Hypervisor-basierte Partitionierung von Mixed-Criticality-Anwendungen in heterogenen eingebetteten Systemen

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)
angenommene

DISSERTATION

von

M.Sc. Florian Schade

Tag der mündlichen Prüfung: 10. Juni 2024

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

Hypervisor-basierte Partitionierung
von Mixed-Criticality-Anwendungen
in heterogenen eingebetteten Systemen

1. Auflage: Juli 2024

Zusammenfassung

Eingebettete Systeme bilden die Grundlage für viele Innovationen der letzten Jahrzehnte. Mit der fortschreitenden Digitalisierung nehmen ihr Funktionsumfang und ihre Bedeutung weiter zu. Aufgrund des damit einhergehenden Anstiegs an benötigter Rechenleistung wird zunehmend leistungsfähigere Ausführungshardware benötigt. Unter technischen und ökonomischen Gesichtspunkten wird zudem die integrierte Umsetzung neuer und bislang verteilter Funktionen auf zentralen Hardwareplattformen angestrebt. Diese basieren meist auf Multicore-Prozessoren (MCP) und Multi-Prozessor Systems-on-Chip (MPSoC), deren Leistungsfähigkeit maßgeblich auf paralleler Ausführung basiert.

Die Integration von Funktionen, an welche unterschiedliche Anforderungen hinsichtlich Echtzeitverhalten und Zuverlässigkeit gestellt sind, führt zu *Systemen gemischter Echtzeitanforderungen* bzw. *gemischter Kritikalität* (Mixed-Criticality-Systeme). Deren zuverlässige Implementierung auf MCP und MPSoC stellt aufgrund der geteilten Nutzung von Plattformressourcen eine bedeutende Herausforderung dar. Insbesondere können an solchen Ressourcen *Interferenzeffekte* auftreten, welche die Softwareausführung auf parallel aktiven Prozessorkernen verzögern. Embedded Hypervisors unterstützen die angestrebte Integration verschiedener Funktionen auf einer Hardwareplattform, indem sie die Ausführung verschiedener Betriebssystem- und Anwendungssoftware in logisch getrennten Partitionen ermöglichen. Durch die Möglichkeit, die Ressourcennutzung auf Partitionsebene einzuschränken, sind sie insbesondere im Kontext von Systemen gemischter Kritikalität von Bedeutung. Um hierbei die Einhaltung von Echtzeitanforderungen garantieren zu können, muss die Nutzung paralleler Ausführungseinheiten auf MCP und MPSoC zur Vermeidung von Interferenzeffekten meist stark eingeschränkt werden. Dies führt zu Unterauslastung der Hardware und verringert somit die Effizienz des Gesamtsystems.

In dieser Arbeit werden daher Konzepte zur Interferenzbeherrschung in Hypervisor-partitionierten MPSoC vorgestellt und untersucht, welche die Umsetzung von Systemen gemischter Echtzeitanforderungen ermöglichen und dabei die Ausnutzung paralleler Recheneinheiten der Plattform optimieren. Hierbei wird neben der Integration solcher Systeme auf MCP-Komponenten auch ihre Umsetzung unter Nutzung von programmierbaren Logikkomponenten heterogener MPSoC betrachtet.

Zur Beherrschung von Interferenz zwischen Kernen von MCP-Clustern im MPSoC

wird ein reaktiver Scheduling-Ansatz vorgestellt. Basierend auf der Überwachung des Ausführungsfortschritts von Echtzeit-Tasks zur Laufzeit wird die parallele Ausführung unkritischer Tasks ermöglicht, solange die auftretenden Interferenzeffekte die Einhaltung der Echtzeitanforderungen nicht gefährden. Zur Bestimmung des Ausführungsfortschritts werden Ausführungs-Trace-Daten verwendet. Dies ermöglicht die effiziente Implementierung der Überwachungskomponente unter Nutzung MPSoC-interner Ressourcen und erlaubt die Integration von Echtzeit-Tasks ohne weitere Modifikation. Das Konzept wird unter Verwendung eines industriellen Hypervisors implementiert, hinsichtlich seiner Effektivität und Effizienz in Abhängigkeit verschiedener Systemparameter untersucht und einem konservativen Scheduling-Ansatz gegenübergestellt. Die Evaluation zeigt, dass der vorgestellte Ansatz die Einhaltung der Echtzeitanforderungen in allen untersuchten Szenarien sicherstellen konnte und gleichzeitig zu einer deutlichen Erhöhung des Zeitanteils paralleler Ausführung führte.

Als Alternative zur prozessorbasierten Implementierung wird eine Architektur vorgestellt, welche die Umsetzung der Echtzeitanwendungen in programmierbaren Logikkomponenten (Field-Programmable Gate Arrays, FPGA) heterogener MPSoC ermöglicht. Sie erweitert die durch den Hypervisor realisierte Partitionierung auf den FPGA und ermöglicht so die Nutzung anwendungsspezifischer, FPGA-basierter Funktionen im Partitionskontext. Die Umsetzung von Echtzeitanwendungen wird durch die direkte Anbindung externer Komponenten an diese FPGA-Funktionen ermöglicht. Durch einen Mechanismus zur flexiblen Anpassung der hierzu verwendeten Kommunikationsschnittstelle sowie durch die modulare Austauschbarkeit von Partitionssoftware und FPGA-Funktionen zur Laufzeit ist die Architektur auf die Anforderungen hochdynamischer Einsatzumgebungen ausgerichtet. Dies wird am Beispiel hochflexibler Produktionssystemkonzepte dargestellt.

Die vorliegende Arbeit geht in verschiedener Hinsicht über den Stand der Technik hinaus. So wird eine im Kontext des reaktiven Schedulings neue Monitoring-Methode untersucht, welche die ressourcenschonende Überwachung von Echtzeit-Tasks erlaubt. Gleichzeitig ermöglicht sie die MPSoC-interne Umsetzung reaktiven Schedulings, ohne eine Modifikation der zu überwachenden Anwendungen zu erfordern. Darüber hinaus stellt die FPGA-basierte Umsetzung von Echtzeitfunktionen im Partitionskontext durch Anbindung externer Komponenten über rekonfigurierbare Schnittstellen eine wesentliche Neuerung dar. Insgesamt trägt die vorliegende Arbeit somit zu einer effizienteren Umsetzbarkeit von Systemen mit gemischten Echtzeitanforderungen auf mittels Hypervisors partitionierten heterogenen MPSoC bei. Sie leistet somit einen Beitrag zur effizienteren Partitionierung von Mixed-Criticality-Anwendungen in heterogenen eingebetteten Systemen.

Abstract

Embedded systems have been the technological foundation for many innovations in the recent decades. As the trend towards digitalization advances, their functional scope and importance continue to grow. Due to the corresponding increase in required computational power, increasingly powerful processing platforms are needed. At the same time, technical and economic considerations mandate the integrated implementation of new and formerly distributed functionality on centralized hardware platforms. These platforms are primarily based on multicore processor (MCP) and multi-processor system-on-chip (MPSoC) devices, which achieve high performance through parallel computation.

The integration of functionality with different real-time and reliability requirements results in mixed-real-time systems and mixed-criticality systems. The reliable implementation of these systems on MCP and MPSoC devices is a major challenge due to the shared use of platform resources. In particular, the use of shared resources can lead to interference effects, leading to delays in the software execution on parallel processing cores. Embedded hypervisors enable the execution of operating system and application software in logically-separated partitions and thus support the desired integration of functionality on shared hardware platforms. By allowing developers to impose access restrictions to system resources on a partition level, they play a particularly important role in the context of mixed-criticality systems. To avoid interference effects, the use of parallel execution in MCP and MPSoC devices has to be severely limited, where real-time behavior must be guaranteed. This leads to underutilization of the hardware and thus reduces the overall system efficiency.

To overcome this issue, this work presents and evaluates concepts for interference control that allow for the implementation of mixed-real-time systems in hypervisor-partitioned MPSoC devices while optimizing the use of the execution platform's parallel processing potential. In doing so, the integration of mixed-criticality systems on MCP components as well as their implementation based on programmable logic in heterogeneous MPSoC devices are considered.

To control inter-core interference in MCP components of the MPSoC, a reactive scheduling approach is presented. By tracking the execution progress of real-time tasks at runtime, it allows for parallel execution of non-critical tasks, as long as interference-induced delays do not compromise compliance with the real-time requirements. To monitor the

task progress, execution trace information is used. This allows for an efficient monitor implementation using MPSoC-internal resources and enables the integration of real-time tasks without requiring modification. The concept is implemented using an industrial hypervisor, evaluated for its effectiveness and efficiency, and compared to a conservative scheduling approach. In doing so, the influence of various system parameters on its efficiency is investigated. The evaluation indicates that the proposed scheduling approach successfully ensured compliance with the given real-time requirements in all examined scenarios and at the same time has led to a significant increase in parallel execution.

As an alternative to the processor-based implementation, an architecture is presented that enables the realization of real-time functionality in field-programmable gate array (FPGA) components in heterogeneous MPSoC devices. By extending the hypervisor-based partitioning to the FPGA, it enables the use of FPGA-based, application-specific functionality within the context of the hypervisor partition. The implementation of real-time functionality is enabled by facilitating direct communication between external components and FPGA functions. A mechanism that supports the flexible exchange of the used communication protocols as well as the modular exchange of partition software and FPGA-based functionality ensures the applicability of the architecture in highly dynamic application environments. This is illustrated by the example of highly flexible production system concepts.

The present work extends the state of the art in several aspects. In the context of reactive scheduling, it investigates the use of a novel monitoring mechanism that allows for resource-efficient monitor implementations. At the same time, it enables the implementation of reactive scheduling on MPSoC devices without requiring modifications to the monitored applications. Furthermore, the FPGA-based realization of real-time functionality that is part of the hypervisor partition context and enabled by reconfigurable interfaces to external components is a significant contribution. As a result, this work contributes towards a more efficient implementation of mixed real-time systems using hypervisor-partitioned heterogeneous MPSoC platforms, and thus, towards the more efficient partitioning of mixed-criticality applications in heterogeneous embedded systems.

Vorwort

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) am Karlsruher Institut für Technologie (KIT). Im Rahmen verschiedener Forschungsprojekte konnte ich in dieser Zeit spannende Erfahrungen zu verschiedensten Aspekten eingebetteter Systeme und der damit verknüpften Forschung machen.

Für die Betreuung meines Promotionsvorhabens, für sein Vertrauen und die von ihm ermöglichten Freiheiten im Rahmen meiner Arbeit am Institut gilt mein besonderer Dank daher Prof. Jürgen Becker. Weiterhin danke ich Prof. Jürgen Fleischer für die gemeinsamen Forschungsprojekte sowie für die Übernahme des Korreferats. Auch den weiteren Mitgliedern des Prüfungsausschusses Prof. Mike Barth, Prof. Cornelius Neumann und Prof. Ivan Peric danke ich für die Unterstützung meines Promotionsvorhabens.

Meinen derzeitigen und ehemaligen Kollegen am Institut danke ich herzlich für die angenehme Zusammenarbeit, die interessanten Diskussionen, die unterhaltsamen Kinobesuche, Spieleabende, Bastelaktionen und vieles weitere. Ebenfalls danke ich allen von mir angeleiteten Studierenden, welche im Rahmen von Abschlussarbeiten und HiWi-Tätigkeiten zum Gelingen dieser Arbeit beigetragen haben.

Abschließend möchte ich meiner Familie und meiner Freundin Laura für die fortwährende Unterstützung und den Rückhalt während dieser Zeit danken und insbesondere meiner Mutter Trixi für ihre Unterstützung beim Korrekturlesen dieser Arbeit.

Karlsruhe, im Juni 2024
Florian Schade

Inhaltsverzeichnis

1 Einleitung

Die fortschreitende Digitalisierung führt zu vielfältigen Neuerungen in verschiedensten Lebensbereichen. Im Mobilitätssektor fördern Unterstützungssysteme wie Advanced Driver Assistance Systems (ADAS) den Fahrkomfort und die Sicherheit, während autonome Fahrfunktionen immer weitere Teile der Fahraufgabe automatisieren. Im Bereich der produzierenden Industrie verspricht die zunehmende Datenverarbeitung und Vernetzung unter der Bezeichnung *Industrie 4.0* Produktivitätsgewinne, die Möglichkeit individualisierter Produktion sowie eine schnelle Anpassbarkeit an die Bedürfnisse des Marktes [1]. Weitere Neuerungen betreffen unter anderem die Vernetzung und Automatisierung von Haushalten (Smart Home) [2] oder ganzer Städte (Smart City) [3], um Komfort- oder Effizienzsteigerungen zu erreichen.

Die Grundlage dieser Neuerungen bilden eingebettete Systeme, d.h. Rechensysteme, welche in übergeordnete Systeme integriert sind und überwiegend softwaregetrieben Systemfunktionen erfüllen. Eingebettete Systeme erfassen mittels Sensoren den Zustand des übergeordneten Systems und dessen Umwelt, steuern physische Prozesse und kommunizieren mit weiteren Komponenten. Im Hinblick auf diese Interaktion eingebetteter Systeme mit physischen Prozessen bezeichnet man die übergeordneten Systeme auch als *Cyber-physische Systeme* (CPS) [4, 5], im Kontext der Produktionstechnik spezifischer als Cyber-physische Produktionssysteme (Cyber-physical production systems, CPPS) [6].

Durch die enge Verzahnung der Datenverarbeitung mit physischen Prozessen müssen eingebettete Systeme in CPS meist spezielle Anforderungen erfüllen. Dies betrifft insbesondere das Zeitverhalten der Softwareausführung im System. Während die Reaktionslatenz der Software auf auftretende Ereignisse in klassischen informationstechnischen Anwendungen vorwiegend ein Optimierungskriterium darstellt, ist sie in vielen eingebetteten Systemen funktionsrelevant [5]. Ist dies der Fall, bezeichnet man das eingebettete System als *Echtzeitsystem*. Besonders relevant ist die Einhaltung des *Echtzeitverhaltens*, wenn das betrachtete System besonderen Anforderungen hinsichtlich funktionaler Sicherheit unterliegt und eine verspätete Reaktion unakzeptable Risiken mit sich bringt. Beispiele hierfür sind Sicherungssysteme wie Airbags in Fahrzeugen oder Notabschaltungen in Produktionsanlagen, aber auch für Betriebsfunktionen relevante Komponenten wie die Strom- und Drehzahlregelung digitaler Antriebssysteme. Während Airbags nach einer Kollision in einem spezifizierten, situationsabhängigen Zeitfenster von wenigen Millisekun-

den ausgelöst werden müssen, um die gewünschte Schutzwirkung zu entfalten [7], kann die Verletzung von Zeitvorgaben bei Steuerungskomponenten bspw. durch Aufschwingen des gesteuerten Systems zu erheblichen Schäden für Mensch, Technik oder Umwelt führen [8].

Der oben genannte Digitalisierungstrend geht mit einem rasanten Zuwachs an softwaregetriebenen Funktionen in den betrachteten Systemen einher. In der Automobilbranche wird auf dem Weg hin zu vollautomatisiertem Fahren ein rasanter Anstieg des benötigten Softwareumfangs erwartet [9]. Dementsprechend ist auch eine starke Zunahme der erforderlichen Rechenleistung eingebetteter Rechenplattformen zu erwarten, welche nicht zuletzt auch durch den zunehmenden Einsatz Machine-Learning-basierter Funktionen verstärkt wird.

Hardwareseitig stehen diesem Trend leistungsfähige Datenverarbeitungsplattformen wie Multicore-Prozessoren (MCP) und Multiprozessor-Systems-on-Chip (MPSoC) gegenüber, welche parallele Datenverarbeitung in einem Prozessor bzw. durch mehrere auf einem Chip integrierte Recheneinheiten ermöglichen. Sie umgehen durch diese Parallelität den als *Ende des Dennard-Scaling* bezeichneten Effekt, welcher das Ende der Skalierbarkeit der Leistung von Single-Core-Prozessoren durch Erhöhung ihrer Taktfrequenz bedeutete. Bis um das Jahr 2005 [10] folgte die Transistortechnologieentwicklung dem als *Dennard-Scaling* bezeichneten Trend, nach welchem die Leistungsdichte von Transistorschaltungen bei einer Reduktion der Strukturgröße annähernd konstant bleibt [11]. Dies ermöglichte die Erhöhung der Taktfrequenz von Prozessoren und somit das Erreichen höherer Rechenleistung bei ähnlicher elektrischer Leistungsaufnahme. Mit dem Ende des Dennard-Scaling würde eine weitere Erhöhung der Taktfrequenz zu einem merklichen Anstieg der Leistungsaufnahme führen, was den Fokus der Prozessorentwicklung auf parallele Ausführung bei niedrigeren Taktfrequenzen in Multicore-Prozessoren lenkte. MPSoC führen die Idee der Integration paralleler Ausführungseinheiten auf einem Chip fort, indem sie mehrere MCP-Cluster mit weiteren Komponenten wie Grafikprozessoren (GPU, Graphics Processing Unit) oder programmierbarer Logik, meist Field-Programmable Gate Array (FPGA), sowie Speicher und Peripheriekomponenten auf einem Chip verbinden.

Die Verfügbarkeit solcher leistungsfähigen Recheneinheiten ermöglicht zum einen die Umsetzung rechenintensiver Anwendungen, zum anderen ermöglicht sie auch die Integration verschiedener Datenverarbeitungsfunktionen auf zentralen Rechenplattformen. Dies ist erstrebenswert, da somit die Anzahl benötigter Recheneinheiten reduziert werden kann, was wiederum den Vernetzungsaufwand und Kosten reduzieren kann. Im Kontext mobiler eingebetteter Systeme ist hierbei insbesondere die Reduktion des benötigten Bauraums, des Gewichts und der Leistungsaufnahme vorteilhaft.

Der Trend hin zu steigender Integration ist unter anderem in der Automobilindustrie anhand der Entwicklung der Elektrischen/Elektronischen Architektur (EE-Architektur) zu beobachten. Diese entwickelt sich von einer verteilten Architektur mit einer Vielzahl an

vernetzten Steuergeräten hin zu einer domänenzentralisierten Architektur, in welcher leistungsfähige Domänensteuergeräte komplexe Funktionen realisieren und vormals verteilte Funktionen integrieren. Langfristig wird davon ausgegangen, dass sich eine fahrzeugweit zentralisierte EE-Architektur durchsetzt, in welcher die Unterteilung in Domänen nur noch virtuell stattfindet. [12]

In der Produktionstechnik ist die genannte Integration verschiedener Funktionen bspw. im Kontext des Konzepts der *dezentralen Steuerung* relevant, welches im Rahmen der als Industrie 4.0 bezeichneten Entwicklungen angestrebt wird [8, 13]. Der Ausführungsort von Softwarefunktionen wird dabei vom Standort der Maschinen entkoppelt, sodass Steuerungs- und Datenverarbeitungsfunktionen verschiedener Maschinen auf maschinenunabhängigen Rechenplattformen integriert werden können. Dies vereinfacht die Skalierung verfügbarer Rechenleistung auf Basis der wechselnden Anforderungen flexibler Produktionssysteme [8].

1.1 Motivation

Sowohl der zunehmende Funktionsumfang der Datenverarbeitung innerhalb vernetzter eingebetteter Systeme als auch die Integration verschiedener Funktionalität auf einzelnen Ausführungsplattformen führen zu einer hohen Systemkomplexität. Insbesondere im Hinblick auf Anwendungen, an welche Anforderungen hinsichtlich funktionaler Sicherheit und Echtzeitverhalten gestellt sind, stellt die Nutzung von Multi-Core-Prozessoren und Multicore-basierten Plattformen Entwickler vor Herausforderungen. Dies wird im Fall von *Mixed-Criticality-Systemen* (MCS) sichtbar, in welchen Funktionen unterschiedlicher Kritikalität als Teile eines übergeordneten Systems ausgeführt werden.

In MCS muss sichergestellt werden, dass sich Anwendungen unterschiedlicher Kritikalität nur im Rahmen des spezifizierten Verhaltens beeinflussen. Eine Fehlfunktion einer weniger kritischen Anwendung darf sich nicht negativ auf Anwendungen mit höherer Kritikalität auswirken. Werden diese auf einer gemeinsamen Plattform, wie einem Multicore-Prozessor oder MPSoC, ausgeführt, müssen daher Maßnahmen getroffen werden, um die Anwendungen voneinander zu isolieren. Eingebettete Hypervisors unterstützen dies, indem sie die Rechenressourcen einer prozessorbasierten Rechenplattform mittels Virtualisierungstechniken partitionieren. Innerhalb der Partitionen stellen sie Ausführungsumgebungen bereit, aus welchen nur definierte Teile der Rechenressourcen genutzt werden können. Somit erlauben sie die isolierte Ausführung von Anwendungen auf geteilten Plattformen. Die hierbei verwalteten Rechenressourcen umfassen dabei typischerweise Rechenzeit auf Prozessorkernen, Arbeitsspeicher sowie Peripheriegeräte der Ausführungsplattform. Weiterhin werden Kommunikationsmechanismen zwischen den Partitionen bereitgestellt.

In Multicore-Systemen teilen sich mehrere Kerne gemeinsame Ressourcen wie Caches, Schnittstellen zum Systembus und Speicher. Somit kann es zu konkurrierender Nutzung dieser Ressourcen kommen, was eine Verzögerung der Anwendungsausführung auf einzelnen Kernen zur Folge hat. Dieser Effekt wird als *Interferenz* bezeichnet. Im Kontext von Echtzeitsystemen können Interferenzeffekte zur Verzögerung zeitkritischer Anwendungen führen, was das Nichteinhalten von Deadlines zur Folgen haben kann. Bei der Implementierung von MCS auf MCP und MPSoC müssen daher Maßnahmen ergriffen werden, diese Interferenzeffekte auf ein vertretbares Maß zu reduzieren bzw. zu vermeiden, um die korrekte Ausführung von Echtzeitanwendungen sicherzustellen.

Aufgrund der Komplexität moderner Hardwareplattformen ist eine präzise Vorhersage der durch Interferenz verursachten Verzögerungen der Anwendungsausführung zur Entwurfszeit oft nicht möglich. Dies ist auch der Fall, wenn nur eingeschränkte Informationen über Teile der auszuführenden Software vorliegen. Aus diesen Gründen muss die Nutzung paralleler Ausführungseinheiten zur Entwurfszeit meist stark eingeschränkt werden, um Echtzeitverhalten garantieren zu können. Dies führt zu Unterauslastung der Rechenhardware und verringert somit die Effizienz des Gesamtsystems.

Da die alleinige Nutzung von Hypervisors zur hinreichenden Interferenzbeherrschung für eine parallele Ausführung von Anwendungen mit gemischten Echtzeitanforderungen auf MCP-basierten Plattformen oft nicht ausreicht und die Einschränkung paralleler Ausführung zu einer deutlich reduzierter Performanz führt, stellt Interferenz für Multicore-basierte MCS ein zentrales Problem dar.

1.2 Zielsetzung und Beitrag der Arbeit

Diese Arbeit widmet sich dem Problem der Interferenzbeherrschung auf hypervisorverwalteten MPSoC-Plattformen. Sie soll eine Grundlage schaffen, welche die effiziente Integration von Anwendungen mit unterschiedlichen Echtzeitanforderungen (Mixed-Real-Time) auf diesen Plattformen ermöglicht. Hierbei muss die Einhaltung des Echtzeitverhaltens sichergestellt werden, während gleichzeitig die Systemeffizienz durch Ausnutzung von Parallelität optimiert werden soll.

Um dies zu erreichen, werden zunächst die Auswirkungen von Interferenzeffekten an der Speicherinfrastruktur auf das Zeitverhalten der Softwareausführung auf Multicore-Plattformen untersucht. Dies erfolgt am Beispiel einer Steuerungseinheit zur Nachrüstung von Industrie-4.0-Funktionen in Produktionsanlagen, welche auf einem Hypervisor ohne interferenzreduzierende Maßnahmen basiert. Anschließend wird ein Konzept zur effizienten Ausführung von Mixed-Real-Time-Anwendungen auf hypervisorverwalteten MCP in commercial-off-the-shelf (COTS) MPSoC vorgestellt. Es begrenzt die Auswir-

kungen von Interferenzeffekten mittels eines tracedatenbasierten Monitoring-Ansatzes und der dynamischen Anpassung des Schedulings paralleler Softwareausführung auf Hypervisorebene. Somit optimiert es die Ausnutzung von Parallelität im MCP und stellt gleichzeitig sicher, dass auftretende Interferenzeffekte das Echtzeitverhalten kritischer Anwendungen nicht gefährden. Dieses Konzept wird auf Basis einer aktuellen MPSoC-Plattform und eines industriellen Embedded Hypervisors umgesetzt und hinsichtlich der erreichbaren Systemeffizienz mit einem konservativen Scheduling-Ansatz zur Interferenzbeherrschung verglichen. Abschließend wird eine Architektur vorgestellt, welche die FPGA-basierte Implementierung von Echtzeit-Funktionen zur Interaktion mit der Umgebung im Hypervisor-Partitionskontext ermöglicht. Sie erweitert die durch den Hypervisor umgesetzte Partitionierung auf FPGA-Komponenten des MPSoC. Dies ermöglicht die modulare Umsetzung von Anwendungslogik im FPGA unter Beibehaltung der Partitionierung und der Rekonfigurierbarkeit zur Laufzeit. Durch einen Mechanismus zur adaptiven Anbindung dieser Anwendungslogik an externe Komponenten ermöglicht die Architektur die Umsetzung von Echtzeit-Interaktion mit der Systemumgebung unter Umgehung von Interferenzkanälen mit dem MCP. Die Architektur wird im Kontext einer Edge-Einheit zur dezentralen Datenverarbeitung in hochflexiblen Produktionssystemen diskutiert und ihre Anwendbarkeit zur flexiblen Anbindung von Sensoren und Aktoren im Kontext solcher Anlagen dargestellt.

2 Grundlagen

2.1 Echtzeit- und Mixed-Criticality-Systeme

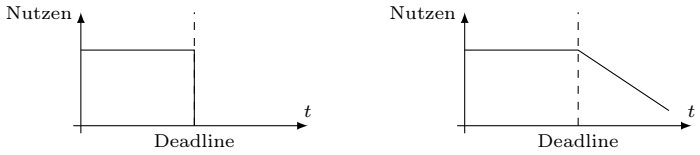
Die vorliegende Arbeit befasst sich mit der Umsetzung von Echtzeit- und Mixed-Criticality Systemen im Kontext eingebetteter Systeme. Im Folgenden werden die hierzu benötigten Konzepte und Begriffe beschrieben und in weiteren Kontext gesetzt.

2.1.1 Echtzeitverhalten und Worst-Case Execution Time

Eingebettete Systeme müssen oft Echtzeitanforderungen erfüllen. Hierunter versteht man die Anforderung an das System, innerhalb einer definierten Zeit auf bestimmte Ereignisse reagieren zu können. Der Grund für Echtzeitanforderungen liegt oft in der Interaktion des eingebetteten Systems mit der physischen Umwelt oder umgebenden digitalen Systemen. So muss beispielsweise die Zündung eines Airbags in einem genau spezifizierten Zeitfenster nach Feststellen einer Kollision erfolgen [7] oder spezifiziertes Zeitverhalten in Kommunikationssystemen exakt eingehalten werden.

Abstrakt können die auslösenden Ereignisse als Eingabe eines Rechensystems angesehen werden, während die resultierende Reaktion als dessen Ausgabe bezeichnet werden kann. Somit ist in Echtzeitsystemen nicht nur die Korrektheit der Ausgabe im Sinne des richtigen Ergebnisses Voraussetzung für die korrekte Funktion des Systems, sondern auch die rechtzeitige Verfügbarkeit dieses Ergebnisses. Diese ist typischerweise in Form einer Deadline spezifiziert, bis zu welcher die Ausgabe erfolgt sein muss.

Wird die definierte Deadline nicht eingehalten, beeinträchtigt dies die Funktion des Echtzeitsystems. Nach der Auswirkung verpasster Deadlines unterscheidet man zwischen *harter Echtzeit* (engl. hard real-time) und *weicher Echtzeit* (engl. soft real-time). Dies ist in Abbildung 2.1 dargestellt. Im Fall harter Echtzeit kommt eine verspätete Ausgabe einem Versagen des Systems gleich, was je nach Anwendungsfall schwere Konsequenzen nach sich ziehen kann. Im Fall weicher Echtzeit führt eine verspätete Ausgabe zu einer verminderten Qualität der durch das System erbrachten Funktion. Vereinzelte verspätete Ausgaben sind in solchen Systemen tolerierbar, solange die Deadline im Mittel eingehalten wird. [14]



(a) Harte Echtzeit (b) Weiche Echtzeit

Abbildung 2.1: Beispielhafte Darstellung des Nutzens der Programmausgabe unter verschiedenen Echtzeitbedingungen (eigene Abb. nach [15])

Eine bei der Auslegung von Echtzeitsystemen zentrale Größe ist die *Worst-Case Execution Time* (WCET). Als WCET eines Programms wird die maximale Ausführungszeit des betrachteten Programms in der betrachteten Ausführungsumgebung bezeichnet. Die Ausführungszeit eines Programms hängt sowohl von der Eingabe als auch vom Zustand des Programms und seiner Ausführungsumgebung ab. Dies umfasst insbesondere auch den Zustand der Ausführungshardware, d.h. Caches, Prozessor-Pipelines und weitere Komponenten, welcher einen starken Einfluss auf die Ausführungszeit haben kann. Aufgrund der Größe des resultierenden Zustandsraums ist es meist nicht möglich, diesen vollständig zu analysieren und so die WCET exakt zu bestimmen. Stattdessen werden Techniken zur Abschätzung der WCET bzw. zur Bestimmung einer möglichst niedrigen *(engen)* oberen Schranke der WCET eingesetzt. Etablierte Ansätze hierzu lassen sich, wie in [14] beschrieben, in die Kategorien *Statische Analyse*, *Dynamische Timing-Analyse* und *hybride Ansätze* einteilen, welche im Folgenden kurz zusammengefasst werden.

Im Rahmen einer statischen Analyse wird eine obere Schranke der WCET durch Analyse des Programmcodes in Kombination mit einem Modell der Ausführungshardware bestimmt. Dies erfolgt zumeist durch eine Kontrollflussanalyse, welche mögliche Pfade durch das zu analysierende Programm identifiziert. Für diese Pfade werden anschließend mittels eines detaillierten Modells der Ausführungsplattform obere Schranken der Ausführungszeit bestimmt. Abschließend wird aus diesen Ausführungszeiten und den Ergebnissen der Kontrollflussanalysen eine obere Schranke der WCET bestimmt. Während eine statische Analyse oftmals in der Lage ist, eine obere Schranke der WCET zu bestimmen, kann diese deutlich über der tatsächlichen WCET des Programms liegen. Dies liegt daran, dass die Programmlaufzeit bei fehlenden Informationen stets pessimistisch abgeschätzt werden muss, um das Erreichen einer oberen Schranke zu garantieren.

Im Gegensatz zur statischen Analyse basiert die dynamische Timing-Analyse auf Messungen der Ausführungszeit auf der Zielhardware oder auf mittels einer zyklengenauen Hardwaresimulation bestimmten Ausführungszeiten. Basierend auf einer Menge von Testvektoren, welche die Eingabe und den Zustand des Programms sowie den Zustand der Ausführungsumgebung beschreiben, werden verschiedene Ausführungen erfasst. Wie

bereits genannt, kann die Laufzeit des Programms aufgrund der Größe des Zustandsraums in der Praxis nicht für alle Startzustände analysiert werden. Weiterhin ist es teilweise schwierig, bestimmte Startzustände zu realisieren, insbesondere im Hinblick auf den internen Zustand der Ausführungshardware. Aus diesem Grund liefert die dynamische Timing-Analyse eine untere Schranke der WCET. Zur Abschätzung der WCET wird in der Praxis häufig ein erfahrungsbasierter Sicherheitszuschlag (bspw. 20 % [16]) angesetzt. Die resultierende WCET-Abschätzung stellt jedoch keine garantierte obere Schranke der Ausführungszeit dar.

Zur Verbesserung der WCET-Schätzung kombinieren hybride Ansätze Konzepte der statischen Analyse und der dynamischen Timing-Analyse. So werden bspw. Ausführungszeiten von Programmabschnitten messtechnisch erfasst und auf Basis statischer Kontrollflussanalysen zu WCET-Abschätzungen kombiniert. Während solche Ansätze helfen, Nachteile der reinen statischen Analyse bzw. reinen dynamischen Timing-Analyse zu umgehen, können aktuelle hybride Analysetools nicht garantieren, eine obere Schranke der WCET zu finden [14].

2.1.2 Mixed-Criticality-Systeme

Im Folgenden werden die Grundlagen zu Mixed-Criticality-Systemen beschrieben. Hierzu wird zunächst der Begriff der *Sicherheit* eingeführt und *sicherheitskritische Systeme* betrachtet, welche die Basis von MCS bilden.

2.1.2.1 Sicherheit

Der Begriff *Sicherheit* bezeichnet im Allgemeinen die Abwesenheit von unakzeptablen Risiken bzw. erheblichen Gefahren für Menschen oder Sachwerte. Bezogen auf technische Systeme wird der Begriff im Deutschen sowohl in Bezug auf die funktionale Sicherheit bzw. Betriebssicherheit (engl. *safety*) als auch auf in Bezug auf die Angriffssicherheit (engl. *security*) genutzt [17]. Funktionale Sicherheit bzw. Betriebssicherheit bezeichnet hierbei den Schutz der Umgebung vor dem betrachteten System und hängt somit mit der korrekten Funktion des Systems zusammen, während Angriffssicherheit auf den Schutz des Systems vor seiner Umwelt abzielt. In dieser Arbeit wird der Begriff Sicherheit vorwiegend im Kontext der funktionalen Sicherheit bzw.Betriebssicherheit genutzt, synonym zum englischen Begriff *safety*.

Entsprechend dem o.g. Verständnis von Sicherheit sind *sicherheitskritische Systeme* durch die Eigenschaft charakterisiert, dass eine Fehlfunktion des Systems zu Personenschäden, wirtschaftlichen Verlusten oder Umweltschäden führen kann [18]. In solchen Systemen müssen daher Maßnahmen getroffen werden, das Risiko eines Schadens (infolge einer

solchen Fehlfunktion) auf ein tolerierbares Maß zu reduzieren. Das größte noch vertretbare Risiko wird hierbei als Grenzrisiko bezeichnet.

Die Entwicklung sicherheitskritischer Systeme ist durch verschiedene Normen geregelt. Die Sicherheitsgrundnorm IEC 61508 [19], welche in Deutschland als DIN EN 61508 übernommen wurde, definiert branchenübergreifende Anforderungen an sicherheitskritische Systeme und ihren Entwicklungsprozess. Basierend auf dieser Grundnorm existieren verschiedene branchenspezifische Normen.

Im Rahmen dieser Normen sind Sicherheits-Integritätslevel definiert, über welche die Anforderungen an die Wirksamkeit der Sicherheitsfunktionen im System klassifiziert werden. Prominente Beispiele sind die in der Avionikbranche genutzten Design Assurance Level DAL A bis DAL E (SAE ARP 4754a [20]), Safety Integrity Level SIL 1 bis SIL 4 in der industriellen Steuerungstechnik (IEC 61508 [19]) oder Automotive Safety Integrity Level ASIL A bis ASIL D in der Automobilbranche (ISO 26262 [21]). Diese Integritätslevel bilden die Grundlage für Anforderungen an das System und den Entwicklungsprozess, welche darauf abzielen, das Risiko von Fehlfunktionen auf ein dem Integritätslevel entsprechendes Maß zu reduzieren [22].

2.1.2.2 Mixed-Criticality-Systeme

Als Mixed-Criticality-System (MCS, deutsch: System gemischter Kritikalität) wird ein System bezeichnet, welches sowohl Komponenten hoher Kritikalität als auch weniger kritische Komponenten umfasst. Der Begriff Kritikalität wird hierbei zumeist als Maß für den benötigten Schutz vor Fehlfunktionen einer Systemkomponente im Sinne der funktionalen Sicherheit [23] verstanden, wird jedoch in Wissenschaft und Standardisierung nicht einheitlich verwendet [24].

Da sicherheitskritische Systeme zumeist physische Prozesse kontrollieren (was in vielen Fällen die Grundlage für ihr Schadenspotenzial darstellt), handelt es sich bei ihnen häufig gleichzeitig um Echtzeitsysteme [23]. Axer et al. [25] stellen hierbei Anforderungen hinsichtlich der korrekten Systemausgabe (Integrität des Berechnungsergebnisses) und Anforderungen hinsichtlich der Rechtzeitigkeit der Systemausgabe (Echtzeitverhalten) als zwei Dimensionen der Kritikalität dar. So werden beispielsweise Ampelanlagen als sicherheitskritische, aber nicht echtzeitkritische Systeme klassifiziert, während mobile Kommunikationssysteme wie Universal Mobile Telecommunications System (UMTS) oder Long-Term Evolution (LTE) in nicht sicherheitskritischen Kontexten eingesetzt werden können, aber Echtzeitanforderungen erfüllen müssen. Neben Betriebssicherheit und Echtzeitverhalten können außerdem auch die Angriffssicherheit [26] sowie weitere Aspekte der Zuverlässigkeit (engl. dependability nach [27]) [23] eines Systems als Gegenstand der Kritikalität im MCS betrachtet werden.

Im Kontext eingebetteter Systeme und CPS entstehen MCS häufig durch die Integration verschiedener Funktionalität auf geteilten Hardwareplattformen. Mit zunehmender Verfügbarkeit leistungsfähiger Hardwareplattformen ist diese Konsolidierung von Recheneinheiten wirtschaftlich attraktiv, da sie eine effiziente Nutzung der Rechenressourcen ermöglicht und somit zu einer Reduktion der Anzahl benötigter Recheneinheiten führt. Dies kann wiederum zu einer Reduktion des Vernetzungsaufwands sowie Einsparungen hinsichtlich Gewicht, Bauraum und Leistungsaufnahme des Systems führen. Gleichzeitig erfordert die Integration von Komponenten unterschiedlicher Kritikalität die Berücksichtigung von Effekten, welche durch die Nutzung geteilter Ressourcen durch verschiedene Anwendungen entstehen.

Normen zur Entwicklung sicherheitskritischer Systeme geben Orientierung hinsichtlich der Integration von Komponenten mit unterschiedlichen Sicherheits-Integritätsleveln. Sie geben vor, dass das Gesamtsystem nach den Anforderungen der Komponente mit dem höchsten Sicherheits-Integritätslevel zu behandeln ist. In der Praxis führt dies zu einem erheblichen Mehraufwand in der Systementwicklung. Aus diesem Grund ermöglichen die Normen ein Abweichen von dieser Vorgabe, sofern ausreichende Unabhängigkeit der Komponenten gezeigt werden kann („sufficient independence“ in IEC 61508, „independence or freedom from interference“ in ISO 26262). [23]

Aus diesem Grund ist die Abgrenzung verschiedener Systemkomponenten in MCS eine zentrale Herausforderung. Diese Abgrenzung wird als *Segregation* von Komponenten oder *Partitionierung* des Systems bezeichnet und wird in Abschnitt 2.3 weiter erläutert.

2.2 Ausführungsplattformen

Der zunehmende Bedarf an Rechenleistung in eingebetteten Systemen erfordert den Einsatz zunehmend leistungsfähigerer Ausführungsplattformen. Während eingebettete Systeme in der Vergangenheit überwiegend auf Einkernprozessoren umgesetzt wurden, werden zunehmend Multicore-basierte Architekturen eingesetzt. Gleichzeitig eröffnet die enge Einbindung weiterer Komponenten wie Field-Programmable Gate Array (FPGA) in MPSoC neue Möglichkeiten.

Im Folgenden werden die für das Verständnis dieser Arbeit benötigten Eigenschaften und Komponenten der betrachteten eingebetteten Ausführungsplattformen vorgestellt.

2.2.1 Multicore-Prozessoren und MPSoC

In diesem Abschnitt erfolgt die Definition und Vorstellung zentraler Eigenschaften und Klassen von Multicore-Prozessoren und Multi-Processor System-on-Chip (MPSoC). Für

ausführlichere Beschreibungen sei auf [28] und [29] verwiesen.

Multiprozessorsysteme nach [28] sind Rechensysteme, welche mehrere Prozessorkerne integrieren, die über einen gemeinsamen Adressraum auf geteilt genutzten Speicher zugreifen. Sie sind somit in der Lage, mehrere Ausführungsstränge (Threads) parallel auszuführen und ermöglichen somit die Umsetzung von Parallelismus auf Threadebene (engl. thread-level parallelism). Sind mehrere Prozessorkerne auf einem einzelnen Chip integriert, werden diese als Mehrkernprozessoren bzw. *Multicore-Prozessoren* bezeichnet.

Multicore-Prozessoren können homogen oder heterogen aufgebaut sein. Homogene Multicore-Prozessoren bestehen aus gleichartigen Prozessorkernen. Diese werden zumeist gemeinsam von einem Betriebssystem verwaltet. Durch den einheitlichen Aufbau können Tasks Kernen beliebig zugewiesen oder zwischen Kernen migriert werden.

Heterogene Multicore-Prozessoren umfassen Kerne unterschiedlicher Mikroarchitektur[1], welche für unterschiedliche Anwendungen oder Betriebszustände optimiert sein können. Diese Unterschiede können bei der Zuweisung von Tasks zu Kernen berücksichtigt werden, um die Performance des Gesamtsystems zu optimieren. Als Beispiel für eine heterogene Multicore-Prozessorarchitektur sei an dieser Stelle die big.LITTLE-Architektur [31] von ARM genannt. In dieser Architektur werden leistungsfähige Prozessorkerne mit energiesparenden Kernen gleicher Befehlssatzarchitektur[1] kombiniert, welche abwechselnd genutzt werden können, um bei stark variierenden Rechenleistungsanforderungen eine hohe Energieeffizienz zu erreichen [32].

Werden neben einem Prozessor weitere anwendungsspezifische Komponenten auf einem Chip realisiert, wird das entstehende System als *Einchipsystem* (engl. *System-on-Chip (SoC)*) bezeichnet. Sind hierbei mehrere Prozessorkerne integriert, spricht man von einem *Mehrprozessor-Einchipsystem* (engl. *Multi-Processor System-on-Chip (MPSoC)*). Die Komponenten eines (MP)SoC können sowohl als feste Hardwareschaltungen realisiert sein oder mittels rekonfigurierbarer Hardware (FPGA, vgl. Abschnitt 2.2.3) realisiert werden. Mittels rekonfigurierbarer Hardware umgesetzte SoC bezeichnet man diese auch als *rekonfigurierbare SoC*.

2.2.2 Speicherarchitekturen in Multicore- und Multiprozessorsystemen

Im Folgenden wird ein Überblick über die relevanten Eigenschaften von Speicherarchitekturen in MCP und Multiprozessorsystemen gegeben.

[1] Die *Befehlssatzarchitektur* (engl. *Instruction Set Architecture (ISA)*) beschreibt die Schnittstelle zwischen der Prozessorhardware und der darauf ausgeführten Software. Sie beschränkt sich somit auf die Funktionalität des Prozessors und abstrahiert von dessen Hardwareumsetzung. Die *Mikroarchitektur* beschreibt den Aufbau der Prozessorhardware, welche die Befehlssatzarchitektur implementiert. [29, 30]

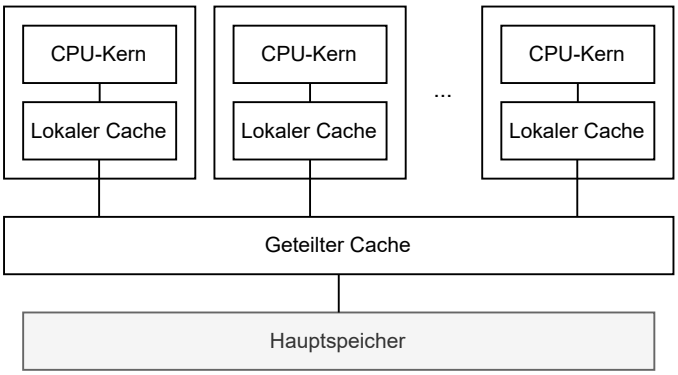


Abbildung 2.2: Multiprozessorsystem mit zentralem geteiltem Speicher (nach [28])

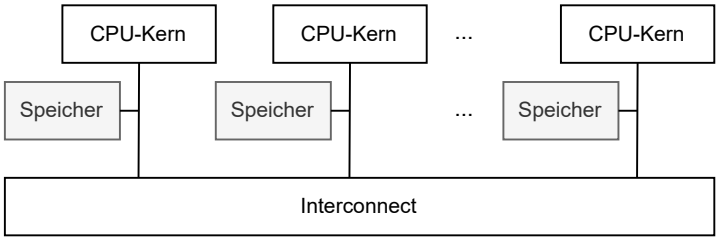


Abbildung 2.3: Multiprozessorsystem mit verteiltem geteiltem Speicher (nach [28])

2.2.2.1 Zentralisierte und verteilte Speicherarchitekturen

In Multicore- und Multiprozessorsystemen lassen sich zwei Ansätze zur Platzierung von Speicher im Rechensystem unterscheiden.

In *symmetrischen* oder *zentralisierten* Multiprozessorsystemen greifen alle Kerne des Hauptprozessors (Hauptprozessor (engl. Central Processing Unit), CPU) auf einen zentralen, geteilt genutzten Hauptspeicher zu (s. Abbildung 2.2). Der Speicherzugriff erfolgt hierbei für alle Kerne und für den gesamten Speicherbereich auf gleichem Weg. Dies wird als *uniform memory access (UMA)* bezeichnet und hat zur Folge, dass für alle Kerne einheitliche Latenzen beim Speicherzugriff erreicht werden. Während der zentralisierte Ansatz in Multicore-Systemen mit moderater Anzahl an Kernen verbreitet ist, wird die Speicheranbindung mit zunehmender Anzahl an Kernen zum begrenzenden Faktor. [33]

Multiprozessorsysteme mit verteiltem, geteiltem Speicher lösen dieses Problem, indem Speicher nahe an jedem CPU-Kern vorgehalten wird. Beim Zugriff auf lokalen Speicher können hierbei hohe Bandbreiten und niedrige Zugriffslatenzen erreicht werden. Der

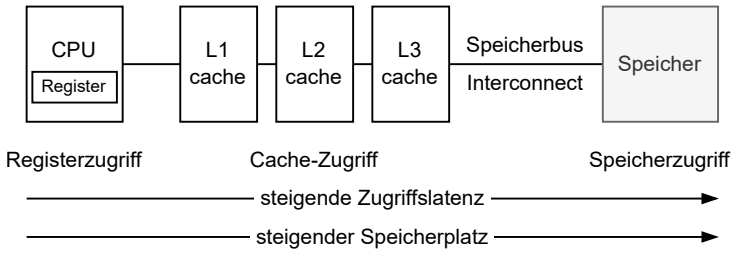


Abbildung 2.4: Speicherhierarchie mit drei Cache-Stufen (eigene Abb. nach [28])

Zugriff auf Speicher anderer Kerne ist über ein *Interconnect* möglich, führt jedoch zu höherer Latenz als lokale Speicherzugriffe. Dementsprechend wird dieser Ansatz auch als *non-uniform memory access (NUMA)* bezeichnet.

2.2.2.2 Caches

Zur Beschleunigung von Speicherzugriffen werden in Rechnersystemen Cache-Speicher eingesetzt. Hierbei handelt es sich um vergleichsweise kleine, auf niedrige Zugriffslatenz optimierte Speicher, in welchen Daten langsamerer Speicher zwischengespeichert werden. Sie sind zwischen CPU-Kerne und den langsamen Speicher geschaltet und arbeiten aus Softwaresicht transparent.

Sind die beim Zugriff auf eine Speicheradresse durch die CPU angefragten Daten im Cache vorhanden, werden diese aus dem Cache bereitgestellt. Dies wird als *Cache Hit* bezeichnet. Sind sie nicht vorhanden *(Cache Miss)*, müssen sie zunächst aus dem nachfolgenden, langsameren Speicher geladen und im Cache gespeichert werden. Hierbei werden Daten blockweise aus dem Speicher geholt, da örtlich nahe liegende Daten nach dem Lokalitätsprinzip[2] mit hoher Wahrscheinlichkeit ebenfalls zeitnah benötigt werden.

Aufgrund der begrenzten Speicherkapazität eines Caches muss im Fall eines Cache Miss ein bestehender Block mit den neu abgerufenen Daten überschrieben werden. Die Methode zur Auswahl des zu ersetzenden Blocks wird als *Verdrängungsstrategie* bezeichnet. Gängige Strategien umfassen die zufällige Auswahl eines Blocks, die Auswahl des am längsten ungenutzten Blocks *(least recently used (LRU))* oder die Auswahl des am längsten im Cache gespeicherten Blocks *(first in, first out (FIFO))* [28].

