs“. In: *2009 International Symposium on VLSI Design, Automation and Test*. Apr. 2009, S. 183–186. DOI: 10.1109/VDAT.2009.5158125.

[60] J. Andersson, J. Gaisler und R. Weigand. „Next generation multipurpose microprocessor“. In: *Int. Conf. on Data Systems in Aerospace (DASIA), Hungary*. 2010.

[61] L. Shannon, E. Matthews, N. C. Doyle und A. Fedorova. „Performance monitoring for multicore embedded computing systems on FPGAs“. In: *CoRR* abs/1508.07126 (2015). URL: http://arxiv.org/abs/1508.07126.

[62] L. Fiorin, G. Palermo und C. Silvano. „A Monitoring System for NoCs“. In: *Proceedings of the Third International Workshop on Network on Chip Architectures*. NoCArc '10. Atlanta, Georgia, USA:

ACM, 2010, S. 25–30. ISBN: 978-1-4503-0397-2. DOI: 10.1145/1921249.1921257. URL: http://doi.acm.org/10.1145/1921249.1921257.

[63] M. A. Al Faruque, T. Ebi und J. Henkel. „ROAdNoC: Runtime Observability for an Adaptive Network on Chip Architecture". In: *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '08. San Jose, California: IEEE Press, 2008, S. 543–548. ISBN: 978-1-4244-2820-5. URL: http://dl.acm.org/citation.cfm?id=1509456.1509577.

[64] P. G. D. Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini und G. D. Micheli. „A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework". In: *2006 IFIP International Conference on Very Large Scale Integration*. Okt. 2006, S. 140–145. DOI: 10.1109/VLSISOC.2006.313218.

[65] J. Nowotsch und M. Paulitsch. „Leveraging Multi-core Computing Architectures in Avionics". In: *2012 Ninth European Dependable Computing Conference*. Mai 2012, S. 132–143. DOI: 10.1109/EDCC.2012.27.

[66] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener und M. Schmidt. „Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement". In: *2014 26th Euromicro Conference on Real-Time Systems*. Juli 2014, S. 109–118. DOI: 10.1109/ECRTS.2014.20.

[67] J. Nowotsch, M. Paulitsch, A. Henrichsen, W. Pongratz und A. Schacht. „Monitoring and WCET Analysis in COTS multi-core-SoC-based Mixed-criticality Systems". In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '14. Dresden, Germany: European Design und Automation Association, 2014, 67:1–67:5. ISBN: 978-3-9815370-2-4. URL: http://dl.acm.org/citation.cfm?id=2616606.2616689.

[68] M. El Shobaki. „On-chip monitoring for non-intrusive hardware/software observability“. Diss. Uppsala universitet, 2004.

[69] J. Zhao, S. Madduri, R. Vadlamani, W. Burleson und R. Tessier. „A Dedicated Monitoring Infrastructure for Multicore Processors“. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.6 (Juni 2011), S. 1011–1022. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2010.2043964.

[70] N. Ho, P. Kaufmann und M. Platzner. „A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms“. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sep. 2014, S. 1–4. DOI: 10.1109/FPL.2014.6927437.

[71] G. Kornaros und D. Pnevmatikatos. „Real-Time Monitoring of Multicore SoCs through Specialized Hardware Agents on NoC Network Interfaces“. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. Mai 2012, S. 248–255. DOI: 10.1109/IPDPSW.2012.27.

[72] C. Ciordas, K. Goossens, A. Radulescu und T. Basten. „NoC monitoring: Impact on the design flow“. In: *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*. IEEE. 2006, 4–pp.

[73] R. B. Mouhoub und O. Hammami. „NoC monitoring hardware support for fast NoC design space exploration and potential NoC partial dynamic reconfiguration“. In: *Industrial Embedded Systems, 2006. IES'06. International Symposium on*. IEEE. 2006, S. 1–10.

[74] Xilinx. *AXI Performance Monitor v5.0 - LogiCORE IP Product Guide*. Okt. 2016.