[2] Das *Lokalitätsprinzip* beschreibt eine typische Eigenschaft von Programmen im Hinblick auf deren Speicherzugriffe. Es wurde beobachtet, dass genutzte Daten und Instruktionen häufig mit kurzem zeitlichem Abstand erneut genutzt bzw. aus dem Speicher abgefragt werden *(zeitliche Lokalität)*. Weiterhin wurde beobachtet, dass Programme nach einem Speicherzugriff häufig auf nahe gelegene Speicheradressen zugreifen *(örtliche Lokalität)*. Zur Veranschaulichung beschreiben Hennessy et al. die Faustregel, dass Programme typischerweise 90 % ihrer Laufzeit innerhalb eines Zehntels ihres Codeumfangs verbringen. [28]

In leistungsfähigen Systemen werden zumeist mehrstufige Caches eingesetzt. Die Caches werden hierbei wie in Abbildung 2.4 dargestellt mit zunehmender Entfernung von der CPU als Level-1- bis Level-n-Cache bezeichnet, der Cache mit dem höchsten Level wird auch als Last-Level Cache (LLC) bezeichnet. Mit zunehmender Entfernung von der CPU nimmt hierbei die Geschwindigkeit des Speicherzugriffs ab und die Speicherkapazität der Caches zu. L1-Caches werden häufig für Instruktionen und Daten getrennt ausgeführt. In Multicore-Prozessoren sind L1-Caches, teilweise auch L2-Caches, für jeden Kern separat vorhanden. Höhere Cache-Level werden in der Regel als geteilte Caches ausgeführt, welche von allen Kernen des Prozessors gemeinsam genutzt werden. Für separat ausgeführte Caches muss hierbei sichergestellt werden, dass Cache-Einträge eines Kerns bei Veränderung des zugehörigen Speicherbereichs durch einen anderen Kern aktualisiert oder als ungültig markiert werden. Dies wird durch den Einsatz von Cache-Kohärenz-Systemen erreicht und in [28] beschrieben.

2.2.3 Field-Programmable Gate Arrays

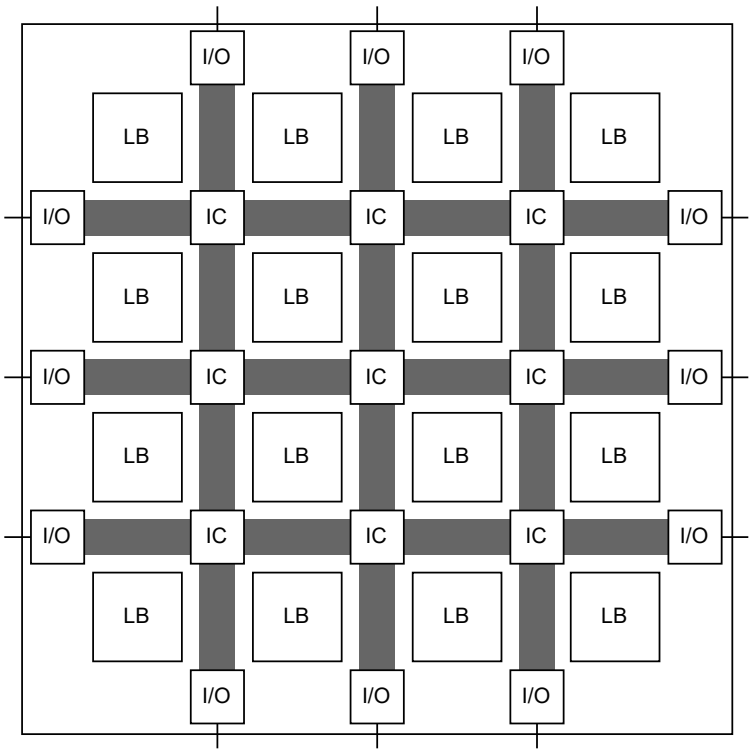
Im Folgenden wird ein grundlegender Überblick über die Funktion und den Aufbau von Field-Programmable Gate Arrays (FPGA) sowie ihre Integration in übergeordnete Systeme gegeben. Die Ausführungen beschränken sich auf die zum Verständnis der vorliegenden Arbeit nötigen Aspekte. Für eine detailliertere Einführung in die FPGA-Technologie sei auf [34] verwiesen.

Field-Programmable Gate Arrays sind integrierte Schaltungen, welche die Umsetzung digitaler Logikschaltungen[3] durch Konfiguration zur Laufzeit ermöglichen. Sie stellen somit eine Alternative zu anwendungsspezifischen integrierten Schaltkreisen (application-specific integrated circuits, ASIC) dar, deren Funktion auf Hardwareebene durch ihre Struktur definiert ist und somit im Rahmen der Fertigung fest vorgegeben wird. FPGAs finden somit häufig in Fällen Verwendung, in welchen anwendungsspezifische Logikschaltungen benötigt werden, die Entwicklung von ASICs, häufig auf Grund geringer Stückzahl, jedoch nicht wirtschaftlich ist. Darüber hinaus bildet die Rekonfigurierbarkeit von FPGAs die Grundlage für weitere Anwendungsfelder wie die Umsetzung anwendungsspezifischer, zur Laufzeit austauschbarer oder anpassbarer Beschleuniger zur schnelleren oder effizienteren Ausführung von Datenverarbeitung.

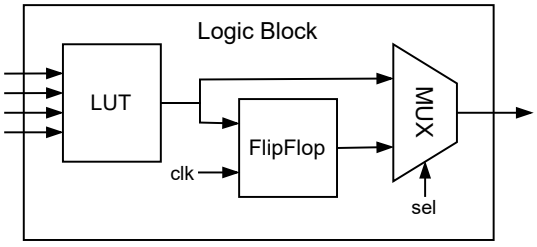
2.2.3.1 Grundlegender Aufbau

Abbildung 2.5 stellt den Aufbau eines FPGA vereinfacht dar. FPGAs bestehen meist aus einer regelmäßigen Anordnung von konfigurierbaren Logikblöcken (LB), welche

[3] Der Übersichtlichkeit halber werden unter dem Begriff *Logikschaltungen* in dieser Arbeit sowohl zustandslose kombinatorische Schaltungen als auch zustandsbehaftete Schaltwerke zusammengefasst.



(a) FPGA-Struktur: Logikblöcke (LB) verbunden über ein Netzwerk von Interconnect-Blöcken (IC). Die Schnittstelle nach außen erfolgt über Input/Output-Blöcke (I/O). (nach [34])



(b) Vereinfachte Darstellung eines Logikblocks (nach [34])

Abbildung 2.5: Vereinfachte Darstellung der Struktur eines FPGA

über eine konfigurierbare Verbindungslogik miteinander verbunden werden können. Die Logikblöcke realisieren hierbei jeweils Teile der Logikschaltung und werden durch das Verbindungsnetzwerk zu größeren Schaltnetzen und Schaltwerken verbunden. Logikblöcke enthalten hierzu folgende zentrale Komponenten (vgl. Abbildung 2.5b):

- Eine Lookup Table (LUT) zur Umsetzung der Logikfunktion. Diese ist häufig als SRAM (Static Random Access Memory) ausgeführt.
- Ein FlipFlop als Zustandsspeicher zur Implementierung von Schaltwerken.
- Ein Multiplexer zur Bestimmung der Ausgabe des LB. Dies ermöglicht die Überbrückung des FlipFlop zur Umsetzung größerer Schaltnetze durch Verkettung mehrerer Logikblöcke.

Neben Logikblöcken umfassen FPGAs häufig auch spezialisierte Blöcke wie Multiplikatoren, digitale Signalprozessoren (DSP) und Speicher (Block RAM, BRAM). Weiterhin umfassen sie Input/Output-Blöcke (I/O-Blöcke), welche die Schnittstelle zwischen der Verbindungslogik und Komponenten außerhalb des FPGA darstellen. Als solche dienen sie u.a. zur Wandlung und Aufbereitung der Signalpegel, zum Synchronisieren und zum Umsetzen des Signals in verschiedene I/O-Standards.

2.2.3.2 Konfiguration und dynamische partielle Rekonfiguration

Im FPGA umzusetzende Schaltungen werden meist mittels Hardwarebeschreibungssprachen (Hardware Description Language, HDL) spezifiziert. Im Rahmen einer automatisierten Synthese wird daraus ein binärer Datenstroms (Bitstream) erzeugt, welcher vom FPGA geladen werden kann und die Konfiguration der LBs und der Verbindungslogik enthält.

Das Laden des Bitstreams erfolgt in vielen Anwendungen im Rahmen der Startphase des umgebenden Systems, kann jedoch auch zur Laufzeit mehrfach angepasst werden *(Rekonfiguration)*, um die implementierte Logikschaltung zu ändern. Moderne FPGAs unterstützen *partielle Rekonfiguration*, wodurch Teile des FPGA rekonfiguriert werden können, während die im verbleibenden Teil liegende Logikschaltung von der Rekonfiguration unbeeinflusst bleibt. Hierzu können im Rahmen der Systementwicklung *partiell rekonfigurierbare Regionen (PRR)* definiert werden. Durch fest definierte Schnittstellen der PRR können darin implementierte Funktionen zur Laufzeit mittels dynamischer partieller Rekonfiguration (DPR) gegen alternative Funktionen ausgetauscht werden. Dies ermöglicht den Austausch von (Teil-)Funktionen, ohne die Ausführung parallel ausgeführter Funktionen zu beeinträchtigen.

2.2.4 Trace-Architekturen

Die Entwicklung komplexer eingebetteter Systeme erfordert häufig, das Verhalten eingebetteter Prozessoren bei der Programmausführung zu beobachten. Im Rahmen von Funktionstests müssen bspw. der korrekte Programmablauf und das Einhalten von Timinganforderungen bestimmter Programmteile nachvollzogen werden. Zeigt das System unerwartetes Verhalten, muss es im Rahmen des Debuggings analysiert werden, um die Ursache der Abweichung identifizieren zu können.

Eine einfache Möglichkeit, Informationen zur Programmausführung zu erhalten, ist die Instrumentierung der Software. Hierbei wird Code eingefügt, welcher bei der Ausführung Artefakte erzeugt, die Aufschluss über das Programmverhalten geben. Im Kontext von eingebetteten Systemen, verteilten Systemen und Echtzeitsystemen kann dies jedoch mit Nachteilen verbunden sein, da die Instrumentierung das Zeitverhalten der beobachteten Software beeinflusst.

Moderne Prozessoren und System-on-Chip umfassen daher Trace-Subsysteme, welche die Aufzeichnung relevanter Ereignisse während der Ausführung von Software erlauben. Sie sind als Teil der Prozessor- bzw. SoC-Hardware implementiert und ermöglichen es, relevante Aspekte der Softwareausführung auf dem Prozessor ohne Modifikation der Software nachzuvollziehen. Der Aufbau dieser Subsysteme ist durch die Trace-Architektur bestimmt. Verbreitete Trace-Architekturen umfassen bspw. Intel Processor Trace [35, Kap. 33], ARM CoreSight [36], sowie Multicore Debug Solution [37]. Gleichzeitig existieren Standards, welche die Schnittstelle zum Zugriff auf Trace-Systeme beschreiben, wie bspw. Nexus 5001 [38].

An Trace-Architekturen werden verschiedene Anforderungen gestellt:

- Zentrales Ziel ist die Bereitstellung von Informationen, welche das Nachvollziehen der Softwareausführung ermöglichen. Relevante Aspekte sind hierbei zumeist der Programmablauf, das Zeitverhalten, Datenzugriffe und der Ausführungskontext.
- Gleichzeitig soll die Datenaufnahme und -verarbeitung durch das Trace-System die Ausführung der Software auf dem Prozessor bzw. SoC möglichst nicht beeinflussen.
- Moderne Prozessoren arbeiten mit hohen Taktraten. Dementsprechend fallen Ausführungsinformationen mit hoher Datenrate an. Gleichzeitig steht Speicher, besonders in eingebetteten Systemen, meist nur in begrenztem Umfang zur Verfügung. Trace-Architekturen müssen daher die Filterung relevanter Ereignisse ermöglichen und sind auf kompakte Codierung der Ausführungsinformationen optimiert.

Abbildung 2.6 stellt die ARM CoreSight Trace-Architektur des Zynq UltraScale+ MPSoC vereinfacht dar. Das Trace-Subsystem erweitert jeden CPU-Kern um eine Einheit zur Tracedatengenerierung. Die generierten Tracedaten werden über einen vom System-

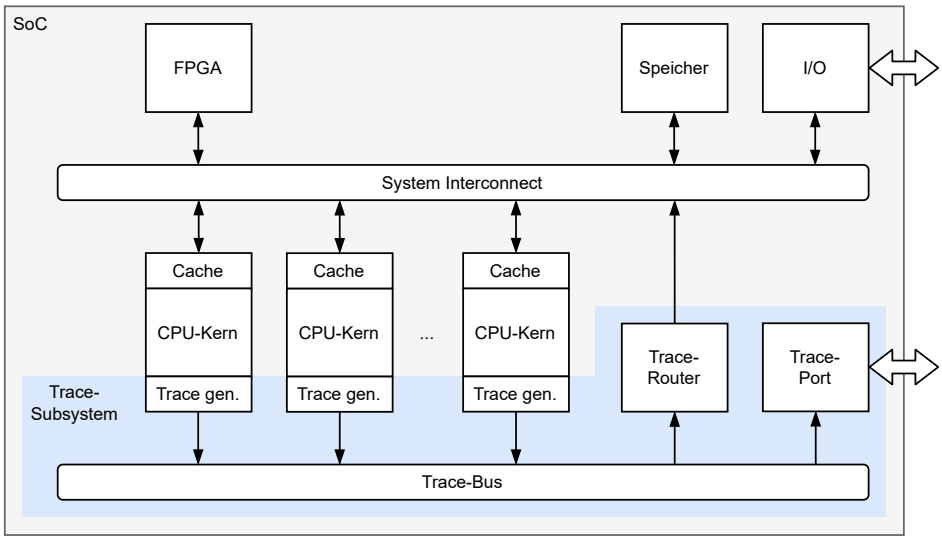


Abbildung 2.6: Vereinfachte Darstellung der Trace-Architektur eines Zynq UltraScale+ MPSoC (vgl. [39])

Interconnect getrennten Trace-Datenbus entweder zu einem Trace-Port oder einem Trace-Router weitergeleitet. Zusätzlich bietet das Trace-Subsystem die Möglichkeit, Trace-Daten in einem Speicher des Trace-Subsystems zwischenzuspeichern. Der Trace-Port stellt eine Schnittstelle nach außen bereit, über welche externe Komponenten zum Ausleiten und Analysieren der Tracedaten angeschlossen werden können. Der Trace-Router ermöglicht das Weiterleiten der Trace-Daten aus dem Trace-Subsystem in weitere Systemteile des SoC. Insbesondere ermöglicht er das Speichern der Trace-Daten im Hauptspeicher, wodurch auch größere Datenmengen gespeichert werden können und weitere Systemkomponenten auf diese zugreifen können. Im Fall des dargestellten Zynq UltraScale+ ist außerdem ein Weiterleiten der Daten in den FPGA des SoC möglich, in welchem die Tracedaten ohne Zwischenspeicherung weiterverarbeitet werden können. Durch die Nutzung eines separaten Bussystems sowie die Möglichkeit, Daten über den Trace-Port auszuleiten oder im Trace-Subsystem zu speichern, bietet die Trace-Architektur die Möglichkeit, Tracedaten mit minimalem Einfluss auf die Softwareausführung auf der CPU zu erfassen.

Trace-Systeme werden häufig im Rahmen der Systementwicklung genutzt. Darüber hinaus werden sie zunehmend im Hinblick auf Anwendungen im Bereich der funktionalen Sicherheit betrachtet. So wird die Nutzbarkeit von Trace- und Debug-Infrastruktur bspw. zur Überwachung der Programmausführung auf Verletzungen zulässiger Kontrollflüsse untersucht (engl. *control flow checking*) [40, 41, 42, 43, 44].

2.2.4.1 Zentrale Funktionen von Trace-Architekturen

Im Folgenden werden zentrale Funktionen von Trace-Architekturen vorgestellt. Die zugehörigen, typischen Trace-Datenpakettypen werden in [45] vorgestellt und sind im Kontext der Funktionen jeweils kurz zusammengefasst.

2.2.4.1.1 Bereitstellen von Kontrollflussinformationen Die Begriffe *Programmablauf* und *Kontrollfluss* bezeichnen die Abfolge der in einem Programm ausgeführten Instruktionen. Der Kontrollfluss wird durch mehrere Aspekte bestimmt. Grundsätzlich verarbeitet eine CPU Instruktionen sequenziell entsprechend ihrer Anordnung im Programmcode. Abweichungen hiervon ergeben sich insbesondere durch

- das Verhalten der Software bei (bedingten) Verzweigungsbefehlen *(Branch-Instruktionen)*,
- auftretende Ereignisse wie Interrupts bzw. Exceptions und
- bestimmtes Programmverhalten, z.B. Schreibzugriffe auf den Befehlszähler *(program counter)*.

Trace-Systeme haben das Ziel, den Kontrollfluss der auf einem Prozessor ausgeführten Software nachvollziehbar zu machen. Um den Umfang der Trace-Daten gering zu halten, wird häufig angenommen, dass der Programmcode zur Rekonstruktion des Kontrollflusses zur Verfügung steht. Dementsprechend werden meist nur dann Trace-Informationen generiert, wenn der Programmablauf nicht anhand des Programmcodes erkennbar ist.

- **Synchronisations-Datenpakete** dienen dazu, den aktuellen Wert des Befehlszählers sowie des Ausführungsmodus des Prozessors mitzuteilen. Sie bilden somit eine Referenz zur Interpretation nachfolgender Trace-Datenpakete. Synchronisations-Pakete werden meist zum Start des Trace-Vorgangs und dann in regelmäßigen Intervallen erzeugt.
- **Branch-Datenpakete** beschreiben zur Laufzeit getroffene Entscheidungen bzgl. des Kontrollflusses. Im Fall einer bedingten Branch-Instruktion mit fester Sprungadresse wird hierbei nur ein Bit benötigt, um zu beschreiben, ob der Sprung durchgeführt wurde. Wird die Sprungadresse zur Laufzeit bestimmt, wird diese als Teil des Branch-Datenpakets aufgezeichnet.
- **Exception-Datenpakete** signalisieren das Auftreten einer Exception und enthalten die zur Rekonstruktion des Kontrollflusses benötigten weiteren Informationen.

2.2.4.1.2 Bereitstellen von Informationen zum Zeitverhalten Um Informationen über das Zeitverhalten der Anwendungsausführung zu bekommen, ermöglichen Trace-Architekturen

meist die Ausgabe von **Timing-Datenpaketen**. Diese können Informationen zur aktuellen Zeit, dem Wert eines Referenz-Timers oder der Anzahl vergangener CPU-Taktzyklen enthalten. Timing-Datenpakete werden typischerweise zeitgesteuert oder in Kombination mit anderen Datenpaketen erzeugt.

2.2.4.1.3 Bereitstellen von Datenflussinformation Mache Trace-Architekturen ermöglichen das Generieren von **Datenfluss-Datenpaketen**, welche Aufschluss über Lese- bzw. Schreiboperationen geben. Aufgrund der hohen Tracedatenrate ist Datenflusstracing oft nur eingeschränkt nutzbar.

2.2.4.1.4 Einschränkung betrachteter Trace-Events Die uneingeschränkte Aufnahme von Traces führt typischerweise zu sehr hohen Datenraten und bei längeren Aufnahmen zu hohen Datenmengen. Gleichzeitig sind meist nur die Traces eines Teils der auf einer CPU ausgeführten Software von Interesse. Trace-Architekturen bieten daher die Möglichkeit, den Betrachtungsbereich der Tracedatenaufnahme durch *Filter* zu beschränken. Hierbei kann meist nach Adressbereichen des Befehlszählers oder dem Ausführungsmodus der CPU gefiltert werden. Weiterhin ermöglichen *Trigger*, die Tracedatenaufnahme bei Auftreten bestimmter Ereignisse zu starten oder zu stoppen.

2.3 Partitionierung

Im Kontext eingebetteter sicherheitskritischer Systeme dient *Partitionierung* (auch als *Segregation* bezeichnet) der sicheren Ausführung verschiedener Systemfunktionen auf einer geteilten Ausführungsplattform. Ziel der Partitionierung ist dabei, Fehlerauswirkungen auf Partitionsebene zu kapseln. So dürfen Fehler, welche in einer Partition auftreten, nicht zu Fehlern in anderen Partitionen führen [46].

Partitionierungsmechanismen zielen hierbei nicht primär auf Fehler der Ausführungshardware ab, sondern setzen zumeist die korrekte Funktion der Ausführungshardware voraus. Stattdessen dienen sie der Beherrschung des zusätzlichen Risikos, welches sich durch die geteilte Nutzung von Ressourcen durch unterschiedliche Anwendungen ergibt: Das Risiko, dass das (Fehl-)Verhalten einer Systemfunktion andere Systemfunktionen auf der Plattform (negativ) beeinflusst. [46]

Die Ursache dieser möglichen gegenseitigen Beeinflussung liegt in der geteilten Nutzung der Plattformenressourcen: So kann eine Anwendung ohne Partitionierungsmaßnahmen bei geteilter Speichernutzung auf Daten und Programmcode anderer Anwendungen zugreifen und diese zur Laufzeit verändern. Bei geteilter Nutzung von Caches kann weiterhin die gegenseitige Verdrängung von Programmcode oder Daten aus dem Cache

die Ausführungszeit parallel laufender Anwendungen erhöhen.

Die genannten Beispiele stellen zwei Klassen der Beeinflussung dar: Die Veränderung von Daten und Programmcode anderer Anwendungen sowie die Beeinträchtigung der Verfügbarkeit von Rechenressourcen für andere Anwendungen. Im Kontext der Partitionierung wird dementsprechend zwischen *räumlicher Partitionierung* und *zeitlicher Partitionierung* unterschieden, welche im Folgenden beschrieben werden.

2.3.1 Räumliche Partitionierung

Räumliche Partitionierung zielt auf die Isolation von Partitionen hinsichtlich ihrer Daten ab. Hierbei soll verhindert werden, dass Anwendungen einer Partition Zugriff auf Daten einer anderen Partition erhalten. Dies betrifft insbesondere im Arbeitsspeicher abgelegte Daten, aber auch Programmcode und Register genutzter Hardwarekomponenten.

2.3.2 Zeitliche Partitionierung und Interferenz

Zeitliche Partitionierung (auch *zeitliche Isolation*) zielt darauf ab, die Beeinflussung des Zeitverhaltens einer Partition durch eine andere Partition zu verhindern [46]. Diese Beeinflussung wird auch als *Interferenz* bezeichnet. Insbesondere soll verhindert werden, dass Aktivitäten innerhalb einer Partition den Zugriff einer anderen Partition auf geteilte Ressourcen beeinflussen [47].

Rushby [46] gibt in einer frühen Arbeit einen Überblick über verschiedene Aspekte, welche zur Umsetzung zeitlicher Partitionierung betrachtet werden müssen. So muss unter anderem das Ausführen von Instruktionen, welche das Zeitverhalten des Gesamtsystems beeinflussen, unterbunden werden. Es müssen Fälle behandelt werden, in welchen Anwendungen ihre zugewiesene Rechenzeit überschreiten oder wechselseitig exklusiv genutzte geteilte Ressourcen zu lange blockieren. Weiterhin muss der Einfluss von eingehenden Interrupts beachtet werden, wenn diese nicht zur aktiven Partition gehören. Außerdem weist Rushby bereits auf mögliche Interferenz an der Speicherinfrastruktur zwischen Prozessor und Direct-Memory-Access-Einheiten (DMA-Einheiten) als mögliche Problemquelle bei der Umsetzung zeitlicher Partitionierung hin. Vergleichbare Interferenzeffekte stellen auch im Kontext von Multicore-Prozessoren eine Herausforderung dar und werden im Kontext dieser Arbeit betrachtet.

In Multicore-Prozessoren kann die parallele Ausführung von Software auf mehreren Kernen zu Interferenz führen. Diese entsteht durch die konkurrierende Nutzung von Plattformressourcen und zeigt sich in einer Verlangsamung der Anwendungsausführung im Vergleich zur exklusiven Ausführung ohne parallel laufende Software. Die Eigenschaften der Ausführungshardware, welche Interferenzeffekte auslösen, werden als *Interferenzkanäle*

bezeichnet. Hardware-Interferenzkanäle auf Multicore-Plattformen umfassen nach [48] Caches, insbesondere geteilte Caches und Cache-Kohärenz-Mechanismen, Datenbusse, geteilte I/O-Komponenten und Interrupts.

2.4 Virtualisierung und Hypervisors

2.4.1 Virtualisierung

Im Kontext rechnerbasierter Datenverarbeitung bezeichnet der Begriff *Virtualisierung* die Abstraktion von realen Objekten hin zu virtuellen Objekten [49]. Anstelle physischer Komponenten, wie CPUs, Arbeits- und Massenspeicher oder Netzwerken, werden mittels einer Abstraktionsschicht virtuelle *(logische)* Instanzen solcher Komponenten bereitgestellt. Sie bilden das Verhalten physischer Komponenten nach und können von darauf aufbauender Software wie ihre physischen Pendants genutzt werden[4].

Durch die Abstraktion von physischer Hardware ermöglicht Virtualisierung die geteilte Nutzung von Ressourcen durch Software, welche auf die exklusive Nutzung dieser Ressourcen ausgelegt ist. Hierdurch kann eine höhere Auslastung der physischen Hardware und damit eine höhere Systemeffizienz erreicht werden. Gleichzeitig können virtuelle Ressourcen flexibler verwaltet werden als physische Ressourcen. Sie können softwaregesteuert angelegt, entfernt, angepasst oder zwischen physischen Systemen migriert werden. Letzteres kann wiederum zur Erhöhung der Verfügbarkeit einer Ressource oder eines Dienstes genutzt werden. Auch können von der physisch verfügbaren Hardware abweichende Komponenten emuliert werden, wodurch die Ausführung von Software ermöglicht wird, welche ursprünglich für andere Zielplattformen entwickelt wurde.

2.4.2 Hypervisors und virtuelle Maschinen

Ein Hypervisor ist eine Softwareschicht, welche auf einem physischen Rechnersystem *(Host-System)* ausgeführt wird und *Virtuelle Maschinen (VM)* bereitstellt und ausführt. VMs werden auch als *Gäste* oder *Partitionen* bezeichnet. Sie stellen virtuelle Rechnersysteme dar und umfassen dementsprechend virtuelle Instanzen der Ressourcen eines Rechners, wie Prozessoren, Speicher und Kommunikationsschnittstellen, auf welche die darauf ausgeführte *Gastsoftware* exklusiven Zugriff hat.

In ihrer grundlegenden Arbeit zur Virtualisierung von Rechnersystemen [50] definieren G. Popek und R. Goldberg virtuelle Maschinen als *effiziente, isolierte Duplikate realer Rechner*. An Hypervisors, in [50] *Virtual Machine Monitor (VMM)* genannt, stellen sie dazu drei zentrale Anforderungen:

[4] Im Fall der Paravirtualisierung erfolgt diese Nachbildung nicht vollständig, s. Abschnitt 2.4.3.1.

- **Äquivalenz** Hypervisors stellen Ausführungsumgebungen bereit, welche sich grundsätzlich identisch zum physischen Rechner verhalten. Programme zeigen bei Ausführung in der VM und Ausführung auf der physischen Hardware das gleiche Verhalten. Abweichungen hiervon sind nur zulässig, wenn sie auf das Zeitverhalten der Software oder die Verfügbarkeit von Rechenressourcen zurückzuführen sind.
- **Effizienz** Die Ausführung von Software in einer VM soll nur geringfügig langsamer sein als auf dem physischen Rechnersystem. Der Großteil der Instruktionen der Gastsoftware kann nativ auf der Host-CPU ausgeführt werden.
- **Ressourcenkontrolle** Der Hypervisor hat die komplette Kontrolle über die Systemressourcen. Gastsoftware hat nur Zugriff auf Ressourcen, welche der VM explizit zugewiesen wurden. Weiterhin ist der Hypervisor ist in der Lage, Zugriff auf Ressourcen wiederzuerlangen, welche zuvor einer VM zugewiesen wurden.

Der Hypervisor erzeugt diese Ausführungsumgebungen unter Nutzung der Hardwareressourcen des Host-Systems. Dies umfasst die Ausführung der Gastsoftware, die Bereitstellung von Speicher sowie Input/Output-Komponenten (I/O-Komponenten) wie Massenspeicher oder Kommunikationsschnittstellen. Die Virtualisierung kann hierbei rein softwarebasiert oder unter Nutzung von Hardwareunterstützung des Host-Systems erfolgen.

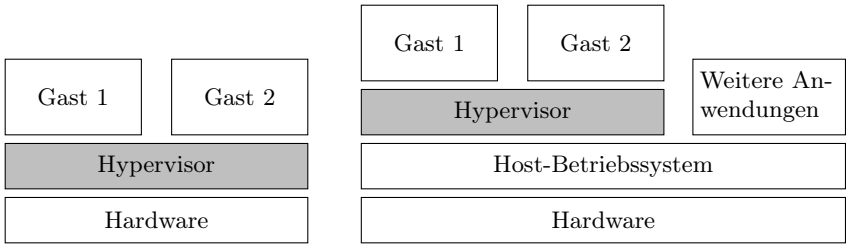
2.4.3 Klassifizierung von Virtualisierungstechniken und Hypervisors

Im Folgenden werden Eigenschaften von Hypervisors und Virtualisierungstechniken vorgestellt, anhand von welchen diese klassifiziert werden. Die Eigenschaften orientieren sich an der von Cinque et al. [51] vorgestellten Taxonomie.

2.4.3.1 Voll- und Paravirtualisierung

Wie in Abschnitt 2.4.2 vorgestellt, fordert die von G. Popek und R. Goldberg eingeführte Definition virtueller Maschinen Äquivalenz zwischen der Ausführungsumgebung der VM und einem physischen Rechnersystem. Dies wird als *Vollvirtualisierung* bezeichnet. Das Verhalten der physischen Rechenressourcen wird hierbei vom Hypervisor vollständig nachgebildet, wodurch sich der Zugriff auf diese Ressourcen durch Gastanwendungen nicht vom Zugriff auf physische Ressourcen unterscheidet. Somit können Anwendungen ohne Anpassung in virtuellen Maschinen ausgeführt werden.

Im Gegensatz zur Vollvirtualisierung bilden Hypervisors Rechenressourcen im Fall der *Paravirtualisierung* nicht vollständig nach. Stattdessen erfolgen bestimmte Ressourcenzugriffe über eine Programmierschnittstelle (engl. application programming interface,



(a) Typ-1-Hypervisor (b) Typ-2-Hypervisor

Abbildung 2.7: Hypervisor-Typen

API), welche dem Gast durch den Hypervisor bereitgestellt wird. Dementsprechend muss Gastsoftware bei der Portierung vom physischen System in eine virtuelle Maschine an diese Schnittstelle angepasst werden. Im Gegensatz zu Vollvirtualisierung reduziert Paravirtualisierung häufig den Aufwand der Virtualisierung und erhöht die Ausführungsgeschwindigkeit der Software in der VM.

2.4.3.2 Typ-1- und Typ-2-Hypervisors

Im Hinblick auf den Softwarestack eines virtualisierten Systems wird zwischen Typ-1-Hypervisors und Typ-2-Hypervisors unterschieden [52]. *Typ-1-Hypervisors* werden direkt auf der Hardware des Rechensystems ausgeführt und somit auch als *Bare-Metal-Hypervisors* bezeichnet (vgl. Abbildung 2.7a). Sie umfassen dementsprechend die für die Ausführung auf der Hardware benötigten Treiber. *Typ-2-Hypervisors* werden stattdessen als Anwendung innerhalb eines darunterliegenden Betriebssystems ausgeführt, welches die verfügbaren Rechenressourcen verwaltet und bereits eine Abstraktion von der Hardware bietet (vgl. Abbildung 2.7b). Dies entkoppelt die Bereitstellung von Rechenressourcen durch das Betriebssystem von der Bereitstellung der virtuellen Ausführungsumgebung durch den Hypervisor und erhöht dadurch dessen Portierbarkeit auf andere Systeme. Weiterhin ermöglicht es die parallele Ausführung weiterer Anwendungen durch das Betriebssystem.

2.4.3.3 Echtzeitfähigkeit

Cinque et al. [51] unterscheiden zwischen echtzeitfähigen und nicht echtzeitfähigen Hypervisors. Hierbei bieten echtzeitfähige Hypervisors die Möglichkeit, Echtzeitverhalten der Gastsoftware zu erreichen. Hierzu werden insbesondere geeignete Schedulingmechanismen bereitgestellt.

2.4.3.4 Statische und dynamische Ressourcenzuweisung

Hypervisors können im Hinblick auf die Dynamik der Zuweisung von Rechenressourcen zu virtuellen Maschinen klassifiziert werden. Hypervisors mit *dynamischer Ressourcenzuweisung* passen die Zuteilung von Ressourcen an VMs zur Laufzeit an die Bedürfnisse der verwalteten VMs an, während Hypervisors mit *statischer Ressourcenzuweisung* ein festes Mapping von Rechenressourcen zu VMs durchsetzen, welches unabhängig vom aktuellen Bedarf der Gastsoftware ist. So führen *statisch partitionierende Hypervisors* häufig eine feste Partitionierung der Systemressourcen durch, indem sie physische CPU-Kerne und I/O-Komponenten fest VMs zuweisen. Da eine statische Ressourcenzuweisung die Nachvollziehbarkeit der Systempartitionierung vereinfacht, wird diese in sicherheitskritischen Systemen häufig bevorzugt. Auch geht eine statische Ressourcenzuweisung meist zu einem geringeren Virtualisierungs-Overhead und einer kleineren Codebasis des Hypervisors einher. [51]

2.4.3.5 Hypervisors für Eingebettete Systeme

Hypervisors für eingebettete Systeme (engl. auch *Embedded Hypervisors*) sind auf den Einsatz in eingebetteten Systemen ausgerichtet. Die Anforderungen solcher Systeme und daraus abgeleitete Anforderungen an Hypervisors werden in [53] diskutiert und sind im Folgenden zusammengefasst.

Eingebettete Systeme zeichnen sich durch mehrere Eigenschaften aus. Da sie zur Steuerung physischer Prozesse eingesetzt werden, bestehen zumeist für einen Teil der im eingebetteten System erbrachten Funktion Echtzeitanforderungen. Weiterhin stehen Rechenressourcen nur in begrenztem Umfang zur Verfügung. Im Fall mobiler Systeme betrifft dies auch die Ressource Energie. Gleichzeitig besteht der Trend hin zu wachsender Funktionalität und Komplexität der Software in eingebetteten Systemen mit intensivem Datenaustausch zwischen den enthaltenen Softwarekomponenten. Außerdem kann eine zunehmende Öffnung des eingebetteten Systems gegenüber weiteren Systemen und Netzwerken beobachtet werden, z.B. zur Anbindung an Endgeräte des Nutzers oder das Internet. Auch werden zunehmend komplexe Betriebssysteme im eingebetteten Kontext genutzt, um auf bestehenden Diensten und Infrastruktur von General-Purpose-Betriebssystemen (GPOS) aufbauen zu können.

Hieraus ergeben sich verschiedene Anforderungen an Hypervisors in eingebetteten Systemen. Zur sicheren Ausführung von Softwarekomponenten auf geteilter Hardware werden starke Isolationsmechanismen zwischen Partitionen bzw. VMs benötigt. Dies setzt eine möglichst kleine Trusted Computing Base (TCB)[5] des Hypervisors voraus. Um die

[5] Als TCB werden Softwarekomponenten bezeichnet, welche zentrale Sicherheitsfunktionen eines

Kommunikationsanforderungen der Softwarekomponenten zu erfüllen, sind gleichzeitig vom Hypervisor kontrollierte Mechanismen zur definierten Inter-Partitions-Kommunikation (IPC) erforderlich, welche eine Kommunikation mit geringer Latenz und hoher Datenrate ermöglichen. Weiterhin muss der Hypervisor Echtzeitverhalten für echtzeitkritische Funktionen der Gastsoftware ermöglichen. Im Gegensatz zu Hypervisors im Desktop- und Server-Umfeld, welche das Scheduling auf Hypervisorebene getrennt vom Scheduling der Gastsoftware behandeln, muss er die Möglichkeit bieten, für kritische Tasks ein definiertes Scheduling auf Gesamtsystemebene zu erreichen.

Im Hinblick auf die begrenzt verfügbaren Rechenressourcen ist eine effiziente Ressourcennutzung im eingebetteten System nötig. Der Hypervisor sollte daher sowohl hinsichtlich des eigenen Speicherverbrauchs als auch der durch die Virtualisierung verursachten Verzögerung der Gastsoftware gegenüber nativer Ausführung optimiert sein. Aus diesem Grund werden in eingebetteten Systemen überwiegend Typ-1-Hypervisors eingesetzt, da somit auf die Nutzung eines vollwertigen Host-Betriebssystems verzichtet werden kann. Auch vereinfacht die Nutzung von Typ-1-Hypervisors die Realisierung echtzeitfähiger Systeme, da sie geringere Reaktionslatenzen auf eingehende Ereignisse ermöglichen und zu einer geringeren TCB führen.

2.4.4 Umsetzung der Virtualisierung

In diesem Abschnitt werden grundlegende Ansätze zur Umsetzung der Virtualisierung und Partitionierung von Rechnersystemen durch Hypervisors beschrieben.

2.4.4.1 Virtualisierung der CPU und Scheduling

Die Ausführung der Gastsoftware erfolgt aus logischer Sicht auf einer virtuellen CPU. Sofern deren Architektur der des Host-Systems entspricht, kann die Gastsoftware in Teilen direkt auf der Host-CPU ausgeführt werden. Der Hypervisor stellt hierbei sicher, dass Instruktionen, deren Auswirkungen bei direkter Ausführung auf der Host-CPU über die VM hinaus gehen, gesondert behandelt werden. Dies erfolgt so, dass nur der Kontext der ausführenden VM entsprechend der Instruktion verändert wird. Dies kann durch Hardwareunterstützung (vgl. Intel-VT, AMD-V, ARM virtualization extensions) oder durch binäre Übersetzung der betreffenden Instruktionen erreicht werden [54]. Im Fall der Paravirtualisierung werden solche Instruktionen in der Gastsoftware durch explizite Aufrufe des Hypervisors, sog. *Hypercalls*, ersetzt. Weicht die Architektur der virtuellen CPU von der des Host-Systems ab, kann diese durch den Hypervisor emuliert werden.

Systems umsetzen, sodass Fehler und Schwachstellen in dieser Software die Integrität des gesamten Systems gefährden.

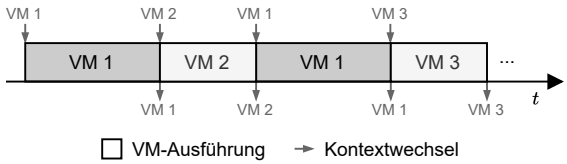


Abbildung 2.8: Scheduling von VMs auf einem Prozessorkern

Wenn die Anzahl virtueller CPU-Kerne (vCPUs) im System die Anzahl physischer Prozessorkerne übersteigt, müssen physische Prozessorkerne zur Ausführung der VMs geteilt genutzt werden. Dies erfolgt mittels Schedulings durch den Hypervisor. Wie in Abbildung 2.8 dargestellt, werden die vCPUs verschiedener VMs hierbei sequenziell auf einem physischen CPU-Kern ausgeführt.

Um trotz sequenzieller Nutzung des physischen Prozessors die Illusion einer exklusiv genutzten vCPU zu erreichen, speichert der Hypervisor im Rahmen des Kontextwechsels zwischen vCPUs den Zustand des physischen Cores zum Ende des Zeitfensters und stellt ihn zum Beginn des nächsten Zeitfensters der VM wieder her. Um hierbei räumliche Partitionierung zwischen den VMs zu erreichen, muss sichergestellt werden, dass keine VM unberechtigt Zugriff auf den (gespeicherten) Zustand der CPU einer anderen VM erhält.

Die Reihenfolge und Länge der Zeitfenster werden von der Scheduler-Komponente des Hypervisors definiert. Verschiedene, im Rahmen dieser Arbeit relevante Schedulingansätze werden in den folgenden Abschnitten vorgestellt.

2.4.4.1.1 Feste, zyklische Zeitfenster Der Avionik-Standard ARINC 653 [55] definiert eine Schnittstelle von Echtzeitbetriebssystemen (real-time operating system, RTOS) zur Integration verschiedener Funktionen auf einer geteilten Ausführungshardware gemäß dem Integrated-Modular-Avionics-Konzept (IMA) [56]. Hierzu ist ein rein zeitgesteuerter Scheduling-Mechanismus definiert, welcher auf festen, zyklischen Zeitfenstern basiert.

Der Schedule basiert auf einem Major Time Frame (MAF) von definierter Dauer, welches vom Scheduler zyklisch ausgeführt wird. Innerhalb des MAF sind Zeitfenster definiert, welche jeweils durch ihr Offset zum Beginn des MAF sowie ihre Länge definiert sind. Während Zeitfenster direkt aufeinander folgen können, sind sich überschneidende Zeitfenster nicht zulässig. Jedem Zeitfenster ist weiterhin eine VM zugewiesen, wobei die gleiche VM mehreren Zeitfenstern zugewiesen sein kann. Die Zuweisung definiert, während welcher Zeiträume die Gastsoftware einer VM auf der CPU ausgeführt werden kann.

Während sich der beschriebene Scheduling-Mechanismus zunächst auf nur eine Ausführungseinheit bezieht, lässt er sich auf Multi-Core-Prozessoren übertragen. Ein solches

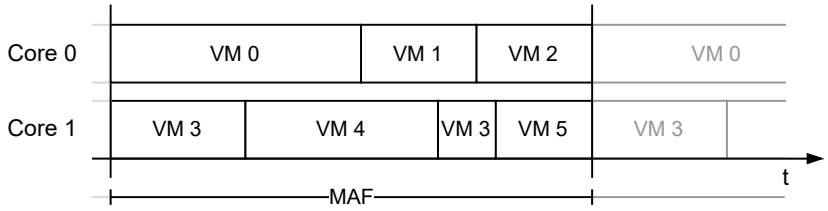


Abbildung 2.9: ARINC 653-Scheduling für mehrere CPU-Kerne

Beispiel ist in Abbildung 2.9 dargestellt. Hierbei umfasst ein MAF für jeden Kern eine Folge von Zeitfenstern. Während einem Kern zugewiesene Zeitfenster weiterhin nicht überlappend platziert werden dürfen, ist die gleichzeitige Ausführung von Zeitfenstern auf verschiedenen Kernen zulässig.

Dieser Scheduling-Mechanismus findet in verschiedenen Hypervisors Anwendung. Beispiele umfassen PikeOS [57, 58], XEN [59], XtratuM [60], Integrity-178B [61] und weitere [62].

2.4.4.1.2 Deferrable Server Schedulingmechanismen, die auf dem Prinzip des *Deferrable Server* [63] basieren, ermöglichen ein dynamischeres Scheduling und zielen auf die effizientere Bearbeitung sporadischer Ereignisse ab. Hierbei wird jeder vCPU ein Zeitbudget und eine Wiederherstellungsperiode zugewiesen. Das Budget wird während der Ausführung der vCPU auf einer physischen CPU kontinuierlich verbraucht. Kann die vCPU nicht ausgeführt werden, bleibt das verbleibende Budget erhalten. Ist das Budget aufgebraucht, kann die vCPU bis zum Ende der Wiederherstellungsperiode nicht weiter ausgeführt werden. Zu Beginn jeder Wiederherstellungsperiode wird das Budget der vCPU wieder auf den ursprünglichen Wert zurückgesetzt.

Auf diesem Konzept basierende Schedulingmechanismen werden bspw. vom Hypervisor XEN unterstützt [59, 64, 65].

2.4.4.1.3 Prioritätsbasiertes Scheduling Bei prioritätsbasiertem Scheduling wird die Schedulingentscheidung anhand der Priorität der auszuführenden Komponenten gefällt. Der Scheduler führt zu jedem Zeitpunkt die Komponente aus, welche die höchste Priorität hat und ausführbar ist. Sind mehrere Komponenten gleicher Priorität ausführbar, werden diese häufig nach dem Round-Robin-Verfahren ausgeführt. Prioritätsbasiertes Scheduling wird häufig auf Prozess- oder Threadebene angewandt.

Prioritätsbasiertes Scheduling wird bspw. im Hypervisor PikeOS als zweite Schedulingebene innerhalb eines Scheduling-Zeitfensters genutzt [57].

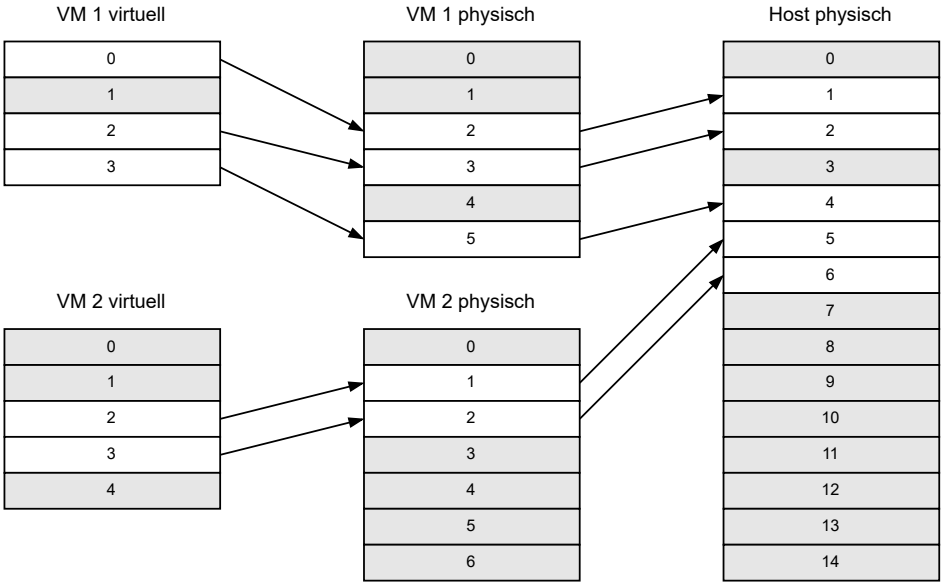


Abbildung 2.10: Zweistufige Abbildung virtueller Speicheradressen

2.4.4.1.4 Statische Ressourcenzuweisung Im Fall statischer Ressourcenzuweisung wird jeder virtuelle CPU-Kern auf einen dedizierten physischen Prozessor-Kern abgebildet. Somit werden alle VM ohne Unterbrechung ausgeführt. Statische Ressourcenzuweisung ist somit kein Schedulingverfahren im klassischen Sinn.

Hypervisors mit statischer Ressourcenzuweisung werden auch als *statisch partitionierende Hypervisors* bezeichnet. Ein Beispiel ist der Hypervisor Jailhouse [66].

2.4.4.2 Virtualisierung des Arbeitsspeichers

Die Bereitstellung des virtuellen Arbeitsspeichers einer VM erfolgt durch dessen Abbildung auf Speicherbereiche des Hostsystems. Hierbei wird der Adressbereich des virtuellen Speichers auf Speicheradressen des Host-Systems abgebildet. In Verbindung mit einer virtuellen Speicherverwaltung auf Betriebssystemebene ergeben sich somit zwei Adressübersetzungen (vgl. Abbildung 2.10). Plattformen mit Unterstützung für Speichervirtualisierung ermöglichen die Realisierung mehrstufiger Abbildungen durch die Memory Management Unit (MMU). Steht nur eine einstufige MMU zur Verfügung, kann *Shadow Paging* eingesetzt werden [67]. Hierbei berechnet der Hypervisor aus der Abbildungstabelle (engl. page table) des Gastbetriebssystems und dem Adressmapping auf

Hypervisorebene eine einstufige Abbildungstabelle, welche mittels der MMU umgesetzt wird. Weiterhin stellt er sicher, dass diese bei Änderungen durch das Gastbetriebssystem konsistent gehalten wird.

2.4.4.3 Virtualisierung von I/O-Komponenten

Die Virtualisierung von I/O-Komponenten umfasst verschiedene Aspekte, wie den Zugriff auf Register der I/O-Komponenten, die Verarbeitung von durch die Komponente ausgelöste Interrupts und die Behandlung direkter Speicherzugriffe (DMA) durch die Komponente. In [26] werden Ansätze zur Umsetzung von I/O-Virtualisierung aus Sicht der Absicherung eingebetteter Systeme beschrieben, welche im Folgenden zusammengefasst werden.

I/O-Komponenten können durch Emulation virtualisiert werden. Zugriffe der Gastsoftware auf die Komponenten werden hierbei durch den Hypervisor abgefangen und das Verhalten der Komponenten nachgebildet. I/O-Zugriffe können so zunächst durch den Hypervisor geprüft, ggf. angepasst und anschließend an die physische I/O-Komponente weitergeleitet werden. Der Emulationsansatz ermöglicht eine geteilte Nutzung der I/O-Komponente durch mehrere VMs und stellt gleichzeitig die Isolation der VMs sicher. Er ist jedoch mit vergleichsweise hohem Overhead verbunden, da jeder Zugriff auf die I/O-Komponente durch den Hypervisor behandelt werden muss. Weiterhin erfordert der Ansatz, dass auf Ebene des Hypervisors entsprechende Gerätetreiber mit Virtualisierungsfunktion zur Verfügung stehen.

Eine Alternative stellt das Durchschleifen von I/O-Komponenten in VMs dar (engl. *pass-through*). Eine VM erhält hierbei exklusiven, direkten Zugriff auf die I/O-Komponente. Hierdurch wird eine deutlich höhere Performance erreicht als im Fall der Emulation, die Nutzung der Komponente ist jedoch auf eine VM beschränkt. Auch im Hinblick auf die Isolation von VMs birgt das Durchschleifen Risiken, da die inkorrekte Nutzung von I/O-Komponenten die Systemstabilität beeinträchtigen kann. Dies betrifft insbesondere Komponenten, welche direkten Speicherzugriff (DMA) unterstützen, da für DMA-Speicherzugriffe typischerweise physische Adressierung genutzt wird. Gastsoftware mit direktem Zugriff auf eine solche Komponente kann somit durch Auslösen von DMA-Speicherzugriffen Speicherbereiche außerhalb ihrer VM lesen oder verändern. Um dies zu unterbinden, bieten moderne Plattformen Hardwareunterstützung in Form von IOMMUs (Input/Output Memory Management Unit, in ARM SoC auch System Memory Management Unit (SMMU) genannt). IOMMUs ermöglichen die Umsetzung virtueller Speicheradressierung für Speicherzugriffe durch Komponenten abseits der CPU (vgl. Abbildung 2.11). Analog zur MMU erfolgt hierbei eine Abbildung virtueller Speicheradressen auf physische Speicheradressen anhand konfigurierbarer Abbildungstabellen. Werden hierbei für eine I/O-Komponente in der IOMMU die gleichen Adressabbildungen kon-

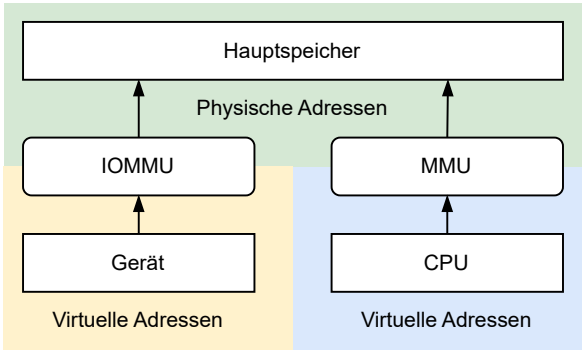


Abbildung 2.11: IOMMU und MMU zur Umsetzung virtueller Speicheradressierung (eigene Abb. nach [26])

figuriert wie für eine VM in der MMU, wird ein einheitliches virtuelles Speicherlayout für die VM und die I/O-Komponente erreicht. Dies hat zur Folge, dass Gastsoftware beim Zugriff auf die I/O-Komponente virtuelle Adressen nutzen kann. Gleichzeitig sind DMA-Transaktionen der I/O-Komponente auf die der VM zugewiesenen Speicherbereiche begrenzt.

Soweit vorhanden, kann auch weitergehende Hardwareunterstützung zur Virtualisierung von I/O-Komponenten genutzt werden. So ermöglicht Single-Root-I/O-Virtualization (SR-IOV), eine Erweiterung der Peripheral-Component-Interface-(PCI)-Express-Spezifikation, die Virtualisierung darüber angebundener Komponenten. Das Multiplexing der Zugriffe auf die Hardwarekomponente erfolgt hierbei auf Hardwareebene.

2.4.5 Betrachtete Hypervisors

In diesem Abschnitt werden die in dieser Arbeit genutzten Hypervisors vorgestellt.

2.4.5.1 Jailhouse

Jailhouse (vorgestellt in [66], Source Code verfügbar unter [68]) ist ein statisch partitionierender Hypervisor, der auf Einfachheit, geringe Hypervisoraktivität und geringen Codeumfang ausgelegt ist, um seine Analysierbarkeit und Zertifizierbarkeit zu erleichtern. Er ist auf den Einsatz in Echtzeit- und sicherheitskritischen Anwendungen ausgerichtet.

Der Hypervisor ermöglicht die statische Zuweisung von Systemressourcen wie CPU-Kernen, Speicherbereichen und I/O-Komponenten an Partitionen, welche als *cells* bezeichnet werden. Eine Überbuchung von Systemressourcen ist nicht möglich. Zugriffe

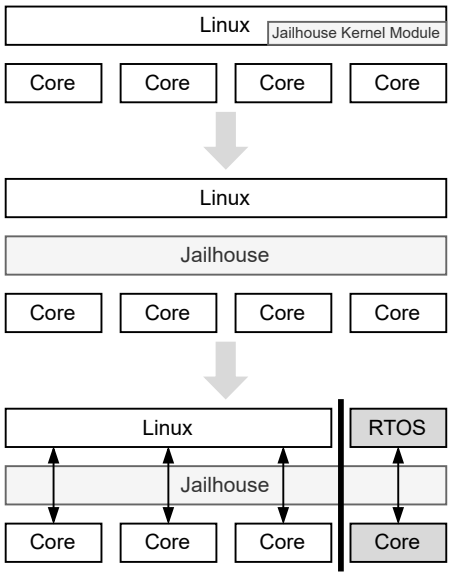


Abbildung 2.12: Initialisierung des Jailhouse-Hypervisors aus einem Linux-Betriebssystem (eigene Abb. nach [66])

der Gastsoftware auf Hardwarekomponenten erfolgen soweit möglich ohne Beteiligung des Hypervisors. Somit bietet er eine Grundlage zur Umsetzung eines partitionierten Asymmetric-Multiprocessing-(AMP)-Systems.

Jailhouse lässt sich nicht klar als Typ-1- oder Typ-2-Hypervisor klassifizieren. Er wird als Kernelmodul innerhalb eines Linux-Betriebssystems initialisiert und führt als solches eine Virtualisierungsschicht unterhalb des Linux-Betriebssystems ein (s. Abbildung 2.12). Das Betriebssystem selbst wird hierbei in eine Partition überführt, wodurch das Anlegen paralleler Partitionen ermöglicht wird. Ist die Virtualisierungsschicht etabliert, arbeitet diese unabhängig vom Betriebssystem. Somit zeigt der Hypervisor in der Startphase Eigenschaften eines Typ-2-Hypervisors, in der Betriebsphase läuft er als Typ-1-Hypervisor direkt auf der Hardware. Dieses Vorgehen erlaubt es Jailhouse, weite Teile der Hardwareinitialisierung in Linux durchzuführen, wodurch der Umfang hardwarespezifischer Treiber im Hypervisor stark reduziert werden kann.

Zur Inter-Partitions-Kommunikation und zur geteilten Nutzung von I/O-Komponenten ist es möglich, Adressbereiche mehreren Partitionen zuzuweisen. Weiterhin ist IPC über das im Rahmen des QEMU-Projekts [69] spezifizierte *ivshmem* [70] möglich, welches mittels eines virtuellen PCI-Geräts u.a. das Auslösen von Inter-Partitions-Interrupts ermöglicht.

2.4.5.2 PikeOS

PikeOS [71, 72] ist ein Typ-1-Hypervisor und Echtzeitbetriebssystem, das nach mehreren Safety-Standards in verschiedenen Anwendungsdomänen zertifiziert ist. Es basiert auf einem Separation-Kernel zur Umsetzung von der Partitionierung.

Die Partitionierung erfolgt auf Basis von *Resource Partitions*, welche logische Container für auf der Plattform ausgeführte Tasks darstellen. Tasks werden den Resource Partitions in der Konfiguration zugewiesen. Weiterhin wird der Zugriff von Resource Partitions auf Systemressourcen wie Speicher oder I/O-Komponenten konfiguriert.

Zeitliche Partitionierung erfolgt durch einen zweistufigen Scheduling-Ansatz, welcher die in den Abschnitten 2.4.4.1.1 und 2.4.4.1.3 beschriebenen Ansätze verbindet. Hierzu sind *Time Partitions* definiert, welchen jeweils mehrere Resource Partitions zugewiesen werden können. Die erste Stufe des Schedulings bildet ein statischer, zeitfensterbasierter Ansatz. Innerhalb eines MAF ist für jeden Prozessorkern eine Folge von Zeitfenstern definiert, welchen jeweils eine Time Partition zugeordnet sind, wobei eine Time Partition mehreren Zeitfenstern innerhalb eines MAF zugeordnet sein kann. Während eines Zeitfensters können alle Tasks auf dem Prozessorkern ausgeführt werden, die zu Resource Partitions gehören, welche der entsprechenden Time Partition zugewiesen sind. Die Nutzung von Time Partitions ermöglicht somit das Ausführen von Tasks mehrerer Resource Partitions innerhalb des gleichen Zeitfensters. Die zweite Stufe des Schedulings bildet ein prioritätsbasierter Ansatz. Hierfür wird jedem Task eine Priorität zugewiesen. Zur Laufzeit wird während eines Zeitfensters die darin ausführbaren Tasks nach Priorität sortiert ausgeführt. [58, 73]

2.4.5.3 Xen

Xen ist ein Typ-1-Hypervisor, welcher im Rahmen des Xen Projects [74] als Open-Source-Software entwickelt wird und in verschiedenen Domänen eingesetzt wird. Seine Anwendungsfelder umfassen insbesondere die Server- und Desktop-Virtualisierung, aber auch eingebettete Systeme und die Automobil- und Luftfahrtindustrie [74].

Der Hypervisor basiert auf einem Mikrokernel, Hardwaretreiber werden somit in VMs ausgeführt. Die von Xen bereitgestellten VMs werden auch als *Domänen* bezeichnet. Xen nutzt eine privilegierte VM *(dom0)*, in welcher zur Virtualisierung benötigte System-Services, Gerätetreiber und Software zur Steuerung des Hypervisors ausgeführt werden. Unprivilegierte Domänen zur Ausführung der Gastsoftware werden als $domU_n$ bezeichnet. Domänen können zur Laufzeit angelegt und entfernt werden. Ihnen können durch Konfiguration Systemressourcen wie virtuelle CPU-Kerne (vCPUs) und Speicher sowie Zugriff auf I/O-Komponenten zugewiesen werden. [75]

Auf x86-Prozessorarchitekturen unterstützt Xen verschiedene Virtualisierungsmodi, die von hardwaregestützter Vollvirtualisierung bis hin zu Paravirtualisierung auf Plattformen ohne Hardwareunterstützung reichen. Auf ARM-Plattformen wird ein Virtualisierungsmodus unterstützt, welcher hardwareunterstütze Virtualisierung mit Paravirtualisierung verbindet. [75]

Zum Scheduling der vCPUs auf physische CPU-Kernen unterstützt XEN verschiedene Scheduling-Ansätze [59]. Der standardmäßig genutzte *Credit*-Scheduler sowie der *Credit2*-Scheduler sind gewichtete Fair-Share-Scheduler. Auf Echtzeitanwendungen ausgerichtet sind die Scheduler *RTDS* [64, 65] und *ARINC653*, welche auf dem Prinzip des Deferrable Servers (vgl. Abschnitt 2.4.4.1.2) basieren bzw. Scheduling nach dem ARINC-653-Standard (vgl. Abschnitt 2.4.4.1.1) umsetzen. Durch Nutzung des *Null*-Schedulers wird jede vCPU statisch einem physischen CPU-Kern zugewiesen (vgl. Abschnitt 2.4.4.1.4). Hiermit können geringe Reaktionslatenzen erreicht und Overhead durch Scheduling vermieden werden. [76]

Auch der Zugriff auf I/O-Komponenten kann in XEN auf verschiedene Arten realisiert werden [75]:

- Bei *paravirtualisiertem I/O-Zugriff* nutzt die Gastsoftware einen zweigeteilten Treiber, um auf die I/O-Komponente zuzugreifen. Der Frontend-Treiber ist Teil der Gastsoftware und kommuniziert mit dem Backend-Treiber. Der Backend-Treiber liegt in einer separaten Domäne (der *Driver Domain*, z.B. dom0), welche Zugriff auf die physische Komponente hat. Der Backend-Treiber führt die Arbitrierung eingehender Zugriffe durch und steuert die I/O-Komponente an.
- Im Fall der *Emulation* nutzt die Gastsoftware einen nativen Gerätetreiber (vgl. Abschnitt 2.4.4.3). Die Emulation des Geräts erfolgt hierbei in dom0.
- Weiterhin wird Durchschleifen unterstützt (vgl. Abschnitt 2.4.4.3). Xen unterstützt hierbei die Nutzung der IOMMU.

3 Stand der Technik

Die Umsetzung von Mixed-Criticality-Systemen auf Multicore-Prozessoren und MPSoC ist Gegenstand intensiver Forschung. Die zentralen Herausforderungen ergeben sich hierbei aus widersprüchlichen Anforderungen der Systempartitionierung und der geteilten Nutzung von Ressourcen [77]. Einen ausführlichen Forschungsüberblick zu diesen Herausforderungen geben Alan Burns und Robert Davis in der regelmäßig aktualisierten Übersicht [77].

Im Folgenden liegt der Fokus auf den für die vorliegende Arbeit relevanten Teilbereichen der Forschung zu Mixed-Criticality-Systemen: der zeitlichen Partitionierung von COTS-Multicore-Systemen bzw. der Integration von Funktionen gemischter Echtzeitanforderungen auf MPSoC. Hierzu wird zunächst der Stand der Forschung zur zeitlichen Partitionierung von Multicore-Systemen betrachtet, wobei ein besonderer Fokus auf reaktive, Monitoring-basierte Ansätze gelegt wird. Anschließend wird ein Überblick über Forschung zu weiteren genutzten Technologien gegeben: der Anwendung von Trace-Systemen zum Online-Monitoring von Softwaresystemen sowie der Integration von FPGA-basierten Funktionen wie Hardwarebeschleunigern in Hypervisor-partitionierte Systeme.

3.1 Zeitliche Partitionierung in Multicore-Systemen

Eine zentrale Herausforderung bei der Umsetzung von Mixed-Criticality-Systemen auf MCP stellt die Sicherstellung von Echtzeiteigenschaften softwarebasierter Funktionen dar. Zwischen parallel ausgeführten Tasks auftretende Interferenzeffekte als Folge konkurrierender Nutzung von Rechenressourcen können die Softwareausführung signifikant verlangsamen und somit die Ausführungszeit zeitkritischer Funktionen verlängern [78]. Das Ausmaß der Verlangsamung ist hierbei sowohl vom Verhalten der parallel ausgeführten Tasks als auch von den Arbitrierungsmechanismen der Ausführungshardware abhängig.

Zur Erzeugung eines Schedules, welcher die vollständige und rechtzeitige Ausführung von Echtzeit-Tasks sicherstellt, müssen die maximalen Ausführungszeiten (WCET) dieser Tasks unter Berücksichtigung der Interferenz zwischen den MCP-Kernen bekannt sein. Die Berücksichtigung dieser Interferenzeffekte kann hierbei durch konservative Abschätzung der resultierenden Ausführungsverzögerung erfolgen. So kann bspw. für jeden Zugriff

auf eine geteilte Ressource die Ausführungszeit bei maximaler Interferenz angenommen werden (vgl. [79]), was jedoch in der Regel zu einer starken Überabschätzung der realen Ausführungszeit führt und so den durch den Einsatz mehrerer Kerne erhofften Performanzgewinn vereiteln kann. Alternativ können die Interferenzeffekte im Rahmen einer genauen Analyse der Softwareausführung im Ausführungskontext berücksichtigt werden. Dies erfordert detaillierte Informationen über die Ausführungsplattform, die parallel ausgeführte Software sowie die Systemumgebung, welche insbesondere bei der Nutzung von COTS-Hardware jedoch häufig nicht zur Verfügung stehen. [80]

Forschung zur zeitlichen Partitionierung von Multicore-Systemen zielt darauf ab, die Auswirkungen paralleler Softwareausführung auf das Zeitverhalten des einzelnen Tasks zu begrenzen. Dies wird als *zeitliche Isolation* (*Timing Isolation* bzw. *Temporal Isolation*) bezeichnet.

Ansätze zur zeitlichen Partitionierung von bzw. Isolation in Multicore-Systemen lassen sich hinsichtlich verschiedener Kriterien klassifizieren. In [81] und [82] werden verschiedene Interferenzkanäle von COTS MCP identifiziert und entsprechende Mitigationsansätze genannt. Die Ansätze sind hierbei nach der Interferenz verursachenden Hardwarekomponente kategorisiert und umfassen u.a. den Systembus, Speicher, Speicherbus und Speicher-Controller, Caches, (Co-)Prozessoren und Peripheriegeräte. Analog zu [83] lassen sich *proaktive* und *reaktive* Ansätze unterscheiden. Während proaktive Ansätze auf Interferenzvermeidung abzielen, lassen reaktive Ansätze Interferenzeffekte zu, stellen jedoch sicher, dass die Deadline von Echtzeit-Tasks eingehalten wird. Weiterhin lässt sich unterscheiden, wie der Zugriff auf Interferenz verursachende Komponenten begrenzt wird. Maiza et al. [84] unterscheiden hierbei mit Fokus auf die Speicherinfrastruktur phasenbasierte Ausführungsmodelle, Ansätze zur Regelung der Speicherbandbreite, statische Scheduling-Techniken und hardwareunterstützte Ansätze.

Im Folgenden werden bestehende Ansätze zur Interferenzreduktion und -beschränkung in Multicore-Systemen in Anlehnung an diese Klassifizierungen dargestellt. Es erfolgt zunächst ein Überblick über proaktive Ansätze zur Interferenzbeherrschung. Hierbei wird zwischen Ansätzen, welche auf der räumlichen Partitionierung der Interferenz verursachenden Ressourcen basieren (Abschnitt 3.1.1), und Ansätzen, welche die zeitlichen Abfolge und Frequenz von Zugriffen auf diese Ressourcen begrenzen (Abschnitt 3.1.2), unterschieden. Abschließend werden reaktive, Monitoring-basierte Ansätze vorgestellt (Abschnitt 3.1.3).

Da die vorliegende Arbeit auf den Einsatz von COTS MCP abzielt, werden Ansätze, welche auf der Nutzung neuartiger Hardwarekomponenten basieren, nicht betrachtet. Für einen weitergehenden Forschungsüberblick über die Themenfelder der Interferenzbeherrschung und Timing-Verifikation auf MCP sei auf [84] und [82] verwiesen.

3.1.1 Räumliche Partitionierung Interferenz verursachender Ressourcen

Ansätze zur räumlichen Partitionierung Interferenz verursachender Ressourcen basieren auf der statischen, exklusiven Zuweisung von (Teil-)Ressourcen zu Tasks bzw. CPU-Kernen. Sie setzen daher voraus, dass mehrere unabhängige Instanzen der (Teil-)Ressource zur Verfügung stehen. Die räumliche Partitionierung dieser Ressourcen kann insbesondere vorteilhaft sein, wenn der abwechselnde Zugriff auf die Ressource durch verschiedener Tasks bzw. Kerne zu signifikant höheren Zugriffslatenzen führt als mehrere sequenzielle Zugriffe durch einen einzelnen Task bzw. Kern.

3.1.1.1 Cache-Locking und Cache-Partitionierung

Geteilt genutzte Caches können aufgrund verschiedener Effekte zu starken Interferenzeffekten führen [81]. Ein zentraler Effekt ist gegenseitige die Verdrängung zwischengespeicherter Daten parallel ausgeführter Anwendung aus dem Cache, welche zu einer erhöhten Anzahl an Cache-Misses und somit deutlichen Verlangsamung der Anwendungsausführung führen können.

Zentrale Ansätze zur Beherrschung von Interferenz durch die geteilte Nutzung von Caches umfassen das Deaktivieren von Caches, Cache-Locking sowie Cache-Partitionierung.

Soweit von der Hardwareplattform unterstützt, können Caches deaktiviert werden, um Cache-bedingte Interferenzeffekte zu vermeiden [81]. Dies geschieht jedoch auf Kosten der Performance, da somit alle Speicherzugriffe aus langsameren nachgelagerten Speicherkomponenten, häufig dem RAM (Random Access Memory), bedient werden.

Cache-Locking ermöglicht im Fall konkurrierender Nutzung des Caches durch mehrere Anwendungen, mittels Software festzulegen, welche Daten im Cache zwischengespeichert werden, und somit ein unerwünschtes Verdrängen von Daten aus dem Cache zu unterbinden. In Caches, welche Cache-Locking unterstützen, können Cache-Lines als gesperrt markiert werden. Gesperrte Cache-Lines behalten die gespeicherten Daten bei, sie können nicht durch konkurrierende Zugriffe auf Daten aus anderen Adressbereichen verdrängt werden [85]. Mittels Cache-Locking kann die Vorhersagbarkeit der Ausführungszeit von Anwendungen auf MCP deutlich verbessert werden [86, 87]. Einen Überblick über die Forschung zu Cache-Locking geben [82, 85].

Als Cache-Partitionierung wird die Unterteilung des Caches in Partitionen und das (exklusive) Zuweisen dieser zu Tasks oder CPU-Kernen bezeichnet [85]. Dies dient dazu, Interferenz am Cache zu reduzieren, die Ausführungsperformance von Tasks zu erhöhen oder die Analysierbarkeit von Tasks hinsichtlich ihrer WCET zu verbessern [85], indem die gegenseitige Verdrängung von Daten im Cache durch verschiedene Tasks bzw. CPU-Kerne ausgeschlossen wird.

Cache-Partitionierungstechniken lassen sich hinsichtlich verschiedener Parameter unterscheiden [82, 88]. Während statische Partitionierungstechniken eine feste, zur Entwurfszeit bestimmte Zuweisung von Cache-Partitionen umsetzen, erfolgt die Zuweisung bei dynamischen Partitionierungstechniken zur Laufzeit [88]. Hierbei führen statische Ansätze zu besserer Vorhersagbarkeit des Systemverhaltens, können jedoch zu geringerer Auslastung des Caches führen und somit die Performanz beeinträchtigen [82]. Gleichzeitig lassen sich Ansätze hinsichtlich der Granularität der Partitionierung unterscheiden. Mittal et al. [88] unterscheiden für Set-Assoziative Caches zwischen Partitionierung auf Basis von Cache-Sets, Cache-Ways und Cache-Blöcken. Da Cache-Sets anhand der Speicheradresse gewählt werden, kann Set-basierte Cache-Partitionierung durch Software erreicht werden, während für Cache-Way-basierte Partitionierung Hardwareunterstützung benötigt wird[1].

Zur softwarebasierten Partitionierung von Caches wird meist Page Coloring [90] eingesetzt [85]. Page Coloring erfolgt als Teil der virtuellen Speicherverwaltung von Betriebssystemen, in welcher der Speicher in Form von Pages (Speicherseiten) organisiert ist. Zur Umsetzung der Partitionierung werden Pages verschiedener Anwendungen hierbei auf unterschiedliche Cache-Partitionen abgebildet. Dies erfolgt durch geeignete Wahl physischer Speicheradressen, auf welche die Pages im Rahmen der Adressübersetzung abgebildet werden (vgl. Abschnitt 2.4.4.2), wobei vorausgesetzt wird, dass es sich um einen physisch adressierten Cache handelt. Der Einsatz von Page Coloring in Multicore-Echtzeitsystemen wurde u.a. in [91, 92, 93, 94] betrachtet. Neben der Umsetzung auf Betriebssystemebene wird auch der Einsatz auf Hypervisorebene untersucht [95, 96, 97, 98, 99, 100, 101]. Einen weiteren Überblick über die Forschung zur Cache-Partitionierung geben [82, 85, 88].

Bei geteilter Nutzung von Speicherbereichen können darüber hinaus auch Cache-Kohärenzmechanismen zu Interferenzeffekten führen. Um dies zu vermeiden, können Cache-Kohärenzmechanismen auf manchen Hardwareplattformen deaktiviert werden [81] und, wenn benötigt, softwarebasierte Kohärenzmechanismen implementiert werden [102]. Alternativ ist die Nutzung anderer Methoden zur Inter-Prozessor-Kommunikation möglich, z.B. durch Message-Passing-Mechanismen, sofern solche von Betriebssystem oder Hardware bereitgestellt werden [81].

3.1.1.2 Partitionierung von Speicherkomponenten

Durch parallele Zugriffe von CPU- und DMA-Komponenten können auch geteilt genutzte Speicherkomponenten wie DRAM-Chips (Dynamic Random Access Memory) und Speichercontroller zu Interferenzeffekten führen [103].

Eine Interferenzquelle hierbei sind bspw. die Zeilenbuffer von DRAM-Bänken. Beim

[1] Beispielsweise unterstützen moderne Intel-Prozessoren mit Cache Allocation Technology (CAT) Hardware-Mechanismen zur Cache-Partitionierung [89], welche u.a. durch Cache-Way-Partitionierung umgesetzt sein können [35, 88].

Zugriff auf eine DRAM-Bank wird die benötigte Zeile in einen Buffer geladen, bevor sie gelesen oder geschrieben werden kann. Somit ist die Speicherzugriffszeit davon abhängig, ob die angefragte Adresse bereits im Buffer liegt. Wenn parallel ausgeführte Anwendungen in Speicherbereichen arbeiten, welche auf verschiedene Speicherzeilen der gleichen DRAM-Bank abgebildet sind, kann dies zu wechselseitigem Verdrängen der DRAM-Zeile aus dem Buffer führen. Dies wiederum führt zu höheren Speicherzugriffszeiten als bei exklusiver Ausführung. [81, 82]

Räumlich partitionierende Ansätze zur Interferenzreduktion an Speicherkomponenten umfassen daher Ansätze zur Partitionierung der Speicherhardware. Analog zu Cache-Partitionierung können auch die im System vorhandenen DRAM-Bänke partitioniert werden. Durch exklusive Zuweisung von CPU-Kernen zu DRAM-Bänken kann die Vorhersagbarkeit und Analysierbarkeit der Speicherzugriffszeiten verbessert werden. DRAM-Partitionierung wurde bspw. in [104, 105] anhand eines Linux-basierten Speicher-Allokators untersucht und in [100] in Kombination mit Cache-Partitionierung auf Hypervisorebene umgesetzt [82].

3.1.2 Zeitliche Partitionierung des Ressourcenzugriffs

Ansätze zur zeitlichen Partitionierung Interferenz verursachender Ressourcen basieren auf der Zuweisung von Zeitanteilen zur Ressourcennutzung an Tasks oder CPU-Kerne.

3.1.2.1 Phasenbasierte Softwareausführung

Phasenbasierte Ansätze zur zeitlichen Partitionierung teilen Tasks in Ausführungsphasen ein, welche sich hinsichtlich der Nutzung geteilter Ressourcen unterscheiden. Durch koordiniertes Scheduling der Phasenausführung auf allen Kernen des MCP können parallele Zugriffe auf geteilte Ressourcen und somit Interferenz vermieden werden. Das Scheduling kann hierbei entweder zur Laufzeit (online) erfolgen oder zur Entwurfszeit (offline) statisch festgelegt werden.

Das in [106, 107] vorgestellte AER-Ausführungsmodell teilt jede Taskausführung, bzw. jede Ausführungsiteration eines periodischen Tasks, in drei aufeinanderfolgende Phasen ein: *acquisition*, *execution* und *restitution*. Die Unterteilung erfolgt so, dass nur während der *acquisition*-Phase und der *restitution*-Phase Zugriffe auf geteilte Ressourcen erfolgen, während die Anwendungsausführung in der *execution*-Phase ausschließlich lokale Ressourcen des CPU-Kerns nutzt. In [108] wird dieses Konzept zur Vermeidung von Interferenz am geteilten Cache bei der Kommunikation zwischen Tasks auf verschiedenen Kernen eines Multicore-Prozessors genutzt. Dies wird durch einen statischen Schedule erreicht, in welchem maximal eine *acquisition*- oder *restitution*-Phase gleichzeitig ausgeführt wird.

Auch das Predictable Execution Model (PREM) [109] teilt die Taskausführung in Phasen ein, um Interferenzeffekte durch geteilte Nutzung der Speicherinfrastruktur zu vermeiden. Hierbei berücksichtigt es neben der CPU auch direkte Speicherzugriffe durch I/O-Komponenten. PREM unterscheidet zunächst zwischen *predictable intervals*, während welcher Kompatibilität mit dem Ausführungsmodell gefordert ist, und *compatible intervals*, während welcher Software nahezu uneingeschränkt ausgeführt werden kann. Innerhalb von *predictable intervals* ist die Taskausführung wiederum in Speicherzugriffsphasen und Berechnungsphasen unterteilt. Während der Speicherzugriffsphasen können Zugriffe auf geteilte Speicher erfolgen, um die zur Berechnung nötigen Daten in lokale Speicher zu laden. Während der Berechnungsphasen wird sichergestellt, dass keine Zugriffe auf geteilte Ressourcen der Speicherinfrastruktur erfolgen. Wird die parallele Ausführung von *compatible intervals*, Speicherzugriffsphasen oder Speicherzugriffen durch I/O-Komponenten verhindert, können Interferenzeffekte durch gleichzeitige Nutzung der Speicherinfrastruktur vermieden werden. Zur Umsetzung des phasenweisen Speicherzugriffs durch I/O-Komponenten nutzt PREM hierbei einen hardwareunterstützten Ansatz [110]. In [109] wird PREM im Kontext von Single-Core-Plattformen vorgestellt, Konzepte zur Anwendung von PREM in MCP-basierten Systemen folgen in [111, 112]. Die Umsetzung eines Schedulings von PREM-Tasks in hypervisorverwalteten Systemen wird in [101] vorgestellt.

Für einen weiteren Forschungsüberblick zu phasenbasierten Ausführungsmodellen bis 2018 sei auf [84, S. 56:10ff] verwiesen. Darüber hinaus gehende, neuere Beiträge betreffen u.a. prioritätsbasiertes Online-Scheduling von PREM-Tasks in Hypervisor-partitionierten MCP-Systemen [101] sowie präemptives (unterbrechendes) Scheduling von PREM/AER-Tasks [113, 114, 115, 116].

3.1.2.2 Begrenzung der Zugriffsrate auf geteilte Ressourcen

In diesem Abschnitt vorgestellte Ansätze begrenzen die Zugriffsrate von parallel ausgeführten Tasks auf geteilte Ressourcen. Sie verfolgen das Ziel, gleichmäßigere Zugriffslatenzen zu erreichen [117] oder die Anzahl der Zugriffe, welche durch Interferenzeffekte erhöhte Zugriffslatenzen aufweisen können, zu beschränken und somit engere WCET-Abschätzungen zu ermöglichen [79, 118]. Solche Ansätze werden insbesondere in Bezug auf Speicherzugriffe durch konkurrierende Recheneinheiten wie CPU-Kerne eingesetzt. In vielen Ansätzen sind Caches hierbei ausgenommen; für diese wird zumeist auf Cache-Partitionierungstechniken verwiesen (vgl. Abschnitt 3.1.1.1). Einen weiteren Überblick über die Forschung zur Interferenzbeherrschung durch Begrenzung der Zugriffsrate geben auch [82, Abs. III.A.2 und III.C.1] und [84, Abs. 3.2].

Die Begrenzung der Zugriffsrate auf die Speicherinfrastruktur wird auch als Drosselung

der Speicherbandbreite bezeichnet. Auf COTS-Systemen sind hierfür softwarebasierte Ansätze vorteilhaft, welche ohne Modifikation der Hardware genutzt werden können. Die Begrenzung der Speicherbandbreite erfolgt durch Festlegen eines Budgets an erlaubten Speicherzugriffen, welches zur Laufzeit durch ausgeführte Speicherzugriffe verbraucht und periodisch wieder aufgefüllt wird. Ist das Budget innerhalb einer Periode erschöpft, wird der zugehörige Task oder CPU-Kern deaktiviert, sodass keine weiteren Speicherzugriffe von ihm ausgehen.

Zur Überwachung der Anzahl durchgeführter Speicherzugriffe werden auf COTS-Systemen zumeist Hardware Performance Counter eingesetzt, welche Cache Misses des LLC erfassen [119]. Die Anzahl an LLC-Misses wird hierbei als Maß für die Anzahl durchgeführter Speicherzugriffe genutzt[2]. Das Abschalten von Tasks oder CPU-Kernen erfolgt zumeist durch eine Komponente des Betriebssystems oder Hypervisors. Als Alternative zum vollständigen Abschalten von Kernen wurde auch eine Reduktion der CPU-Taktfrequenz untersucht [121].

Ansätze zur Vermeidung von Interferenz an der Speicherinfrastruktur durch Drosselung der Speicherbandbreite lassen sich hinsichtlich verschiedener Eigenschaften klassifizieren. Zunächst lässt sich eine Unterscheidung hinsichtlich der Anzahl für kritische Tasks nutzbarer CPU-Kerne treffen. So werden kritische Tasks in einem frühen Ansatz von Yun et al. [122] nur auf einem CPU-Kern ausgeführt, während die Bandbreite der verbleibenden Kerne beschränkt wird. Im Gegensatz dazu erlauben spätere Ansätze die parallele Ausführung kritischer Prozesse auf mehreren Kernen. Memguard [117, 123] verfolgt hierbei den oben genannten Ansatz, die verfügbare Bandbreite der Speicherinfrastruktur so auf alle Kerne aufzuteilen, dass die mittlere Zugriffslatenz der eines Single-Core-Systems entspricht. Auch in [79, 118] wird die Bandbreite aller CPU-Kerne beschränkt, indem die Anzahl an Speicherzugriffen pro Ausführungsperiode jedes Tasks auf einen task-spezifischen, zur Entwurfszeit definierten Maximalwert begrenzt wird. Dies wird zur Bestimmung einer engeren WCET-Abschätzung (*interference sensitive WCET*, isWCET) für Anwendungen mit harten Echtzeitanforderungen genutzt.

Hinsichtlich der Zuweisung von Bandbreiten zu Tasks können statische und dynamische Ansätze unterschieden werden. Bei statischen Ansätzen werden offline bestimmte, feste Bandbreiten bzw. Speicherzugriffsbudgets zugewiesen (vgl. [79, 118]), während diese Budgets bei dynamischen Ansätzen zur Laufzeit bestimmt oder angepasst werden. So bietet Memguard [117, 123] neben der rein statischen Zuweisung von Budgets einen dynamischen Mechanismus, welcher die tatsächlich von Tasks benötigte Bandbreite

[2] Yun et al. [117] merken hierzu an, dass diese Performance Counters ggf. nicht alle Speicherzugriffe erfassen. Für die in [117] untersuchte Plattform sind bspw. Speicherzugriffe durch Prefetching oder LLC write-backs ausgenommen. Auch folgern Saeed et al. [120] aus Messungen der Auslastung von DRAM-Controllern und LLC-Misses, dass die reine Betrachtung von LLC-Misses kein optimales Maß zur Bestimmung der Auslastung von DRAM-Controllern und zur Vorhersage von Interferenz an der Speicherinfrastruktur ist.

zur Laufzeit überwachen und voraussichtlich nicht benötigte Bandbreite an andere Tasks abgeben kann. Die Autoren stellen hierbei klar, dass dieser Mechanismus zur Erfüllung harter Echtzeitanforderungen deaktiviert werden muss. Auch wird in [124] eine Erweiterung des Ansatzes [79, 118] vorgestellt, welche den Tasks auch nach dem Aufbrauchen ihres Speicherzugriffsbudgets weitere Zugriffe ermöglicht, sofern hierdurch die Einhaltung der bestimmten isWCETs nicht gefährdet wird. Dies ist möglich, sofern die bereits durchgeführten Speicherzugriffe zu geringeren Verzögerungen geführt haben als bei der WCET-Bestimmung angenommen.

Die bisher genannten Ansätze basieren entweder auf einer zur Entwurfszeit festgelegten Klassifizierung von Prozessen als zeitkritisch und unkritisch oder unterscheiden nicht hinsichtlich dieser Eigenschaft. Im Gegensatz dazu ermöglichen die in [121, 125] vorgestellten Ansätze, diese Klassifizierung von Prozessen zur Laufzeit zu ändern. Hierzu werden in [125] *Memory-Performance Critical Sections* eines Prozesses definiert, innerhalb welcher die Bandbreite paralleler unkritischer Tasks gedrosselt werden kann, um interferenzbedingte Ausführungsverzögerungen in diesen Abschnitten zu vermeiden. In [121] wird ein ähnlicher Ansatz verfolgt, wobei neben CPU-basierter Datenverarbeitung auch GPU-basierte Datenverarbeitung betrachtet wird. Die Drosselung der Speicherzugriffe erfolgt in [121] durch Reduzieren der Taktfrequenz von CPU-Kernen, während sie in den anderen Ansätzen durch das Pausieren der Taskausführung erfolgt.

Zuletzt lassen sich die Ansätze hinsichtlich ihrer Implementierungsebene im Software-Stack unterscheiden. Während viele Implementierungen auf dem Betriebssystem Linux aufbauen [121, 122, 125], bestehen auch Implementierungen in RTOS [79][3] und Hypervisors [126].

Die Anwendung eines auf Bandbreitenbegrenzung basierten Ansatzes zur Interferenzbeherrschung wird in [127] im Kontext eines Avionik-Use-Cases beschrieben. Die Bandbreitenbegrenzung erfolgt hierbei in Kombination mit einem Task-Scheduling nach ARINC 653 (vgl. Abschnitt 2.4.4.1.1), wobei individuelle Speicherzugriffsbudgets pro Zeitfenster festgelegt werden können. Die Bandbreitenbegrenzung erfolgte hierbei analog zu Memguard [123], die Methode zur WCET-Abschätzung folgt dem in [79] vorgestellten Ansatz.

Die bisher genannten Ansätze basieren auf der Überwachung von LLC-Misses zur Messung der Speicherauslastung. In einer neueren Arbeit nutzen Saeed et al. [120] die Fähigkeit neuerer Hardwareplattformen aus, die Auslastung von DRAM-Controllern direkt zu messen. Sie beobachten, dass die reine Betrachtung von LLC-Misses kein optimales Maß zur Bestimmung der Auslastung von DRAM-Controllern und zur Vorhersage von

[3] Der von Nowotsch et al. [79] vorgestellte Ansatz ist in PikeOS implementiert, welches sowohl Hypervisor- als auch RTOS-Funktionalität erfüllt (vgl. Abschnitt 2.4.5.2). In [79] wird es als RTOS klassifiziert.

Interferenz an der Speicherinfrastruktur ist. Sie schlagen daher einen dynamischen Regelungsansatz vor, welcher die Auslastung der DRAM-Controller überwacht und zur Regelung der Speicherbandbreitebudgets der CPU-Kerne nutzt.

Im Hinblick auf moderne MPSoC-Architekturen stellen Zuepke et al. [128] mit MemPol eine Architektur vor, in welcher die Bandbreitenbegrenzung von CPU-Kernen eines MCP durch einen separaten Prozessor des MPSoC erfolgt. Hierdurch wird zusätzlicher Overhead auf dem MCP vermieden. Die Überwachung der Speicherzugriffe erfolgt weiterhin über Hardware Performance Counter, zum temporären Abschalten von CPU-Kernen wird hier jedoch die Debug-Infrastruktur der Plattform genutzt.

Während verschiedene Ansätze zur hardwareunterstützten Beschränkung der Speicherbandbreite vorgeschlagen wurden, werden diese im Hinblick auf die Nutzung von COTS-Plattformen nicht vorgestellt. Für einen Überblick zu diesen Ansätzen sei auf Surveys wie [82, 84] verwiesen. Im Hinblick auf verbreitete Prozessorarchitekturen seien an dieser Stelle Architektur-Features wie Arm Memory System Resource Partitioning and Monitoring (MPAM) [129, 130] und Intel Resource-Direktor-Technik (RDT) [131, 132] genannt, welche zur Partitionierung neuerer Plattformen und insbesondere zur Beschränkung von Ressourcenzugriffen genutzt werden können.

3.1.3 Reaktive Monitoring-Ansätze

Als *reaktive Monitoring-Ansätze* zur Beherrschung von Interferenz in MCP-Systemen werden im Kontext dieser Arbeit Ansätze bezeichnet, welche die Softwareausführung auf Basis einer Überwachung des *Ausführungsfortschritts von Echtzeit-Tasks* zur Laufzeit steuern. Im Gegensatz zu Ansätzen, welche auf der reinen Begrenzung der Speicherzugriffsrate basieren (vgl. Abschnitt 3.1.2.2) und daher auf Basis pessimistischer Abschätzung der durch parallele Speicherzugriffe verursachten Verzögerung von Echtzeit-Tasks operieren, bietet die Überwachung des Ausführungsfortschritts von Echtzeit-Tasks die Möglichkeit, auf unterschiedliche Verzögerungen durch variierende Speicherzugriffsmuster parallel ausgeführter Tasks zu reagieren.

In [133, 134] stellen Kritikakou et al. einen solchen reaktiven Monitoring-Ansatz zur Umsetzung hart echtzeitfähiger Systeme auf MCP vor. Der Ansatz basiert auf der Überwachung des Ausführungsfortschritts an ausgewählten Beobachtungspunkten innerhalb des Kontrollflussgraphen des Echtzeit-Tasks (ET). Anhand von offline bestimmten WCET-Abschätzungen bei isolierter Ausführung des ET bzw. bei maximaler Interferenz wird an jedem Überwachungspunkt geprüft, ob das Einhalten der Deadline bei weiterer paralleler Ausführung unkritischer Tasks bis zum nächsten Überwachungspunkt sichergestellt ist. Wird diese Bedingung verletzt, werden unkritische, parallel ausgeführte Tasks bis zur Fertigstellung des ET unterbrochen.

Während in [134] die Umsetzung der Überwachung durch eine hardwarebasierte Monitoring-Komponente vorgeschlagen wird, wird in [135] ein rein softwarebasierter Ansatz vorgestellt und umgesetzt. Der Ansatz basiert auf der Instrumentierung von Echtzeit-Tasks mit Monitoring-Logik. Die Monitoring-Software wird an den jeweiligen Überwachungspunkten der Tasks eingebracht. Zur Laufzeit signalisiert sie einer zentralen Orchestrator-Komponente, wenn das Einhalten der Task-Deadline durch weitere parallele Ausführung gefährdet wird. Die Orchestrator-Komponente deaktiviert in diesem Fall die Ausführung unkritischer Tasks, bis die Ausführung des signalisierenden ET abgeschlossen ist. Der Ansatz wird in [136] erweitert: Um den Ressourcenbedarf des Monitorings zu reduzieren wird ein Ansatz vorgestellt, welcher die Ausführung der Monitoring-Software auf eine Untermenge der Überwachungspunkte beschränkt, welche zur Laufzeit bestimmt wird.

Im Gegensatz zur Überwachung des ET-Fortschritts an festen Überwachungspunkten schlagen Freitag et al. [137, 138, 139] einen auf *Fingerprinting* basierenden Regelungsansatz zur Interferenzbeherrschung vor. Durch Vergleich der zur Laufzeit gemessenen Signalkurven von Hardware Performance Countern mit zur Entwurfszeit bestimmten Signalverläufen kann der Fortschritt und die Ausführungsgeschwindigkeit des Echtzeit-Tasks bestimmt werden. Die Überwachung erfolgt hierbei durch eine externe Hardware-Komponente, welche die Performance Counter über die Debug- und Trace-Schnittstelle der MCP-Plattform ausliest. Zur Reduktion auftretender Interferenzeffekte werden parallele CPU-Kerne über die Debug-Schnittstelle in kurzen Zeitscheiben nach dem Prinzip der Pulsweitenmodulation pausiert oder durch Reduktion der Taktfrequenz verlangsamt.

Crespo et al. stellen in [140, 141] einen Ansatz zur Timing-Isolation auf hypervisorverwalteten MCP-Plattformen vor, welche einen Schedule mit festen, zyklischen Zeitfenstern umsetzen (vgl. Abschnitt 2.4.4.1.1). Hierzu wird sowohl der Ausführungsfortschritt des kritischen Tasks als auch das Verhalten der parallel ausgeführten unkritischen Tasks überwacht. Während der Ausführung kritischer Tasks wird die Ausführungsgeschwindigkeit durch Messung der verarbeiteten Instruktionen pro Zeit überwacht und parallel ausgeführte Partitionen bei Unterschreitung eines definierten Grenzwertes deaktiviert. Gleichzeitig wird die Speicherzugriffsrate unkritischer Partitionen überwacht und die Ausführung der Partition bei Überschreitung eines Grenzwerts bis zum nächsten Zeitfenster des statischen Schedules gestoppt.

In [142] wird ein probabilistischer Ansatz zur Beherrschung von interferenzbedingter Ausführungszeitverlängerung auf MCP vorgestellt. Hierzu werden sowohl Task-Ausführungszeiten als auch die diese bestimmenden Speicherzugriffszeiten als Zufallsvariablen aufgefasst. Der Ansatz zielt darauf ab, die Ausführung des Tasks mit einer definierten Wahrscheinlichkeit vor einer definierten Deadline abzuschließen. Dies wird durch eine Reglerkomponente erreicht, welche die Speicherzugriffslatenzen des Tasks zur Lauf-

zeit mittels Hardware Performance Countern erfasst und durch temporäres Abschalten interferierender Kerne beeinflusst.

3.2 Tracedatenbasiertes Online-Monitoring

Debug und Trace-Infrastruktur eingebetteter Hardwareplattformen erlauben die Überwachung und Beeinflussung der Softwareausführung auf Prozessoren. Sie werden zumeist im Rahmen der Systementwicklung genutzt, bspw. im Rahmen des Debuggings oder der Optimierung. Die Debug- und Trace-Schnittstelle eines eingebetteten Systems ist hierzu zumeist als externe Schnittstelle ausgeführt, welche über Trace-Hardware an das Entwicklungssystem angebunden werden kann.

Moderne (MP)SoC erlauben darüber hinaus den Zugriff auf die Debug- und Trace-Infrastruktur über interne Schnittstellen. Somit kann auf der Hardwareplattform ausgeführte Software sowohl die eigene Ausführung als auch die Ausführung von anderen Softwarekomponenten auf der Plattform überwachen.

Die Nutzung dieser Infrastruktur zur Laufzeit wurde bereits in verschiedenen Forschungsarbeiten betrachtet. Eine viel untersuchte Anwendung von Trace-Daten ist die Kontrollflussüberwachung [42, 43, 44, 143, 144, 145, 146, 147, 148]. Hierbei wird der Kontrollfluss einer ausgeführten Anwendung zur Laufzeit durch eine Monitoring-Komponente überwacht, um unzulässiges Programmverhalten festzustellen. Dies schließt bspw. unerwartete Sprünge im Programmablauf ein, welche durch Single-Event-Upsets oder Manipulation der Softwareausführung verursacht sein können. In [43, 145, 143, 146, 147] erfolgt die Verarbeitung der Trace-Informationen hierzu beispielsweise in einer Hardwarekomponente, welche auf einem im SoC integrierten FPGA instanziiert wurde. In [148, 149] erfolgt die Überwachung einer Anwendung innerhalb einer mittels eines Separation Kernels partitionierten Umgebung (vgl. Hypervisors in Abschnitt 2.4.2) durch eine Monitoring-Komponente, welche in einer zweiten Partition oberhalb des Kernels ausgeführt wird.

Weitere Anwendungen liegen im Bereich der Angriffssicherheit (Security). So untersuchten Wahab et al. [150, 151, 152] die Nutzung von Trace-Daten zur Umsetzung von Dynamic Information Flow Tracking, einem Ansatz zur Erkennung unerwünschter Datenflüsse in softwarebasierten Systemen. In [153] wurde Online Trace Monitoring zur Erkennung von Code Reuse Attacks untersucht, welche auf der Manipulation des Kontrollflusses eines Programms basieren. Code Reuse Attacks bilden auch die Motivation für weitere bereits genannte Arbeiten wie [148].

Im Bezug auf das Scheduling von Echtzeit-Tasks in MCS machen die in [128, 137, 138, 139] vorgestellten Ansätze von der Debug-Schnittstelle Gebrauch, um die Softwareausführung auf einzelnen CPU-Kernen zu deaktivieren und somit Einfluss auf die Stärke von Inter-

ferenzeffekten zwischen den Kernen zu nehmen. In [137, 138, 139] nutzen Freitag et al. die Trace-Infrastruktur weiterhin zur Ausleitung von Daten der Hardware Performance Counter.

3.3 Integration FPGA-basierter Funktionen in partitionierte Systeme

Die Einbindung FPGA-basierter Funktionen bzw. *Anwendungslogik* in Hypervisor-partitionierten Systemen zielt darauf ab, auch in partitionierten und virtualisierten Umgebungen von den Vorteilen rekonfigurierbarer Logik zu profitieren. Dementsprechend ist die Virtualisierung von FPGAs Gegenstand intensiver Forschung [154, 155, 156]. Während ein starker Forschungsfokus auf Rechenzentrums- und Cloud-Anwendungen liegt [157], gewinnt die FPGA-Integration auch im Umfeld partitionierter eingebetteter Systeme an Bedeutung, insbesondere im Hinblick auf den Trend zur Funktionsintegration auf leistungsfähigen, FPGA-unterstützten Hardwareplattformen [158].

Die verschiedenen Ansätze zur Virtualisierung von FPGA verfolgen nach [156] die folgenden grundsätzlichen Ziele:

- **Abstraktion**: Durch die Bereitstellung einer Abstraktion von der FPGA-Hardware soll ihre Nutzung vereinfacht und Schnittstellen vereinheitlicht werden.
- **Geteilte Nutzung (Mandantenfähigkeit)**: Das effiziente Teilen von FPGA-Ressourcen zwischen mehreren Nutzern bzw. Anwendungen soll ermöglicht werden.
- **Ressourcenmanagement**: FPGA-Ressourcen sollen bedarfsgerecht bereitgestellt werden. Dies umfasst u.a. Scheduling sowie die Überwachung der Ressourcenauslastung.
- **Isolation**: Gegenseitige Beeinflussung verschiedener auf dem FPGA umgesetzter Anwendungslogik soll vermieden werden, um trotz Nutzung geteilter Hardware die unabhängige Ausführung der Anwendungslogik zu ermöglichen.

Die vorliegende Arbeit behandelt die Partitionierung der Rechenressourcen eingebetteter, MPSoC-basierter Rechenplattformen. Hierbei stehen weniger die Abstraktion und Vereinheitlichung von FPGA-Schnittstellen im Fokus, sondern hohe Flexibilität hinsichtlich der im FPGA umsetzbaren Anwendungslogik und die zuverlässige Isolation dieser. Daher beschränkt sich der Forschungsüberblick im Folgenden auf die Ziele der geteilten Nutzung von FPGAs und der dabei notwendigen Isolation. Für einen weitergehenden Überblick, welcher auch die weiteren Ziele behandelt, wird auf [154, 155, 156, 158] verwiesen.

3.3.1 FPGA-Multiplexing

Die geteilte Nutzung eines FPGA durch mehrere Anwendungen wird auch als *FPGA-Multiplexing* bezeichnet. Grundsätzlich wird zwischen räumlichem und zeitlichem Multiplexing unterschieden.