[75] M. Koibuchi, H. Matsutani, H. Amano und T. M. Pinkston. „A Lightweight Fault-Tolerant Mechanism for Network-on-Chip“. In: *Proceedings of the Second ACM/IEEE International Symposium*

on Networks-on-Chip. NOCS '08. Washington, DC, USA: IEEE Computer Society, 2008, S. 13–22. ISBN: 978-0-7695-3098-7. URL: http://dl.acm.org/citation.cfm?id=1397757.1397982.

[76] H. Shimada, A. Courbot, Y. Kinebuchi und T. Nakajima. „A Lightweight Monitoring Service for Multi-core Embedded Systems". In: *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Mai 2010, S. 202–209. DOI: 10.1109/ISORC.2010.12.

[77] R. Azimi, D. K. Tam, L. Soares und M. Stumm. „Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring". In: *SIGOPS Oper. Syst. Rev.* 43.2 (Apr. 2009), S. 56–65. ISSN: 0163-5980. DOI: 10.1145/1531793.1531803. URL: http://doi.acm.org/10.1145/1531793.1531803.

[78] W. Zhang. „Design and Implementation of Multi-core Support for an Embedded Real-time Operating System for Space Applications". Diss. KTH Royal Institute of Technology, 2015.

[79] N. Navet, A. Monot, B. Bavoux und F. Simonot-Lion. „Multi-source and Multicore Automotive ECUs-OS Protection Mechanisms and Scheduling". In: *International Symposium on Industrial Electronics-ISIE 2010*. 2010.

[80] A. Aguiar und F. Hessel. „Embedded systems' virtualization: The next challenge?" In: *Rapid System Prototyping, 21st IEEE Int. Symposium on*. Juni 2010, S. 1–7. DOI: 10.1109/RSP.2010.5656430.

[81] G. Heiser. „The Role of Virtualization in Embedded Systems". In: *Proceedings of 1st Workshop on Isolation and Integration in Embedded Systems*. ACM, 2008.

[82] J. Sugerman, G. Venkitachalam und B.-H. Lim. „Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor". In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*.

[83] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima und A. Mallick. „Xen 3.0 and the Art of Virtualization“. In: *Proceedings of the 2005 Ottawa Linux Symposium*. Juli 2005.

[84] A. Menon, J. R. Santos, Y. Turner, G. (Janakiraman und W. Zwaenepoel. „Diagnosing Performance Overheads in the Xen Virtual Machine Environment“. In: *Proceedings of the 1st ACM/USENIX Int. Conference on Virtual Execution Environments '05*. Chicago, IL, USA. ISBN: 1-59593-047-7.

[85] J. Kim, S. Lee und H. Jin. „Fieldbus Virtualization for Integrated Modular Avionics“. In: *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference*.

[86] S. Gansel, S. Schnitzer, F. Dürr, K. Rothermel und C. Maihöfer. „Towards Virtualization Concepts for Novel Automotive HMI Systems“. English. In: *Embedded Systems: Design, Analysis and Verification*. 2013. ISBN: 978-3-642-38852-1. DOI: 10.1007/978-3-642-38853-8_18.

[87] H. Raj und K. Schwan. „High Performance and Scalable I/O Virtualization via Self-virtualized Devices“. In: *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. HPDC '07. Monterey, California, USA, 2007. ISBN: 978-1-59593-673-8.

[88] C. Herber, D. Reinhardt, A. Richter und A. Herkersdorf. „HW/SW trade-offs in I/O virtualization for Controller Area Network“. In: *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE. 2015, S. 1–6.

[89] D. Reinhardt, M. Güntner und S. Obermeir. „Virtualized Communication Controllers in Safety-Related Automotive Embedded Systems“. English. In: *Architecture of Computing Systems - ARCS*

2015. Lecture Notes in Computer Science. 2015. ISBN: 978-3-319-16085-6. DOI: 10.1007/978-3-319-16086-3_14.

[90] K. D. Pham, A. Jain, J. Cui, S. Fahmy und D. Maskell. „Microkernel hypervisor for a hybrid ARM-FPGA platform“. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013*. DOI: 10.1109/ASAP.2013.6567578.

[91] M. Vuletic, L. Righetti, L. Pozzi und P. Ienne. „Operating system support for interface virtualisation of reconfigurable coprocessors“. In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*.