Räumliches Multiplexing bezeichnet die parallele Umsetzung verschiedener Anwendungslogik auf dem FPGA, welche verschiedenen Nutzern bzw. Partitionen zugeordnet sind. Dies wird durch die Nutzung unterschiedlicher Ressourcen bzw. Bereiche der FPGA-Struktur ermöglicht. Die Umsetzung räumlichen Multiplexings kann durch ein *monolithisches FPGA-Design* oder durch ein *modulares FPGA-Design* erfolgen werden.

In einem monolithischen FPGA-Design werden die umzusetzenden Funktionen in einem einzelnen, zusammenhängenden Design implementiert, welches für den zu verwendenden FPGA synthetisiert wird. Die Anwendungslogik kann im Rahmen der Synthese frei im FPGA platziert werden, was die Umsetzung dieser unter effizienter Nutzung der verfügbaren FPGA-Ressourcen ermöglicht.

Im Fall des partitionierten Designs werden partiell rekonfigurierbare FPGA-Regionen (PRR) definiert, innerhalb welcher die Anwendungslogik platziert wird. Der hierdurch ermöglichte Austausch von der Anwendungslogik zur Laufzeit wird bspw. zur Umsetzung zeitlichen FPGA-Multiplexings genutzt. Die Nutzung von PRR führt jedoch in der Regel zu suboptimaler Auslastung der FPGA-Ressourcen, da PRR anhand der größten darin zu platzierenden Anwendungslogik dimensioniert werden müssen.

Zeitliches Multiplexing bezeichnet die sequenzielle Nutzung des FPGA durch verschiedene Nutzer bzw. Partitionen. Es kann sowohl auf Ebene der im FPGA implementierten Anwendungslogik umgesetzt werden als auch auf Ebene der FPGA-Ressourcen selbst.

Auf Anwendungsebene bezeichnet zeitliches Multiplexing die geteilte Nutzung im FPGA implementierter Anwendungslogik-Komponenten, z.B. von Beschleunigern, durch verschiedene Nutzer bzw. Partitionen. In der Regel erfordert dies eine Ressourcenverwaltungskomponente, welche den Zugriff auf die Anwendungslogik kontrolliert. Die Anwendungslogik kann hierbei sowohl als Teil eines monolithischen FPGA-Designs umgesetzt sein (vgl. [159]) als auch in einem modularen Design (vgl. [160, 161]).

Zeitliches Multiplexing auf FPGA-Ebene bezeichnet die sequenzielle Nutzung rekonfigurierbarer Regionen durch verschiedene Anwendungen bzw. Partitionen. Für die jeweilige Nutzungsdauer der PRR durch eine Anwendung bzw. Partition kann hierbei anwendungsspezifische Logik in die FPGA-Region geladen werden. Ein Hypervisor, welcher FPGA-Multiplexing mittels eines modularen FPGA-Designs sowohl auf Anwendungs- als auch auf FPGA-Ebene umsetzt, wird in [160, 161] vorgestellt.

Bei zeitlichem Multiplexing kann weiterhin zwischen präemptiven und kollaborativen

Ansätzen unterschieden werden. Bei präemptivem Multiplexing kann der Hypervisor den Partitionen den Zugang zu den FPGA-Ressourcen jederzeit entziehen, um diese Ressourcen anderen Partitionen zur Verfügung zu stellen. Hierbei muss der Zustand der Anwendungslogik in der Regel zunächst gespeichert werden, um die Verarbeitung zu einem späteren Zeitpunkt fortzusetzen. Bei kollaborativem Multiplexing verfügt eine Partition dagegen so lange über die Ressource, bis sie diese an den Hypervisor zurückgibt.

3.3.2 Isolation FPGA-basierter Funktionen

Soll FPGA-basierte Anwendungslogik in den Partitionskontext Hypervisor-verwalteter Systeme eingebunden werden, stellt der FPGA eine zusätzliche, zwischen Partitionen geteilt genutzte Ressource dar. Um die räumliche und zeitliche Partitionierung der Systempartitionen aufrechtzuerhalten, muss Isolation zwischen den Anwendungslogikkomponenten sichergestellt werden. Hierbei muss sowohl eine mögliche gegenseitige Beeinflussung von auf dem FPGA parallel oder sequenziell ausgeführter Anwendungslogik verschiedener Partitionen betrachtet werden als auch die Einflussnahme dieser Logik auf Komponenten außerhalb des FPGA, welche fremden Partitionen zugeordnet sind. Nach [156] lassen sich dabei *funktionale Isolation*, *Performance-Isolation* und *Fehlerisolation* unterscheiden.

3.3.2.1 Funktionale Isolation

Funktionale Isolation bezeichnet die Aspekte der unabhängigen Verwaltbarkeit von Partitionen und der Beschränkung von Partitionen hinsichtlich der erreichbaren Systemressourcen. FPGA-Bereiche und Anwendungslogik, welche unterschiedlichen Partitionen zugeordnet sind, sollen somit unabhängig voneinander (re-)konfiguriert werden können. Gleichzeitig muss sichergestellt sein, dass im FPGA implementierte Anwendungslogik nur auf die der zugehörigen Partition zugewiesenen Ressourcen zugreifen kann.

Die unabhängige Rekonfiguration von FPGA-Bereichen und FPGA-basierten Funktionen kann bei Multiplexing auf FPGA-Ebene durch den Einsatz dynamischer partieller Rekonfiguration erfolgen. Diese erlaubt die individuelle Rekonfiguration statisch definierter PRR, ohne den Ablauf der in weiteren FPGA-Bereichen implementierten Logik zu stören. Die PRR kann während der Rekonfiguration mittels Isolationskomponenten wie [162] von der umliegenden Logik entkoppelt werden, um Fehlverhalten durch Störsignale an den Schnittstellen während der Rekonfiguration zu vermeiden.

Hinsichtlich der Partitionierung von Systemressourcen muss insbesondere der Zugriff der FPGA-Anwendungslogik auf Speicher und I/O-Komponenten betrachtet werden. Dies ist vor allem relevant, wenn aus der Anwendungslogik heraus direkte Zugriffe auf diese Komponenten erfolgen können (Direct Memory Access (DMA)).

In Systemen ohne DMA erfolgt die Datenübertragung zur FPGA-basierten Anwendungslogik softwaregesteuert. Somit können nur Daten übertragen werden, welche durch die die Anwendungslogik nutzende Software selbst erreichbar sind (vgl. Abschnitt 2.4.4.2). Wird das System von einem Hypervisor partitioniert, sind diese durch die Partitionierung begrenzt.

Ist die Anwendungslogik DMA-fähig, kann sie eigenständig auf Speicher und Peripheriegeräte zugreifen. In partitionierten Systemen sind daher zusätzliche Maßnahmen nötig, Zugriffe auf Ressourcen außerhalb der Partitionsgrenzen zu verhindern. Für über PCI angebundene FPGAs kann hierzu Single-Root-I/O-Virtualization (SR-IOV) [163] in Verbindung mit der I/O-Virtualisierungslösung der Hardwareplattform genutzt werden. In [164] wird hierzu ein Ansatz vorgestellt, welcher dieses für eine große Anzahl von (virtuellen) FPGA-basierten Funktionen ermöglicht. In [165] werden durch FPGA-basierte Anwendungslogik ausgelöste Speicherzugriffe mit einer ID markiert, anhand welcher später eine Adressübersetzung auf die der Partition zugehörigen Speicherbereiche durchgeführt werden kann. In [159] wird die Integration von im FPGA eines MPSoC implementierten Funktionen auf Hypervisorebene betrachtet. Hierbei erfolgt die Adressübersetzung und Sicherstellung der Speicherisolation durch die IOMMU des MPSoC. Durch *page table slicing* werden hierzu Speicherzugriffe der Anwendungslogik verschiedener Partitionen auf disjunkte Adressbereiche abgebildet, um die Unterscheidung verschiedener Anwendungslogik-Komponenten durch die IOMMU zu ermöglichen[4]. Die IOMMU (bzw. SMMU) wird ebenfalls in der auf den Zynq UltraScale+ MPSoC ausgerichteten FPGA-Abstraktionsschicht ZUCL 2.0 [167] genutzt, welche die geteilte Nutzung des FPGA auf Betriebssystemebene verwaltet[5].

3.3.2.2 Performance-Isolation

Als Performance-Isolation (Leistungsisolation) wird die Unabhängigkeit der Leistung der im FPGA implementierten Anwendungslogik einer Partition vom Verhalten anderer Partitionen bezeichnet. Insbesondere soll ihr Zeitverhalten nicht durch fremde Partitionen beeinflusst werden können.

Im Gegensatz zur Softwareausführung auf der CPU erfolgt die Umsetzung verschiedener

[4] Durch *page table slicing* wird in [159] die auch in [166] genannte Einschränkung der betrachteten Hardware-Architektur umgangen, dass alle aus dem FPGA stammenden Zugriffe auf den Speicherbus aus Sicht der IOMMU (bzw. SMMU) von der gleichen Masterkomponente des Speicherbusses stammen. Ohne weitere Maßnahmen ist somit nur eine Übersetzungstabelle für alle aus dem FPGA stammenden Speicherzugriffe nutzbar.

[5] Während in ZUCL [168] die Verwaltung der FPGA-Abstraktionsschicht auf Basis eines Linux-Betriebssystems beschrieben ist, ist die Beschreibung der softwareseitigen Verwaltung in ZUCL 2.0 [167] generischer gehalten. Die Verwaltung erfolgt hierbei aus einem *User Space* und einem *Kernel Space*. Es wird daher davon ausgegangen, dass auch hier die Verwaltung auf Betriebssystemebene erfolgt.

Funktionen im FPGA durch Nutzung räumlich getrennter Ressourcen. Insbesondere bei der Nutzung unterschiedlicher PRR zur Umsetzung verschiedener Funktionen sind die zur Umsetzung einer Funktion genutzten Logikmodule von denen anderer Funktionen getrennt, wodurch eine gegenseitige Performance-Beeinflussung vermieden wird[6] [170]. Aus diesem Grund konzentriert sich Forschung zur Performance-Isolation zwischen FPGA-basierten Funktionen überwiegend auf geteilt genutzte Systemressourcen, welche außerhalb der PRR liegen.

Im Hinblick auf die von FPGA-basierten Funktionen geteilt genutzte Speicherarchitektur wurden verschiedene Konzepte vorgestellt. Diese basieren häufig auf im FPGA instanziierbaren Hardwaremodulen, welche den Zugriff der Anwendungslogik auf den Speicherbus kontrollieren. So beschränken die in [166, 171] vorgestellten Module die Bandbreite der Schnittstelle zum Speicherbus, um Mindestbandbreiten beim Speicherzugriff garantieren zu können. Weiterhin existieren Module, welche auf eine faire Aufteilung der verfügbaren Bandbreite unter konkurrierenden Anwendungslogik-Komponenten abzielen [166, 172]. Das in [166] vorgestellte *AXI Hyperconnect* (Advaned eXtensible Interface Hyperconnect) kombiniert beide Ansätze.

3.3.2.3 Fehlerisolation

Fehlerisolation zwischen FPGA-basierten Funktionen zielt nach [156] darauf ab, die Auswirkungen von Hardware- und Implementierungsfehlern auf die Ausführung von Anwendungslogik im FPGA zu reduzieren. Während in Cloud-Systemen mit mehreren Rechnern und FPGAs im Fall eines Hardwaredefekts eine Migration von Anwendungslogik auf andere Hardware möglich ist [173], steht im Kontext einzelner, FPGA-unterstützter SoC die Begrenzung der Auswirkung von Fehlverhalten einzelner Anwendungslogik-Komponenten im Fokus.

Um zu verhindern, dass solches Fehlverhalten zu Instabilität des Gesamtsystems führt, wurden Hardwaremodule entwickelt, welche an der Schnittstelle zwischen Anwendungslogik und dem umgebenden System im FPGA instanziiert werden können, bspw. an der PRR-Schnittstelle. Hierzu zählen die in Abschnitt 3.3.2.1 bereits vorgestellten Entkopplungsmodule, welche die PRR während der Rekonfiguration isolieren. Um Fehlverhalten der Anwendungslogik festzustellen, erlauben Module wie [174] die Überwachung der Kommunikation auf dem Speicherbus und melden Abweichungen vom genutzten Busprotokoll (AXI in [174]). Um solches Fehlverhalten zu kapseln, existieren darüber hinaus Module wie AXISAFETY [175], welche zwischen Anwendungslogik und Speicherbus

[6] Es sei angemerkt, dass sich auch räumlich getrennte Logik im FPGA gegenseitig beeinflussen kann. Dies kann z.B. durch thermische Überlastung des FPGA in Folge hoher Verlustleistung durch einzelne Funktionen oder durch starke Beanspruchung geteilter Spannungsversorgungen [169] erfolgen. Es ist davon auszugehen, dass Systemintegratoren sicherstellen müssen, dass Funktionen mit solchen Eigenschaften ausgeschlossen sind.

platziert werden und auch bei fehlerhaften Protokollimplementierungen in der Anwendungslogik sicherstellen, dass auf Speicherbusseite nur protokollkonforme Zugriffe erfolgen. In ähnlicher Weise können Module integriert werden, welche in Bezug auf das Speicherbusprotokoll zulässiges, aber meist unerwünschtes Verhalten filtern. So erkennt und verhindert der AXI Stall Monitor [176] unerwartet langes Blockieren des Systembus.

3.4 Einordnung und Abgrenzung der Arbeit

Die vorliegende Arbeit widmet sich dem Problem der Interferenzbeherrschung auf hypervisorverwalteten Multicore-Prozessoren in COTS MPSoC-Plattformen. Es werden Konzepte vorgestellt und untersucht, welche die Integration von Anwendungen mit unterschiedlichen Echtzeitanforderungen (Mixed-Real-Time) auf diesen Plattformen ermöglichen.

Ein Schwerpunkt liegt hierbei auf der effizienten Integration vormals verteilter Mixed-Real-Time-Software auf Hypervisor-partitionierten Multicore-Clustern solcher Plattformen. Hierzu wird ein Konzept vorgestellt, welches die Auswirkungen von Interferenzeffekten mittels eines tracedatenbasierten Monitoring-Ansatzes und der dynamischen Anpassung des Schedulings paralleler Softwareausführung auf Hypervisorebene begrenzt. Es ist somit den dynamischen Schedulingansätzen und den reaktiven Maßnahmen zur zeitlichen Partitionierung zuzuordnen.

Verwandte Forschungsarbeiten zur reaktiven, Monitoring-basierten Interferenzbeherrschung sind in Abschnitt 3.1.3 vorgestellt. Die vorliegende Arbeit grenzt sich hinsichtlich verschiedener Aspekte von diesen ab. Durch den Einsatz des dynamischen Schedulings auf Hypervisorebene und den Verzicht auf Instrumentierung der ausgeführten Software unterstützt der entwickelte Ansatz die Integration unveränderter Anwendungssoftware, was Zertifizierung oder Qualifizierungsprozesse vereinfachen kann. Im Gegensatz dazu erfordern bestehende, softwarebasierte Ansätze wie [135, 136] die Modifikation kritischer Anwendungen, um die Fortschrittsüberwachung zu realisieren. Des Weiteren benötigt ein großer Teil der vorgeschlagenen reaktiven Ansätze individuelle oder SoC-externe Hardware-Komponenten zur Umsetzung des Monitorings. So wird in [133] eine in Hardware ausgeführte Monitoring-Komponente genutzt. Auch der in [137, 138, 139] vorgestellte Ansatz nutzt einen SoC-externen Hardware-Monitor, bestehend aus einem FPGA mit Vorverarbeitungslogik und Soft-CPU. In [177] wird eine Architekturalternative des Ansatzes genannt, in welcher die Überwachung durch einen Kern des MCP selbst erfolgt, diese wird jedoch nicht weiter verfolgt. Die in der vorliegenden Arbeit betrachtete Umsetzung des Monitors in einem COTS MPSoC mit mehreren Prozessor-Clustern wurde nicht diskutiert. Weiterhin setzen die Ansätze [137, 138, 139, 177] mit Fingerprinting auf einen vergleichsweise daten- und rechenintensiven Ansatz zur Fortschrittserkennung von Echtzeit-Tasks, da dieser Ansatz die kontinuierliche, hochfrequente Überwachung von

Performance Countern und den Vergleich von Signalverläufen erfordert. Der in der vorliegenden Arbeit genutzte, auf Ausführungs-Trace-Daten basierende Ansatz generiert nur zu relevanten Zeitpunkten Trace-Daten, welche weitere Verarbeitung erfordern. Zudem liefern diese Trace-Daten direkt Informationen über die aktuell ausgeführte Instruktion, welche zur Bestimmung des Ausführungsfortschritt genutzt werden können. Andere Ansätze wie [140, 141] basieren auf der Überwachung der Ausführungsgeschwindigkeit des kritischen Tasks bzw. der Dauer der durch ihn verursachten Speicherzugriffe [142] anstatt des Ausführungsfortschritts. Prinzipbedingt können solche Ansätze nicht auf wechselnde, bspw. Input-abhängige Rechenaufwände kritischer Tasks reagieren. Es ist daher von einer reduzierten Effizienz des Schedulings im Vergleich zu Ausführungsfortschritt-basierten Ansätzen auszugehen.

Im Gegensatz zu diesen bestehenden Ansätzen wird in Kapitel 5 ein Konzept vorgestellt, welches die effiziente Integration von Mixed-Real-Time-Systemen auf Hypervisor-partitionierten MCP innerhalb eines COTS MPSoC ohne Modifikation der Tasks ermöglicht. Durch die Nutzung von Ausführungs-Trace-Daten ist ein ereignisorientiertes Online-Monitoring des Ausführungsfortschritts möglich, was effiziente Monitor-Implementierungen zulässt. Während die Debug- und Trace-Infrastruktur von MCP in früheren Arbeiten zur Ausleitung von Performance-Counter-Daten [128, 137, 138, 139] und zur Unterbrechung der Softwareausführung auf einzelnen CPU-Kernen [137, 138, 139] genutzt wird, ist die Nutzung von Ausführungs-Trace-Daten zur Umsetzung dynamischen Schedulings ein Alleinstellungsmerkmal der vorliegenden Arbeit.

Als Alternative zur Umsetzung der Echtzeitfunktionen auf dem MCP werden in Kapitel 6 Konzepte vorgestellt, welche die Umsetzung der Echtzeit-Interaktion mit der Umgebung im FPGA des Hypervisor-partitionierten MPSoC ermöglichen. Sie erweitern die vom Hypervisor umgesetzte Partitionierung der SoC-Ressourcen auf den FPGA. Somit erlauben sie die geteilte Nutzung des FPGA durch Anwendungen, welche verschiedenen Partitionen zugeordnet sind, sowie den modularen Austausch von Partitionssoftware samt zugehöriger FPGA-Anwendungslogik zur Laufzeit. Um die Umsetzung von Steuerungsaufgaben im FPGA zu unterstützen, ermöglichen sie den Zugriff auf Speicher und Peripheriekomponenten des SoC aus der Anwendungslogik heraus (DMA). Außerdem ermöglichen sie die direkte Anbindung verschiedener, bei Bedarf zur Laufzeit austauschbarer externer Komponenten, wie Sensoren, Aktoren und Kommunikationsschnittstellen, an den FPGA.

Der hierzu vorgestellte Ansatz zur FPGA-Partitionierung und Hypervisor-Integration setzt räumliches Multiplexing mittels einer Shell um. Im Gegensatz zu statischen Ansätzen wie [159] ermöglicht er den Austausch von FPGA-Anwendungslogik zur Laufzeit mittels DPR (zeitliches Multiplexing). Im Gegensatz zu DPR-basierten Ansätzen wie [160, 161] findet die Rekonfiguration jedoch nur beim Laden oder Entladen der Partition statt, um die im FPGA implementierte Echtzeitfunktionen nicht zu unterbrechen. Um trotz

DMA-fähiger Schnittstelle zur Speicherinfrastruktur des SoC funktionale Isolation sicherzustellen, nutzt das vorgestellte Konzept die IOMMU/SMMU der SoC-Plattform. Im Gegensatz zu Optimus [159], nach Angaben der Autoren der erste skalierbare Hypervisor zur FPGA-Virtualisierung mit DMA-Unterstützung, erfolgt dies jedoch ohne *page table slicing* (vgl. Abschnitt 3.3.2.1) und kann somit ohne Einschränkungen des nutzbaren virtuellen Adressbereichs erreicht werden.

Insbesondere im Hinblick auf die unterstützten Anwendungen grenzen sich die in dieser Arbeit vorgestellten Konzepte vom Stand der Technik ab. Während der Fokus der Forschung zur FPGA-Virtualisierung bzw. FPGA-Partitionierung auf MPSoC überwiegend auf der Anbindung von Beschleunigerfunktionen an CPU-basierte Software liegt ([159, 160, 161, 178, 179]), ermöglichen die vorgestellten Konzepte darüber hinaus die direkte Anbindung externer Komponenten an den FPGA. Zusätzlich unterstützen sie hierbei die Anbindung zur Laufzeit austauschbarer Komponenten über verschiedene Schnittstellen. Sie bieten somit die Möglichkeit, echtzeitkritische Steueraufgaben im Kontext hochflexibler eingebetteter Systeme auf dem FPGA umzusetzen und somit Interferenzkanäle mit weiteren SoC-Komponenten zu umgehen.

4 Interferenz an der Speicherinfrastruktur am Beispiel des *Smart Controller*

In diesem Kapitel werden die Auswirkungen von Interferenz an der Speicherinfrastruktur auf das Zeitverhalten der Softwareausführung auf Multicore-Prozessoren untersucht. Dies erfolgt am Beispiel einer Steuerungseinheit zur Nachrüstung von Industrie-4.0-Funktionen in Produktionsanlagen.

Die im Folgenden betrachtete Steuerungseinheit und ihre Evaluation basieren auf der in der gemeinsamen Veröffentlichung [BGS+19] vorgestellten *Smart-Controller*-Architektur. Ihre Evaluation basiert auf der in der eigenen Veröffentlichung [SBFB21] beschriebenen Evaluation.

4.1 Umfeld und Zielsetzung

Prozessüberwachung und *Predictive Maintenance* sind zentrale Ansätze zur Effizienzoptimierung industrieller Produktionssysteme. Als Prozessüberwachung (engl. *process monitoring* oder *condition monitoring*) wird die Überwachung von Fertigungsprozessen verstanden, auf Basis welcher eine direkte Beeinflussung des laufenden Produktionsvorgangs oder der Produktionsstrategie erfolgen kann [180]. Predictive Maintenance bezeichnet hingegen die regelmäßige Überwachung von Maschinen zur Identifikation optimaler Wartungszeitpunkte anhand des Maschinenzustands [181].

Beide Ansätze werden durch das Einbringen von Sensorik in Produktionsanlagen sowie deren Vernetzung ermöglicht. Aufgrund der langen Lebensdauer industrieller Maschinen erfolgt die Integration der Sensoren häufig im Rahmen einer Nachrüstung. Wenn die bestehende Steuerungsinfrastruktur an der Maschine nicht mit den benötigten Sensoren kompatibel ist, kann es hierbei notwendig sein, zusätzliche Steuerungs- bzw. Datenverarbeitungseinheiten ins Produktionssystem einzubringen, welche die benötigten Schnittstellen bereitstellen und die Verarbeitung und Weiterleitung der erfassten Daten umsetzen.

In [BGS+19] wird ein Konzept einer solchen Steuerungseinheit vorgestellt. Es dient zur Integration nachgerüsteter Sensoren in Produktionssysteme und zur maschinennahen Umsetzung von Prozessüberwachungs- und Predictive-Maintenance-Funktionen. Hierzu wird eine Hypervisor-basierte Softwarearchitektur vorgeschlagen, welche sowohl die Aus-

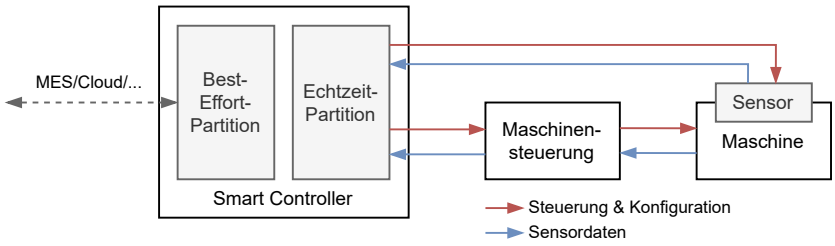


Abbildung 4.1: Integration eines Sensors mittels des *Smart Controllers* in ein Produktionssystem (eigene Darstellung nach [BGS+19])

führung von Prozessüberwachungsanwendungen mit Anforderungen hinsichtlich niedriger Reaktionslatenz als auch die Umsetzung latenzunkritischer Predictive-Maintenance-Anwendungen auf einer Plattform unterstützt.

Anhand der in [SBFB21] vorgestellten MCP-basierten Implementierung dieses Konzepts sollen im Folgenden die Auswirkungen von Interferenzeffekten auf die Reaktionslatenz zeitkritischer Software auf Hypervisor-partitionierten Multicore-Systemen exemplarisch untersucht werden.

4.2 Konzept und Umsetzung der Steuereinheit

Im Folgenden werden zentrale Eigenschaften der in [BGS+19] vorgestellten Smart-Controller-Architektur zusammengefasst. Anschließend wird ihre Umsetzung auf einer MCP-Plattform beschrieben, welche die Grundlage für die in Abschnitt 4.3 folgende Evaluation darstellt.

4.2.1 Die *Smart Controller*-Architektur

Der *Smart Controller* ist eine Steuerungseinheit, welche zur Integration nachzurüstender Sensoren in ein Produktionssystem eingebracht wird (vgl. Abbildung 4.1). Sie ist über eine geeignete Schnittstelle mit dem Sensor verbunden, über welche Sensordaten empfangen werden und die ggf. benötigte Konfiguration des Sensors erfolgt. Weiterhin ist die Einheit mit der Maschinensteuerung verbunden, um Zugriff auf Statusinformationen der Steuerung zu erhalten und im Rahmen der Prozessüberwachung Einfluss auf den Ablauf des Produktionsvorgangs zu nehmen. Zudem ermöglicht diese Verbindung den Zugriff auf Daten von an die Steuerung angebundenen Sensoren. Zuletzt ist die Steuereinheit mit übergeordneten Komponenten wie Manufacturing Execution Systems (MES) oder cloudbasierten Diensten verbunden, an welche die erfassten Sensordaten weitergeleitet

werden und von welchen Befehle und Konfigurationen empfangen werden können.

Grundlage des Smart Controllers bildet eine MCP-Hardwareplattform, welche mittels eines Hypervisors in eine Echtzeit- und eine Best-Effort-Partition unterteilt wird. In der Echtzeit-Partition werden Anwendungen implementiert, welche mit niedriger Latenz auf eingehende Sensordaten reagieren müssen. Dies trifft bspw. auf Funktionen zur Prozessüberwachung zu. Um die Umsetzung solcher Anwendungen zu unterstützen, bildet ein RTOS die Grundlage dieser Partition.

In der Best-Effort-Partition wird dagegen ein General-Purpose-Betriebssystem (General-Purpose Operating System, GPOS) wie bspw. Linux eingesetzt, auf welchem Anwendungen ohne Echtzeitanforderungen ausgeführt werden. Dies umfasst bspw. Anwendungen, welche Sensordaten zwecks langfristiger Analysen vorverarbeiten und speichern, wie sie häufig im Kontext von Predictive-Maintenance-Ansätzen zu finden sind. Der Einsatz eines GPOS vereinfacht hierbei die Entwicklung und Verwaltung der dort ausgeführten Anwendungen, indem er die Nutzung eines breiten Ökosystems an Software und Diensten ermöglicht.

Wie in Abbildung 4.1 dargestellt, erfolgt der Zugriff auf die Schnittstellen zum Sensor und zur Maschinensteuerung aus der Echtzeit-Partition heraus, um die Reaktionslatenz latenzkritischer Anwendungen zu minimieren. Die Anbindung an übergeordnete Systeme erfolgt über die Best-Effort-Partition, um hierbei von den Funktionen des GPOS profitieren zu können.

Um die Reaktionslatenz der Steuereinheit und den Ressourcenbedarf der Partitionierung gering zu halten, wird ein statisch partitionierender Hypervisor eingesetzt. Dieser stellt auch einen IPC-Mechanismus bereit, über welchen die in der Echtzeit-Partition erfassten Daten an die Best-Effort-Partition weitergeleitet werden können.

4.2.2 Umsetzung auf Basis eines Einplatinenrechners

Die in Abschnitt 4.2.1 beschriebene Architektur wurde auf Basis des Einplatinenrechners Lemaker Banana Pi M1 [182] implementiert. In [SBFB21] ist ihre Nutzung zur Umsetzung einer Überlasterkennung anhand eines per Controller Area Network (CAN) angebundenen Sensors an der Hauptspindel einer Werkzeugmaschine beschrieben. Im Folgenden liegt der Fokus der Beschreibung auf der im Hinblick auf die Evaluation relevanten Implementierung des Software-Stacks.

Der gewählte Einplatinenrechner basiert auf einem Allwinner-A20-SoC. Der Rechner umfasst einen Dual-Core ARM Cortex-A7 Prozessor, 1 GB DDR3 RAM und u.a. eine zur Anbindung des Sensors nutzbare CAN-Schnittstelle. Der implementierte Software-Stack ist in Abbildung 4.2 dargestellt. Die Rechenplattform wird mittels des Jailhouse-

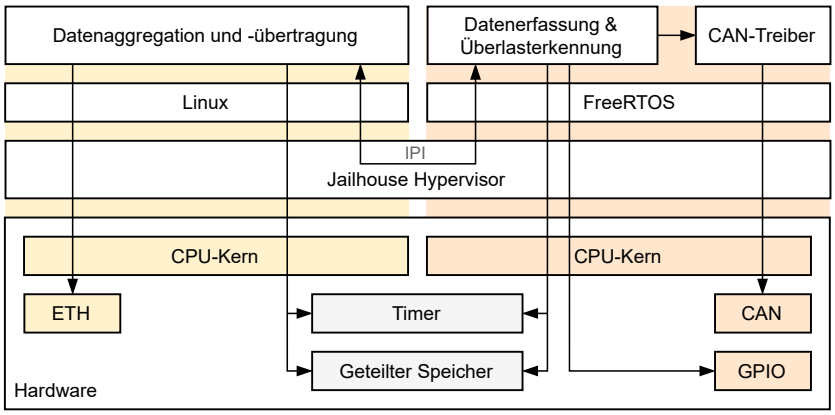


Abbildung 4.2: Umsetzung der *Smart Controller*-Architektur zur Integration eines Sensors über die CAN-Schnittstelle. Die Partitionierung ist farblich angedeutet, geteilt genutzte Komponenten grau dargestellt (eigene Darstellung in Anlehnung an [SBFB21]).

Hypervisors (vgl. Abschnitt 2.4.5.1) partitioniert, wobei jeder Partition statisch ein CPU-Kern, Speicher sowie benötigte Peripheriekomponenten zugewiesen sind.

In der Echtzeit-Partition wird das Echtzeitbetriebssystem FreeRTOS [183] ausgeführt, innerhalb dessen Tasks zur Sensordatenerfassung und zur Umsetzung der Überlasterkennung implementiert sind. Zur Interaktion mit dem Sensor umfassen diese die benötigten Geträtetreiber für das CAN-Peripheral. Weiterhin besteht Zugriff auf die General-Purpose Input/Output (GPIO)-Komponente, über welche digitale Eingangs- und Ausgangssignale gelesen bzw. gesetzt werden können. Im Kontext der Überlasterkennung erfolgt hierüber die Ansteuerung einer Signalleitung zur Maschinensteuerung, über welche der Überlastzustand gemeldet wird. In der Best-Effort-Partition wird ein Linux OS (Betriebssystem, engl. operating system) eingesetzt, innerhalb dessen Prozesse zur Datenaggregation und Weiterleitung der erfassten Daten über die Ethernet-Schnittstelle an ein übergeordnetes Edge-Gerät ausgeführt werden.

Die Übertragung der empfangenen Sensordaten zwischen Echtzeit- und Best-Effort-Partition erfolgt über das RPMsg-Protokoll. Dieses ermöglicht die Umsetzung der Kommunikation zwischen den Partitionen über einen geteilten Speicherbereich und eine Hypervisorschnittstelle, welche das Auslösen von Inter-Partitions-Interrupts (IPI) an die jeweils andere Partition ermöglicht. Ein von beiden Partitionen erreichbarer Hardwaretimer wird zur Zeitsynchronisierung zwischen den Partitionen genutzt.

4.3 Evaluation der Auswirkungen von Interferenzeffekten

Im Folgenden werden die Auswirkungen von Interferenzeffekten auf die Softwareausführung auf COTS-Multicore-Plattformen betrachtet. Es wird untersucht, in welchem Umfang sich das Auftreten von Interferenz auf die Reaktionszeit latenzkritischer Anwendungen in MCP-basierten eingebetteten Systemen auswirkt. Weiterhin wird untersucht, welchen Einfluss verschiedene Basissoftwarearchitekturen und Implementierungsalternativen der Anwendungen auf die Interferenzabhängigkeit der Reaktionslatenz haben.

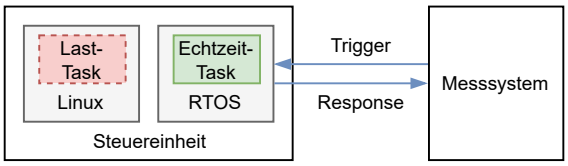
Die Untersuchung erfolgt exemplarisch anhand der in Abschnitt 4.2.2 vorgestellten Plattform. Um die durch die Basissoftware und Hardwareplattform verursachten Reaktionslatenzen zu bestimmen, wird eine minimale Anwendung zur Evaluation genutzt, welche auf ein eingehendes digitales Signal schnellstmöglich mit einem Antwortsignal reagiert. Für eine Evaluation der mit der Plattform erreichbaren Reaktionslatenzen komplexerer Anwendungen, wie einer CAN-basierten Überlasterkennung, sei auf [SBFB21] verwiesen.

4.3.1 Evaluationsaufbau

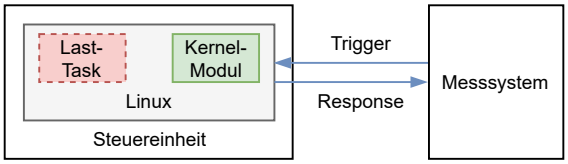
Wie eingehend beschrieben, wird zur Evaluation der Reaktionslatenz eine minimale Anwendung (im Folgenden als *Evaluationsanwendung* bezeichnet) eingesetzt, welche auf ein eingehendes digitales Signal schnellstmöglich mit einem Antwortsignal reagiert. Das Eingangssignal wird im Folgenden als *Trigger*-Signal bezeichnet. Ein Pegelwechsel des Signals führt zu einem Interrupt, welcher von der Evaluationsanwendung verarbeitet wird. Als Reaktion auf den festgestellten Pegelwechsel des Eingangssignals setzt die Anwendung ein Antwortsignal *(Response)*.

Die Evaluationsanwendung wurde auf der Hardwareplattform in mehreren Implementierungsalternativen umgesetzt. Diese stellen verschiedene Möglichkeiten zur Umsetzung von Anwendungen dar, welche interruptbasiert auf eingehende Sensordaten reagieren. Im Folgenden sind die betrachteten Implementierungen nach der genutzten Basissoftwarearchitektur gruppiert beschrieben. Abbildung 4.3 stellt die entsprechenden Messaufbauten dar.

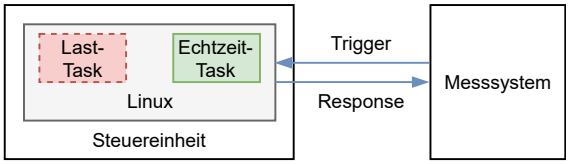
1. **RTOS-basierte Umsetzung im Hypervisor-partitionierten System:** Die Evaluationsanwendung wird auf Basis der in Abschnitt 4.2 beschriebenen Architektur umgesetzt. Die Anwendung wird im RTOS der Echtzeit-Partition implementiert, welche auf einem CPU-Kern ausgeführt wird und Zugriff auf die GPIO-Komponente hat. Auf dem zweiten CPU-Kern wird parallel ein Linux OS ausgeführt, innerhalb dessen eine Lastanwendung ausgeführt werden kann. Hinsichtlich der Implementierung der Evaluationsanwendung innerhalb des RTOS werden zwei Alternativen untersucht:



(a) RTOS-basierte Umsetzung im partitionierten Aufbau



(b) Umsetzung im Linux-Kernel



(c) Umsetzung als Linux-Anwendung

Abbildung 4.3: Messaufbau für betrachtete Implementierungsalternativen der Evaluationsanwendung

a) Das Response-Signal wird direkt in einem vom RTOS aufgerufenen Interrupt-Handler gesetzt. Die Verarbeitung des Eingangssignals erfolgt somit im Interruptkontext.

b) Das Response-Signal wird von einem hoch priorisierten Task gesetzt, welcher vom Betriebssystem als Reaktion auf den Interrupt aktiviert wird (vgl. Abbildung 4.3a). Die Verarbeitung des Signals erfolgt somit außerhalb des Interruptkontexts.

2. **Linux-basierte Umsetzung**: Auf der Hardwareplattform wird ein Linux OS ausgeführt, welches alle Plattformressourcen verwaltet. Es wird kein Hypervisor eingesetzt. Hinsichtlich der Implementierung der Evaluationsanwendung innerhalb des OS werden zwei Alternativen untersucht:

   a) Die Anwendung wird als Kernel-Modul implementiert, welches einen Interrupt-Handler registriert (vgl. Abbildung 4.3b). In diesem wird das Response-Signal gesetzt. Die Verarbeitung des Signals erfolgt somit im Interruptkontext im Kernel Mode.

b) Die Anwendung wird als Kernel-Modul implementiert, welches einen threadbasierten Interrupt-Handler registriert (vgl. Abbildung 4.3b). In diesem wird das Response-Signal gesetzt. Die Verarbeitung des Signals erfolgt somit zwar ebenfalls im Kernel Mode, jedoch außerhalb des Interruptkontexts.

c) Die Anwendung wird als Userspace-Anwendung implementiert, welche unter Nutzung der *libgpio*-Bibliothek auf den Pegelwechsel des Request-Signals reagiert und das Response-Signal setzt (vgl. Abbildung 4.3c). Die Verarbeitung des Signals erfolgt somit vollständig im User Mode.

3. **Echtzeit-Linux-basierte Umsetzung**: Die Umsetzung der Anwendung erfolgt analog zur Linux-basierten Umsetzung. Im Gegensatz zu den zuvor genannten Implementierungen wird ein Kernel genutzt, welcher mittels der PREEMPT_RT Patches [184] auf Echtzeitfähigkeit optimiert wurde. Da diese Patches u.a. auf die Minimierung der Codeausführung im Interruptkontext abzielen, erfolgte keine Umsetzung analog zu Implementierung 2a. Somit werden folgende Alternativen betrachtet:

   a) Analog zu Implementierung 2b wird die Anwendung als Kernel-Modul implementiert, welches zur Umsetzung der Funktion einen threadbasierten Interrupt-Handler nutzt (vgl. Abbildung 4.3b).

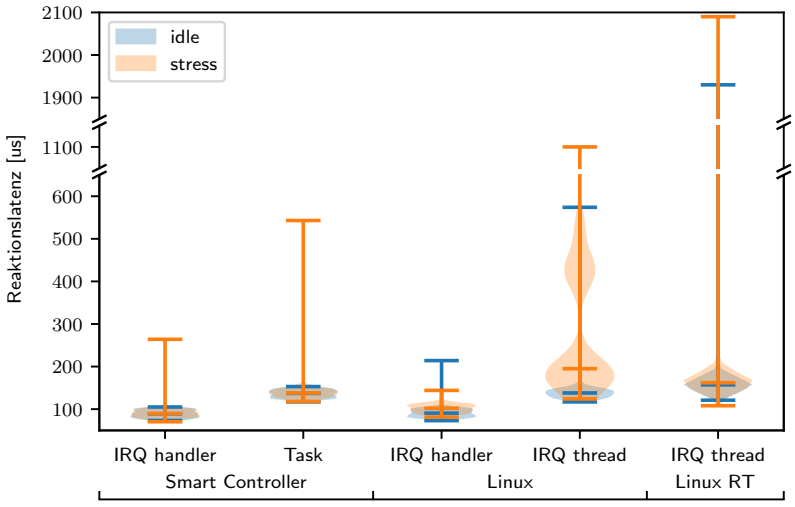
   b) Analog zu Implementierung 2c wird die Anwendung im Userspace unter Nutzung der *libgpio*-Bibliothek implementiert (vgl. Abbildung 4.3c). Die Anwendung wird mit hoher Priorität ausgeführt.

Zur Evaluation von Interferenzeffekten wird die Lastanwendung `stress` [185] im Userspace des Linux OS genutzt. Sie erzeugt Last auf CPU-Kernen, Speicher-Komponenten sowie I/O-Last durch wiederholte arithmetische Berechnungen, Speicherzugriffe und zyklische Aufrufe von `sync()`.

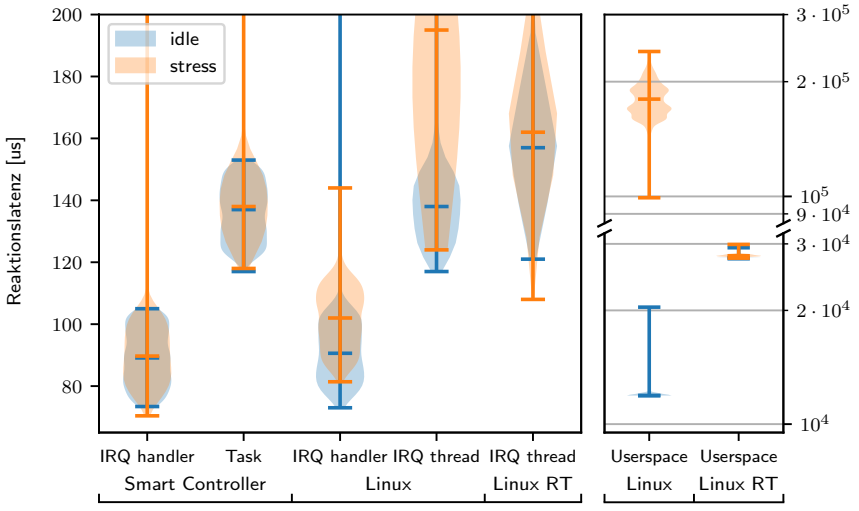
Die Messung erfolgt durch ein externes Messsystem, welches das Trigger-Signal setzt und die Zeit bis zum Empfang des Response-Signals erfasst. Die Messfunktion ist in einem FPGA als Logikschaltung implementiert, welche mit einer Taktfrequenz von 500 MHz eine theoretische Auflösung von 2 ns erreicht. Störeinflüsse durch die Signallaufzeiten zum Einplatinenrechner werden gegenüber den im Rahmen der Messung beobachteten Reaktionslatenzen als vernachlässigbar angesehen, insbesondere da für die durchgeführte Evaluation nur die Differenz gemessener Signallaufzeiten relevant ist.

4.3.2 Reaktionszeit in Abhängigkeit der Systemlast

Unter Nutzung des in Abschnitt 4.3.1 beschriebenen Messaufbaus wurden 1000 Messungen je Konfiguration durchgeführt. Die Verteilung der hierbei beobachteten Reaktionslatenzen



(a) Kernelspace- und RTOS-Implementierungen



(b) Detailansicht der Kernelspace- sowie RTOS-Implementierungen und Userspace-Implementierungen

Abbildung 4.4: Verteilung der gemessenen Reaktionslatenzen verschiedener Implementierungsalternativen im belasteten (stress) und unbelasteten (idle) Fall.

ist in Abbildung 4.4 für die verschiedenen Implementierungen gegenübergestellt. Zu jeder Verteilung sind Median, Minimum und Maximum in Form horizontaler Markierungen dargestellt, die anhand der Messwerte geschätzte Verteilung der Reaktionslatenzen ist durch farbige Flächen visualisiert (vgl. [186]). In Abbildung 4.4 sind auf der in Abschnitt 4.2 vorgestellten Architektur aufbauende Implementierungen mit *Smart Controller* bezeichnet. Linux-Implementierungen unter Nutzung des nativen Kernels sind mit *Linux* bezeichnet, während Implementierungen unter Nutzung der PREEMPT_RT-Patches mit *Linux RT* bezeichnet sind. Die Implementierungen 1a und 2a sind hierbei als *IRQ handler*, Implementierungen 1b, 2b und 3a als (RTOS-) *Task* bzw. *IRQ thread* und Implementierungen 2c und 3b als *Userspace* bezeichnet.

4.3.2.1 Reaktionslatenzen im unbelasteten Fall

Zunächst ist erkennbar, dass sich eine Abstufung der gemessenen Reaktionslatenzen ergibt. Die erste Stufe bilden die IRQ-Handler-basierten Implementierungen. Sie führen zu den niedrigsten mittleren Reaktionslatenzen. Im unbelasteten Fall liegt der Median der gemessenen Latenzen dieser Implementierungen im Bereich von 89 µs bis 91 µs. Die nächste Stufe bilden Implementierungen, welche die Verarbeitung des eingehenden Signals in einem RTOS-Task bzw. Kernel-Thread umsetzen, welche zu etwas höheren mittleren Latenzen führen. Ihr Median liegt im unbelasteten Fall bei 137 µs bis 157 µs. Zuletzt führen Userspace-Implementierungen im Vergleich mit den RTOS- bzw. Kernel-basierten Implementierungen zu vielfach höheren Reaktionslatenzen. Diese liegen im unbelasteten Fall im Bereich weniger zehn Millisekunden.

Der Anstieg der Reaktionslatenzen zwischen den Stufen ist u.a. durch zusätzlich benötigte Kontextwechsel sowie die Möglichkeit von Unterbrechungen durch weitere Interrupts oder priorisierte Threads nachvollziehbar. Der Einfluss von Last auf die Verteilung der Reaktionslatenzen ist stark implementierungsabhängig und wird im Folgenden detaillierter betrachtet.

4.3.2.2 Auswirkungen paralleler Last auf die mittlere Reaktionslatenz

Wird das System parallel zur Evaluationsanwendung mittels der Lastanwendung ausgelastet, steigen die mittleren Reaktionszeiten aller Implementierungen an. Der Umfang des Anstiegs hängt stark von der jeweiligen Implementierung ab. Während auf dem nativen Linux-Kernel basierende Kernel-Implementierungen eine Zunahme der mittleren Reaktionszeiten von 12 % bzw. 41 % zeigen, erhöht sich die Reaktionszeit der Userspace-Anwendung um ein Vielfaches (1400 %). Für Implementierungen, die auf dem RT-Kernel basieren, zeigen sich deutlich geringere Auswirkungen: In der kernelbasierten Umsetzung steigt die mittlere Reaktionszeit lediglich um 3 %, in der User-Space-Implementierung nur

um 1 %. Ähnliche Ergebnisse werden im Hypervisor-partitionierten System erreicht: Die mittlere Reaktionszeit nimmt hier für beide Implementierungsalternativen um 0.7 % zu.

4.3.2.3 Auswirkungen paralleler Last auf die maximale Reaktionslatenz

Im Hinblick auf die im Fall harter Echtzeitanforderungen relevante maximale Reaktionslatenz zeigt sich ebenfalls eine Lastabhängigkeit: Sowohl für die Hypervisor-partitionierten Implementierungen als auch die Linux-RT-Implementierungen ist ein Anstieg der maximalen Reaktionszeiten unter Last erkennbar. Für erstere liegt dieser Anstieg bei 151 % für die IRQ-Handler-basierte Implementierung und 255 % für die Umsetzung als RTOS-Task. Die maximalen Reaktionszeiten liegen somit bei 264 µs bzw. 543 µs. Für letztere zeigt sich eine deutlich geringere relative Zunahme der maximalen Reaktionslatenz (8 % für die Kernel-Implementierung und 2 % für die Userspace-Implementierung) bei jedoch höheren Absolutwerten um 2 ms bzw. 30 ms. Für die native Linux-Implementierung steigt die maximale gemessene Reaktionszeit unter Last für die threadbasierte Interrupt-Handler-Implementierung um 92 % bzw. für die Userspace-Implementierung um 1100 % an. Für die Interrupt-Handler-basierte Implementierung wurde im Lastszenario jedoch eine geringere Maximallatenz beobachtet als im unbelasteten Fall. Eine Betrachtung der Messwertverteilungen beider Fälle legt nahe, dass es sich bei dem im unbelasteten Fall gemessenen Maximalwert um einen selten auftretenden Fall handelt. Da der genutzte OS-Kernel nicht auf Echtzeitfähigkeit ausgelegt ist, sind solche Ausreißer nachvollziehbar. Sie entstehen bspw. durch vergleichsweise lange laufende, nicht unterbrechbare Programmabschnitte des Kernels, während welcher Interrupts deaktiviert sind. Dies verzögert die Verarbeitung des Trigger-Interrupts, bis das Ende des Programmabschnitts erreicht ist.

4.3.2.4 Diskussion und Schlussfolgerungen

Im Hinblick auf die dreistufige Einteilung der betrachteten Implementierungen in Interrupt-Handler-basierte Implementierungen, RTOS-Task- bzw. threadbasierte Interrupt-Handler-Implementierungen und Userspace-Implementierungen ist zu erkennen, dass der Einfluss paralleler Rechenlast auf die Reaktionszeit der Anwendung mit zunehmender Stufe steigt. Dies ist durch den zunehmenden Rechenaufwand und die zunehmende Anfälligkeit für Unterbrechungen durch parallele Ereignisse bzw. Tasks nachvollziehbar.

Der Einsatz eines auf Echtzeitverhalten optimierten Linux-Kernels führt gegenüber den nativen Linux-Implementierungen zu einer deutlich schmaleren Verteilung der beobachteten Reaktionslatenzen. Insbesondere bei der Userspace-Implementierung ist zudem eine deutliche Reduktion des Einflusses paralleler Last auf die Reaktionslatenz beobachtbar. Dies geht jedoch mit deutlich höheren mittleren Reaktionslatenzen im unbelasteten Fall einher.

Die beobachtete Lastabhängigkeit der Reaktionszeit der Linux-Implementierungen geht nicht nur auf Interferenzeffekte zwischen den vom Betriebssystem verwalteten CPU-Kernen zurück. Da die Threads der Lastanwendung in den Linux-Szenarien auf beiden CPU-Kernen ausgeführt werden, ist anzunehmen, dass die Verzögerung der Evaluationsanwendung zu einem großen Teil durch die parallele Ausführung dieser ressourcenintensiven Threads auf demselben CPU-Kern verursacht wird. Mögliche Ursachen für eine solche Verzögerung umfassen unter anderem das Verhalten des OS-Schedulers bei hoher Auslastung sowie durch die Lastanwendung verursachte Interrupts, deren direkte Verarbeitung durch den Kernel die Ausführung der Evaluationsanwendung unterbrechen können.

Im Gegensatz zu den Linux-basierten Implementierungen ist die Lastabhängigkeit der Reaktionszeit der RTOS-basierten Implementierungen im Hypervisor-partitionierten System nicht auf solche Auslastungseffekte des Betriebssystems oder des von der Testanwendung genutzten CPU-Kerns zurückzuführen. Ein Vergleich der in Abbildung 4.4b dargestellten Verteilungen der Reaktionszeitmessungen dieser Implementierungen zeigt, dass RTOS-Implementierungen im Hypervisor-partitionierten System in deutlich geringerem Maße von der eingebrachten Lastanwendung beeinflusst werden als die entsprechenden Linux-Implementierungen. Dennoch ist auch in der RTOS-Partition eine Verschiebung der Messwertverteilung hin zu höheren Latenzen erkennbar. Die dieser Verschiebung zugrunde liegende Verzögerung der Programmausführung auf dem RTOS ist Auswirkung der Interferenz zwischen den CPU-Kernen, welche durch die starke Auslastung der der Linux-Partition zugewiesenen Ressourcen verursacht ist. Sie führt für beide RTOS-Implementierungsalternativen zu dem in Abschnitt 4.3.2.2 beschriebenen Anstieg der mittleren Ausführungszeit um 0.7 % bzw. der maximal gemessenen Latenz um 151 % bzw. 255 %.

4.4 Zusammenfassung und Fazit

In diesem Kapitel wurden die Auswirkungen von Interferenzeffekten auf das Zeitverhalten der Softwareausführung auf einer Multicore-basierten Steuerungseinheit exemplarisch untersucht. Durch Ende-zu-Ende-Messungen wurde die Reaktionszeit der Steuerungseinheit auf externe Ereignisse anhand einer einfachen Evaluationsanwendung untersucht. Hierbei wurden verschiedene Implementierungsalternativen der Anwendung betrachtet, welche entweder in einer mittels eines Hypervisors statisch partitionierten Softwarearchitektur oder auf einem Linux-Betriebssystem ohne Partitionierung umgesetzt wurden.

Während der Einsatz der partitionierten Architektur in den betrachteten Szenarien zu niedrigeren mittleren Reaktionszeiten führt als rein Linux-basierte Umsetzungen, ist dennoch eine Abhängigkeit der Reaktionszeit von der Auslastung anderer Partitionen erkennbar. Dies zeigt, dass durch den Einsatz des Hypervisors im untersuchten Szenario keine vollständige zeitliche Partitionierung erreicht wurde. Im Hinblick auf die maximal

beobachteten Reaktionszeiten ist eine starke Abhängigkeit von der Auslastung paralleler Partitionen zu erkennen. Während dies für Anwendungen mit weichen Echtzeitanforderungen annehmbar ist, stellt dies bei der Umsetzung von Anwendungen mit harten Echtzeitanforderungen ein Problem dar. Für die effiziente Integration solcher Anwendungen sind demnach weitere Maßnahmen nötig, um die Auswirkungen von Interferenzeffekten zu reduzieren und hinreichende zeitliche Partitionierung sicherzustellen.

5 Effiziente Timing-Isolation durch Trace-Monitoring kritischer Anwendungen

In diesem Kapitel wird ein Konzept zur Interferenzbeherrschung zwischen Kernen von Multicore-Prozessoren vorgestellt, welches die effiziente Umsetzung von Systemen mit gemischten Echtzeitanforderungen auf Hypervisor-partitionierten MCP-Plattformen ermöglichen soll. Basierend auf der kontinuierlichen Überwachung des Ausführungsfortschritts von Echtzeitanwendungen wird die parallele Ausführung von Echtzeit- und unkritischen Anwendungen erlaubt, solange das Einhalten von Deadlines durch auftretende Interferenzeffekte nicht gefährdet ist.

Das vorgestellte Konzept wurde zur Anwendung in Mixed-Criticality-Systemen entworfen, welche durch die Integration bestehender und neuer Funktionen auf einer MCP-Plattform entstehen. Aus diesem Grund setzt es auf einer Hypervisor-Komponente zur Partitionierung der Plattform auf, welche insbesondere die Integration verschiedener Anwendungen und Betriebssysteme unterstützt. Der Überwachungsansatz erlaubt die unmodifizierte Integration von Echtzeit-Anwendungen und erfordert keine Informationen über das Ausführungsverhalten unkritischer Anwendungen.

Das Konzept wird prototypisch auf einem MPSoC unter Nutzung eines industriellen Hypervisors umgesetzt, hinsichtlich seiner Effektivität und Effizienz evaluiert und einem konservativen Scheduling-Ansatz gegenübergestellt. Abschließend erfolgt eine Diskussion der unter Anwendung des Konzepts erreichbaren Echtzeiteigenschaften sowie Schlussfolgerungen zur Leistungsfähigkeit und dem Ressourcenbedarf des Ansatzes.

Das in diesem Kapitel beschriebene Konzept zur Interferenzbeherrschung wurde in der eigenen Veröffentlichung [SSBT22] publiziert. Die hier vorgestellte Implementierung ist gegenüber der bereits veröffentlichten Variante im Hinblick auf ihre Reaktionslatenz optimiert. Auch der Umfang und Detailgrad der Evaluation geht über die bestehende Veröffentlichung hinaus.

5.1 Umfeld und Zielsetzung

Steuerungs- und Regelungsaufgaben in eingebetteten Echtzeitsystemen werden meist in Form periodischer Tasks implementiert. In jeder Ausführungsperiode des Tasks werden

auf Basis der Eingabedaten und des gespeicherten Zustands Steuerentscheidungen getroffen und ausgegeben. Der Umfang der hierzu nötigen Berechnungen kann dabei stark von den Eingabedaten abhängen. Dies ist beispielsweise der Fall, wenn Berechnungen nur in bestimmten Betriebszuständen vorgenommen werden müssen oder wenn gespeicherte Berechnungsergebnisse aus vorherigen Ausführungsperioden bei gleich bleibenden Eingangsdaten wiederverwendet werden können.

Sollen Tasks, an welche harte Echtzeitanforderungen gestellt sind (Echtzeit-Tasks, ET), und unkritische Tasks (sog. Best-Effort-Tasks, BET) auf einem Multicore-Prozessor integriert werden, muss sichergestellt werden, dass die Deadlines der ET stets eingehalten werden. Durch Interferenzeffekte zwischen den Kernen kann sich die Ausführungszeit von Tasks bei paralleler Ausführung von Tasks auf mehreren Kernen eines Multicore-Prozessors um ein Vielfaches verlängern [78]. Gleichzeitig ist es schwierig, eine enge WCET-Abschätzung für diesen Fall zu bestimmen, insbesondere wenn die Ressourcenzugriffe eines der Tasks nicht bekannt sind. Dies stellt ein zentrales Hindernis bei der effizienten Umsetzung von Echtzeitsystemen auf Multicore-Prozessoren dar.

Ein konservativer Ansatz zur Lösung dieser Problematik basiert daher auf der Ausführung von Echtzeit-Tasks auf einem Kern des Multicore-Prozessors, während alle weiteren Kerne deaktiviert sind [81, 187]. Hierdurch wird die durch diese Kerne verursachte Interferenz eliminiert [48] und ein Verhalten ähnlich eines Single-Core-Prozessors erreicht. Dies geschieht jedoch auf Kosten der Performance, da es die parallele Ausführung von Software verhindert.

Das in diesem Kapitel vorgestellte Konzept verfolgt daher das Ziel, die Ausnutzung der Rechenleistung von Multicore-Prozessoren zu optimieren und gleichzeitig das Einhalten der Deadlines von Echtzeit-Tasks sicherzustellen. Somit ergeben sich die folgenden zentralen Anforderungen:

Anforderung 1: Optimierung paralleler Ausführung
Um die Ausnutzung der Rechenleistung des Multicore-Prozessors zu optimieren, soll ein möglichst hoher Zeitanteil paralleler Ausführung von Echtzeit- und Best-Effort-Tasks erreicht werden.

Anforderung 2: Wahrung des Echtzeitverhaltens
Um die Echtzeitanforderungen an Echtzeit-Tasks zu erfüllen, muss die Einhaltung der Deadlines dieser Tasks sichergestellt werden.

5.2 Konzept

Im Folgenden wird das Konzept zur Interferenzbeherrschung auf Multicore-Prozessoren vorgestellt. Dies erfolgt in den folgenden Unterkapiteln zur einfacheren Nachvollziehbarkeit

zunächst für einen Echtzeit-Task. In Abschnitt 5.2.7 wird die Anwendung des Konzepts auf Systeme mit mehreren Echtzeit-Tasks beschrieben.

5.2.1 Annahmen und Voraussetzungen

Das im Folgenden dargestellte Konzept bezieht sich auf Systeme, innerhalb derer mehrere Anwendungen (Tasks) auf einem Multicore-Prozessor ausgeführt werden sollen. Hierbei kann es sich um Mixed-Criticality-Systeme handeln, deren Anforderungen hinsichtlich räumlicher Partitionierung mittels eines Hypervisors erfüllt werden können. Die Anwendungen lassen sich hierbei nach den an sie gestellten Echtzeitanforderungen in zwei Klassen unterteilen:

- **Echtzeitanwendungen (Echtzeit-Tasks, ET)** sind Anwendungen, an welche Echtzeitanforderungen gestellt sind. Wie im vorherigen Abschnitt motiviert, wird angenommen, dass ET periodisch ausgeführt werden und in jeder Ausführungsperiode eine definierte Deadline haben, zu welcher ihre Ausführung abgeschlossen sein muss. Weiterhin wird vorausgesetzt, dass ET genau einen Prozessorkern nutzen und wie im nachfolgend beschrieben hinsichtlich ihrer WCET analysierbar sind.
- **Unkritische Anwendungen (Best-Effort-Tasks, BET)** sind Anwendungen, an welche keine Echtzeitanforderungen gestellt werden. Ihre Ausführung kann dementsprechend jederzeit unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden, ohne Anforderungen an das System zu verletzen. Hinsichtlich des Laufzeitverhaltens und der Analysierbarkeit von BET-Anwendungen bestehen keine weiteren Einschränkungen. Auch bestehen keine Einschränkungen bzgl. der Anzahl parallel genutzter Prozessorkerne. BETs können somit auch komplexe Softwarestacks umfassen, z.B. ein Betriebssystem mit mehreren, dynamisch austauschbaren Anwendungen[1].

Aus der Forderung, BET jederzeit unterbrechen zu können, folgt, dass die korrekte und rechtzeitige Ausführung von ET unabhängig vom Ausführungsfortschritt der BET sein muss. Dementsprechend müssen Verzögerungen der Ausführung von ET durch BET, welche u.a. durch die wechselseitige, blockierende Inanspruchnahme von Ressourcen wie I/O-Komponenten entstehen können, ausgeschlossen werden. Gibt es Datenabhängigkeiten zwischen BET und ET, muss sichergestellt sein, dass ausbleibende oder spät verfügbare Daten nicht zu einer unvorhersehbaren Verzögerung von ET führen.

Hinsichtlich der Analysierbarkeit von ET wird vorausgesetzt, dass partielle WCET-Abschätzungen $WCET_{part}(I)$ bestimmt werden können, welche eine obere Schranke der

[1] Auch wenn die als unkritische Anwendungen klassifizierten Softwarekomponenten somit über die Definition eines *Tasks* hinausgehen können, werden sie im Folgenden der Kürze wegen unter der Bezeichnung *Best-Effort-Task BET* zusammengefasst.

ET-Ausführungsdauer ab der ersten Ausführung der Instruktion I im ET-Programm bis zum Ende der Ausführung des ET in der Ausführungsperiode angeben. Zur Bestimmung der WCET kann die exklusive Ausführung des ET auf einem Kern des MCP ab Instruktion I angenommen werden. Es muss jedoch davon ausgegangen werden, dass sich geteilt genutzte Ressourcen, wie z.B. Caches, zu Beginn des betrachteten Ausführungsteils, d.h. beim ersten Erreichen der Instruktion I, durch zuvor erfolgte Zugriffe paralleler Anwendungen in einem unbekannten Zustand befinden. Zusätzlich ist hinsichtlich der WCET-Abschätzung gefordert, dass $WCET_{part}(I) \geq WCET_{part}(J)$ für alle Instruktionen J gilt, welche im Ablauf einer ET-Ausführungsperiode nach I verarbeitet werden. Hieraus folgt, dass die Ausführung des ET nach Erreichen der Instruktion J innerhalb der Dauer $WCET_{part}(I)$ bei exklusiver Ausführung abgeschlossen werden kann, auch wenn bis zum Erreichen der Instruktion J parallele Anwendungen ausgeführt wurden.

Hinsichtlich der Ausführungsplattform wird vorausgesetzt, dass der MCP mit einer Trace-Infrastruktur ausgestattet ist, welche Kontrollflussinformationen bereitstellt (vgl. Abschnitt 2.2.4.1.1).

Wie einleitend motiviert, wird angenommen, dass der Umfang der vom ET durchgeführten Berechnungen von dessen Eingabewert und Zustand abhängt und somit über verschiedene Ausführungsperioden hinweg sowohl kürzere als auch längere Ausführungszeiten beobachtet werden können. Diese Annahme ist jedoch keine zwingende Voraussetzung für die Anwendung des Konzepts.

5.2.2 Technischer Ansatz

Der in diesem Kapitel vorgestellte Ansatz zur Optimierung paralleler Ausführung von ET und BET auf MCP basiert auf den im Folgenden dargestellten Überlegungen.

Zunächst wird die exklusive Ausführung des ET auf einem Kern des MCP betrachtet. Die WCET eines ET stellt hierbei eine obere Schranke der Ausführungsdauer des ET für alle möglichen Inputs und erreichbaren interne Zustände des Tasks dar. Die tatsächliche Dauer einer Ausführung hängt von den Eingabewerten und dem Zustand in der jeweiligen Ausführungsperiode ab. Sie wird insbesondere durch Kontrollflussentscheidungen beeinflusst, welche auf Basis dieser Daten zur Laufzeit getroffen werden.

Führt der MCP parallel zum ET weitere Tasks aus, können durch die geteilte Nutzung der Rechenressourcen Interferenzeffekte zwischen den Kernen des MCP auftreten, welche die Ausführung des ET verzögern (vgl. Abschnitt 2.3.2). Die Ausführungsgeschwindigkeit des ET ist hierbei abhängig von der Stärke der auftretenden Interferenz. Diese ist wiederum vom Verhalten des ET sowie der parallel ausgeführten Tasks im Hinblick auf die Nutzung der geteilten Ressourcen abhängig. Da das Ressourcennutzungsverhalten des BET a priori

als nicht bekannt angenommen wird, ist auch das Ausmaß der auftretenden Interferenzeffekte und somit die resultierende Verzögerung der ET-Ausführung zur Entwurfszeit nicht genau bestimmbar.

Um dennoch parallele Ausführung auf dem MCP zu ermöglichen, werden die Auswirkungen der Interferenz auf den Ausführungsfortschritt des ET zur Laufzeit überwacht. Auf Basis dieser Information wird die Ausführung paralleler BET so gesteuert, dass die Deadline des ET eingehalten werden kann.

Die Überwachung des Ausführungsfortschritts des ET erfolgt durch die Beobachtung von Kontrollflussentscheidungen an bedingten Verzweigungsinstruktionen (Conditional Branch Instructions, CBI) des ET. Diese werden mittels der Trace-Infrastruktur der MCP-Plattform erfasst. Die Gründe für die Nutzung der Trace-Infrastruktur werden im Folgenden dargelegt:

- Die Nutzung der Trace-Infrastruktur erlaubt die Beobachtung von Kontrollflussentscheidungen an CBI zur Laufzeit. Diese liefern sowohl Informationen über den Ausführungsfortschritt des ET als auch über Entscheidungen, welche Auswirkungen auf die verbleibende Laufzeit des ET haben.
- Mittels der Filterfunktion der Trace-Infrastruktur kann die Tracedatenerzeugung auf ausgewählte CBI beschränkt werden. Hierdurch reduziert sich die zur Überwachung zu verarbeitende Datenmenge auf wenige Trace-Datenpakete pro überwachter CBI.
- Die Möglichkeit der Auswahl überwachter CBI erlaubt es dem Systementwickler, die Granularität der Überwachung des Ausführungsfortschritts zu definieren. Je mehr CBIs überwacht werden, desto feingranularer kann der Ausführungsfortschritt zur Laufzeit ermittelt werden. Die Häufigkeit, mit welcher überwachte CBI im ET-Code auftreten, wird als *CBI-Rate* bezeichnet.
- Trace-Infrastrukturen sind darauf ausgerichtet, die Ausführung von Software auf dem überwachten Prozessor nicht zu beeinflussen. Dies ist hilfreich, um Interferenz zwischen dem Überwachungssystem und dem ET zu vermeiden.

Zur Steuerung der BET-Ausführung wird eine *Zeitreserve* definiert. Die Zeitreserve ist ein Zeitintervall, dessen Länge ausreicht, um die Ausführung des ET bei exklusiver Ausführung auf dem MCP abzuschließen. Die parallele Ausführung von ET und BET wird nur erlaubt, wenn die verbleibende Zeit bis zur Deadline länger ist als diese Zeitreserve. Ist dies nicht der Fall, wird die Ausführung der BET unterbrochen, sodass der ET exklusiv auf dem MCP ausgeführt wird. Wird ein Ausführungsfortschritt der ET festgestellt, kann die Zeitreserve angepasst werden. Die Anpassung der Zeitreserve erfolgt hierbei auf Basis partieller WCET-Abschätzungen des ET, welche die maximale verbleibende Ausführungsdauer des ET ab den überwachten CBI bei exklusiver Ausführung angeben. Diese werden im Rahmen der Systementwicklung bestimmt.

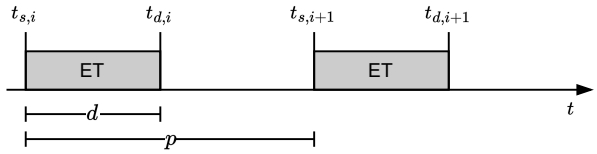


Abbildung 5.1: Scheduling-Parameter eines Echtzeit-Tasks

5.2.3 Task-Modell

Im Folgenden wird das zur Beschreibung des vorliegenden Konzepts nötige Task-Modell vorgestellt. Es umfasst sowohl relevante Scheduling-Parameter als auch Kontrollflusseigenschaften der ET.

Im Hinblick auf das Scheduling werden ET als periodische Tasks modelliert. Somit ist jedem ET eine Periode p und eine relative Deadline d zugewiesen, wobei d die Länge der Zeitspanne beschreibt, innerhalb welcher der Task nach Start seiner Ausführung abgeschlossen sein muss. Mit $t_{s,i}$ wird der Startzeitpunkt des Tasks in der i-ten Ausführungsperiode bezeichnet. Somit ergibt sich für den Zeitpunkt der Deadline in der i-ten Ausführungsperiode $t_{d,i} = t_{s,i} + d$. Für den Zusammenhang zwischen zwei Startzeitpunkten gilt in der Regel $t_{s,i+1} = t_{s,i} + p$. Diese Zusammenhänge sind in Abbildung 5.1 verdeutlicht.

Gemäß der Annahmen kann jeder Task hinsichtlich seiner WCET bei exklusiver Ausführung auf einem Kern des MCP analysiert werden. Die WCET-Abschätzung einer vollständigen Ausführung des ET wird mit $W_{total} = WCET_{part}(S)$ bezeichnet. Sie entspricht einer partiellen WCET-Abschätzung ab der ersten ausgeführten Instruktion S des ET. Die relative Deadline jedes ET muss so gewählt werden, dass $d \geq W_{total}$ gilt.

Im Hinblick auf seinen Programmablauf kann jeder ET in Form eines Kontrollflussgraphen (Control Flow Graph, CFG) dargestellt werden. Dieser stellt mögliche Abläufe der Programmausführung des ET dar. Knoten des Kontrollflussgraphen entsprechen den Basisblöcken des Programmcodes, innerhalb welcher die Programmausführung sequenziell erfolgt. Kanten stellen mögliche Sprünge zwischen den Basisblöcken dar. Sie ergeben sich aus Kontrollflussinstruktionen, welche die letzte Instruktion des jeweiligen Basisblocks bilden.

Im Rahmen des Konzepts wird angenommen, dass es ausgehend vom Einstiegspunkt des ET mehrere Wege durch den Kontrollflussgraphen des ET gibt, welche mit der ersten Instruktion S des ersten Basisblocks des Tasks beginnen und mit der letzten Instruktion des letzten Basisblocks (E) enden. Dies ist in Abbildung 5.2 anhand von zwei Wegen exemplarisch dargestellt. Auf Basis der Annahme, dass der vom ET auszu-

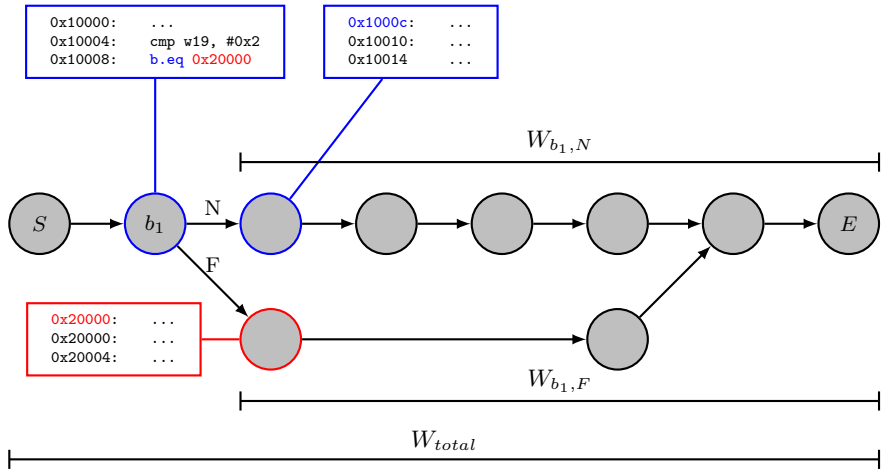


Abbildung 5.2: Exemplarische Darstellung des Kontrollflusses eines Echtzeit-Tasks mit annotierten (partiellen) WCET-Abschätzungen W. CBI wie b_1 führen zu Verzweigungen im Kontrollflussgraph.

führende Berechnungsumfang von seiner Eingabe und seinem Zustand abhängt, kann angenommen werden, dass sich einige dieser Wege hinsichtlich ihrer Ausführungszeit deutlich unterscheiden. Somit kann eine Menge B von CBI identifiziert werden, welche einen signifikanten Einfluss auf die Ausführungszeit des ET hat. An jeder CBI $b \in B$ ergibt sich hinsichtlich der weiteren Ausführung des ET eine Fallunterscheidung: Ist die Bedingung zur Laufzeit erfüllt, wird ein Sprung an eine andere Instruktionsadresse durchgeführt – der Kontrollfluss *folgt* der Verzweigung (Fall F). Ist sie nicht erfüllt, wird die Ausführung an der nachfolgenden Instruktion fortgesetzt – der Kontrollfluss folgt der Verzweigung *nicht* (Fall N).

Aufgrund der geforderten Analysierbarkeit des ET lassen sich für jede CBI $b \in B$ die partiellen WCET-Abschätzungen $W_{b,F} = WCET_{part}(I_{b,F})$ und $W_{b,N} = WCET_{part}(I_{b,N})$ bestimmen. Diese entsprechen für den Fall F bzw. N jeweils einer oberen Schranke der verbleibenden Ausführungszeit des ET ab der auf b folgenden Instruktion $I_{b,F}$ bzw. $I_{b,N}$.

5.2.4 Systemübersicht

Abbildung 5.3 stellt die zentralen Komponenten des Systems zur Umsetzung der Interferenzbeherrschung dar. Auf dem Multicore-Prozessor wird ein Hypervisor ausgeführt, welcher alle Kerne des Prozessors verwaltet und die Ausführung von ET und BET auf diesen Kernen kontrolliert. ET und BET sind jeweils in separaten Hypervisor-Partitionen umgesetzt, wodurch der Hypervisor die räumliche Partitionierung (nach Abschnitt 2.3.1)

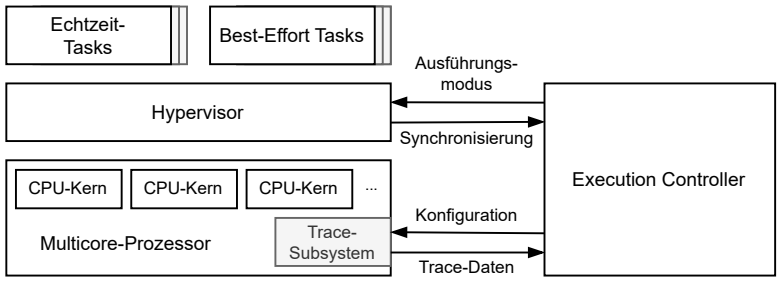


Abbildung 5.3: Monitoring-System zur Timing-Isolation auf Multicore-Prozessoren (aufbauend auf [SSBT22])

des Systems umsetzt. Er implementiert weiterhin ein Scheduling auf Basis von festen, zyklischen Zeitfenstern (vgl. Abschnitt 2.4.4.1.1).

Neben dem Multicore-Prozessor umfasst das System weiterhin einen *Execution Controller*. Der Execution Controller überwacht die Ausführung des Echtzeit-Tasks und steuert basierend darauf die Ausführung des BET. Die Überwachung erfolgt anhand von Ausführungs-Trace-Daten, welche durch das Trace-Subsystem des Multicore-Prozessors generiert werden. Dieses wird vom Execution Controller konfiguriert und liefert anschließend Trace-Daten, welche wiederum vom Execution Controller empfangen und ausgewertet werden. Die Steuerung der BET-Ausführung erfolgt durch Signalisierung eines Ausführungsmodus (vgl. Abschnitt 5.2.5.3) an den Hypervisor, welcher diesen anschließend umsetzt. Der Execution Controller ist hierzu mit der Scheduler-Komponente des Hypervisors synchronisiert.

5.2.5 Scheduling-Ansatz

Im Folgenden wird das von Hypervisor und Execution Controller umgesetzte Scheduling-Verfahren vorgestellt. Hierzu wird zunächst auf die Konfiguration des Hypervisor-Schedules sowie das Informationsmodell des Execution Controller eingegangen. Anschließend wird das Konzept der Ausführungsmodi erläutert und das Scheduling-Verhalten von Hypervisor und Execution Controller zur Laufzeit beschrieben.

5.2.5.1 Konfiguration des Hypervisors

Auf dem Hypervisor werden ET und BET in separaten Partitionen implementiert. Dies ermöglicht die räumliche Isolation der Tasks durch Konfiguration der für die Partitionen erreichbaren Ressourcen der Ausführungsplattform.

Hinsichtlich des Schedulings wird ein zyklischer Schedule basierend auf festen Zeitfenstern (vgl. Abschnitt 2.4.4.1.1) konfiguriert. Hierbei wird jedem ET ein Zeitfenster $W = (c, s, D)$ zugeordnet, welches durch den CPU-Kern c, den Offset s des Zeitfensters innerhalb des MAF sowie der Dauer D definiert ist. Die Parametrisierung der Zeitfenster erfolgt anhand der Ausführungsparameter des zugehörigen ET. Insbesondere wird das Offset des Zeitfensters s anhand des Startzeitpunkts einer ET-Ausführung $t_{s,i}$ gewählt, die Dauer des Zeitfensters entspricht der relativen Deadline des ET ($D = d$), sodass $s + D = t_{d,i}$ gilt. Hierbei wird angenommen, dass die Länge des MAF der ET-Periode p entspricht. Entspricht die Länge des MAF einem Mehrfachen der ET-Periode, werden entsprechend mehrere ET-Zeitfenster pro MAF definiert.

Wie in Abschnitt 5.2.3 beschrieben, muss $D = d \geq W_{total}$ gewählt werden, sodass der Abschluss der Taskausführung vor der Deadline im Fall dauerhafter exklusiver Ausführung sichergestellt ist. Die Deadline und somit das Zeitfenster zur ET-Ausführung sollte im Rahmen der Konfiguration, soweit möglich, so gewählt werden, dass es ein *initiales Zeitbudget für parallele Ausführung* t_{slack} umfasst, d.h. $D = d = W_{total} + t_{slack}$. Eine solches Zeitbudget erlaubt es, bereits zu Beginn des ET-Zeitfensters die parallele Ausführung von ET und BET zuzulassen und diese auch in Situationen leichter Interferenz zu erhalten.

Während für jede ET-Ausführung ein einzelnes, zusammenhängendes Zeitfenster definiert werden muss, kann das Offset dieses Zeitfensters im MAF frei gewählt werden. Werden mehrere ET-Zeitfenster im MAF definiert, müssen diese auf dem gleichen Prozessorkern c ausgeführt werden. Dieser Kern wird im Folgenden als *ET-Kern* bezeichnet.

Zeitfenster für BET können auf den verbleibenden Kernen beliebig platziert werden. Insbesondere können BET-Zeitfenster parallel zu ET-Zeitfenstern platziert werden. Außerhalb der von ET-Zeitfenstern belegten Zeiträume können BET-Zeitfenster auch auf dem ET-Kern angelegt werden.

5.2.5.2 Konfiguration des Execution Controller

Die Konfiguration des Execution Controllers erfordert zunächst eine Analyse des ET hinsichtlich zu überwachender CBI. Relevant sind insbesondere CBI, welche einen starken Einfluss auf die verbleibende Ausführungszeit des Echtzeit-Tasks haben. Wird an diesen CBIs der Pfad mit geringerer Ausführungszeit gewählt, ermöglicht dies eine Reduktion der Zeitreserve, welche eine Erhöhung des Zeitbudgets zur parallelen Task-Ausführung ermöglichen kann. Darüber hinaus kann es hilfreich sein, zusätzlich CBI anhand ihrer zeitlichen Lage im Kontrollfluss des ET zu wählen, um eine gewünschte CBI-Rate zu erreichen.

Abbildung 5.4 stellt die zentralen Elemente der Konfiguration des Execution Controller

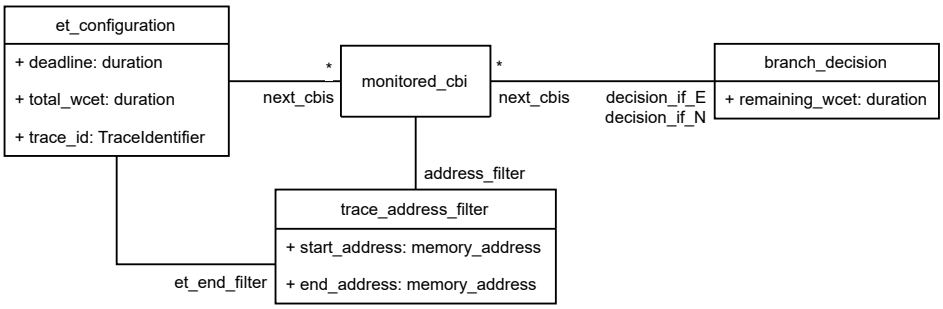


Abbildung 5.4: Konfiguration des Execution Controller

dar. Die Konfiguration umfasst die Deadline d und die WCET W_{total} des ET, Merkmale zur eindeutigen Identifikation des zum ET gehörenden Trace-Datenstroms sowie eine Trace-Filter-Konfiguration, um das Ende der ET-Ausführung zu erkennen. Weiterhin umfasst sie für jede zu überwachende CBI $b \in B$ die partiellen WCET-Abschätzungen $W_{b,F}$ und $W_{b,N}$ sowie eine Trace-Filter-Konfiguration. Die Filter-Konfiguration beschreibt den vom Trace-Subsystem zu überwachenden Instruktionsadressbereich und dient dazu, die Tracedatenerzeugung auf die relevanten CBI zu begrenzen.

Da Trace-Architekturen meist nur eine begrenzte Anzahl an Filtereinheiten umfassen, ist es in der Regel nicht möglich, alle zu überwachenden CBI $b \in B$ gleichzeitig durch Trace-Filter zu selektieren. Aus diesem Grund überwacht der Execution Controller zu jedem Zeitpunkt nur eine Teilmenge von B sowie das Ende des ET. Die zu überwachenden CBI sind daher in Form eines gerichteten Graphen abgelegt. Hierzu ist eine ist eine Teilmenge $B_s \subseteq B$ definiert, welche die zu Beginn der ET-Ausführung zu überwachenden CBI angibt. Für jede CBI $b \in B$ sind weiterhin zwei Teilmengen $B_{b,F} \subseteq B$ und $B_{b,N} \subseteq B$ definiert, welche für die beiden Verzweigungs-Entscheidungen F und N (vgl. Abschnitt 5.2.3) die nächsten zu überwachenden CBIs angeben.

5.2.5.3 Ausführungsmodi

Hinsichtlich der Ausführung von ET und BET werden zwei Ausführungsmodi unterschieden.

- **Parallele Ausführung:** ET und BET werden parallel auf verschiedenen Kernen des MCP ausgeführt.
- **Exklusive Ausführung:** Der ET auf dem ET-Kern ausgeführt. Alle weiteren Kerne des MCP sind inaktiv, die BET-Ausführung ist pausiert.

5.2.5.4 Laufzeitverhalten

Im Folgenden wird das zur Laufzeit durch Execution Controller und Hypervisor umgesetzte Scheduling detailliert vorgestellt. Abbildung 5.5 visualisiert hierzu die Betriebszustände des Execution Controller.

Nach dem Start des Systems befindet sich der Execution Controller im unsynchronisierten Zustand. In diesem sind alle CPU-Kerne aktiv. Der Hypervisor setzt den konfigurierten, zeitfensterbasierten Schedule mit paralleler Ausführung um.

Zu Beginn jedes Zeitfensters eines ET, zum Zeitpunkt $t = t_{s,i}$ in der i-ten ET-Ausführungsperiode, beginnt die Ausführung des ET auf dem ET-Kern. Parallel werden zunächst, falls im Hypervisor-Schedule konfiguriert, auf den verbleibenden Kernen BET ausgeführt. Gleichzeitig sendet der Hypervisor ein Synchronisierungssignal an den Execution Controller, welcher daraufhin in den synchronisierten Zustand wechselt.

Im synchronisierten Zustand überwacht der Execution Controller den Fortschritt des ET sowie die seit Beginn des ET-Ausführungszeitfensters verstrichene Zeit und steuert die Ausführung der BET. Hierzu wird zunächst der Zeitpunkt $t_{x,i}$ ermittelt, zu welchem der Übergang vom parallelen hin zum exklusiven Ausführungsmodus erfolgt. Dieser wird initial so gewählt, dass zwischen $t_{x,i}$ und der Deadline des ET $t_{d,i}$ eine Zeitreserve w_{res} der Dauer W_{total} zur exklusiven Ausführung besteht. Hierdurch wird sichergestellt, dass der ET bis zur Deadline vollständig ausgeführt werden kann.

$$w_{res} \leftarrow W_{total}$$
$$t_{x,i} \leftarrow t_{d,i} - W_{total}$$

Weiterhin wird das Trace-Subsystem zur Überwachung der ersten relevanten CBI B_s konfiguriert.

Während der Ausführung des ET verarbeitet der Execution Controller die eingehenden Trace-Daten. Sobald auf dem MCP eine CBI $b_j \in B$ ausgeführt wird, erzeugt die Trace-Infrastruktur ein Branch-Datenpaket, welches Informationen über die Branch-Entscheidung enthält. Sobald dieses vom Execution Controller empfangen und decodiert wurde, aktualisiert dieser $t_{x,i}$, sodass die Zeitreserve nun der partiellen WCET-Abschätzung für die gewählte Branch-Entscheidung $d_{j,i} \in \{F, N\}$ entspricht:

$$w_{res} \leftarrow W_{b_j,d_{j,i}}$$
$$t_{x,i} \leftarrow t_{d,i} - W_{b_j,d_{j,i}}$$

Auch werden die Filter des Trace-Systems zur Überwachung der nächsten relevanten CBI $B_{b_j,d_{j,i}}$ konfiguriert.

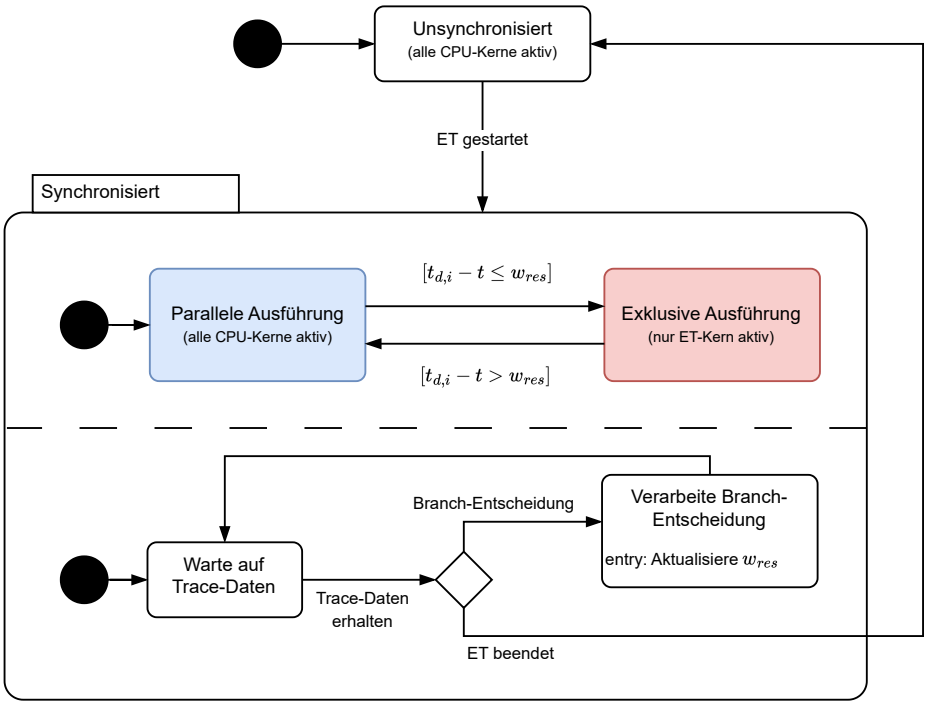


Abbildung 5.5: Funktionsweise des Execution Controller

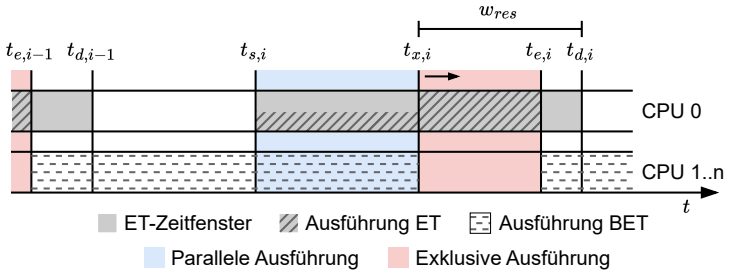


Abbildung 5.6: Resultierendes Scheduling-Verhalten

Zum Zeitpunkt $t_{x,i}$ signalisiert der Execution Controller dem Hypervisor den Übergang in den exklusiven Ausführungsmodus. Dieser stoppt daraufhin die Ausführung aller BET auf den BET-Kernen. Auch im exklusiven Ausführungsmodus wird der Fortschritt des ET durch den Execution Controller weiterhin überwacht. Wird $t_{x,i}$ in Folge einer beobachteten Branch-Entscheidung zurück in die Zukunft versetzt, löst der Execution Controller den Übergang zurück in den parallelen Ausführungsmodus aus.

Stellt der Execution Controller den Abschluss der ET-Ausführung fest ($t = t_{e,i}$), wechselt er in den unsynchronisierten Zustand. Im Zuge dessen signalisiert er dem Hypervisor, die Ausführung von BET zuzulassen (parallele Ausführung) und wartet anschließend auf ein erneutes Synchronisierungssignal.

Abbildung 5.6 visualisiert den Ansatz anhand einer exemplarischen Ausführung auf dem MCP. Zum Zeitpunkt $t_{s,i}$ beginnt die Ausführung des ET im parallelen Ausführungsmodus. Durch die parallele Ausführung von ET und BET entstehen Interferenzeffekte, welche eine reduzierte Ausführungsgeschwindigkeit des ET zur Folge haben. Zum Zeitpunkt $t_{x,i}$ erfolgt der Übergang zum exklusiven Ausführungsmodus. In diesem sind die BET gestoppt, der ET läuft mit voller Geschwindigkeit. Auf Basis des zur Laufzeit erkannten Ausführungsfortschritts des ET kann $t_{x,i}$ hierbei mehrfach verschoben werden. Zum Zeitpunkt $t_{e,i}$ endet die Ausführung des ET. Der Execution Controller wechselt in den unsynchronisierten Zustand, die Ausführung von BET ist wieder erlaubt.

5.2.6 Abstrakte Betrachtung des Schedulingverhaltens

Das in Abschnitt 5.2.5.4 beschriebene Schedulingverhalten lässt sich abstrakt anhand eines Budgets zur parallelen Ausführung von ET und BET auf dem MCS beschreiben. Hierzu wird ein Ausführungszeitbudget p angenommen, welches während der parallelen Ausführung von ET und BET verbraucht wird.

Die initiale Festlegung von $t_{x,i}$ zu Beginn des ET-Zeitfensters entspricht einer Initialisierung des Budgets anhand des konfigurierten initialen Zeitbudgets t_{slack}:

$$p \leftarrow D - W_{total} = t_{slack}$$

Solange $p > 0$ ist, werden ET und BET parallel ausgeführt und das Budget verbraucht. Bei $p = 0$ erfolgt der Wechsel in den exklusiven Ausführungsmodus.

Wird zum Zeitpunkt t ein Ausführungsfortschritt des ET festgestellt, kann w_{res} zu w'_{res} reduziert und $t_{x,i}$ zu $t'_{x,i}$ aktualisiert werden. Liegt das aktualisierte $t'_{x,i}$ in der Zukunft ($t'_{x,i} > t$), entspricht dies einem Auffüllen des Ausführungszeitbudgets p auf die

verbleibende Zeitdauer bis zur Deadline abzüglich der aktualisierten Zeitreserve:

$$p \leftarrow t'_{x,i} - t = t_{d,i} - w'_{res} - t$$

Liegt das aktualisierte $t'_{x,i}$ in der Vergangenheit, wird das Budget nicht aufgefüllt.

Mit dem Abschluss der ET-Ausführung zum Zeitpunkt $t_{e,i}$ wird das System in den parallelen Ausführungsmodus versetzt. Da der ET abgeschlossen ist, ist die weitere BET-Ausführung bis zum nächsten ET-Zeitfenster ohne Berücksichtigung des Ausführungszeitbudgets möglich.

5.2.7 Unterstützung mehrerer Echtzeit-Tasks

Im Folgenden werden die Anwendbarkeit des Konzepts bei Integration mehrerer ET auf dem MCP diskutiert.

Grundsätzlich ist das vorgeschlagene Konzept auf Systeme mit mehreren ET anwendbar. Bei der Definition des Schedules muss hierbei sichergestellt werden, dass ET nur auf einem CPU-Kern, dem ET-Kern, platziert werden und dass jeder ET-Ausführung ein zusammenhängendes Zeitfenster zugeordnet ist. Innerhalb eines MAF können mehrere Ausführungsperioden desselben ET vorgesehen werden. Dies ermöglicht die Integration von ET mit unterschiedlichen Periodendauern. In diesem Fall kann die MAF-Dauer als Hyperperiode aller ET-Perioden bestimmt werden.

Werden mehrere ET unterstützt, muss das Synchronisierungssignal zwischen Hypervisor und Execution Controller eine Identifikation des gestarteten ET ermöglichen. Somit kann der Execution Controller im Rahmen der Synchronisierung die passende Task-Konfiguration laden und mit diese wie in Abschnitt 5.2.5.4 beschrieben umsetzen.

5.3 Prototypische Umsetzung

Das in Abschnitt 5.2 vorgestellte Konzept wurde auf einem Zynq UltraScale+ MPSoC [39] auf dem ZCU102 Entwicklungsboard [188] unter Nutzung des Hypervisors PikeOS (s. Abschnitt 2.4.5.2) implementiert. Beide kommerziell verfügbaren Komponenten wurden dabei nur im Rahmen der vom Hersteller vorgesehenen Konfigurations- bzw. Anpassungsmöglichkeiten genutzt. Die Implementierung erfolgte prototypisch für einen Echtzeit-Task, eine Erweiterung auf mehrere ET ist basierend auf den in Abschnitt 5.2.7 beschriebenen Überlegungen möglich.

5.3.1 Ausführungsplattform

Das Zynq UltraScale+ MPSoC umfasst zwei zentrale Prozessorcluster: Die Application Processing Unit (APU) umfasst 4 ARM Cortex-A53-Kerne, die Real-time Processing Unit (RPU) besteht aus 2 ARM Cortex-R5-Kernen. Beide Prozessorcluster sind an ein Trace-Subsystem vom Typ ARM CoreSight angebunden, welches Instruktionstracing ermöglicht.

Im Hinblick auf die Speicherarchitektur bietet die APU für jeden CPU-Kern 32 kB L1-Instruktions-Cache und 32 kB L1-Daten-Cache. Weiterhin umfasst sie 1 MB L2-Cache, welcher von allen APU-Kernen geteilt genutzt wird. Die RPU bietet für jeden Kern 32 kB L1-Caches für Instruktionen und Daten sowie 128 kB Tightly Coupled Memory (TCM), welcher zur Umsetzung von zeitdeterministischen Anwendungen genutzt werden kann. Komponenten innerhalb des MPSoC sind über mehrere Interconnects miteinander verbunden. Ein DDR-Speichercontroller ermöglicht die Anbindung externen Speichers. Im Fall des ZCU102 sind hierüber 4 GB DDR4-RAM angebunden.

5.3.2 Systemüberblick

Abbildung 5.7 stellt die Umsetzung des Konzepts auf der MPSoC-Plattform dar. Auf der APU wird der Hypervisor ausgeführt, welcher alle Prozessorkerne des Clusters verwaltet. CPU-Kern 0 wird als ET-Kern genutzt, während auf den weiteren CPU-Kernen (BET-Kernen) die BET ausgeführt werden. Der Hypervisor-Kernel wurde um ein Modul erweitert, welches die Kommunikationsschnittstelle zum Execution Controller implementiert und die Ausführungsmodi umsetzt.

Auf der RPU wird der Execution Controller ausgeführt. Er überwacht die Ausführung des ET und steuert die BET-Ausführung durch Signalisierung des Ausführungsmodus an den Hypervisor. Um die Kommunikationslatenz zwischen Hypervisor und Execution Controller zu minimieren, erfolgt diese Signalisierung über Inter-Prozessor-Interrupts (IPI).

Zur Überwachung des ET wird die CoreSight Trace-Infrastruktur der Hardwareplattform genutzt. Sie umfasst für jeden Kern der APU eine Embedded Trace Macrocell (ETM), welche Prozessorereignisse überwacht und entsprechende Trace-Datenpakete generiert. Die Trace-Datenpakete werden an eine Embedded Trace FIFO (ETF) Komponente weitergeleitet, in welcher sie zwischengespeichert und vom Execution Controller abgerufen werden. Die zur Überwachung genutzten Trace-Komponenten werden vom Execution Controller beim Systemstart initial konfiguriert und zur Laufzeit verwaltet.

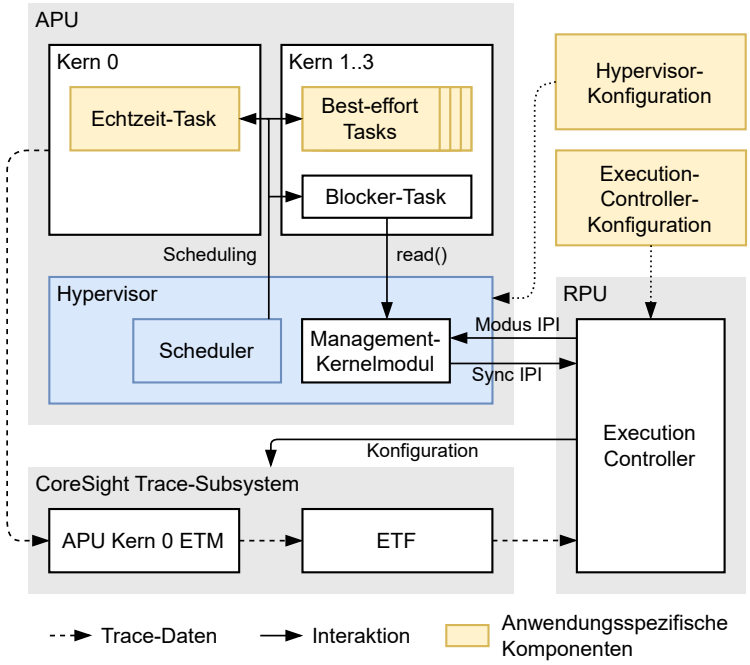


Abbildung 5.7: Umsetzung des Trace-basierten Schedulingkonzepts auf dem Zynq UltraScale+ MPSoC (eigene Darstellung nach [SSBT22])

5.3.3 Umsetzung der Ausführungsmodi

Die Umsetzung der Ausführungsmodi erfolgt durch den Hypervisor. Hierbei wird der zweistufige Schedulingmechanismus von PikeOS genutzt sowie die Möglichkeit, eigene Kernelmodule zu integrieren.