[92] C. Main. „Virtualization on Multicore for Industrial Real-Time Operating Systems [From Mind to Market]“. In: *IEEE Industrial Electronics Magazine* 4.3 (Sep. 2010), S. 4–6. ISSN: 1932-4529. DOI: 10.1109/MIE.2010.937935.

[93] S. Trujillo, A. Crespo und A. Alonso. „MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems“. In: *2013 Euromicro Conference on Digital System Design*. Sep. 2013, S. 260–265. DOI: 10.1109/DSD.2013.37.

[94] K. Gilles, S. Groesbrink, D. Baldin und T. Kerstan. „Proteus Hypervisor: Full Virtualization and Paravirtualization for Multicore Embedded Systems“. In: *Embedded Systems: Design, Analysis and Verification: 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*. Hrsg. von G. Schirner, M. Götz, A. Rettberg, M. C. Zanella und F. J. Rammig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 293–305. ISBN: 978-3-642-38853-8. DOI: 10.1007/978-3-642-38853-8_27. URL: https://doi.org/10.1007/978-3-642-38853-8_27.

[95] F. K. Bapp. *Überwachungskonzepte für den Einsatz von COTS Multicores in der Avionik*, Diplomarbeit, Karlsruhe, 2012.

[96] Xilinx. *ML505/ML506/ML507 Evaluation Platform - User Guide*. User guide. Mai 2011.

[97] S. I. Inc. *The SPARC Architecture Manual Version 8*. 1992.

[98] SPARC. *The SPARC Architecture Manual*. Version 8.

[99] „IEEE Standard for Test Access Port and Boundary-Scan Architecture“. In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (Mai 2013), S. 1–444. DOI: 10.1109/IEEESTD.2013.6515989.

[100] Freescale. *i.MX 6Dual/6Quad Applications Processor Reference Manual*. Rev. 1, Apr. 2013.

[101] ARM. *ARM Cortex-A9 MPCore - Technical Reference Manual*. Rev. r4p1, Jan. 2016.

[102] Xilinx. *Zynq-7000 All Programmable SoC - Technical Reference Manual*. Rev. 1.11, Sep. 2016.

[103] International Standardization Organization. *ISO 11898-1:2015 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. 2. Aufl. Dez. 2015.

[104] Xilinx. *LogiCORE IP AXI Timer v2.0*. Product Guide. 2014.

[105] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo und L. Sha. „The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety“. In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2009, S. 99–107. DOI: 10.1109/RTAS.2009.20.

[106] M. Hübner, D. Göhringer, J. Noguera und J. Becker. „Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs“. In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE. 2010, S. 1–8.

[107] M. Hübner und J. Becker. „Exploiting dynamic and partial reconfiguration for FPGAs: toolflow, architecture and system integration“. In: *Proceedings of the 19th annual symposium on Integrated circuits and systems design*. ACM. 2006, S. 1–4.

[108] K. Paulsson, M. Hubner und J. Becker. „Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration“. In: *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*. IEEE. 2006, S. 288–291.

[109] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner und J. Becker. „A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput“. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE. 2008, S. 535–538.

[110] Xilinx. *Zynq UltraScale+ MPSoC - Technical Reference Manual*. Rev. 1.5, März 2017.

[111] R. Isermann. *Fahrdynamik-Regelung: Modellbildung, Fahrerassistenzsysteme, Mechatronik*. ATZ/MTZ-Fachbuch. Vieweg+Teubner Verlag, 2007. ISBN: 9783834890498. URL: https://books.google.de/books?id=LLqlkqmXryMC.

Betreute studentische Arbeiten

[Bir13] T. Birmele. „Konzept und Implementierung eines Monitors zur Zustandsüberwachung von COTS Multicoreprozessoren“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2013.

[Sch13] S. Schultschick. „Analyse und Vergleich von State-of-the-Art MPSoC-Architekturen“. Seminararbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2013.

[Luc14] M. M. Luciano. „Konzept und Implementierung einer FPGA basierten Traktionskontrolle und Torque Vectoring für den Einsatz in Elektrofahrzeugen“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2014.