Im zeitfensterbasierten Schedule sind Ausführungszeitfenster für ET und BET definiert. Parallel zum ET-Zeitfenster auf Kern 0 werden BET-Zeitfenster auf den weiteren Kernen platziert. Dies führt grundsätzlich zu einer parallelen Ausführung von ET und BET durch den Scheduler des Hypervisors.

Das Unterbrechen der BET und das Deaktivieren der BET-Kerne erfolgt mittels eines *Blocker-Tasks*. Dieser Task ist einer speziellen Time Partition (vgl. Abschnitt 2.4.5.2) zugeordnet, wodurch er unabhängig vom zeitfensterbasierten Schedule jederzeit ausgeführt werden kann. Der Blocker-Task erzeugt für jeden BET-Kern einen Blocker-Thread, welchem eine höhere Priorität zugewiesen ist als die maximale für BET-Tasks wählbare Priorität. Ist der Blocker-Thread aktiv, verhindert er somit die Ausführung von BET-Threads auf dem Prozessorkern. Ist er inaktiv, werden die niedriger priorisierten BET-

Threads ausgeführt. Die Blocker-Threads selbst führen zyklisch eine *Wait-for-Interrupt*-Instruktion aus. Hierdurch wird der Prozessorkern in einen Energiesparmodus versetzt, sodass er selbst keine Interferenz verursacht.

Die Ausführung der Blocker-Threads wird über ein Kernelmodul im Hypervisor gesteuert. Dieses stellt eine `read()`-Funktion bereit, welche von den Blocker-Threads aufgerufen wird. Soll der Blocker-Thread deaktiviert werden, markiert das Kernelmodul diesen beim Aufruf von `read()` durch die eine Kernel-API des Hypervisors als blockiert, wodurch die Ausführung von BET-Threads ermöglicht wird. Soll die Ausführung des Blocker-Threads fortgesetzt werden, markiert das Kernel-Modul den blockierten Thread als lauffähig, wodurch dieser laufende BET-Threads aufgrund seiner höheren Priorität verdrängt.

Der Wechsel der Ausführungsmodi wird in der bestehenden Implementierung durch einen IPI ausgelöst, welcher vom Kernelmodul verarbeitet wird und zwischen exklusivem und parallelem Ausführungsmodus umschaltet. Darüber hinaus wird der Ausführungsmodus zum Start einer ET-Partition auf den parallelen Modus zurückgesetzt.

5.3.4 Umsetzung des Execution Controller

Der Execution Controller ist als Bare-Metal-Anwendung auf der RPU des MPSoC implementiert. Bei Systemstart initialisiert er das Trace-Subsystem und synchronisiert sich anschließend mit dem Hypervisor. Dies erfolgt durch einen IPI, welcher den Beginn des ET-Zeitfensters anzeigt. Im Rahmen der Synchronisation startet der Execution Controller einen Hardware-Timer als lokale Zeitreferenz, bestimmt $t_{x,i}$ und konfiguriert für diesen Zeitpunkt einen Timer-Interrupt. Wird der Timer-Interrupt zu diesem Zeitpunkt ausgelöst, signalisiert der Execution Controller dem Hypervisor mittels IPI den Wechsel in den exklusiven Ausführungsmodus. Nach der Synchronisierung konfiguriert der Execution Controller die Trace-Filter entsprechend der konfigurierten, zu überwachenden CBI und ruft Trace-Daten durch Polling ab.

Werden Trace-Informationen zu einer CBI empfangen, aktualisiert der Execution Controller $t_{x,i}$ und konfiguriert den Timer-Interrupt entsprechend. Anschließend konfiguriert er die Trace-Filter anhand der nächsten zu überwachenden CBI. Wird $t_{x,i}$ hierbei in die Zukunft versetzt, während sich das System im exklusiven Ausführungsmodus befindet, signalisiert der Execution Controller dem Hypervisor mittels IPI den Übergang zurück zum parallelen Ausführungsmodus.

Wird anhand der Trace-Daten das Ende des ET erkannt, versetzt der Execution Controller das System, wenn nötig, zurück in den parallelen Ausführungsmodus. Anschließend wartet er von neuem auf den Synchronisierungsinterrupt.

5.4 Evaluation

Die Evaluation des vorgestellten Konzepts erfolgt experimentell auf Basis der in Abschnitt 5.3 vorgestellten Implementierung auf dem Zynq UltraScale+ MPSoC. Sie zielt darauf ab, die Effektivität des vorgestellten Konzepts zu prüfen (vgl. Anforderung 2) und zentrale Zusammenhänge im Bezug auf die erreichbare Systemeffizienz (vgl. Anforderung 1) zu ermitteln. Die im Folgenden untersuchten Aspekte umfassen:

- die Sicherstellung des Einhaltens der Deadline von Echtzeit-Tasks,
- die Abhängigkeit der erreichbaren Dauer paralleler Ausführung sowie des erreichbaren Datendurchsatzes paralleler Anwendungen von den folgenden Parametern:
  - der Stärke der Interferenzeffekte zwischen den Kernen,
  - des konfigurierten initialen Zeitbudgets t_{slack} für parallele Ausführung,
  - der zeitlichen Häufigkeit überwachter CBI, d.h. der CBI-Rate,
  - der Anzahl aktiver Prozessorkerne und
  - der Laufzeit des gewählten ET-Pfades.

Die erreichten Ergebnisse werden jeweils mit einem konservativeren, prioritätsbasierten Ansatz verglichen. Weiterhin wird die Reaktionslatenz des Überwachungs- und Steuerungssystems untersucht.

Die Evaluation erfolgt auf Basis von parametrisierbaren, für die Evaluation entwickelten, Benchmark-Anwendungen, welche die Steuerung der durch sie ausgelösten Speicherzugriffe und somit der verursachten Interferenz an der Speicherinfrastruktur ermöglichen. Auf Basis dieser Anwendungen wird ein Basis-Evaluationsaufbau entwickelt, welcher anschließend hinsichtlich verschiedener Parameter variiert wird, um deren Einfluss auf die Performanz des vorgestellten Systems zu bestimmen.

5.4.1 Evaluationsaufbau und Parametrisierung

Im Folgenden wird der Versuchsaufbau zur Evaluation des vorgestellten Ansatzes beschrieben. Er basiert auf der in Abschnitt 5.3 vorgestellten Implementierung.

Der Evaluationsaufbau umfasst einen Echtzeit-Task sowie eine konfigurierbare Anzahl n_{BET} an Best-Effort-Tasks, welche auf der APU des Zynq UltraScale+ MPSoC in separaten Hypervisor-Partitionen ausgeführt werden.

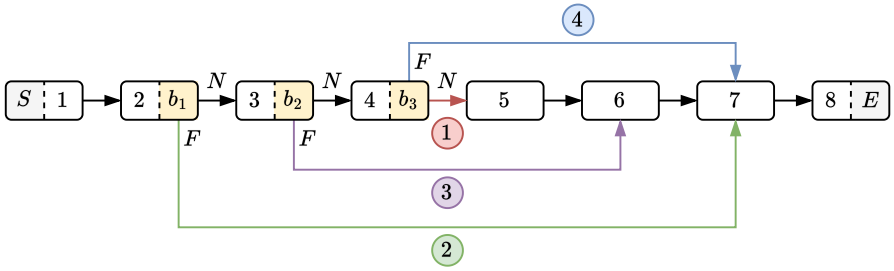


Abbildung 5.8: Kontrollflussgraph des Evaluations-ET mit $n_{CBI} = 3$. Kontrollflussblöcke werden durch nummerierte Rechtecke dargestellt, b_i bezeichnet die bedingte Verzweigungsinstruktion am Ende eines Blocks. Die vier möglichen Pfade durch den CFG sind farblich markiert.

5.4.1.1 Evaluationstasks

Der zur Evaluation genutzte Echtzeit-Task wird periodisch ausgeführt. Er enthält n_{CBI} bedingte Verzweigungsinstruktionen, welche den Kontrollfluss in Abhängigkeit der Inputdaten des Tasks bestimmen. Zwischen den CBI wird jeweils eine definierte Anzahl an Arbeitspaketen ausgeführt, welche sequenzielle Berechnungen des Tasks darstellen. Im Folgenden wird, sofern nicht abweichend beschrieben, der in Abbildung 5.8 dargestellte Echtzeit-Task zur Evaluation genutzt. Mit $n_{CBI} = 3$ CBI ergeben sich vier Ausführungspfade durch den Kontrollflussgraphen des Tasks, welche mit ① bis ④ referenziert werden.

Neben dem Echtzeit-Task ausgeführte Best-Effort-Tasks simulieren Anwendungen ohne Echtzeitanforderungen, welche parallel zum Echtzeit-Task ausgeführt werden können. Diese führen kontinuierlich Arbeitspakete aus und zeichnen die Anzahl ausgeführter Arbeitspakete auf.

Um verschiedene Anwendungscharakteristika im Hinblick auf die Speichernutzung untersuchen zu können, können in ET und BET verschiedenartige Arbeitspakete ausgeführt werden. Diese sind im Folgenden beschrieben. Jedes Arbeitspaket ist auf eine Ausführungszeit von annähernd 1 ms bei exklusiver Ausführung auf einem Kern des MCP ausgelegt.

- **CPU-intensive Anwendungen:** Ein CPU-intensives Arbeitspaket führt zyklische Berechnungen der Wurzelfunktion von Zufallszahlen aus.
- **Speicherintensive Anwendungen** Ein speicherintensives Arbeitspaket ist durch zyklisches Schreiben und Lesen eines zusammenhängenden Speicherbereichs realisiert. Um Interferenzeffekte an der Speicherinfrastruktur zu maximieren, wird ein Speicherbereich von 1 MB mit einer Schrittweite von 64 B geschrieben bzw. gelesen.

Diese Parameter entsprechen der Größe und Cache-Line-Länge des L2-Caches der Ausführungsplattform.

- **Gemischte Anwendungen (zufällig)** Um Anwendungen mit gemischter Speichernutzung abzubilden, werden zufallsgesteuert CPU- und speicherintensive Arbeitspakete ausgeführt. Die Auftrittswahrscheinlichkeit CPU- und speicherintensiver Arbeitspakete ist parametrisierbar.
- **Gemischte Anwendungen (deterministisch)** Um Echtzeit-Anwendungen mit gemischter Speichernutzung abzubilden, wurde ein Algorithmus implementiert, welcher eine deterministische Abfolge von Arbeitspaketen generiert, die sich einer parametrisierbaren Verteilung von CPU- und speicherintensiven Arbeitspaketen annähert. So ist sichergestellt, dass wiederholte Ausführungen des Echtzeit-Tasks bei gleichen Input-Parametern zur gleichen Abfolge an CPU- und speicherintensiven Arbeitspaketen führen.

In Arbeitspaketen mit parametrisierbarer CPU- und Speicherintensität gibt der Parameter M die Auftrittswahrscheinlichkeit eines speicherintensiven Arbeitspakets an. Die Auftrittswahrscheinlichkeit des CPU-intensiven Arbeitspakets beträgt somit $1 - M$. Bezogen auf einen Task T wird M_T auch als *Speicherintensität* des Tasks bezeichnet.

5.4.1.2 WCET-Analyse des Echtzeit-Tasks

Zur Parametrisierung des Schedules und Konfiguration des Execution Controllers wurden WCET-Analysen des in Abbildung 5.8 dargestellten Echtzeit-Tasks mit verschiedenen Konfigurationen hinsichtlich der Speicherintensität M_{ET} durchgeführt. Für jede Konfiguration wurden sowohl Messungen der Gesamtlaufzeit einer Taskausführung als auch der partiellen Laufzeiten für die WCET-Abschätzungen $W_{b_i,F}$ bzw. $W_{b_i,N}$ durchgeführt (vgl. Abschnitt 5.2.3). Der Echtzeit-Task wurde hierbei exklusiv auf einem Kern des MCP ausgeführt. Die Abschätzung der (partiellen) WCET erfolgte experimentell durch wiederholte exklusive Ausführung der zu analysierenden Abschnitte des Echtzeit-Tasks. Als WCET-Abschätzung wurde die maximale gemessene Ausführungszeit über 500 Iterationen zzgl. eines Sicherheitszuschlags von 5 % angenommen.

Tabelle 5.1 zeigt die maximal gemessenen Ausführungszeiten der betrachteten Pfade bzw. Pfadabschnitte des Echtzeit-Tasks pro Konfiguration sowie die daraus berechneten WCET-Abschätzungen. Es ist erkennbar, dass die maximalen Ausführungszeiten jedes Abschnitts bei Veränderung der Speicherintensität um maximal eine Millisekunde variieren. Eine genauere Analyse ergibt eine Spanne von 0.32 % bis 2.97 % des jeweiligen Maximalwerts. Aufgrund dieser geringen Varianz wurde für die weitere Evaluation für jeden Abschnitt eine feste, von der Speicherintensität unabhängige WCET der Echtzeit-Anwendung angenommen, welche in der Spalte WCET dargestellt ist. Als längster Pfad bestimmt ①

Tabelle 5.1: Maximale gemessene Ausführungszeiten in Abhängigkeit der Speicherintensität M_{ET} und (partielle) WCET-Abschätzungen der Pfade durch den Echtzeit-Task sowie relevanter Ausführungsabschnitte (vgl. Abbildung 5.8)

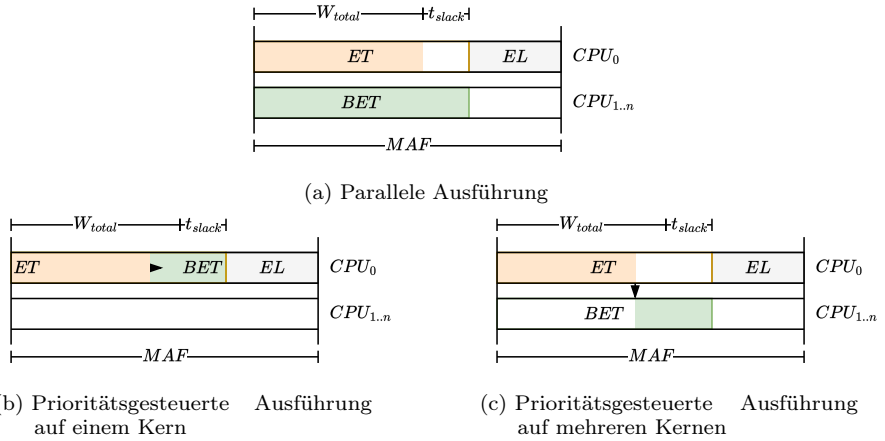
Pfad(abschnitt)	**Maximale Ausführungszeit [ms] nach M_{ET}**						**WCET**
	0 %	**20 %**	**40 %**	**60 %**	**80 %**	**100 %**	**[ms]**
①	133	134	133	133	133	134	141
②	28	29	28	28	28	29	30
③	53	54	54	53	53	54	57
④	83	84	83	83	83	84	88
$W_{b_1,F} = W_{b_3,F}$	25	26	26	26	25	26	27
$W_{b_1,N}$	130	131	130	130	130	131	137
$W_{b_2,F}$	45	46	46	46	45	46	48
$W_{b_2,N}$	125	126	125	125	125	126	132
$W_{b_3,N}$	75	76	75	75	75	76	79

die WCET einer gesamten Taskausführung. Diese beträgt somit $W_{total} = 141\,\text{ms}$.

5.4.1.3 Definition des Schedules und Referenzansatz

Die Schedule-Konfiguration des Hypervisors erfolgte auf Basis der WCET-Analyse des Echtzeit-Tasks. Abbildung 5.9a stellt den Schedule zur Umsetzung des vorgestellten Konzepts dar. Auf dem ersten CPU-Kern (CPU_0) wird der ET ausgeführt. Diesem ist ein Zeitfenster der Länge $D = W_{total} + t_{slack}$ zugewiesen, dessen Ende der Deadline des ET entspricht. Parallel zu diesem Zeitfenster sind auf den anderen CPU-Kernen BET-Zeitfenster der gleichen Länge definiert. Die Anzahl paralleler BET-Zeitfenster ist durch n_{BET} bestimmt. Die Ausführung der BET innerhalb dieser Zeitfenster wird nach dem vorgestellten Konzept gesteuert. Im Anschluss an das ET- und BET-Zeitfenster wird ein Evaluations-Task (Evaluationslogik, EL) ausgeführt, welcher prüft, ob die Ausführung des ET innerhalb des Zeitfensters abgeschlossen wurde, und die Anzahl der von den BET verarbeiteten Arbeitspaketen erfasst. Die Zeitfenster bilden zusammen das Major Frame (MAF), welches zyklisch wiederholt wird.

Der Trace-Monitoring-Ansatz wird mit einem prioritätsbasierten Referenzansatz verglichen. Im prioritätsbasierten Ansatz werden ET und BET sequenziell ausgeführt. Für $n_{BET} = 1$ erfolgt dies, indem ET und ein BET dem gleichen Zeitfenster zugewiesen werden und dem ET eine höhere Priorität zugewiesen wird als dem BET. Der Scheduler des Hypervisors führt somit den ET zunächst exklusiv aus. Sobald der ET abgeschlossen ist ($t_{e,i}$ in Abbildung 5.6), beginnt die Ausführung des BET für die verbleibende Zeit



(a) Parallele Ausführung

(b) Prioritätsgesteuerte Ausführung auf einem Kern

(c) Prioritätsgesteuerte Ausführung auf mehreren Kernen

Abbildung 5.9: Konfigurationen des Hypervisor-Schedules für das Trace-Monitoring-basierte Scheduling bzw. ein prioritätsbasiertes Scheduling.

des Ausführungszeitfensters (vgl. Abbildung 5.9b). Für $n_{BET} > 1$ sind BET-Zeitfenster parallel zum ET-Zeitfenster auf den weiteren Kernen des MCP angelegt. Die Ausführung der BET innerhalb dieser Zeitfenster wird zur Laufzeit so gesteuert, dass der ET zunächst exklusiv ausgeführt wird und nach dessen Ende die Ausführung der BET auf allen BET-Kernen beginnt (vgl. Abbildung 5.9c).

5.4.1.4 Parametervariation

Auf Basis des beschriebenen Evaluationsaufbaus wurden verschiedene Evaluationsszenarien untersucht. Zunächst wurde unter Nutzung des in Abbildung 5.8 dargestellten ET ($n_{CBI} = 3$) und einem Best-Effort-Task ($n_{BET} = 1$) der sich aus der Variation der folgenden Parameter ergebende Parameterraum untersucht:

- Speicherintensität des Echtzeit-Tasks M_{ET}: Die Speicherintensität des Echtzeit-Tasks wurde von 0 % bis 100 % in Schritten von 20 % variiert.
- Speicherintensität des Best-Effort-Tasks M_{BET}: Die Speicherintensität des Best-Effort-Tasks wurde von 0 % bis 100 % in Schritten von 20 % variiert.
- Deadline des Echtzeit-Tasks: Die Deadline des Echtzeit-Tasks ist als $D = W_{total} + t_{slack}$ definiert. t_{slack} wurde hierbei von 0 ms bis 20 ms in Schritten von 5 ms variiert.

Für jede Parameterkonfiguration wurden 80 Messungen durchgeführt, wobei jeder Pfad durch die Echtzeit-Anwendung gleich häufig durchlaufen wurde. Somit wurden jeweils 20 Messungen pro Pfad durch die Echtzeit-Anwendung aufgenommen.

Tabelle 5.2: Erfolgsquote hinsichtlich der Einhaltung der Deadline bei unkontrollierter paralleler Ausführung des ET und BET in Abhängigkeit der Speicherintensität

Speicherintensität [%]	**Erfolgsrate [%]**					
$\mathbf{M_{BET}}$ \ $\mathbf{M_{ET}}$	**0 %**	**20 %**	**40 %**	**60 %**	**80 %**	**100 %**
0 %	100	100	100	100	100	100
20 %	100	75	75	74	66	73
40 %	100	75	51	50	50	51
60 %	100	65	50	38	33	50
80 %	100	50	50	25	25	28
100 %	100	50	43	25	25	18

5.4.2 Einhaltung der Deadline

Die zentrale Anforderung an den vorgestellten Ansatz ist es, das Einhalten der Deadline von Echtzeit-Tasks sicherzustellen. Im Rahmen aller Untersuchungen wurde daher erfasst, ob die Deadline des Echtzeit-Tasks eingehalten wurde.

Im Folgenden wird die Einhaltung der Deadline unter Nutzung des Trace-Monitoring-Ansatzes, des prioritätsbasierten Referenzansatzes sowie der unkontrollierten parallelen Ausführung von Tasks gegenübergestellt. Dies erfolgt auf Basis der in Abschnitt 5.4.1.4 beschriebenen Messreihe, wobei für die Auswertung nur Messungen mit $t_{slack} = 0\,\text{ms}$ betrachtet wurden.

Für den Fall der unkontrollierten parallelen Ausführung von ET und BET zeigt Tabelle 5.2 die Erfolgsquote E der Einhaltung der Deadlines in Abhängigkeit der Speicherintensität der beiden Tasks. Die Erfolgsquote berechnet sich hierbei als

$$E = \frac{n_e}{n_e + n_n}$$

mit der Anzahl der Messungen n_e, bei denen die Deadline eingehalten wurde, und der Anzahl der Messungen n_n, bei denen die Deadline nicht eingehalten wurde. Es ist erkennbar, dass die Deadline in allen Fällen eingehalten wurde, in denen mindestens ein Task rein CPU-intensive Arbeitspakete ausführte ($M = 0\,\%$). Dies ist nachvollziehbar, da zur Ausführung des CPU-intensiven Arbeitspakets überwiegend lokale Ressourcen des CPU-Kerns genutzt werden und somit eine geringe Abhängigkeit der Ausführungszeit von der Verfügbarkeit geteilter Ressourcen besteht. Mit zunehmender Speicherintensität beider Tasks steigt die Häufigkeit verpasster Deadlines. Dies ist durch die zunehmende parallele Nutzung der Speicherinfrastruktur erklärbar, welche zu Interferenzeffekten

in Form von Verzögerungen der Ausführung der beiden Tasks führt. Da die Deadline im untersuchten Aufbau der Single-Core-WCET des Echtzeit-Tasks entspricht, kann bereits eine geringfügige Verzögerung des Echtzeit-Tasks zum Überschreiten der Deadline führen. Dies ist insbesondere dann der Fall, wenn der längste Ausführungspfad durch die Echtzeit-Anwendung gewählt wird.

Unter Nutzung des Trace-Monitoring-Ansatzes sowie des prioritätsbasierten Ansatzes wurde die Deadline unabhängig von Speicherintensität und gewähltem Pfad in allen Fällen eingehalten ($E = 100\,\%$).

Die Untersuchungen bestätigen, dass bei unkontrollierter paralleler Ausführung von ET und BET auf COTS Multi-Core-Prozessoren Interferenzeffekte an der Speicherinfrastruktur auftreten können, die zu einem Überschreiten festgelegter Deadlines führen können. Das Ausmaß der Verzögerung ist dabei von der Speicherintensität der parallel ausgeführten Anwendungen abhängig. Die Untersuchung deutet darauf hin, dass der vorgestellte Ansatz die Einhaltung der Deadline sicher stellen kann, obwohl er die zeitweise parallele Ausführung des ET und des BET zulässt. Auch der prioritätsbasierte Ansatz führte zur Einhaltung der Deadline, wobei dieser keine parallele Ausführung von Tasks neben dem Echtzeit-Task zulässt.

5.4.3 Zeitanteil paralleler Ausführung

Im Folgenden wird der mit dem vorgestellten Ansatz erreichbare Zeitanteil paralleler Ausführung in Abhängigkeit relevanter Systemparameter untersucht. Hierzu erfolgt zunächst eine theoretische Betrachtung der erreichbaren BET-Ausführungszeit, anhand welcher die gemessenen Zusammenhänge nachvollzogen werden können.

Während in Abschnitt 5.4.3.2 die Abhängigkeit des BET-Durchsatzes von den individuellen Speicherintensitäten M_{ET} und M_{BET} der parallelen Tasks untersucht wird, erfolgt die Untersuchung der weiteren Zusammenhänge in Abhängigkeit der *gemeinsamen Speicherintensität* $M = M_{ET} = M_{BET}$. Hierzu werden nur Messungen betrachtet, in welchen $M_{ET} = M_{BET}$ gilt.

5.4.3.1 Grundsätzliche Überlegungen zur erreichbaren BET-Ausführungszeit

Die im beschriebenen Evaluationsszenario mit dem Trace-Monitoring-Ansatz erreichbare BET-Ausführungszeit während eines MAF setzt sich aus einer interferenzunabhängigen und einer interferenzabhängigen Komponente zusammen.

Die interferenzunabhängige Komponente entsteht durch parallele Ausführung, welche möglich wird durch

- das initiale Ausführungszeitbudget t_{slack},
- eine schnellere Ausführung des ET als im Rahmen der WCET-Abschätzung angenommen. Diese kann wiederum verursacht sein durch
  - die Wahl eines kurzen Ausführungspfades CFG (② bis ④ in Abbildung 5.8) oder
  - pessimistische Annahmen bei der WCET-Schätzung.

Die interferenzunabhängige Komponente der BET-Ausführungszeit wird in jedem Fall erreicht. Dies erfolgt insbesondere auch im (theoretischen) Extremfall, in welchem die parallele Ausführung der BET den ET-Fortschritt vollständig blockiert. In diesem Extremfall wird ein einmal vergebenes Budget p für parallele Ausführung verbraucht, woraufhin der BET unterbrochen wird und die Ausführung des ET fortgesetzt wird.

Die interferenzabhängige Komponente entsteht in Situationen, in welchen der ET trotz paralleler Ausführung des BET einen Ausführungsfortschritt macht. Durch diesen Ausführungsfortschritt wird, in einem ansonsten gleichen Szenario, die nächste CBI zu einem um ein Δt früheren Zeitpunkt erreicht als im genannten Extremfall. Dieser Zeitgewinn Δt kann bei Erreichen der CBI dem Ausführungszeitbudget p zugeschlagen werden und wird somit Teil der erreichten BET-Ausführungszeit.

5.4.3.2 Abhängigkeit der BET-Ausführungszeit von der Speicherintensität

Im Folgenden wird die Abhängigkeit der erreichten BET-Ausführungszeit von den Speicherintensitäten M_{ET} und M_{BET} und somit von der Stärke der auftretenden Interferenzeffekte zwischen den Tasks untersucht. Dies erfolgt auf Basis der in Abschnitt 5.4.1.4 beschriebenen Messreihe mit der Einschränkung, dass $t_{slack} = 0\,\text{ms}$ gewählt wurde. Abbildung 5.10 stellt die gemessenen mittleren BET-Ausführungszeiten $\bar{t}_{BET}$ für den Trace-Monitoring-Ansatz und den prioritätsbasierten Ansatz gegenüber.

Zunächst ist erkennbar, dass der Trace-Monitoring-Ansatz eine im Vergleich zum prioritätsbasierten Ansatz längere Ausführung des BET ermöglicht. In den untersuchten Szenarien werden mit dem prioritätsbasierten Ansatz mittlere BET-Ausführungszeiten von 67 ms bis 68 ms erreicht. Der Trace-Monitoring-Ansatz führt in den gleichen Szenarien zu Werten zwischen 82 ms und 95 ms. Dies entspricht einer Steigerung von 22 % bis 41 % gegenüber dem prioritätsbasierten Ansatz (vgl. Abbildung 5.11).

Weiterhin zeigt sich, dass $\bar{t}_{BET}$ bei Veränderung von M_{ET} und M_{BET} mit dem prioritätsbasierten Ansatz annähernd konstant bleibt, während sie mit dem Trace-Monitoring-Ansatz mit zunehmender Speicherintensität beider Tasks deutlich sinkt. Für den prioritätsbasierten Ansatz lässt sich dieser Effekt durch die sequenzielle Ausführung von ET und BET erklären. Hierbei treten aufgrund der exklusiven Nutzung der Rechenressourcen

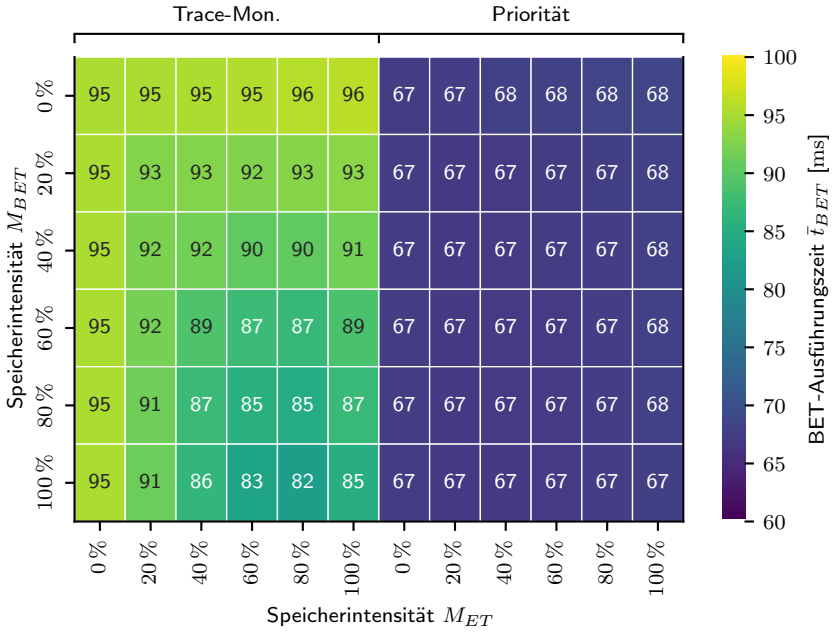


Abbildung 5.10: Mittlere BET-Ausführungszeit in Abhängigkeit der Speicherintensitäten M_{ET} und M_{BET} für den Trace-Monitoring-Ansatz und den prioritätsbasierten Ansatz.

keine signifikanten Interferenzeffekte auf. Beim Umschalten zwischen den Tasks können Verzögerungen aufgrund geteilt genutzter Caches auftreten, diese haben im untersuchten Szenario jedoch nur einen geringen Einfluss. Stattdessen sind die in Tabelle 5.1 dargestellten Unterschiede in der Ausführungszeit speicher- und CPU-intensiver Arbeitspakete als Ursache für die leichte Varianz der BET-Ausführungszeit des ET anzunehmen.

Für den Trace-Monitoring-Ansatz lässt sich eine deutliche Abhängigkeit der BET-Ausführungszeit von der Speicherintensität und somit vom Ausmaß der Interferenzeffekte beobachten. Sie ist durch die in Abschnitt 5.4.3.1 genannten Überlegungen zur interferenzabhängigen Komponente der BET-Ausführungszeit zu erklären. Weiterhin ist erkennbar, dass das Minimum der mittleren BET-Ausführungszeit nicht wie zunächst zu erwarten bei $M_{ET} = M_{BET} = 100\,\%$ liegt, sondern bei $M_{ET} = 80\,\%$ und $M_{BET} = 100\,\%$ erreicht wird. Betrachtet man $\bar{t}_{BET}$ in Abhängigkeit von M_{BET} bzw. M_{ET}, sind folgende Zusammenhänge beobachtbar: Mit steigenden M_{BET} sinkt $\bar{t}_{BET}$ bei festem M_{CT} stetig. Dies ist aufgrund der zunehmenden Intensität der Interferenzeffekte zu erwarten. Betrachtet man $\bar{t}_{BET}$ bei festem $M_{BET} > 20\,\%$, ist nach dem zu erwartenden initialen Abfallen von $\bar{t}_{BET}$

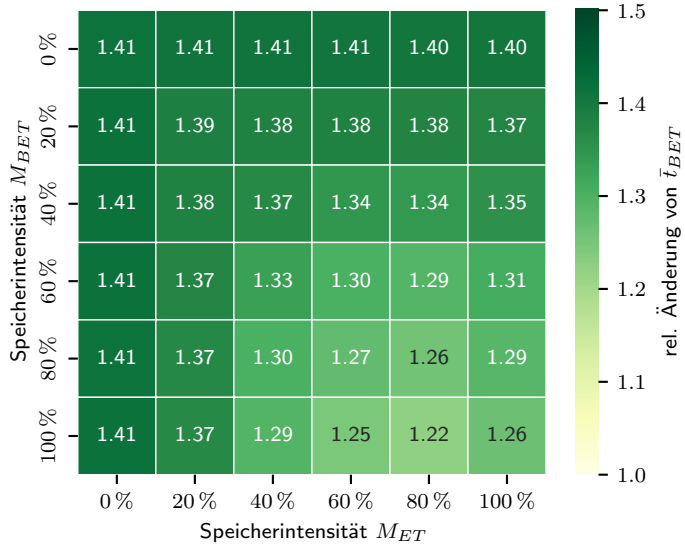


Abbildung 5.11: Relative Änderung der BET-Ausführungszeit gegenüber dem prioritätsbasierten Ansatz in Abhängigkeit der Speicherintensität.

mit steigendem M_{ET} beim Übergang von $M_{ET} = 80\,\%$ zu $M_{ET} = 100\,\%$ ein Anstieg von $\bar{t}_{BET}$ erkennbar. Die Ursache dieses Effekts wurde nicht abschließend untersucht, da er sich nicht negativ auf die Effektivität des Trace-Monitoring-Ansatzes auswirkt. Als mögliche Ursachen werden Eigenschaften der Ausführungshardware sowie ihr Einfluss auf den Zeitpunkt des Erreichens von überwachten CBI vermutet. Relevant können hierbei Eigenschaften sein, durch welche kontinuierliche Speicherzugriffe gegenüber zeitlich unterbrochenen Zugriffen effizienter verarbeitet werden (Burst-Funktionen) oder bei der Arbitrierung priorisiert werden.

5.4.3.3 Abhängigkeit der BET-Ausführungszeit vom initialen Ausführungszeitbudget und der ET-Pfadlänge

Im Folgenden wird zunächst die Abhängigkeit der mittleren BET-Ausführungszeit $\bar{t}_{BET}$ vom initialen Ausführungszeitbudget t_{slack} untersucht. Die Betrachtung findet unter Berücksichtigung des gewählten ET-Pfades statt. Es erfolgt ein Vergleich zwischen dem Trace-Monitoring-basierten Ansatz und dem prioritätsbasierten Ansatz.

Abbildung 5.12 zeigt $\bar{t}_{BET}$ in Abhängigkeit von t_{slack} sowie des gewählten Pfades. Für den Trace-Monitoring-Ansatz ist $\bar{t}_{BET}$ jeweils für den Fall starker und schwacher Interferenz

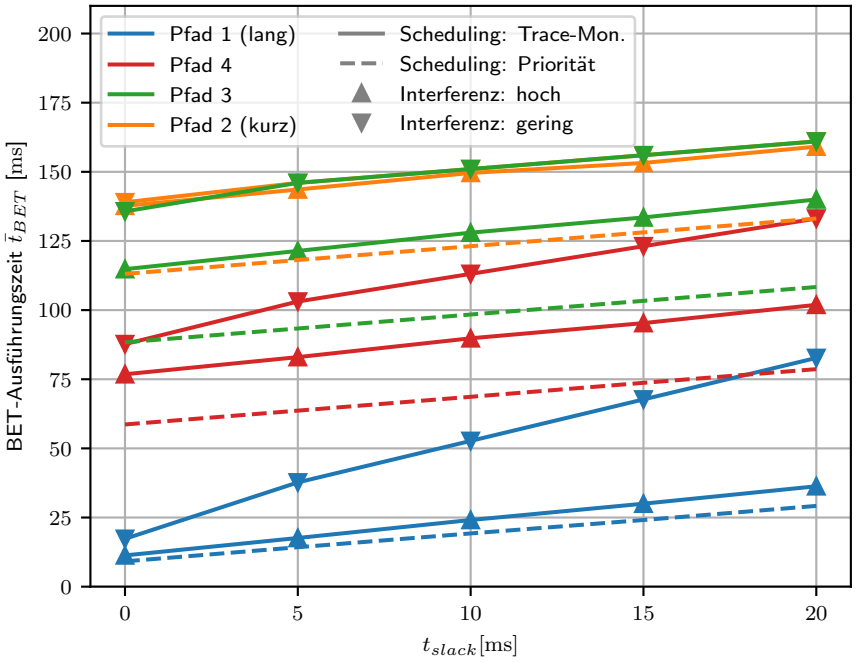


Abbildung 5.12: Mittlere BET-Ausführungszeit in Abhängigkeit des gewählten ET-Pfades und von t_{slack} bei hoher ($M = 100\,\%$) bzw. geringer ($M = 0\,\%$) Interferenzstärke.

getrennt dargestellt.

Zunächst ist erkennbar, dass die erreichte BET-Ausführungszeit stark vom gewählten ET-Pfad abhängt. Dies ist der unterschiedlichen Ausführungszeit der Pfade geschuldet (vgl. Tabelle 5.1). Durch die feste Dauer des ET-Zeitfensters führen kurze ET-Pfade zu einer längeren BET-Ausführung als lange Pfade.

Weiterhin zeigt sich eine Abhängigkeit zwischen $\bar{t}_{BET}$ und t_{slack}. Sowohl der prioritätsbasierte als auch der Trace-Monitoring-Ansatz führen zu einer steigenden $\bar{t}_{BET}$ mit steigendem t_{slack}. Für den prioritätsbasierten Ansatz erscheint dieser Zusammenhang linear. Dies lässt sich anhand der sequenziellen Ausführung des ET und BET nachvollziehen. Für den prioritätsbasierten Ansatz gilt $t_{BET} = D - t_{ET}$ mit der Ausführungszeit des Echtzeit-Tasks t_{ET} bzw. des Best-Effort-Tasks t_{BET} und der Länge D des ET-Zeitfensters. Da sich eine Änderung von t_{slack} nicht auf t_{ET} auswirkt und die Länge des Zeitfensters als $D = W_{total} + t_{slack}$ definiert ist, führt eine Erhöhung von t_{slack} um Δt_{slack} zu einer entsprechenden Erhöhung von t_{BET} um $\Delta t_{BET} = \Delta t_{slack}$.

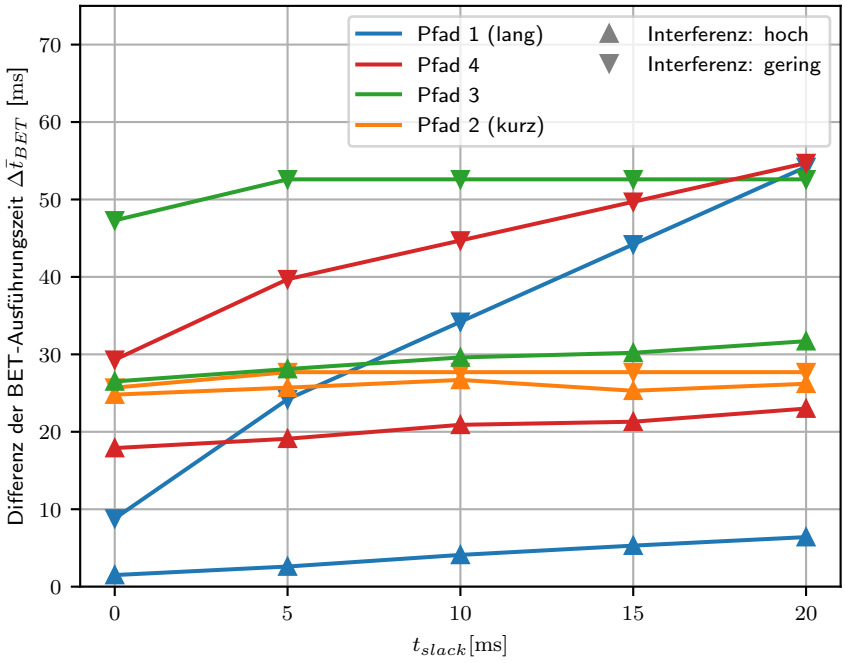


Abbildung 5.13: Differenz der erreichbaren mittleren BET-Ausführungszeit des Trace-Monitoring-Ansatzes gegenüber dem prioritätsbasierten Ansatz in Abhängigkeit von t_{slack} für verschiedene ET-Pfade bei hoher ($M = 100\,\%$) bzw. geringer ($M = 0\,\%$) Interferenzstärke.

Der Trace-Monitoring-Ansatz führt in den betrachteten Szenarien zu höheren mittleren BET-Ausführungszeiten als der prioritätsbasierte Ansatz. Dies ist auf Basis der in Abschnitt 5.4.3.1 angestellten Überlegungen zur interferenzabhängigen Komponente der Ausführungszeit nachvollziehbar: Während mit dem prioritätsbasierten Ansatz durch die sequenzielle Ausführung von ET und BET nur die interferenzunabhängige Komponente der BET-Ausführungszeit erreicht wird, erreicht der Trace-Monitoring-Ansatz durch parallele Ausführung längere BET-Ausführungszeiten.

Im Gegensatz zum prioritätsbasierten Ansatz ist beim Trace-Monitoring-Ansatz zu erkennen, dass die Auswirkung der Wahl von t_{slack} auf $\bar{t}_{BET}$ von weiteren Parametern abhängt. Zur Veranschaulichung dieser Zusammenhänge ist in Abbildung 5.13 die Differenz $\Delta\bar{t}_{BET}(t_{slack}) = \bar{t}_{BET,T}(t_{slack}) - \bar{t}_{BET,P}(t_{slack})$ der mittleren BET-Ausführungszeit zwischen dem Trace-Monitoring-Ansatz und dem prioritätsbasierten Ansatz für Situationen starker bzw. schwacher Interferenz dargestellt. Durch die Betrachtung der Differenz

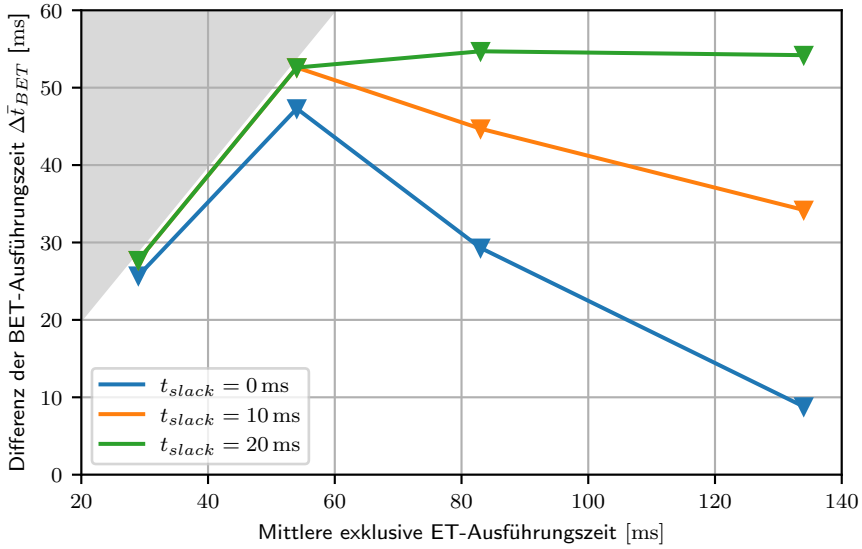


Abbildung 5.14: Differenz der erreichbaren mittleren BET-Ausführungszeit des Trace-Monitoring-basierten Ansatzes gegenüber dem prioritätsbasierten Ansatz in Abhängigkeit der exklusiven Ausführungszeit des gewählten ET-Pfads für verschiedene t_{slack} bei geringer Interferenzstärke ($M = 0\,\%$).

ist die Darstellung um die interferenzunabhängige Komponente der BET-Ausführungszeit bereinigt. Die somit dargestellte, interferenzabhängige Komponente $\Delta\bar{t}_{BET}$ zeigt wie erwartet eine starke Abhängigkeit von der Speicherintensität der Anwendungen bzw. der resultierenden Stärke der Interferenzeffekte als auch vom gewählten ET-Pfad und von t_{slack}.

Für den Fall starker Interferenz ist erkennbar, dass sich $\Delta\bar{t}_{BET}$ nur in geringem Maße mit t_{slack} ändert. Das legt nahe, dass die parallele Ausführung des BET die Ausführung des ET zwar stark verzögert, jedoch nicht komplett blockiert. Eine Erhöhung von t_{slack} um Δt_{slack} verlängert die Dauer paralleler Ausführung um $\Delta t_{slack} + \Delta t_{iak}$, wobei Δt_{slack} eine Änderung der interferenzunabhängigen Komponente darstellt und somit in Abbildung 5.13 nicht erkennbar ist. Δt_{iak} stellt die Änderung der interferenzabhängigen Ausführungszeitkomponente dar, welche sich durch die zusätzliche parallele Ausführungszeit ergibt. Aufgrund der starken Interferenz ist diese gering.

Für den Fall schwacher Interferenz ist für verschiedene Pfade eine unterschiedlich starke Abhängigkeit zwischen $\Delta\bar{t}_{BET}$ und t_{slack} zu beobachten. Bei Pfaden mit langer Ausführungszeit ① ist eine starke Abhängigkeit beobachtbar, bei kurzen Pfaden ② scheint die

interferenzabhängige BET-Ausführungszeitkomponente nahezu unabhängig von t_{slack}. Eine Betrachtung dieser Komponente in Abhängigkeit der Ausführungszeit des gewählten ET-Pfades (siehe Abbildung 5.14) zeigt diesen Zusammenhang deutlich: Eine Veränderung von t_{slack} zwischen 0 ms und 20 ms führt bei geringer Interferenzstärke zu einer Änderung der interferenzabhängigen BET-Ausführungszeitkomponente um 1.4 ms für den kürzesten Pfad bzw. 45.4 ms für den längsten Pfad.

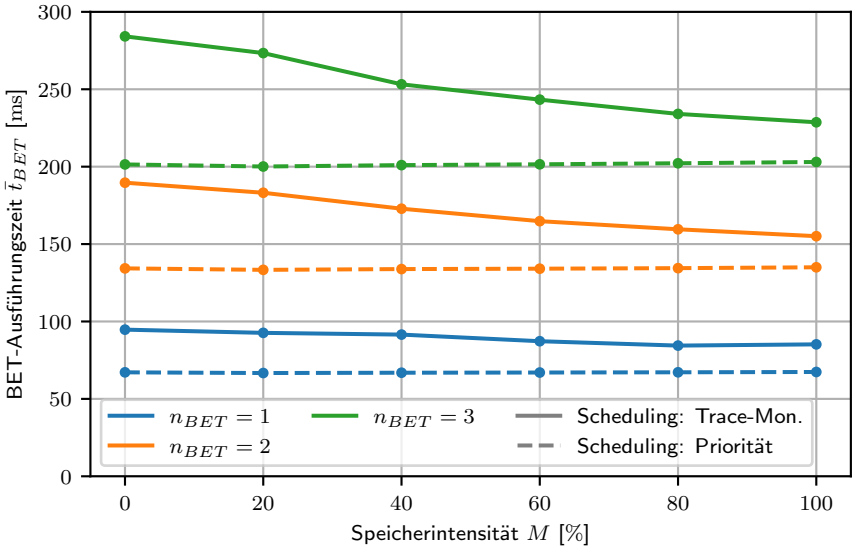
Der beobachtete Effekt lässt sich wie folgt nachvollziehen: Im Fall geringer Interferenz verzögert die parallele Ausführung von BET und ET die ET-Ausführung nur geringfügig. Ein einmal erreichtes Zeitbudget p, welches parallele Ausführung ermöglicht, kann auf Basis dieses ET-Fortschritts regelmäßig wieder aufgefüllt werden. Dies geschieht jedoch nur an einer überwachten CBI. Der Zeitanteil paralleler Ausführung zwischen zwei überwachten CBI wird durch das Verhältnis von p und dem zeitlichen Abstand der CBI bestimmt. Dauerhafte parallele Ausführung wird erreicht, wenn das p ausreicht, um die Zeit bis zum Erreichen der nächsten überwachten CBI zu überbrücken. Die Wahl von t_{slack} definiert den Initialwert von p und somit das Budget welche zu Beginn des ET-Zeitfensters für parallele Ausführung zur Verfügung steht. Wird an einer überwachten CBI der kürzere Ausführungspfad gewählt, wird der daraus entstehende (Worst-Case-)Zeitgewinn dem Budget p zugeschlagen. Dies erhöht die Dauer paralleler Ausführung zwischen zwei überwachten CBI. Wird stattdessen der lange Pfad gewählt, entfällt diese Budgeterhöhung. Dementsprechend hat t_{slack} für längere Pfade einen stärkeren Einfluss auf den Zeitanteil paralleler Ausführung zwischen überwachten CBI als für kurze Pfade.

Eine obere Grenze der interferenzabhängigen Komponente der BET-Ausführungszeit stellt die Ausführungszeit des ET dar. Ist eine dauerhafte parallele Ausführung von ET und BET möglich, entspricht die interferenzabhängige BET-Ausführungszeitkomponente der Laufzeit des ET. Dies ist in Abbildung 5.14 in grau dargestellt.