[Wey14] N. Weyand. „Entwicklung einer ergonomischen Tastatur mit USB-Interface“. Bachelorarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2014.

[Mül14] T. Müllensiefen. „Weiterentwicklung einer mechatronischen Kupplungsaktuierung im Umfeld eines Formula Student Rennwagens“. Diplomarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2014.

[Krü15] M. Krüger. „Evaluation und Beurteilung einer State-of-the-Art programmable SoC Architectur anhand des Xilinx Zynq“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2015.

[Hot15] T. Hotfilter. „Konzeptionierung und Aufbau eines Steuergeräts für Elektrofahrzeuge auf Basis einer flexiblen und heterogenen Rechenplattform im Umfeld eines Formula Student Rennwagens“. Bachelorarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2015.

[Sto15] H. Stoll. „Hardwareunterstützung für gemeinsam genutzte Ressourcen in heterogenen Multicores basierend auf der Zynq Architektur“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2015.

[Sin16] C. Singe. „Durchgängiger Modellierungsansatz einer heterogenen Multicore-Architektur und Umsetzung am Beispiel eines Automotive-Steuergeräts“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2016.

[Hla16] A. Hlawatsch. „Konzeptionierung und Prototypische Umsetzung eines Telemetriesystems für rotierende Anwendungen“. Bachelorarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2016.

[Rep16] R. Reppich. „Konzeption und Implementierung eines FPGA-basierten Time-to-Digital-Converters sowie experimenteller Vergleich mit einem Referenzdesign hinsichtlich Robustheit“. Masterarbeit. Karlsruher Institut für Techno-

logie (KIT), Institut für Technik der Informationsverarbeitung, 2016.

[Sch16] B. Schmitz-Rode. „Softwareintegration hochdynamischer Motorregelalgorithmen auf einem Steuergerät auf Basis einer flexiblen Rechenplattform im Umfeld eines Formula Student Elektroantriebs“. Bachelorarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2016.

[Kön16] S. König. „Aufbau eines FPGA-basierten Kamera- Spiegelersatzsystems“. Bachelorarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2016.

[Dör17] T. Dörr. „Entwicklung einer heterogenen Hardware/Software-Architektur für Fail-Operational-Systeme“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2017.

[Bal17] L. Balzer. „Entwicklung einer Hardwareerweiterung für sicherheitskritische heterogene MPSoC-Architekturen unter Verwendung dynamisch partieller Rekonfiguration“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2017.

[Pan17] J. Pan. „Information Security for an Industry 4.0 Communication Network“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung and CDHK, Tongji University, Shanghai, China, 2017.

[Gün17] T. Günther. „Design and emulation of a synthesizable High Speed Pulse Density Modulation module testbench for ADAS hardware support“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2017.

[Sch17] V. Schumacher. „Konzeptionierung und Realisierung einer FPGA-basierten Augensicherheitsüberwachung für scannende LiDAR-Systeme“. Masterarbeit. Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung, 2017.

Veröffentlichungen

Konferenz- und Journalbeiträge

[Bap+18] F. K. Bapp, T. Sandmann, T. Dörr, F. Schade und J. Becker. „Towards Fail-Operational Systems on Controller Level using Heterogeneous Multicore Architectures and Hardware Support“. In: *WCX: SAE World Congress Experience*. SAE International, 2018.

[BB18a] F. K. Bapp und J. Becker. „Advances in Avionic Platforms: Multi-Core Systems“. In: *Aeronautics and Recent Advances in Information and Communication Technologies – Towards Flight 4.0*. Hrsg. von U. Durak, J. Becker, S. Hartmann, A. Reinhardt, C. Torens und N. Voros. Springer International Publishing, 2018.

[Bap+17] F. K. Bapp et al. „A Non-Invasive Cyberrisk in Cooperative Driving“. In: *TüV Tagung Fahrerassistenz* (Aug. 2017).

[Bap+16] F. K. Bapp, O. Sander, T. Sandmann, H. Stoll und J. Becker. „Programmable Logic as Device Virtualization Layer in Heterogeneous Multicore Architectures“. In: *Applied Reconfigurable Computing: 12th International Symposium, ARC 2016 Mangaratiba, RJ, Brazil, March 22–24, 2016 Proceedings*. Hrsg. von V. Bonato, C. Bouganis und M.