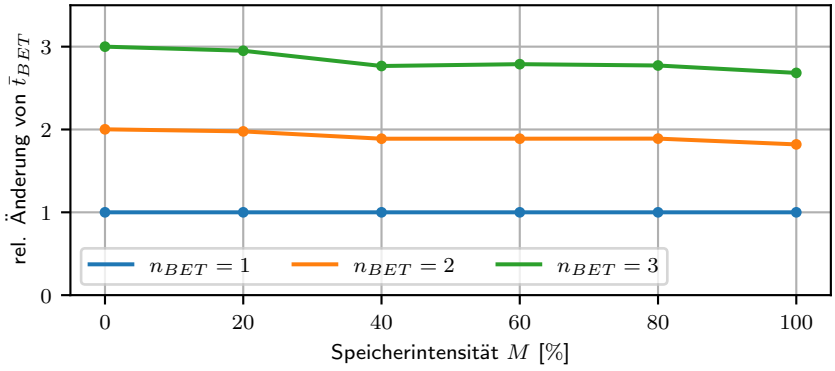
5.4.3.4 Abhängigkeit der BET-Ausführungszeit von der Anzahl genutzter Prozessorkerne

Im Folgenden wird die Abhängigkeit der erreichbaren BET-Ausführungszeit von der Anzahl der Prozessorkerne zur parallelen Ausführung von BET-Tasks betrachtet. Hierzu wurde der in Abschnitt 5.4.1 vorgestellte Versuchsaufbau genutzt, wobei $t_{slack} = 0\,\text{ms}$ gewählt wurde. Die Anzahl parallel ausgeführter BET-Tasks wurde zwischen $n_{BET} = 1$ und $n_{BET} = 3$ variiert, wobei für jede Konfiguration die Auswirkung der Speicherintensität der ET- und BET-Tasks auf die BET-Ausführungszeit untersucht wurde. Es wird die Summe der Ausführungszeiten aller parallel ausgeführten BET-Tasks betrachtet.

Abbildung 5.15 zeigt die BET-Ausführungszeit pro MAF in Abhängigkeit von n_{BET} sowie der Speicherintensität der Tasks. Für den prioritätsbasierten Ansatz ist erkennbar, dass die erreichte BET-Ausführungszeit von der Speicherintensität der Tasks unabhängig



(a) BET-Ausführungszeit für prioritätsbasierten und Trace-Monitoring-Ansatz



(b) Relative Änderung der BET-Ausführungszeit $\bar{t}_{BET}(n_{BET})/\bar{t}_{BET}(1)$ für den Trace-Monitoring-Ansatz

Abbildung 5.15: BET-Ausführungszeit in Abhängigkeit der Interferenzstärke und der Anzahl parallel ausgeführter BET n_{BET}

ist. Dies resultiert aus der sequenziellen Ausführung von ET und BET. Zwar können Interferenzeffekte zwischen parallel ausgeführten BET auftreten, diese beeinflussen jedoch nur die Ausführungsgeschwindigkeit der BET. Sie beeinflussen nicht die Zeitdauer, welche zu ihrer Ausführung zur Verfügung steht. Hieraus folgt, dass die erreichte BET-Ausführungszeit im prioritätsbasierten Fall linear mit der Anzahl parallel ausgeführter BET-Tasks n_{BET} und somit der Anzahl genutzter CPU-Kerne steigt. Für den auf $\bar{t}_{BET}(n_{BET} = 1)$ normierten Skalierungsfaktor der BET-Ausführungszeit in Abhängigkeit der Anzahl parallel ausgeführter BET-Tasks

$$S = \frac{\Delta \bar{t}_{BET}}{\Delta n_{BET} \cdot \bar{t}_{BET}(n_{BET} = 1)}$$

gilt hierbei $S_P \approx 1$.

Für den Trace-Monitoring-Ansatz ist ein Einfluss von Interferenzeffekten auf die BET-Ausführungszeit beobachtbar. Wie bereits in Abschnitt 5.4.3.2 beobachtet, nimmt die BET-Ausführungszeit mit zunehmender Speicherintensität der Tasks ab. Dies lässt sich durch die parallele Ausführung von ET und BET erklären, welche zu einer interferenzabhängigen Verzögerung des ET führt. Diese beeinflusst wiederum die BET-Ausführungszeit.

Im Hinblick auf die Skalierung der BET-Ausführungszeit mit n_{BET} ist erkennbar, dass der oben genannte Skalierungsfaktor unter Nutzung des Trace-Monitoring-Ansatz mit zunehmender Speicherintensität abnimmt: $S_T = f(M)$ (vgl. Abbildung 5.15b). Dies ist aufgrund der mit n_{BET} zunehmenden Anzahl konkurrierender Zugriffe auf geteilte Ressourcen nachvollziehbar. Bei geringer Speicherintensität ($M = 0\,\%$) und somit geringer Interferenz ist hierbei wie im prioritätsbasierten Fall ein linearer Anstieg von $\bar{t}_{BET}$ in Abhängigkeit von n_{BET} mit dem Skalierungsfaktor $S_T(0\,\%) \approx 1$ erkennbar.

5.4.3.5 Abhängigkeit der BET-Ausführungszeit von der CBI-Rate

Im Folgenden wird die Abhängigkeit der erreichbaren BET-Ausführungszeit von der Häufigkeit überwachter CBIs untersucht. Hierzu wurde der in Abschnitt 5.4.1 vorgestellte ET um zusätzliche CBI erweitert. Diese wurden so im ET integriert, dass der maximale zeitliche Abstand der Ausführung aufeinanderfolgender CBI bei exklusiver Ausführung $t_{CBI} = 5\,\text{ms}$ beträgt. Durch Auswahl der zu überwachenden CBI in der Konfiguration des Execution Controller wurden drei Messszenarien erzeugt, welche bei exklusiver Ausführung einen maximalen zeitlichen Abstand überwachter CBI von 20 ms, 10 ms bzw. 5 ms aufweisen. Für diese drei Szenarien wurde die erreichbare BET-Ausführungszeit in Abhängigkeit von t_{slack} sowie der gemeinsamen Speicherintensität M untersucht.

Abbildung 5.16 zeigt die erreichte mittlere BET-Ausführungszeit $\bar{t}_{BET}$ in Abhängigkeit der CBI-Rate und von t_{slack} für minimale bzw. maximale Speicherintensität M. Hierbei

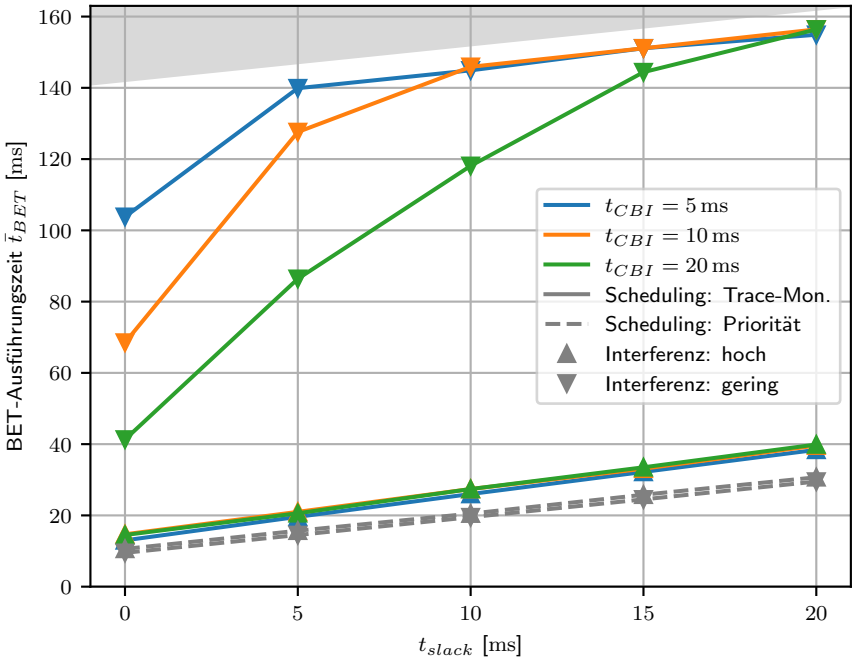


Abbildung 5.16: BET-Ausführungszeit in Abhängigkeit der CBI-Rate und von t_{slack} für den längsten ET-Pfad

sind nur Messungen des längsten untersuchten Pfades ① dargestellt, da dieser das größte Potenzial für parallele Ausführung birgt und der Einfluss der CBI-Rate somit am deutlichsten wird.

Für den prioritätsbasierten Ansatz ist erkennbar, dass die erreichte BET-Ausführungszeit unabhängig von M linear mit t_{slack} steigt. Dieser Effekt wurde bereits in Abschnitt 5.4.3.3 beobachtet und diskutiert.

Für den Trace-Monitoring-Ansatz ist eine Abhängigkeit des Einflusses der CBI-Rate auf $\bar{t}_{BET}$ von der Speicherintensität M und somit von der Stärke der auftretenden Interferenzeffekte erkennbar. Im Fall maximaler Speicherintensität ($M = 100\,\%$) steigt die erreichte BET-Ausführungszeit linear mit t_{slack}, eine Abhängigkeit von t_{CBI} ist hierbei nicht zu beobachten.

Im Fall geringer Speicherintensität ($M = 0\,\%$) ist erkennbar, dass die erreichte BET-Ausführungszeit $\bar{t}_{BET}$ mit abnehmendem t_{CBI} steigt. Theoretisch ist $\bar{t}_{BET}$ nach oben hin auf die von t_{slack} abhängige Dauer D des ET-Zeitfensters beschränkt. Diese beträgt

im betrachteten Szenario $D = 142\,\text{ms} + t_{slack}$. In den betrachteten Fällen wird dieser Grenzwert nicht vollständig erreicht, stattdessen nähert sich $\bar{t}_{BET}$ einem etwas darunter liegenden, von t_{slack} abhängigen Grenzwert an.

Im Hinblick auf die Parametrisierung von t_{slack} und t_{CBI} ist erkennbar, dass der Grenzwert bei $t_{slack} \geq t_{CBI}$ erreicht wird. Dies ist analog zur in Abschnitt 5.4.3.3 beschriebenen Überlegung zur Zeitreserve nachvollziehbar: Übersteigt das initiale Ausführungszeitbudget den zeitlichen Abstand t_{CBI} zwischen zwei überwachten CBI und wird das Ausführungszeitbudget an jeder CBI wieder mindestens auf t_{CBI} aufgefüllt, verbleibt das System dauerhaft im parallelen Ausführungsmodus.

5.4.4 Durchsatz der Best-Effort-Anwendung

Im Folgenden wird die mit dem in Abschnitt 5.2 vorgestellten Konzept erreichbare Performance der BET untersucht. Hierzu wird der mittlere Durchsatz $\bar{P}_{BET}$ der BET-Anwendungen betrachtet. Wie im vorherigen Abschnitt findet diese Evaluation im Vergleich mit dem prioritätsbasierten Ansatz statt.

Die Evaluation erfolgte auf Basis des in Abschnitt 5.4.1 beschriebenen Aufbaus. Als Maß für den Durchsatz der BET wurde die Anzahl verarbeiteter Arbeitspakete pro Zeitfenster und somit pro MAF betrachtet, wobei die Dauer eines Taskzeitfensters D entspricht (vgl. Abschnitt 5.4.1.3).

Analog zu Abschnitt 5.4.3 wird in Abschnitt 5.4.4.1 zunächst die Abhängigkeit des BET-Durchsatzes von den individuellen Speicherintensitäten M_{ET} und M_{BET} der parallelen Tasks untersucht. In den darauf folgenden Abschnitten erfolgt die Untersuchung der weiteren Zusammenhänge in Abhängigkeit der *gemeinsamen Speicherintensität* $M = M_{ET} = M_{BET}$.

5.4.4.1 Abhängigkeit des BET-Durchsatzes von der Speicherintensität

Zunächst wird der Durchsatz der Best-Effort-Anwendung in Abhängigkeit der Speicherintensität von BET und ET betrachtet. Abbildung 5.17 stellt diesen für den vorgestellten Ansatz sowie den prioritätsbasierten Ansatz für $t_{slack} = 0$ gegenüber.

Es ist erkennbar, dass der prioritätsbasierte Ansatz bei Variation von M_{ET} zu gleichbleibendem mittlerem BET-Durchsatz $\bar{P}_{BET,P}$ führt. Dies ist, wie bereits bei der Betrachtung der BET-Ausführungszeit, auf die sequenzielle Ausführung von ET und BET zurückzuführen, wodurch Interferenzeffekte vermieden werden. Im Gegensatz zu den beobachteten BET-Ausführungszeiten ist mit steigendem M_{BET} ein leichter Anstieg von $\bar{P}_{BET,P}$ zu beobachten. Dieser kann durch Unterschiede in der Implementierung der Arbeitspakete

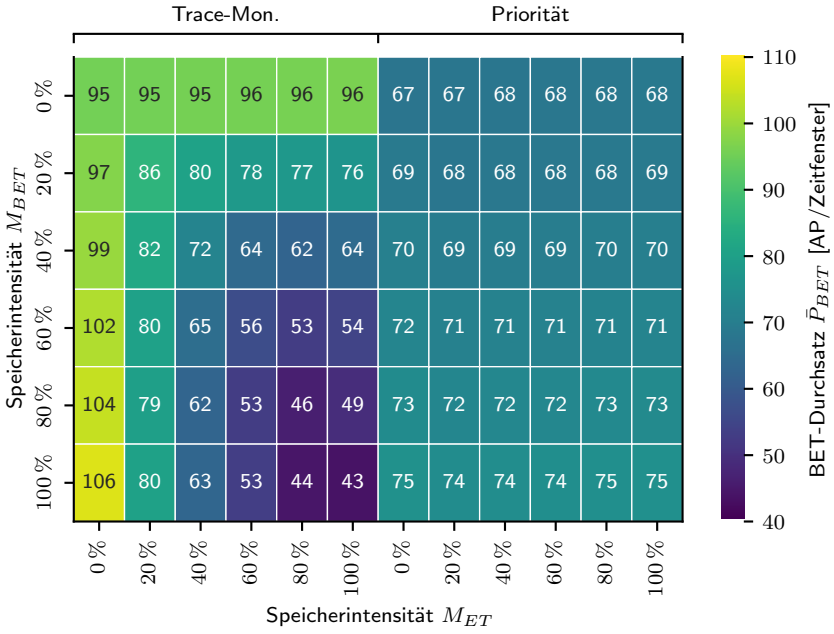


Abbildung 5.17: Mittlerer BET-Durchsatz in Abhängigkeit der Speicherintensität für den Trace-Monitoring-Ansatz und den prioritätsbasierten Ansatz

oder deren Cache-Nutzung bedingt sein, welche die Ausführungsdauer der Arbeitspakete beeinflussen.

Für den Trace-Monitoring-Ansatz zeigt sich eine starke Abhängigkeit des mittleren BET-Durchsatzes $\bar{P}_{BET,T}$ von M_{ET} und M_{BET}. Bei geringer Speicherintensität ist der gemessene mittlere Durchsatz um bis zu 42 % höher als unter Nutzung des prioritätsbasierten Referenzansatzes. Mit steigender Speicherintensität beider Tasks und somit zunehmender Interferenz nimmt $\bar{P}_{BET,T}$ ab. Bei einer Speicherintensität von $M_{ET} = M_{BET} = 40\,\%$ entspricht der erreichte mittlere Durchsatz ungefähr dem mit dem prioritätsbasierten Ansatz erreichten Durchsatz, für höhere Speicherintensitäten ($M_{ET} \geq 40\,\%$ und gleichzeitig $M_{BET} \geq 40\,\%$) liegt er unter dem des Referenzansatzes und erreicht bei maximaler Speicherintensität sein Minimum. Dort beträgt der $\bar{P}_{BET,T}$ noch 58 % des mit dem prioritätsbasierten Ansatz erreichten Durchsatzes. Das Verhältnis der BET-Performance zwischen Trace-Monitoring-Ansatz und prioritätsbasiertem Ansatz ($\bar{P}_{BET,T}/\bar{P}_{BET,P}$) ist in Abbildung 5.18 dargestellt.

Die Abhängigkeit des BET-Durchsatzes von der Interferenzstärke ist anhand zweier Effekte nachvollziehbar: Zum einen wird die Ausführung von BET mit steigender Interfe-

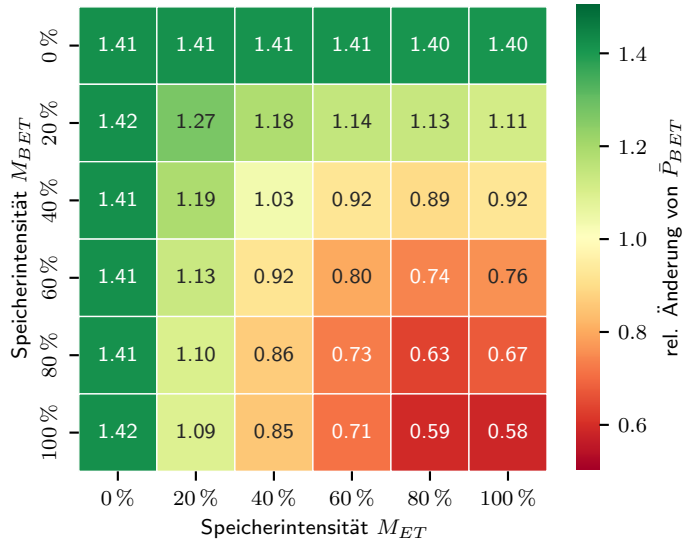


Abbildung 5.18: Relative Änderung des BET-Durchsatzes gegenüber dem prioritätsbasierten Ansatz in Abhängigkeit der Speicherintensität

renzstärke stärker verzögert, ihre Ausführungsgeschwindigkeit sinkt. Dies führt bei gleicher Ausführungsdauer t_{BET} zu einer sinkenden Anzahl verarbeiteter Arbeitspakete und somit einem sinkenden Durchsatz. Zum anderen ist auch die erreichte BET-Ausführungszeit t_{BET} selbst von der Interferenzstärke abhängig (vgl. interferenzabhängige Komponente in Abschnitt 5.4.3.1).

Im Gegensatz zur in Abschnitt 5.4.3 betrachteten BET-Ausführungszeit führt der Trace-Monitoring-Ansatz bei starker Interferenz zu einem geringeren BET-Durchsatz als der prioritätsbasierte Ansatz. Dies tritt auf, sobald sowohl die Speicherintensität des ET als auch des BET einen Grenzwert überschreiten, welcher im durchgeführten Experiment bei $M \approx 40\,\%$ liegt. Dieser Effekt ist nachvollziehbar, da der Trace-Monitoring-Ansatz auf die Maximierung der parallelen Ausführungszeit von ET und BET ausgelegt ist und somit auch den Zeitraum maximiert, in welchem die Taskausführung durch Interferenz verzögert werden kann. In früheren Untersuchungen wurde bereits gezeigt, dass die Ausführungszeit von Anwendungen bei starker Interferenz um ein Mehrfaches langsamer sein kann als bei exklusiver Ausführung [78]. In solchen Fällen führt die sequenzielle Ausführung der Anwendungen, wie hier mit dem prioritätsbasierten Ansatz, demnach zu einem schnelleren Ausführungsabschluss und somit zu einem höheren BET-Durchsatz.

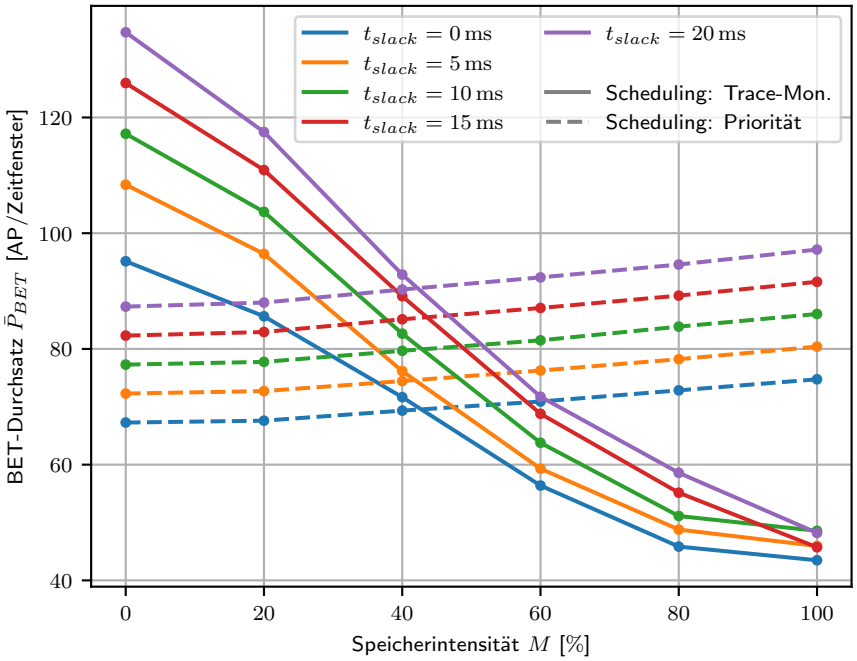


Abbildung 5.19: Mittlerer BET-Durchsatz in Abhängigkeit der gemeinsamen Speicherintensität M und von t_{slack}.

5.4.4.2 Abhängigkeit des BET-Durchsatzes vom initialen Ausführungszeitbudget

Abbildung 5.19 zeigt den mittleren BET-Durchsatz in Abhängigkeit der gemeinsamen Speicherintensität M von ET und BET für verschiedene Werte von t_{slack} und die beiden untersuchten Ansätze.

Für den prioritätsbasierten Ansatz zeigt sich näherungsweise linearer Zusammenhang zwischen $\bar{P}_{BET,P}$ und t_{slack}. Dies ist wie folgt nachvollziehbar: Wie in Abschnitt 5.4.3.3 identifiziert, führt eine Erhöhung von t_{slack} um Δt_{slack} für den prioritätsbasierten Ansatz zu einer Erhöhung der erreichten BET-Ausführungszeit t_{BET} um $\Delta t_{BET} = \Delta t_{slack}$. Im untersuchten Szenario wird der BET exklusiv ausgeführt. Da die im Rahmen der Messung genutzten Arbeitspakete auf eine Laufzeit von ca. 1 ms bei exklusiver Ausführung abgestimmt sind, ist als Folge der Erhöhung von t_{BET} um Δt_{slack} eine Erhöhung des BET-Durchsatzes P_{BET} um $\Delta P_{BET} \approx 1\,\mathrm{AP/ms} \cdot \Delta t_{slack}$ zu erwarten. Ein solcher Trend ist in Abbildung 5.19 erkennbar.

Auch für den Trace-Monitoring-Ansatz führt eine Erhöhung von t_{slack} zumeist zu ei-

ner Erhöhung der BET-Performance $\bar{P}_{BET,T}$. Da t_{slack} bei diesem Ansatz jedoch zur Maximierung der parallelen Ausführungszeit genutzt wird, ist die Auswirkung auf die BET-Performance abhängig von der Stärke der auftretenden Interferenzeffekte zwischen ET und BET. In Abbildung 5.19 ist dieser Effekt erkennbar: Mit zunehmender Speicherintensität sinkt der Einfluss von t_{slack} auf die BET-Performance. Die einzige gemessene Abweichung von diesem Trend liegt bei maximaler Speicherintensität $M = 100\,\%$, bei welcher für $t_{slack} = 10\,\text{ms}$ ein etwas höherer BET-Durchsatz erreicht wurde als für $t_{slack} = 15\,\text{ms}$.

Diese Abhängigkeit des Einflusses von t_{slack} auf $P_{BET,T}$ von der Speicherintensität lässt sich wie folgt nachvollziehen: Die Erhöhung von t_{slack} um Δt_{slack} führt zu einer Erhöhung von t_{BET}, welche aus einer interferenzunabhängigen und einer interferenzabhängigen Komponente besteht (vgl. Abschnitt 5.4.3.1). Während die interferenzunabhängige Komponente stets erreicht wird, sinkt die interferenzabhängige Komponente mit steigender Interferenzstärke. Gleichzeitig sinkt, wie in Abschnitt 5.4.4.1 diskutiert, mit zunehmender Interferenz die Ausführungsgeschwindigkeit des BET. In Kombination führen diese Effekte zu einer deutlichen Abnahme des betrachteten Einflusses von t_{slack} auf $P_{BET,T}$.

Wie in Abschnitt 5.4.3.3 gezeigt, kann durch Zugabe eines Δt_{slack} bei geringer Interferenzstärke eine Erhöhung der BET-Ausführungszeit um $\Delta t_{BET} > \Delta t_{slack}$ erreicht werden. Dementsprechend kann in Abbildung 5.19 für den Fall geringer Interferenz eine Steigerung des BET-Durchsatzes um mehr als $1\,\text{AP/ms} \cdot \Delta t_{slack}$ beobachtet werden, da $\Delta P_{BET,T} \approx 1\,\text{AP/ms} \cdot \Delta t_{BET}$ und $\Delta t_{BET} > \Delta t_{slack}$. Im untersuchten Szenario wurde hierbei eine mittlere Steigerung des BET-Durchsatzes pro Zeitfenster in Abhängigkeit von t_{slack} von $2\,\text{AP/ms}$ gemessen.

5.4.4.3 Abhängigkeit des BET-Durchsatzes von der ET-Pfadlänge und dem initialen Ausführungszeitbudget

Im Folgenden wird der Einfluss der ET-Pfadlänge auf die BET-Performance untersucht. Dies erfolgt im direkten Vergleich des Trace-Monitoring-Ansatzes mit dem prioritätsbasierten Ansatz. Abbildung 5.20 stellt hierzu die Differenz des mittleren BET-Durchsatzes $\Delta\bar{P}_{BET} = \bar{P}_{BET,T} - \bar{P}_{BET,P}$ in Abhängigkeit der mittleren Ausführungszeit des gewählten ET-Pfads sowie des gewählten t_{slack} für zwei Interferenzszenarien dar. Die Darstellung erfolgt analog zu Abbildung 5.14, die Differenz Δt_{BET} der BET-Ausführungszeit darstellt.

Zunächst ist für alle Pfadlängen der bereits in Abschnitt 5.4.4.1 identifizierte Zusammenhang zwischen Speicherintensität bzw. Interferenzstärke und dem erreichten BET-Durchsatz erkennbar: Bei geringer Interferenzstärke führt der Trace-Monitoring-Ansatz zu einem höheren BET-Durchsatz als der prioritätsbasierte Ansatz, bei hoher Interferenz zu einem geringeren Durchsatz.

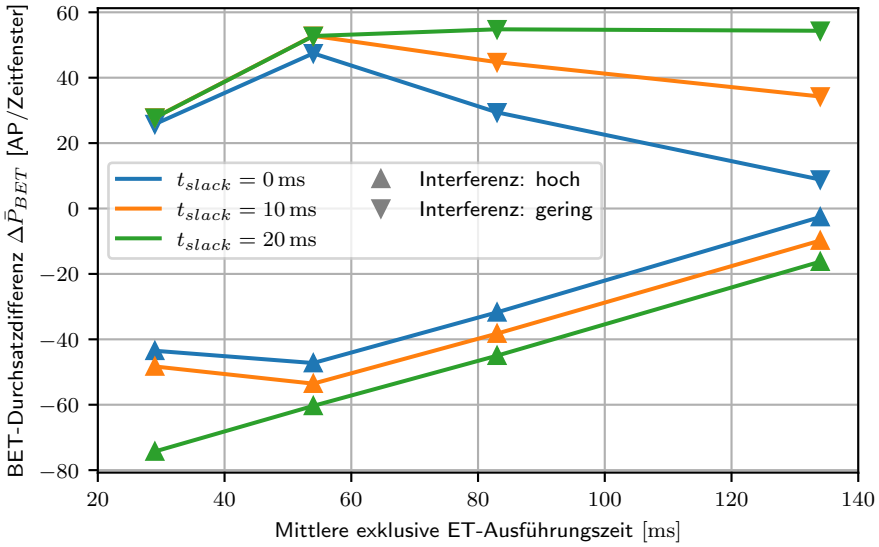


Abbildung 5.20: Differenz des erreichbaren mittleren BET-Durchsatzes des Trace-Monitoring-Ansatzes gegenüber dem prioritätsbasierten Ansatz in Abhängigkeit der exklusiven Ausführungszeit des gewählten ET-Pfades für verschiedene t_{slack} bei hoher ($M = 100\,\%$) und geringer ($M = 0\,\%$) Interferenz.

Für den Fall geringer Interferenzstärke ist weiterhin erkennbar, dass der Einfluss einer Erhöhung von t_{slack} auf den BET-Durchsatz von der mittleren ET-Ausführungszeit abhängt. Während eine Änderung von t_{slack} von 0 ms auf 20 ms bei kurzen ET-Laufzeiten nur geringen Einfluss auf die BET-Durchsatzdifferenz hat (1.95 AP/Frame für den kürzesten ET-Pfad), steigt dieser mit zunehmender ET-Ausführungszeit auf 45.5 AP/Frame für den längsten ET-Pfad an. Dies ist aufgrund des mit zunehmender ET-Laufzeit steigenden Einflusses von t_{slack} auf die BET-Ausführungszeit erklärbar, welcher bereits in Abschnitt 5.4.3.3 festgestellt wurde.

Für den Fall hoher Interferenzstärke ist auffällig, dass sich die dargestellte mittlere BET-Durchsatzdifferenz $\Delta\bar{P}_{BET}$ zwischen Trace-Monitoring-Ansatz und prioritätsbasiertem Ansatz mit zunehmender ET-Laufzeit verringert. Es liegt die Vermutung nahe, dass dies auf die bei längeren ET-Laufzeiten kürzere Zeit paralleler Ausführung zurückzuführen ist. Die Wahl kurzer ET-Pfade führt durch den Trace-Monitoring-Ansatz zu vergleichsweise langer paralleler Ausführung von ET und BET, während welcher die Ausführung des BET aufgrund der auftretenden Interferenzeffekte sehr langsam erfolgt. Im gleichen Szenario führt der prioritätsbasierte Ansatz zu einer kurzen exklusiven Ausführung des ET mit anschließender exklusiver Ausführung des BET für die verbleibende Dauer des Zeitfensters.

Bei starker Verzögerung der Taskausführung aufgrund von Interferenzeffekten kann die exklusive, sequenzielle Ausführung effizienter sein als die parallele Ausführung (vgl. Abschnitt 5.4.4.1). Da lange ET-Pfade zu einer kürzeren Phase paralleler Ausführung führen als kurze ET-Pfade, ist die BET-Durchsatzdifferenz zwischen Trace-Monitoring-Ansatz und prioritätsbasiertem Ansatz für lange ET-Pfade geringer als für kurze ET-Pfade.

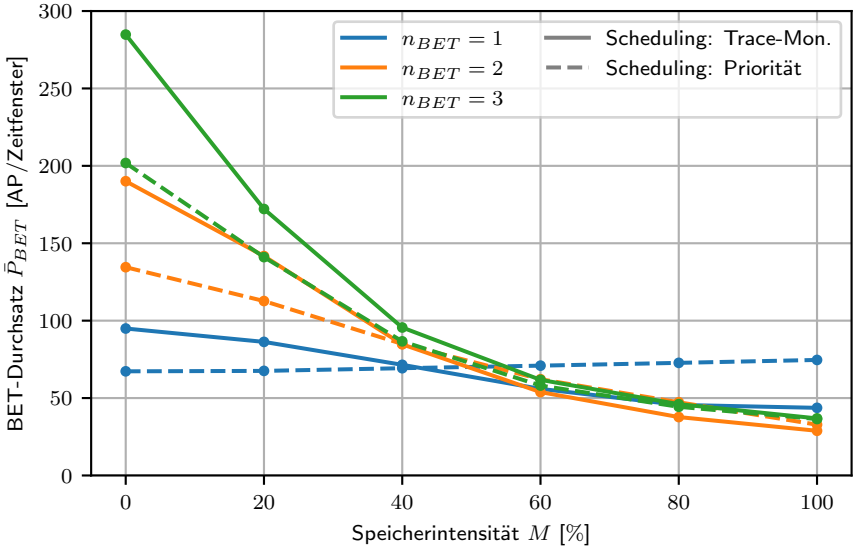
5.4.4.4 Abhängigkeit des BET-Durchsatzes von der Anzahl genutzter Prozessorkerne

Analog zu Abschnitt 5.4.3.4 wird im Folgenden die Abhängigkeit des erreichbaren BET-Durchsatzes $\bar{P}_{BET}$ von der Anzahl der Prozessorkerne zur Ausführung von BET-Tasks betrachtet. Hierzu wurde der in Abschnitt 5.4.1 vorgestellte Versuchsaufbau genutzt, wobei $t_{slack} = 0\,\text{ms}$ gewählt wurde. Die Anzahl parallel ausgeführter BET-Tasks wurde zwischen $n_{BET} = 1$ und $n_{BET} = 3$ variiert, wobei für jede Konfiguration die Auswirkung der gemeinsamen Speicherintensität M der ET- und BET-Tasks auf den BET-Durchsatz untersucht wurde. Als BET-Durchsatz wird in diesem Abschnitt die Summe der verarbeiteten Arbeitspakete aller parallel ausgeführten BET-Tasks pro Zeitfenster verstanden.

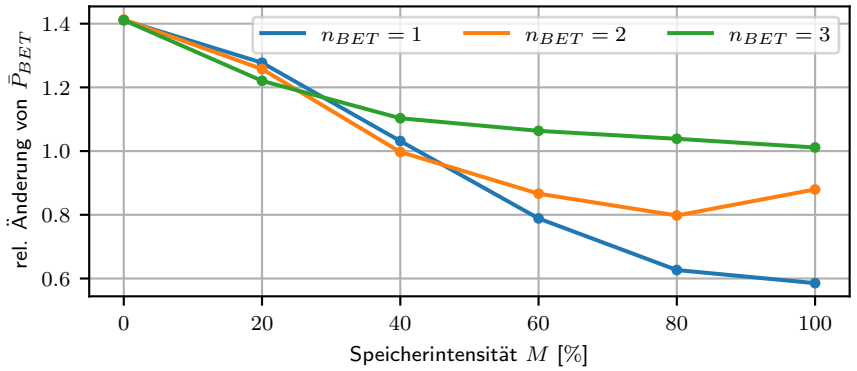
Abbildung 5.21 zeigt den BET-Durchsatz in Abhängigkeit der Anzahl parallel ausgeführter BET sowie der Speicherintensität der Tasks. Zunächst ist erkennbar, dass $\bar{P}_{BET}$ im Fall geringer Interferenz mit n_{BET} steigt. Für $M = 0\,\%$ ist dieser Zusammenhang annähernd linear. Dies ist nachvollziehbar, da die Taskausführung auf den Kernen ohne Nutzung geteilter Ressourcen weitgehend unabhängig verläuft. Weiterhin ist erkennbar, dass der Trace-Monitoring-Ansatz hier zu höheren mittleren Durchsatzwerten führt als der prioritätsbasierte Ansatz. Dies ist durch die interferenzarme Ausführung der BET in Kombination mit längerer BET-Ausführungszeit (vgl. Abschnitt 5.4.3.4) nachvollziehbar.

Mit zunehmender Speicherintensität und somit Interferenzstärke ist ein Abfallen der $\bar{P}_{BET}$-Kurven erkennbar. Für den Trace-Monitoring-Ansatz ist hierbei zunächst erkennbar, dass Einfluss zusätzlicher Kerne auf $\bar{P}_{BET,T}$ mit steigendem n_{BET} sinkt. Dies ist nachvollziehbar, da die Stärke der auftretenden Interferenzeffekte mit jedem zusätzlichen BET-Kern steigt, was wiederum zu einer sinkenden BET-Ausführungszeit und -Ausführungsgeschwindigkeit führt. Im Gegensatz zur BET-Ausführungszeit sinkt der BET-Durchsatz auch bei Nutzung des prioritätsbasierten Ansatzes mit zunehmender Interferenz, jedoch nur sofern $n_{BET} > 1$. Dies ist durch die parallele Ausführung mehrerer BET erklärbar, zwischen welchen Interferenzeffekte auftreten und zu langsamerer Taskausführung führen. Im Fall $n_{BET} = 1$ treten im prioritätsbasierten Fall aufgrund der exklusiven Ausführung des BET keine Interferenzeffekte auf, wodurch $\bar{P}_{BET,P}$ hier von der Speicherintensität unabhängig ist.

Zum Vergleich des erreichten BET-Durchsatzes zwischen den beiden Ansätzen stellt



(a) BET-Durchsatz für den Trace-Monitoring-Ansatz und den prioritätsbasierten Ansatz



(b) Verhältnis zwischen mittlerem BET-Durchsatz des Trace-Monitoring-Ansatzes und des prioritätsbasierten Ansatzes in Abhängigkeit der Speicherintensität

Abbildung 5.21: BET-Durchsatz in Abhängigkeit der Speicherintensität und der Anzahl parallel ausgeführter BET n_{BET}

Abbildung 5.21b das Verhältnis der mittleren BET-Durchsatzwerte $\bar{P}_{BET,T}/\bar{P}_{BET,P}$ beider Ansätze dar. Bei geringer Interferenz ($M < 40\,\%$) zeigt sich ein signifikant höherer Durchsatz bei Nutzung des Trace-Monitoring-Ansatzes. Bei höheren Speicherintensitäten ist ein von der Anzahl paralleler BET-Tasks abhängiges Verhalten zu beobachten. Für $n_{BET} = 1$ ist ein deutlich reduzierter BET-Durchsatz gegenüber dem prioritätsbasierten Ansatz erkennbar. Mit zunehmendem n_{BET} wird die Durchsatzdifferenz der Ansätze jedoch geringer. Für $n_{BET} = 3$ führt der Trace-Monitoring-Ansatz auch bei maximaler Speicherintensität zu einem leicht höheren BET-Durchsatz als der prioritätsbasierte Ansatz.

Eine mögliche Erklärung für dieses Verhaltens ergibt sich aus der Betrachtung der mit n_{BET} zunehmenden Anzahl interferierender Tasks. Für $n_{BET} = 1$ beträgt die Anzahl parallel ausgeführter Tasks im prioritätsbasierten Ansatz $n_{Tasks,P} = n_{BET} = 1$, wodurch der BET exklusiv und somit nahezu unbeeinflusst ausgeführt wird. Mit dem Trace-Monitoring-Ansatz beträgt die Anzahl parallel ausgeführter Tasks durch den ET in diesem Fall bereits $n_{Tasks,T} = n_{BET} + 1 = 2$, wodurch bereits Interferenzeffekte auftreten, welche zu einer signifikanten Differenz im erreichbaren BET-Durchsatz führen. Mit zunehmender Anzahl n_{BET} nähert sich das Verhältnis der Anzahl parallel ausgeführter Tasks der beiden Ansätze zunehmend $n_{Tasks,T}/n_{Tasks,P} \xrightarrow{n_{BET}\to\infty} 1$ an. Es ist daher anzunehmen, dass sich die hierdurch verursachte Verzögerung der Taskausführung ebenfalls annähert.

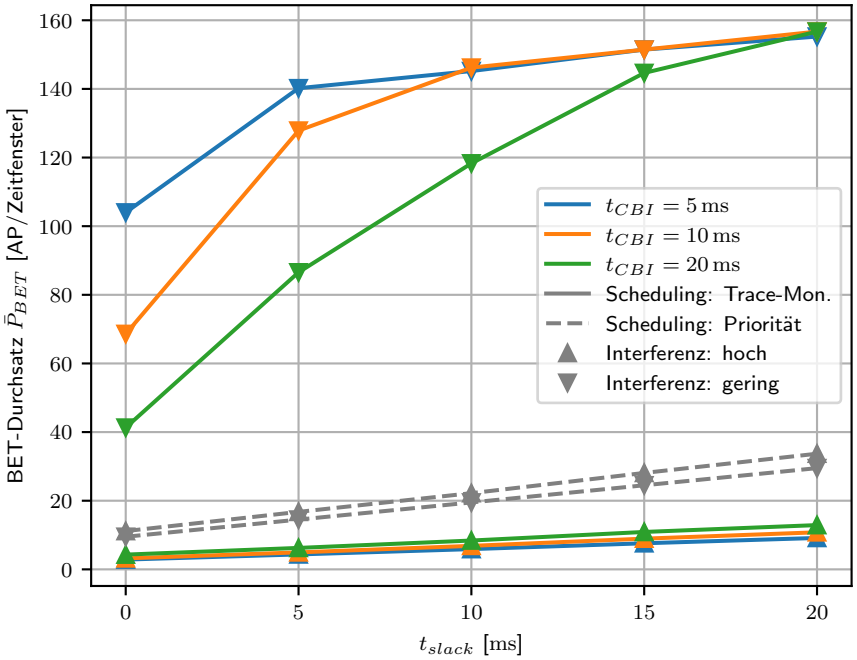
Dass mit dem Trace-Monitoring-Ansatz bei $n_{BET} = 3$ bei hoher Speicherintensität ein höherer BET-Durchsatz erreicht wird als mit dem prioritätsbasierten Ansatz, kann dieser Annäherung der BET-Ausführungsgeschwindigkeiten und damit einer zunehmenden Bedeutung der erreichten BET-Ausführungszeiten der Ansätze geschuldet sein. Wie in Abbildung 5.15a dargestellt, liegt die BET-Ausführungszeit auch bei $n_{BET} = 3$ und $M = 100\,\%$ signifikant über der mit dem prioritätsbasierten Ansatz erreichten Ausführungszeit. Bei ähnlicher Ausführungsgeschwindigkeit geht dies mit höheren Durchsatzraten einher.

5.4.4.5 Abhängigkeit des BET-Durchsatzes von der CBI-Rate

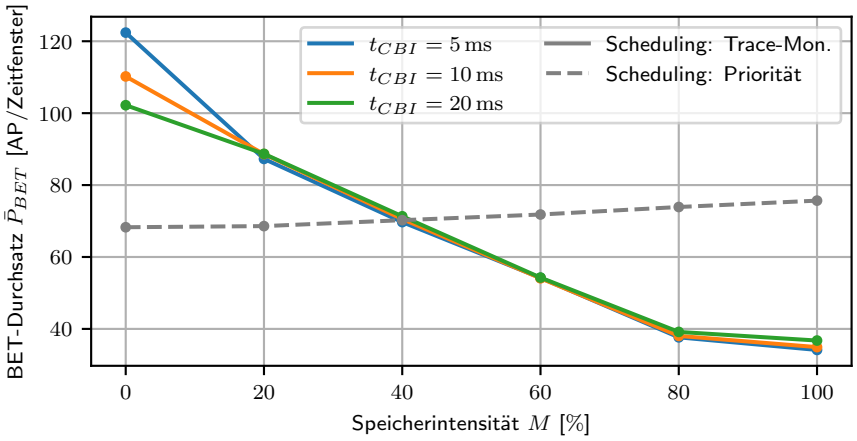
Im Folgenden wird die Abhängigkeit des erreichbaren BET-Durchsatzes von der Häufigkeit überwachter CBIs untersucht. Dies erfolgt analog zu Abschnitt 5.4.3.5 anhand eines um zusätzliche CBI erweiterten ET.

Analog zur Abbildung 5.16 zeigt Abbildung 5.22a den erreichten mittleren BET-Durchsatz $\bar{P}_{BET}$ in Abhängigkeit der CBI-Rate und von t_{slack} für minimale bzw. maximale Speicherintensität M. Hierbei sind nur Messungen des längsten untersuchten Pfades ① dargestellt. Zum Vergleich ist ebenfalls das mit dem prioritätsbasierten Ansatz erreichte $\bar{P}_{BET}$ dargestellt.

Grundsätzlich ist eine starke Ähnlichkeit zu den in Abbildung 5.16 betrachteten Ab-



(a) BET-Durchsatz in Abhängigkeit der CBI-Rate und von t_{slack} für den längsten ET-Pfad



(b) BET-Durchsatz in Abhängigkeit der Interferenzstärke des der CBI-Rate für $t_{slack} = 0\,\text{ms}$

Abbildung 5.22: BET-Durchsatz in Abhängigkeit der CBI-Rate

hängigkeiten der BET-Ausführungs-zeit t_{BET} erkennbar: Bei geringer Interferenzstärke ($M = 0\,\%$) zeigt sich für $\bar{P}_{BET,T}$ ein nahezu identischer Verlauf wie für $t_{BET,T}$. Der BET-Durchsatz nähert sich einem von t_{slack} abhängigen Grenzwert an und erreicht diesen für $t_{slack} \geq t_{CBI}$. Dies ist analog zur Erklärung in Abschnitt 5.4.3.5 nachvollziehbar, da bei geringer Interferenzstärke eine nahezu unverzögerte BET-Ausführung zu erwarten ist und somit ein von t_{BET} abhängiger Durchsatz von ca. 1 AP/ms erreicht wird. Im Gegensatz zur BET-Ausführungszeit führt der Trace-Monitoring-Ansatz bei hoher Interferenz zu geringeren BET-Durchsatzwerten als der prioritätsbasierte Ansatz. Dieser Effekt wurde bereits in Abschnitt 5.4.4.1 betrachtet.

Um den Einfluss der Interferenzstärke auf die Abhängigkeit des BET-Durchsatzes von der CBI-Rate zu untersuchen, stellt 5.22b den mittleren BET-Durchsatz in Abhängigkeit der Speicherintensität M für verschiedene t_{CBI} dar. Hierbei wurden nur Messungen mit $t_{slack} = 0\,\text{ms}$ berücksichtigt. Es ist erkennbar, dass der für $M = 0\,\%$ beobachtete Einfluss der CBI-Rate auf den BET-Durchsatz bereit für $M \geq 20\,\%$ nicht mehr beobachtbar ist. Die Differenz der für $M = 20\,\%$ gemessenen mittleren BET-Durchsatzwerte für verschiedene t_{CBI} beträgt weniger als 3 AP/Frame.

5.4.5 Reaktionslatenz des Execution Controllers

Im Folgenden wird die Reaktionslatenz der Trace-Monitor-Implementierung auf relevante Ereignisse während der Ausführung des ET untersucht. Hierzu wird die Erkennung einer CBI sowie die daraus folgende Einflussnahme auf den Hypervisor-Schedule betrachtet.

Wie in Abschnitt 5.2 beschrieben, nehmen CBI im vorgestellten Ansatz eine zentrale Rolle ein. Trace-Daten, welche bei der Ausführung überwachter CBI erzeugt werden, werden vom Execution Controller zur Erkennung des ET-Fortschritts und somit zur Bestimmung der Zeitreserve zur exklusiven Ausführung genutzt. Auf Basis dieser Zeitreserve wird der Umschaltzeitpunkt zwischen parallelem und exklusivem Ausführungsmodus aktualisiert (vgl. Abschnitt 5.2.5), was, je nach Systemzustand, zusätzliche parallele Ausführung der BET ermöglicht. Eine geringe Reaktionslatenz ist daher insofern vorteilhaft, als dass sie die Wahrscheinlichkeit eines nicht benötigten Umschaltens in den exklusiven Ausführungsmodus verringert und ein schnelles Zurückschalten in den parallelen Ausführungsmodus nach Erreichen einer entsprechenden CBI ermöglicht.

Auf Basis der beiden Systemzustände ergeben sich zwei relevante Latenzen. Ist das System zum Ausführungszeitpunkt der überwachten CBI im parallelen Ausführungsmodus, verbleibt es bei Feststellen eines ET-Ausführungsfortschritts in diesem Modus. Somit umfasst die in dieser Situation relevante Latenz die Generierung und Verarbeitung der Trace-Daten im Trace-Subsystem, den Empfang und die Decodierung der Trace-Daten durch den Execution Controller sowie das Ableiten der Scheduling-Entscheidung durch

1. Eine überwachte CBI wird durch den ET-Kern des MCP ausgeführt.
2. Die Trace-Komponente des ET-Kerns (ETM) erzeugt Synchronisations- und Branch-Pakete.
3. Die Trace-Pakete werden in einer Trace-FIFO (ETF) als Tracedatenstrom formatiert und gespeichert.
4. Der Execution Controller (EC) löst einen Flush in der ETF-Komponente aus, um ausstehende Tracedatenpakete zu speichern.
5. Der EC liest die formatierten Trace-Daten aus dem Buffer der ETF-Komponente aus und parst die darin enthaltenen Trace-Pakete.
6. Der EC identifiziert das relevante Branch-Paket und leitet daraus die Notwendigkeit eines Ausführungsmoduswechsels ab.
7. Der EC kommuniziert den Ausführungsmoduswechsel an den Hypervisor.
8. Der Hypervisor reaktiviert die BET auf den BET-Kernen.

Abbildung 5.23: Verarbeitungsschritte zum Wechsel vom exklusiven zum parallelen Ausführungsmodus nach Ausführung einer überwachten CBI

denselben. Diese Latenz wird als *Detektionslatenz* bezeichnet. Sie umfasst Punkt 1 bis Punkt 6 in der in Abbildung 5.23 dargestellten Verarbeitungskette. Ist das System zum Ausführungszeitpunkt der CBI im exklusiven Ausführungsmodus und ist auf Basis des ET-Fortschritts weitere parallele BET-Ausführung möglich, wird ein Moduswechsel durchgeführt: Der Execution Controller versetzt das System zurück in den parallelen Ausführungsmodus. Die hierbei relevante Latenz umfasst daher neben der Detektionslatenz auch die *Reaktionslatenz*. Diese setzt sich aus Punkt 7 bis Punkt 8 in der in Abbildung 5.23 dargestellten Verarbeitungskette zusammen.