Gorgon. Cham: Springer International Publishing, 2016, S. 273–286. ISBN: 978-3-319-30481-6. DOI: 10.1007/978-3-319-30481-6_22. URL: http://dx.doi.org/10.1007/978-3-319-30481-6_22.

[Bap+15] F. K. Bapp, O. Sander, T. Sandmann, V. V. Duy, S. Baehr und J. Becker. „Adapting Commercial Off-The-Shelf Multicore Processors for Safety-Related Automotive Systems Using Online Monitoring“. In: *SAE Technical Paper*. SAE International, Apr. 2015. DOI: 10.4271/2015-01-0280. URL: http://dx.doi.org/10.4271/2015-01-0280.

[BB18b] J. Becker und F. K. Bapp. „The ARAMiS Project Initiative: Multicore Systems in Safety- and Mixed-Critical Applications“. In: *Applied Reconfigurable Computing: 14th International Symposium, ARC*. 2018.

[Tob+17] S. Tobuschat, A. Kostrzewa, F. K. Bapp und C. Dropmann. „Online monitoring for safety-critical multicore systems“. In: *it-Information Technology* 59.5 (2017), S. 215–222.

[San+17] O. Sander, F. K. Bapp, L. Dieudonne, T. Sandmann und J. Becker. „The promised future of multi-core processors in avionics systems“. In: *CEAS Aeronautical Journal* 8.1 (2017), S. 143–155. ISSN: 1869-5590. DOI: 10.1007/s13272-016-0228-x. URL: http://dx.doi.org/10.1007/s13272-016-0228-x.

[San+14a] O. Sander, F. K. Bapp, T. Sandmann, V. V. Duy, S. Bähr und J. Becker. „Architectural measures against radiation effects in multicore SoC for safety critical applications“. In: *2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*. Aug. 2014, S. 663–666. DOI: 10.1109/MWSCAS.2014.6908502.

[San+14b] O. Sander, T. Sandmann, V. V. Duy, S. Bähr, F. K. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lübbers, J. Hairbucher, A. Richter, C. Herber und A. Herkersdorf. „Hardware virtualization support for shared resources in mixed-criticality multicore systems“. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. März 2014, S. 1–6. DOI: 10.7873/DATE.2014.081.

[Fig+11] P. Figuli, M. Hübner, R. Girardey, F. K. Bapp, T. Bruckschlögl, F. Thoma, J. Henkel und J. Becker. „A heterogeneous SoC architecture with embedded virtual FPGA cores and runtime Core Fusion“. In: *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. Juni 2011, S. 96–103. DOI: 10.1109/AHS.2011.5963922.

Patente

[Ada+17] D. Adam (DE-80539 München), F. K. Bapp (DE-76137 Karlsruhe), S. Münz (DE-74564 Crailsheim), O. Sander (DE-76137 Karlsruhe) und J. Becker (DE-76751 Jockgrim). „Verfahren zum gemeinsamen Übertragen von Daten auf einem Bus in einem Kraftfahrzeug und diesbezügliche Vorrichtungen“. DE102015218202. März 2017. URL: http://www.freepatentsonline.com/DE102015218202.html.

Curriculum Vitae

Dipl.-Ing. Falco K. Bapp

geboren am 1. Juni 1985 in Waiblingen

Ausbildung und beruflicher Werdegang

seit 2012	**Wissenschaftlicher Mitarbeiter**, *Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Institut für Technologie (KIT)*
2004–2012	**Diplomstudium der Elektro- und Informationstechnik**, *Karlsruher Institut für Technologie (KIT).* Studienrichtung: *Systems Engineering* Titel der Diplomarbeit: *Überwachungskonzepte für den Einsatz von COTS Multicores in der Avionik*
2001–2004	**Gymnasium**, *Heinrich Wieland Schule, Pforzheim.* Schwerpunkt: Technik
1996–2001	**Realschule**, *Ludwig Uhland Haupt- und Realschule, Heimsheim.*
1992–1996	**Grundschule**, *Friolzheim.*