Der Messaufbau basiert auf der in Abschnitt 5.3 beschriebenen Umsetzung des Trace-Monitoring-Konzepts. Als Zeitreferenz zwischen der APU und RPU der Ausführungsplattform wurde ein gemeinsamer Hardwaretimer (Triple Timer Counter, TTC) genutzt. Ein Echtzeit-Task wurde implementiert, welche in jeder Ausführungsperiode zunächst eine zufällige Zeit pausiert, anschließend eine überwachte CBI ausführt und direkt darauf den laufenden TTC zurücksetzt. Der Execution Controller wurde um eine Komponente erweitert, welche den TTC beim Systemstart initialisiert und einen TTC-Zeitstempel aufnimmt, sobald das zur CBI zugehörige Branch-Paket erkannt wurde. Weiterhin wurde ein Best-Effort-Task implementiert, welche den Wechsel vom exklusiven zum parallelen Ausführungsmodus erkennt und ebenfalls einen aktuellen Zeitstempel aufnimmt, sobald

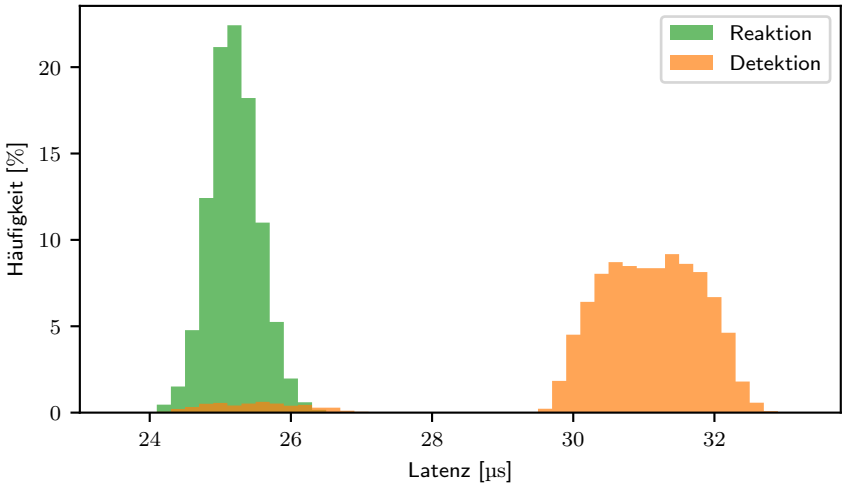


Abbildung 5.24: Verteilung der Detektions- und Reaktionslatenzen

die Ausführung nach einem solchen Moduswechsel fortgesetzt wird.

Mit dem genannten Versuchsaufbau wurden 8000 Messungen durchgeführt. Sie zeigen eine mittlere Detektionslatenz von 30.81 µs und eine mittlere Reaktionslatenz von 25.19 µs (vgl. Tabelle 5.3). Somit ergibt sich eine mittlere Gesamtlatenz von 56.00 µs und eine Maximallatenz von 59.42 µs, innerhalb welcher das System auf die Ausführung einer überwachten CBI mit der entsprechenden Änderung des Ausführungsmodus reagiert.

Abbildung 5.24 zeigt die Verteilung der gemessenen Latenzen. Während die Reaktionslatenz einer schmalen Verteilung um den Mittelwert folgt, zeichnen sich bei der Detektionslatenz zwei Häufungen ab, welche im zeitlichen Abstand von ca. 6 µs auftreten. Die Ursache für diesen Effekt wurde nicht untersucht. Eine mögliche Ursache kann durch das zur Trace-Infrastruktur asynchrone Auslesen der Trace-Daten aus dem Trace-Buffer sein. Aufgrund der Nutzung der Trace-Filterkomponenten wird vor jedem Branch-Paket

Tabelle 5.3: Detektions- und Reaktionslatenz

Messung	**Mittelwert**	**Minimum**	**Maximum**
Detektionslatenz	30.81 µs	24.07 µs	32.87 µs
Reaktionslatenz	25.19 µs	23.96 µs	26.55 µs

einer überwachten CBI ein Synchronisierungs-Paket erzeugt. Sofern zum Zeitpunkt des Auslesens der ETF durch den Execution Controller nur das Synchronisierungs-Paket verfügbar ist, verzögert sich das Auslesen und Verarbeiten des Branch-Pakets bis zur nächsten Iteration des Execution Controller. Befinden sich dagegen sowohl Synchronisierungs- als auch Branch-Paket bereits im ETF-Buffer, können diese in einer Programmiteration ausgelesen und verarbeitet werden.

5.5 Diskussion und Anwendbarkeit

In den folgenden Abschnitten wird diskutiert, in wie weit das vorgestellte Konzept die in Abschnitt 5.1 genannte Zielsetzung erreicht. Hierzu werden die erreichten Echtzeit-Eigenschaften, die Systemperformanz und der Ressourcenbedarf des Trace-Monitoring-Ansatzes diskutiert sowie Trade-Offs und Einschränkungen bei der Systemauslegung dargestellt.

5.5.1 Erreichbare Echtzeit-Eigenschaften

Anwendungen mit harten Echtzeitanforderungen zeichnen sich durch eine maximale zulässige Reaktionszeit eines Datenverarbeitungssystems auf eine Eingabe aus. Auf Softwareebene führt dies dazu, dass Tasks, welche die Datenverarbeitung implementieren, zu einem definierten Zeitpunkt, ihrer Deadline, abgeschlossen sein müssen.

In den betrachteten Systemen mit gemischten Echtzeitanforderungen sind nur an Echtzeit-Tasks Echtzeitanforderungen gestellt. Somit ist die Echtzeitfähigkeit des Systems erfüllt, wenn die Einhaltung der Deadlines aller Echtzeit-Tasks sichergestellt ist. Basierend auf den in Abschnitt 5.2.1 genannten Annahmen und Voraussetzungen kann bei Nutzung des vorgeschlagenen Scheduling-Ansatzes von der Einhaltung dieser Deadlines und somit der Echtzeitfähigkeit des Systems unter folgenden Bedingungen ausgegangen werden:

Bedingung 1: Das System muss wie in Abschnitt 5.2.5.1 und Abschnitt 5.2.5.2 beschrieben konfiguriert sein.

Bedingung 2: Der Wechsel zum exklusiven Ausführungsmodus muss dazu führen, dass der Echtzeit-Task ab dem Umschaltzeitpunkt mit Single-Core-Performance ausgeführt wird.

Bedingung 3: Der Wechsel zum exklusiven Ausführungsmodus muss rechtzeitig erfolgen, sodass die Ausführung des Echtzeit-Tasks in der verbleibenden Zeit bis zur Deadline abgeschlossen werden kann.

Im Folgenden werden diese Bedingungen und die dahinter stehende Argumentation genauer betrachtet.

5.5.1.1 Korrekte Parametrisierung

Hinsichtlich Bedingung 1 wird die korrekte Parametrisierung des Systems vorausgesetzt. Besonderheiten im Hinblick auf die Parametrisierung, welche zur Erfüllung von Echtzeitverhalten beachtet werden müssen, werden in den folgenden Abschnitten diskutiert.

5.5.1.2 Ausführung mit Single-Core-Performance

Für Phasen exklusiver Ausführung des Echtzeit-Tasks wird gefordert, dass der Task mit der Geschwindigkeit ausgeführt wird, welche im Rahmen der WCET-Abschätzungen für den Fall dauerhafter exklusiver Ausführung angenommen wurde (Bedingung 2). Dies entspricht der Annahme, dass nach dem Umschalten keine weiteren Interferenzeffekte auftreten, welche nicht bei der Berechnung der (partiellen) WCET-Abschätzungen berücksichtigt wurden. Diese Anforderung ist nur mit starken Einschränkungen erfüllbar, da durch verschiedene Effekte auch nach dem Moduswechsel Interferenzeffekte auftreten können. Diese lassen sich unterteilen in (1) Effekte, deren Ursache aus der Zeit paralleler Ausführung stammt und mit dem Moduswechsel entfällt, und (2) Effekte, deren Ursache nach dem Wechsel in den exklusiven Ausführungsmodus fortbesteht.

Im Folgenden werden diese Effekte diskutiert und entsprechende Einschränkungen bzw. Alternativen diskutiert.

Wie in Abschnitt 5.2.1 vorausgesetzt, muss sichergestellt werden, dass der ET unabhängig vom Ausführungsfortschritt des BET ausgeführt werden kann. Dies hat insbesondere zur Folge, dass die Ausgabe eines ET nicht von der Verfügbarkeit einer Ausgabe eines BET abhängen darf. Weiterhin darf der Fortschritt eines ET nicht von der Verfügbarkeit einer Ressource abhängen, welche vom BET blockiert werden kann. Dies betrifft sowohl logische Ressourcen als auch Hardwarekomponenten, welche durch inkorrekte oder unvollständige Konfiguration durch den BET nicht vom ET genutzt werden können. Ein Verstoß gegen diese Forderung kann die Ausführung des ET dauerhaft blockieren (vgl. (2)) und somit die Einhaltung der Echtzeitanforderungen gefährden.

Auch exklusiv vom BET genutzte Hardwarekomponenten können zu einer Verzögerung der ET-Ausführung nach (2) führen, wenn die Hardwarekomponenten mit dem ET interferieren können. Dies ist insbesondere bei Komponenten mit DMA-Funktion der Fall, welche durch externe Ereignisse ausgelöste Speicherzugriffe durchführen. Sofern das Auftreten dieser Ereignisse nicht mit dem Abschalten des BET endet, können solche Komponenten auch während der exklusiven Ausführungsphase zu Interferenz führen. Die Nutzung dieser Komponenten sollte daher auf Fälle beschränkt werden, in denen ihr Einfluss auf die ET-Ausführung im Rahmen der WCET-Analyse berücksichtigt werden kann. Sofern dieser Einfluss nicht sicher bestimmt bzw. nach oben abgeschätzt werden

kann, kann die Nutzung solcher Komponenten die Echtzeitfähigkeit beeinträchtigen, da die resultierende exklusive ET-Ausführungsdauer bei unerwartet starker Verzögerung der ET-Ausführung durch die Hardwarekomponente für den rechtzeitigen Abschluss des ET unzureichend sein kann.

Die gleichen Überlegungen gelten bezüglich Interferenz mit dem Execution Controller selbst. Der Execution Controller muss so integriert sein, dass er über seine Steuerungsaufgabe hinaus keinen Einfluss auf die Ausführung des ET hat, da Interferenzeffekte zwischen Execution Controller und ET die ET-Ausführung verzögern können. Weiterhin könnten Interferenzeffekte die Ausführung des Controllers selbst beeinträchtigen und somit zu einer Verletzung von Bedingung 3 führen.

Auch Aspekte der Prozessor- bzw. Hardware-Architektur können in Phasen exklusiver Ausführung zu wechselnden Ausführungszeiten führen. So kann bspw. spekulative Ausführung in verschiedenen Ausführungsiterationen zu unterschiedlichen Laufzeiten führen. Um Echtzeitverhalten sicherstellen zu können, ist daher gefordert, dass die genutzte Ausführungsplattform bei exklusiver Ausführung hinreichend deterministisch ist, um für die Anwendung ausreichende WCET-Abschätzungen bestimmen zu können.

Neben den bisher genannten Effekten sind nach dem Umschalten von paralleler zu exklusiver Ausführung auch Effekte nach (1) zu erwarten. Ursache dieser Effekte ist, dass die parallele Ausführung des BET auch nach dem Abschalten des BET noch Auswirkungen auf die ET-Ausführung haben kann. Da die Ausführung des BET den Systemzustand beeinflusst, ergibt sich zum Umschaltzeitpunkt von paralleler zu exklusiver Ausführung ein Zustand, welcher von dem im Fall dauerhafter exklusiver ET-Ausführung erreichten Zustand abweicht. Anschaulich ist dies am Beispiel eines geteilten Caches zu erklären: Im Rahmen der WCET-Analyse des gesamten ET kann die dauerhafte exklusive Ausführung des ET angenommen werden. Dementsprechend wird von der exklusiven Nutzung des Caches durch den ET ausgegangen. Bei geteilter Nutzung des Caches können zwischengespeicherte Daten des ET durch Speicherzugriffe des BET verdrängt werden. Findet anschließend der Wechsel in den exklusiven Ausführungsmodus statt, müssen die verdrängten Daten beim nächsten Zugriff durch den ET neu aus dem Hauptspeicher geholt werden (Cache Miss). Dieser Speicherzugriff benötigt somit mehr Zeit als der im Rahmen der WCET-Analyse erwartete Speicherzugriff aus dem Cache (Cache Hit).

Der genannte Effekt kann vermieden werden, indem von diesem Effekt betroffene Komponenten soweit möglich deaktiviert oder partitioniert werden. Im Fall von Caches ist bei einer Deaktivierung eine deutlich verringerte Leistung des Prozessorsystems zu erwarten. Daher können hier alternativ Cache-Locking oder Cache-Partitionierungstechniken genutzt werden [189], welche bereits in Hypervisor-partitionierten Systemen integriert wurden [100].

Alternativ muss die durch den Effekt verursachte Verzögerung des ET nach oben abgeschätzt werden und bei der Bestimmung der (partiellen) ET-WCET berücksichtigt werden. Es wird davon ausgegangen, dass dies möglich ist, wenn die WCET-Analysen unter Annahme eines uninitialisierten Systemzustands erfolgen (vgl. Abschnitt 5.2.1).

Zur Parametrisierung des Execution Controllers werden solche WCET-Abschätzungen gefordert. Diese geben hierbei eine Abschätzung der WCET eines Teilpfads des ET an, beginnend ab ausgewählten Instruktionen $\mathbb{I} = \{S, I_{b_1,F}, I_{b_1,N}, I_{b_2,F}, I_{b_2,N}...\}$ bis zur letzten Instruktion der ET-Ausführung. Im realen System erfolgt der Wechsel in den exklusiven Ausführungsmodus zeitgesteuert. Er ist nicht mit der Ausführung der Instruktionen in $\mathbb{I}$ synchronisiert und kann demnach auch an einer Instruktion $J \notin \mathbb{I}$ erfolgen. Somit muss sichergestellt werden, dass die nach dem Wechsel des Ausführungsmodus benötigte ET-Ausführungszeit die verfügbare Zeitreserve nicht überschreitet. Im vorgestellten Konzept wird die Zeitreserve hierzu anhand der partiellen WCET-Abschätzung $WCET_{part}(I)$ der zuletzt ausgeführten Instruktion $I \in \mathbb{I}$ gewählt, welche auf die zuletzt erfasste überwachte CBI folgt. Aus diesem Grund ist gefordert, dass die partielle WCET-Abschätzung, beginnend ab einer Instruktion im ET-Programm, stets größer oder gleich der WCET-Abschätzung im Programmfluss später erstmals ausgeführter Instruktionen ist (vgl. Abschnitt 5.2.1).

5.5.1.3 Rechtzeitiges Umschalten in den exklusiven Ausführungsmodus

Zur Einhaltung von Echtzeitanforderungen muss das Umschalten des Systems vom parallelen zum exklusiven Ausführungsmodus so geschehen, dass die resultierende Dauer exklusiver Ausführung zum Abschluss des Echtzeit-Tasks vor dessen Deadline ausreicht (Bedingung 3). Gemäß Abschnitt 5.2.5.4 ergibt sich der Umschaltzeitpunkt $t_{x,i}$ aus der Deadline $t_{d,i}$ der aktuellen Task-Iteration i und der verbleibenden partiellen WCET-Abschätzung basierend auf dem Fortschritt des ET. Um die Einhaltung der Deadline sicherzustellen, muss der Moduswechsel spätestens zu diesem Zeitpunkt geschehen. Im Folgenden werden daraus Anforderungen an (Teil-)Komponenten des vorgestellten Konzepts abgeleitet.

Gemäß des Konzepts sind am Umschaltvorgang der Execution Controller und der Hypervisor beteiligt. Der Execution Controller ist mit dem vom Hypervisor umgesetzten Schedule synchronisiert, erkennt das Erreichen von $t_{x,i}$ und signalisiert daraufhin den Moduswechsel an den Hypervisor. Dieser deaktiviert daraufhin die BET-Partitionen zum Zeitpunkt $t = t_{x,i} + \epsilon_1$.

Es muss daher sichergestellt werden, dass sowohl die am Moduswechsel beteiligten Module des Execution Controller als auch der Hypervisor Echtzeit-Anforderungen erfüllen. Hierzu muss insbesondere sichergestellt sein, dass keine Interferenzeffekte zwischen Execution

Controller und Hypervisor auftreten bzw. die Auswirkungen dieser begrenzt und bestimmbar sind. Da der beschriebene Ablauf nach Erreichen von $t_{x,i}$ nicht in Nullzeit erfolgt, ist weiterhin die maximale Laufzeit $\max(\epsilon_1)$ dieses Ablaufs zu bestimmen und bei der Konfiguration des Systems zu berücksichtigen. Da zudem die vom Execution Controller genutzte Zeitreferenz gegenüber der des Hypervisor-Schedulers verschoben sein kann, ist weiterhin der Synchronisierungsfehler ϵ_2 zu bestimmen. Synchronisierungsfehlerkomponenten, die sich aus mangelnder Präzision der Taktquellen ergeben, werden hier als vernachlässigbar angenommen, da mit Beginn jedes ET-Zeitfensters eine erneute Synchronisierung erfolgt.

Sind diese maximalen Verzögerungen bestimmt, können sie den (partiellen) WCET-Abschätzungen des Echtzeit-Tasks bei der Konfiguration des Execution Controllers zugeschlagen werden. Hierdurch wird $t_{x,i}$ um $\max(\epsilon_1) + \max(\epsilon_2)$ zu $t'_{x,i}$ vorverlegt, sodass der Moduswechsel zum Zeitpunkt $t'_{x,i} + \max(\epsilon_1) + \max(\epsilon_2) = t_{x,i}$ abgeschlossen ist.

An dieser Stelle sei hervorgehoben, dass keine Echtzeitanforderungen an das Trace-Subsystem gestellt sind. Während aus Trace-Daten abgeleiteten Informationen über den Ausführungsfortschritt des ET zur Verlängerung der Zeit paralleler Ausführung genutzt werden, ist zu jedem Zeitpunkt sichergestellt, dass die verbleibende, zur exklusiven Ausführung reservierte Zeit ausreicht, um die Ausführung des ET vor seiner Deadline abzuschließen. Sollten Trace-Daten zeitweise nicht oder stark verspätet zur Verfügung stehen, beeinflusst das daher nur den BET-Durchsatz des Systems, nicht aber das Echtzeitverhalten.

5.5.1.4 Vermeidung von Interferenz mit dem Execution Controller

Sowohl um Bedingung 2 als auch um Bedingung 3 erfüllen zu können, ist es nötig, Interferenz zwischen dem Hypervisor auf der APU, einschließlich der darauf ausgeführten ET und BET, und dem Execution Controller zu vermeiden. In diesem Abschnitt wird auf diesen Aspekt weiter eingegangen.

Da Trace-Systeme auf die nichtinvasive Erfassung des Programmablaufs auf Prozessoren ausgelegt sind, bilden diese eine geeignete Informationsquelle über den Programmfortschritt. Das im Rahmen der Implementierung genutzte Trace-System ARM CoreSight implementiert ein solches nichtinvasives Tracing [36]. Hierzu werden Trace-Daten durch Hardwarekomponenten erfasst, verarbeitet und übertragen, welche weitgehend unabhängig vom Rest des Systems sind (vgl. Abbildung 2.6 in Abschnitt 2.2.4). Es kann daher davon ausgegangen werden, dass die Verarbeitung von Trace-Daten im Trace-Subsystem keine Auswirkungen auf Ausführung des ET oder auf das Echtzeitverhalten der zur Umsetzung des Moduswechsels genutzten Komponenten hat.

Eine mögliche Interferenzquelle stellt die Schnittstelle zwischen Trace-Subsystem und

Execution Controller dar. In der untersuchten Implementierung greift der Execution Controller auf Tracedaten zu, welche in Buffer-Komponenten gespeichert sind. Wie im Fall der genutzten Hardwareplattform erfolgen solche Zugriffe häufig über das zentrale Interconnect des SoC und können daher bei hoher Belastung des Interconnect zu Interferenzeffekten führen. Im Rahmen der untersuchten Szenarien wurden keine Auswirkungen auf die Einhaltung der ET-Deadline festgestellt (vgl. Abschnitt 5.4.2). In Anwendungen, in welchen eine höhere Auslastung der Interconnects zu erwarten ist, bspw. durch weitere parallele Datenströme auf der Plattform, muss auch diese Komponente als mögliche Interferenzquelle in Betracht gezogen werden und ggf. eine abweichende Implementierung der Tracedatenausleitung gewählt werden. Auf dem in dieser Arbeit betrachteten MPSoC bietet sich eine Implementierung des Execution Controllers im FPGA an. Dies kann mittels eines im FPGA instanziierten Prozessors erfolgen, welcher die Execution-Controller-Software ausführt und wie in der vorgestellten Implementierung mittels IPI mit dem Hypervisor kommuniziert. Alternativ kann die Funktion des Execution Controllers direkt als Logikschaltung im FPGA implementiert werden. Die Machbarkeit der Auswertung von Trace-Daten im FPGA zur Laufzeit wurde bereits in [44] gezeigt. Die Übertragung der Trace-Daten erfolgt hierbei über eine direkte Schnittstelle zwischen Trace-Infrastruktur und FPGA, wodurch Interferenzeffekte mit weiteren Systemkomponenten vermieden werden.

In ähnlicher Weise ist auch der vom Execution Controller genutzte Speicher zu beachten. Sofern Programm- oder Datenspeicher des Execution Controllers in einem geteilt genutzten Speicher liegen, können Zugriffe auf diesen zu Interferenz an Interconnects oder der Speicherkomponente führen. Auf der betrachteten Plattform lässt sich dies durch Nutzung von TCM der RPU umgehen.

5.5.1.5 Betrachtete Interferenzkanäle und Verallgemeinerung

In den untersuchten Evaluationsszenarien wurde die Verlangsamung der Anwendungsausführung als Folge von Interferenz an der Speicherinfrastruktur betrachtet. Dies erfolgte mittels eines speicherintensiven Arbeitspakets, dessen Speicherzugriffsverhalten auf die vollständige Nutzung des geteilten L2-Caches ausgerichtet wurde. Ohne weitere Mechanismen zur Cache-Partitionierung maximiert diese Auslegung die gegenseitige Verdrängung von im Cache gespeicherten Daten parallel ausgeführter Anwendungen. Somit führt sie zu einer starken Verzögerung der Anwendungsausführung gegenüber dem Fall exklusiver Ausführung.

Das vorgestellte Konzept ist jedoch nicht auf die Beherrschung von Interferenz an geteilten Caches begrenzt. Da der Monitoring-Ansatz die Auswirkung von Interferenz auf die ET-Ausführung unabhängig vom ursächlichen Interferenzkanal erkennt, eignet

sich das Konzept auch zur Begrenzung von an anderen Komponenten auftretenden Interferenzeffekten.

Relevant für die Wirksamkeit des Konzepts ist der genutzte Mechanismus zur Interferenzbegrenzung. Wie in Abschnitt 5.5.1.2 betrachtet, ist der vorgestellte Mechanismus anwendbar, sofern das Unterbrechen der BET-Ausführung die Interferenzursache beseitigt, die nach dem Unterbrechen noch auftretende Verzögerung der ET-Ausführung begrenzt und für diese eine obere Schranke bestimmbar ist.

Sofern keine weiteren Komponenten im System vorhanden sind, welche mit vom ET genutzten Ressourcen konkurrieren, deckt das Konzept somit bspw. auch Interferenz an der weiteren Speicherinfrastruktur wie Speichermodulen, Speichercontrollern, Interconnects und Systembussen ab.

5.5.2 Erreichbare Leistung des Multicore-Systems

Im Folgenden wird die erreichbare Leistung des Multicore-Systems unter Anwendung des vorgestellten Konzepts diskutiert. Die Diskussion erfolgt auf Basis der in Abschnitt 5.4 beschriebenen Evaluation.

5.5.2.1 Optimierung paralleler Ausführung

Im Hinblick auf Anforderung 1 wurde die Abhängigkeit der erreichbaren Dauer paralleler Ausführung von ET und BET von verschiedenen Parametern untersucht. Diese umfassen die Speicherintensität der beteiligten Tasks, das initiale Zeitbudget für parallele Ausführung, die CBI-Rate und die Anzahl parallel ausgeführter BET auf verschiedenen Kernen des Multicore-Prozessors. Anhand der untersuchten Szenarien zeigt sich, dass das vorgestellte Konzept im Vergleich mit dem prioritätsbasierten Referenzansatz zu einer deutlich höheren Dauer paralleler Ausführung führen kann. Im in Abschnitt 5.4.3.2 betrachteten Fall mit $t_{slack} = 0\,\text{ms}$ und $n_{BET} = 1$ wurde eine interferenzabhängige Steigerung dieser Dauer um 41 % bis 26 % erreicht. Weiterhin lassen sich folgende Zusammenhänge erkennen:

- Die erreichte Dauer paralleler Ausführung hängt insbesondere von der Stärke der auftretenden Interferenzeffekte zwischen ET und BET ab. Mit zunehmender Stärke nimmt die Ausführungsdauer des BET ab (vgl. Abschnitt 5.4.3.2).
- Das Vorsehen eines initialen Zeitbudgets für parallele Ausführung $t_{slack} > 0$ führt zu einer Verlängerung der Dauer paralleler Ausführung. Diese Verlängerung entspricht mindestens dem gewählten Wert t_{slack}, umfasst jedoch zusätzlich eine interferenzabhängige Komponente. Diese nimmt mit zunehmender Interferenz ab,

ist in den untersuchten Szenarien jedoch auch bei maximaler Speicherintensität noch feststellbar.
Bei geringer Interferenz ist weiterhin eine starke Abhängigkeit dieser Komponente von der Länge des gewählten Ausführungspfades durch den ET erkennbar. Während t_{slack} bei geringer Pfadlänge nur einen geringen Einfluss auf die Dauer der interferenzabhängigen Komponente hat, nimmt dieser mit zunehmender Pfadlänge zu. Es kann daher davon ausgegangen werden, dass Anwendungen, welche häufig ET-Ausführungsdauern nahe an der maximalen ET-Ausführungsdauer erreichen, von der Zugabe eines solchen Budgets in besonderem Maße profitieren (vgl. Abschnitt 5.4.3.3).

- Die Nutzung zusätzlicher Prozessorkerne für die parallele Ausführung weiterer BET-Tasks führt bei geringer Interferenz zu einer zur Kern-Anzahl proportionalen Erhöhung der BET-Ausführungszeit. Mit zunehmender Interferenzstärke sinkt der Skalierungsfaktor, sodass eine Erhöhung der Kern-Anzahl zu einer geringeren Steigerung der BET-Ausführungszeit führt (vgl. Abschnitt 5.4.3.4).
- Die Wahl der CBI-Rate beeinflusst die erreichte BET-Ausführungszeit insbesondere in Situationen geringer Interferenz. Ist $t_{CBI} < t_{slack}$ gewählt, kann in Zeiten geringer Interferenz durchgehend parallele Ausführung von ET und BET erreicht werden.

5.5.2.2 Auswirkungen auf den BET-Durchsatz

Die Auswirkung der in Abschnitt 5.5.2.1 diskutierten Optimierung der parallelen Ausführung von ET und BET auf die Performance der Ausführungsplattform wurde in Abschnitt 5.4.4 untersucht. Als Maß für die Performance wurde der BET-Durchsatz betrachtet, d.h. die Rate, mit welcher BET-Arbeitspakete verarbeitet werden.

Die Evaluation zeigt, dass der Trace-Monitoring-Ansatz bei geringer Speicherintensität und somit geringer Interferenzstärke zu einem bis zu 42 % höheren Durchsatz führt als der prioritätsbasierte Referenzansatz. Mit zunehmender Interferenz sinkt die mit dem Trace-Monitoring-Ansatz erreichte Rate jedoch stark ab und liegt bei maximaler Speicherintensität nur noch bei 58 % des mit dem Referenzansatz erreichten Wertes.

Der im Vergleich zum Referenzansatz geringere Durchsatz bei starker Interferenz ist auf das gewählte Optimierungsziel zurückzuführen. Der vorgestellte Ansatz optimiert die Zeit paralleler Ausführung von ET und BET. Wie in früheren Untersuchungen zur Interferenz auf Multicore-Prozessoren gezeigt, kann die parallele Ausführung von Tasks bei starker Interferenz jedoch zu mehrfach längeren Ausführungszeiten führen als die sequenzielle Ausführung derselben [78]. Dieser Effekt ist in Szenarien beobachtbar, in welchen sowohl ET als auch BET hohe Speicherintensität aufweisen. Mit dem prioritätsbasierten Ansatz

werden ET und BET sequenziell ausgeführt, was in diesen Szenarien zu kürzeren Task-Ausführungszeiten und somit höherem BET-Durchsatz führt.

Die starke Interferenzabhängigkeit des BET-Durchsatzes wurde auch bei den Betrachtungen der Zusammenhänge mit dem initialen Budget für parallele Ausführung, der Anzahl parallel ausgeführter BET-Tasks und der CBI-Rate beobachtet:

- So wurde bei Zugabe einer initialen Zeitreserve t_{slack} bei geringerer Interferenzstärke ein stärkerer Effekt auf den erreichten BET-Durchsatz beobachtet als bei starker Interferenz (vgl. Abschnitt 5.4.4.2).
- Beim Einsatz zusätzlicher BET-Prozessorkerne steigt der BET-Durchsatz bei minimaler Interferenz proportional zur Kern-Anzahl. Mit zunehmender Interferenz sinkt der Einfluss zusätzlicher BET auf den erreichten Durchsatz. Dies ist sowohl aufgrund der verringerten BET-Ausführungszeit als Folge zunehmender Verlangsamung der ET-Ausführung als auch aufgrund von Interferenz zwischen parallel ausgeführten BET-Tasks nachvollziehbar (vgl. Abschnitt 5.4.4.4).
- Während eine Erhöhung der CBI-Rate bei geringer Interferenz zu einer Verbesserung des BET-Durchsatzes führt, konnte dieser Effekt ab einer Speicherintensität von 20 % bereits nicht mehr beobachtet werden (vgl. Abschnitt 5.4.4.5).

5.5.3 Ressourcenbedarf und Effizienzabwägungen

In diesem Abschnitt werden zentrale Anforderungen des Konzepts an die Hardwareplattform diskutiert und Abwägungen zur Optimierung der Systemeffizienz vorgestellt.

Das vorgestellte Konzept basiert auf der Überwachung von Echtzeit-Tasks anhand von Trace-Daten durch einen vom Hypervisor getrennt ausgeführten Execution Controller. Dementsprechend wird auf der Zielplattform eine Trace-Infrastruktur sowie eine Einheit zur Ausführung des Execution Controllers benötigt. Hierbei ist weitgehende Interferenzfreiheit zwischen der Ausführungseinheit des Execution Controllers und den Systemkomponenten zur Ausführung der Nutzanwendungen gefordert.

Sofern das MPSoC entsprechende, vom Anwendungsprozessor getrennte Prozessorkerne bereitstellt, ist eine Implementierung des Execution Controller in Software auf einem solchen Prozessor möglich. Die vollständige Umgehung von Interferenzkanälen stellt hierbei jedoch eine Herausforderung dar. Wie in Abschnitt 5.5.1.4 beschrieben, kann daher eine Implementierung des Execution Controller in im SoC integrierten FPGA-Komponenten, soweit vorhanden, eine vorteilhafte Alternative sein. Wird die Funktion des Execution Controllers vollständig in Form einer Logikschaltung implementiert, ist diese zudem gut hinsichtlich ihrer maximalen Verarbeitungslatenz analysierbar. Sofern keine geeignete Ausführungskomponente innerhalb des SoC zur Verfügung steht, ist auch

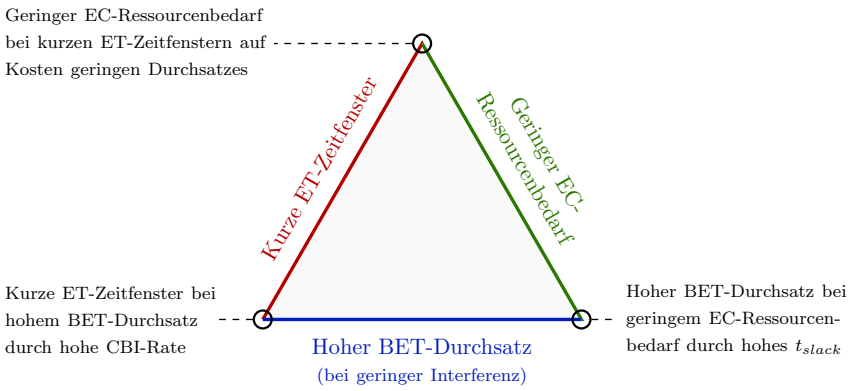


Abbildung 5.25: Trade-Off zwischen dem BET-Durchsatz, der durch t_{slack} bestimmten Dauer der ET-Zeitfenster und dem Ressourcenbedarf des Execution Controller (EC)

eine Implementierung des Controllers als externe Komponente möglich. Einen solcher Ansatz wird in [137] verfolgt.

Im Hinblick auf eine Software-Implementierung des Execution Controllers ist die benötigte CPU-Rechenleistung als vergleichsweise gering einzustufen. Während der in [138] vorgestellte und in [137] angewandte *Fingerprinting*-Ansatz auf einer hochfrequenten[2] periodischen Abtastung und Verarbeitung von CPU-Performance-Zählerwerten basiert, erfolgt die Verarbeitung von Trace-Daten im vorgeschlagenen Ansatz ereignisbasiert. Das auslösende Ereignis ist hierbei das Auftreten einer überwachten CBI. Die Häufigkeit dieses Ereignisses kann vom Systementwickler durch geeignete Wahl der zu überwachenden CBI bestimmt werden. Somit ist der Bedarf des Execution Controller an Rechenleistung direkt vom Systementwickler beeinflussbar. Es sei angemerkt, dass die Überprüfung des Trace-Buffers durch einen in Software ausgeführten Execution Controller in regelmäßigen Abständen Rechenzeit erfordert. Sofern keine Trace-Daten vorliegen, beschränkt sich dies jedoch auf die Prüfung eines Registers der Trace-Komponente. Der periodische Bedarf an Rechenleistung wird daher als sehr gering angesehen.

Wie in Abschnitt 5.4.4.5 festgestellt, beeinflusst die Wahl der überwachten CBI zusammen mit dem gewählten initialen Budget für parallele Ausführung t_{slack} den erreichbaren BET-Durchsatz. Für den Systementwickler ergibt sich somit ein Trade-Off zwischen der vom Execution Controller benötigten Rechenleistung, dem erreichten BET-Durchsatz und der Länge der ET-Zeitfenster im Hypervisor-Schedule (vgl. Abbildung 5.25). Letztere bestimmt wiederum die Effizienz des Schedules in Bezug auf die Anzahl sequenziell

[2] Die in [138] genutzte Abtastrate und Periode der Datenverarbeitung durch die Monitoring-Komponente beträgt 100 µs.

ausführbarer ET-Tasks. Der Trade-Off ergibt sich aus folgenden Zusammenhängen:

- Soll die Anzahl der auf dem Anwendungsprozessor ausgeführten Echtzeit-Tasks maximiert werden, müssen ET-Zeitfenster möglichst kurz gewählt werden. Dies entspricht einer Minimierung von t_{slack}. Eine Reduktion von t_{slack} wirkt sich negativ auf den BET-Durchsatz aus. Dies kann durch Erhöhen der CBI-Rate teilweise kompensiert werden, wodurch sich jedoch der EC-Ressourcenbedarf erhöht.
- Soll der BET-Durchsatz maximiert werden, sollten die überwachten CBI und somit die CBI-Rate so gewählt werden, dass $t_{slack} \geq t_{CBI}$ gilt[3]. Wie in Abschnitt 5.4.4.5 beobachtet, kann in Situationen geringer Interferenz zwischen ET und BET durch diese Wahl eine signifikante Steigerung des BET-Durchsatzes erreicht werden. Da t_{CBI} umgekehrt proportional zur CBI-Rate ist, kann dies durch die Wahl einer hohen CBI-Rate oder durch die Wahl eines entsprechend langen t_{slack} erfüllt werden. Ersteres führt jedoch zu erhöhtem EC-Ressourcenbedarf, letzteres zu längeren ET-Zeitfenstern und reduziert somit die Anzahl auf dem Anwendungsprozessor ausführbarer ET.
- Soll der Ressourcenbedarf des Execution Controllers reduziert werden, erfordert dies eine Verringerung der CBI-Rate. Eine Reduktion der CBI-Rate wirkt sich negativ auf den BET-Durchsatz aus. Dies kann durch Erhöhen von t_{slack} kompensiert werden, wodurch sich jedoch der EC-Ressourcenbedarf erhöht.

In der zur Evaluation genutzten Implementierung wird der Execution Controller als Baremetal-Anwendung exklusiv auf einem separaten Prozessorkern des MPSoC ausgeführt und prüft den Trace-Buffer periodisch auf verfügbare Trace-Daten. Um eine effizientere Nutzung des Prozessors zu erreichen, kann der Execution Controller als Task innerhalb eines Echtzeitbetriebssystems ausgeführt werden. Der Task sollte periodisch aufgerufen werden, um das Polling des Trace-Buffers auszuführen und darin gespeicherte Daten ggf. abzurufen und zu verarbeiten. Da an die Verarbeitung von Trace-Daten keine Echtzeitanforderungen gestellt sind, kann die Polling-Periode grundsätzlich beliebig gewählt werden. Auch kann sie beliebigen Abweichungen (Jitter) unterworfen sein. Mit zunehmender Dauer zwischen zwei Polling-Vorgängen ist jedoch mit einer Verringerung des BET-Durchsatz zu rechnen, da ET-Anwendungsfortschritt später erkannt wird und somit später in den parallelen Ausführungsmodus gewechselt wird. Im Gegensatz zur Tracedatenverarbeitung muss die Logik zum Umschalten des Ausführungsmodus auch in der RTOS-Implementierung Echtzeitfähigkeit erfüllen.

[3] Ist dies nicht erfüllbar, sollte t_{slack} möglichst hoch und t_{CBI} möglichst gering gewählt werden.

5.5.4 Zielerreichung

In Abschnitt 5.1 wurden zwei zentrale Anforderungen an den Trace-Monitoring-Ansatz zur Interferenzbeherrschung definiert, deren Erfüllungsgrad im Rahmen der Evaluation experimentell geprüft und im Rahmen der Diskussion analysiert wurde.

Im Rahmen der Evaluation wurde die Erfüllung von Anforderung 2 mittels parametrisierbarer Benchmark-Anwendungen untersucht. Hierbei wurde die Speicherintensität parallel ausgeführter Anwendungen variiert und somit verschiedene Interferenzszenarien erzeugt. Es konnte gezeigt werden, dass das vorgestellte Konzept in allen betrachteten Szenarien, insbesondere auch bei starken Interferenzeffekten, zu einer Einhaltung der Deadline führte (vgl. Abschnitt 5.4.2). Die unkontrollierte parallele Ausführung der Tasks führte dagegen bereits bei leichter Speicherintensität zu Verletzung der Deadline.

In Abschnitt 5.5.1 wurde das Konzept hinsichtlich erreichbarer Echtzeiteigenschaften analysiert und Implementierungseinschränkungen sowie mögliche Mitigationsansätze diskutiert. Während das Konzept die Einhaltung von Echtzeitanforderungen auf geeigneten und hinreichend analysierbaren Ausführungsplattformen sicherstellen kann, bestehen in der zur Evaluation genutzten Implementierung Interferenzkanäle, welche den Echtzeit-Nachweis für hoch kritische Anwendungen ohne genauere Kenntnis der verwendeten Hardware verhindern können. In der experimentellen Evaluation konnten jedoch keine negativen Auswirkungen auf das Einhalten der ET-Deadlines beobachtet werden. Es ist daher davon auszugehen, dass in den untersuchten Szenarien bei der WCET-Bestimmung ein hinreichend großer Sicherheitszuschlag gewählt wurde, sodass potenzielle Interferenzeffekte nicht sichtbar wurden. Bei stärkerer Auslastung der potenziell Interferenz auslösenden Komponenten, z.B. durch weitere Ausführungseinheiten innerhalb der Plattform, ist dies jedoch erneut zu betrachten.

In Bezug auf Anforderung 1 wurde der vorgestellte Ansatz hinsichtlich der erreichten Zeit paralleler Ausführung von Echtzeit- und Best-Effort-Tasks in verschiedenen Interferenzszenarien unter Nutzung synthetischer Anwendungen evaluiert. Im Vergleich zu einem prioritätsbasierten Referenzansatz konnte, in Abhängigkeit von der Stärke der auftretenden Interferenzeffekte, eine Steigerung der Zeit paralleler Ausführung um 22 % bis 41 %[4] erreicht werden. Weiterhin wurde die Abhängigkeit der erreichten Zeit paralleler Ausführung von verschiedenen Konfigurationsparametern untersucht (Abschnitt 5.5.2.1) und die Auswirkungen der parallelen Ausführung auf die erreichbare Performanz betrachtet (Abschnitt 5.5.2.2).

[4] Bei minimaler Länge des ET-Zeitfensters im Hypervisor-Schedule ($t_{slack} = 0\,\text{ms}$, siehe Abschnitt 5.4.3.2). Bei Festlegung längerer Zeitfenster kann die Zeit paralleler Ausführung weiter gesteigert werden (vgl. Abschnitte 5.4.3.3 und 5.4.3.5).

6 FPGA-basierte Echtzeit-Datenverarbeitung in Hypervisor-verwalteten SoC

In diesem Kapitel werden Konzepte vorgestellt, welche die Partitionierung integrierter FPGA-Komponenten in Hypervisor-verwalteten MPSoC ermöglichen und diese zur Umsetzung von Echtzeit-Datenverarbeitung nutzbar machen. Sie ermöglichen somit sowohl die FPGA-basierte Umsetzung von Beschleunigerkomponenten als Teil von Hypervisor-Partitionen als auch die Umsetzung von Echtzeitfunktionen unter Umgehung von Interferenzkanälen mit weiteren Komponenten der Plattform.

Während die Konzepte in verschiedenen Anwendungsdomänen genutzt werden können, wird ihre Anwendung in diesem Kapitel am Beispiel hochflexibler Produktionssysteme diskutiert.

Die in diesem Kapitel beschriebenen Konzepte wurden teilweise in den eigenen Veröffentlichungen [SKM+23, SKM+22] publiziert. Ihre Umsetzung erfolgte in den betreuten Abschlussarbeiten [Kre21, Kar21].

6.1 Umfeld und Zielsetzung

FPGAs sind integrierte Schaltkreise, welche die Umsetzung beliebiger Logikschaltungen durch Programmierung bzw. Konfiguration ermöglichen und den schnellen Austausch dieser zur Laufzeit (Rekonfiguration) erlauben (vgl. Abschnitt 2.2.3). Somit ermöglichen sie u.a. die Umsetzung von anwendungsoptimierten Recheneinheiten (Hardwarebeschleunigern) oder Schnittstellenlogik.

Die Möglichkeit, Funktionalität auf Ebene von Logikschaltungen zu implementieren, kann im Kontext von Systemen mit harten Echtzeitanforderungen vorteilhaft sein. Dies ist insbesondere der Fall, wenn die Anwendung sehr kurze Reaktionslatenzen erfordert. Durch die feingranulare Spezifikation der Datenverarbeitungslogik ist eine taktgenaue Analyse des Zeitverhaltens der resultierenden FPGA-Implementierung möglich. Weiterhin kann der Schaltungsentwurf hinsichtlich verschiedener Aspekte optimiert werden, wie z.B. Verarbeitungslatenz, Datendurchsatz, Energieaufnahme oder FPGA-Ressourcenverbrauch.

Moderne (MP)SoC, wie die Zynq- und Versal-Produktreihen von AMD bzw. die SoC-FPGA-Produkte von Intel, enthalten FPGA-Komponenten, welche über leistungsfähige Schnittstellen mit den weiteren Komponenten des SoC verbunden sind. Diese On-Chip-Integration ermöglicht hohe Datenraten an der Schnittstelle zwischen FPGA und weiteren SoC-Komponenten und bietet die Möglichkeit, bei der Umsetzung von Funktionen auf dem SoC zwischen Hardware-Implementierungen und Software-Implementierungen zu entscheiden. Gleichzeitig sind die integrierten FPGA-Komponenten meist an externe Schnittstellen des SoC angebunden, wodurch die direkte Kommunikation mit externen Komponenten möglich ist.

Unter Ausnutzung dieser Eigenschaften sollen in diesem Kapitel Hardware/Software-Architekturkonzepte vorgestellt werden, welche die FPGA-basierte Umsetzung anwendungsspezifischer Datenverarbeitungsfunktionen in Hypervisor-partitionierten MPSoC ermöglichen. Sie sollen neben klassischen Hardwarebeschleunigern insbesondere die Umsetzung von Anwendungen zur Interaktion externen Komponenten im FPGA ermöglichen, an welche harte Echtzeitanforderungen gestellt sind. Im Hinblick auf wechselnde Anwendungsszenarien des eingebetteten Systems, wie dem im Folgenden beschriebenen Beispiel, soll die Architektur den modularen Austausch von Anwendungen und FPGA-Implementierungen zur Laufzeit ermöglichen.

6.1.1 Anwendungsfall hochflexible Produktionssysteme

Hochflexible Produktionssystemkonzepte zeichnen sich im Vergleich zu klassischen Produktionssystemen durch ihre höhere Wandelbarkeit aus. Sie zielen auf die effiziente Produktion einer großen Produkt- und Variantenvielfalt ab, um der zunehmenden Individualisierung von Produkten (vgl. *Mass Customization* und *Personalization* [190]) und schnell wechselnden Marktanforderungen zu begegnen.

Ein Beispiel für ein hochflexibles Produktionssystem hierfür ist das in [192] vorgestellte Konzept *Wertstromkinematik (WSK)*. Anstelle spezialisierter Maschinen sieht es die Nutzung universell einsetzbarer, an Industrierobotern orientierten *Kinematiken* vor, mit welchen eine Vielzahl unterschiedlicher Produktionsprozesse durchgeführt werden sollen. Die Unterstützung verschiedener Handhabungs- und Fertigungsschritte wird hierbei durch automatisiert wechselbare Endeffektoren ermöglicht, welche die benötigten Werkzeuge für den jeweiligen Schritt enthalten. Um Prozesse mit hohen Steifigkeitsanforderungen zu ermöglichen, ist die zeitweise Kopplung von Kinematiken vorgesehen. Dies ist in Abbildung 6.1 dargestellt.

Durch die häufige und automatisierte Rekonfiguration stellen solche hochflexiblen Produktionssysteme besondere Anforderungen an die zugehörige Steuerungs- und Datenverarbeitungsarchitektur. Verschiedene Handhabungs- und Fertigungsschritte bedeuten

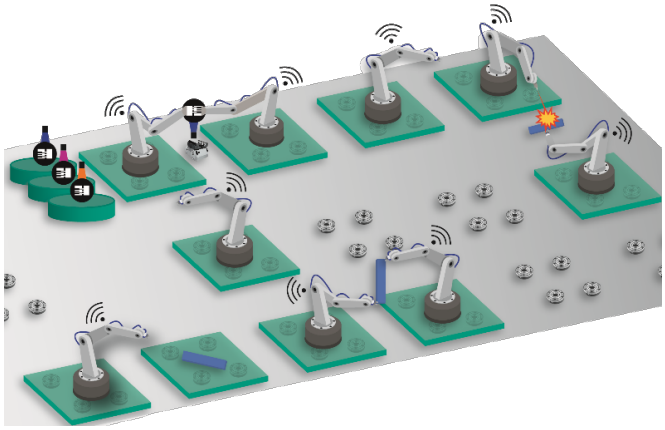


Abbildung 6.1: Visualisierung des hochflexiblen Produktionssystemkonzepts *Wertstromkinematik*. Verschiedene Produktionsschritte werden durch universell einsetzbare Kinematiken umgesetzt. Dies wird durch wechselbare Endeffektoren mit den geeigneten Werkzeugen und Kopplung von Kinematiken ermöglicht. (Ausschnitt aus [191])

unterschiedliche Datenverarbeitungsaufgaben, welche wiederum unterschiedliche Anforderungen an die zu ihrer Umsetzung genutzte Rechenhardware stellen. Diese ergeben sich insbesondere hinsichtlich der erforderlichen Verarbeitungslatenz und des Umfangs der Datenverarbeitung. So ist im Kontext der Prozessüberwachung beispielsweise oft schnelle Datenverarbeitung und teilweise das Erreichen von Echtzeitverhalten nötig, während die Dauer der Datenverarbeitung in Predictive-Maintenance-Anwendungen häufig eine untergeordnete Rolle spielt (vgl. Abschnitt 4.1). Bezüglich des Umfangs der benötigten Datenverarbeitung ist die Auswertung hochauflösender Kamerabilder, zum Beispiel zum Zweck der Qualitätskontrolle, meist deutlich rechenintensiver als die Auswertung von Berührungssonden zur Positionsfeststellung. Im Hinblick auf die Effizienz des Datenverarbeitungssystems ist eine geteilte Nutzung von Recheneinheiten im Produktionssystem vorteilhaft, um längerfristige Unterauslastung dieser bei unterschiedlich verteilten Anforderungsprofilen zu vermeiden.

6.2 FPGA-basierte Umsetzung von Echtzeitfunktionen

Der in diesem Kapitel verfolgte grundlegende Ansatz zur Erreichung von Echtzeitfähigkeit ist die Implementierung von Echtzeitfunktionalität in FPGA-Komponenten. Hierzu werden die an der Echtzeitanwendung auf Systemebene beteiligten Komponenten direkt an den FPGA angebunden. Durch die räumliche Trennung von Echtzeit-Funktionen bzw.

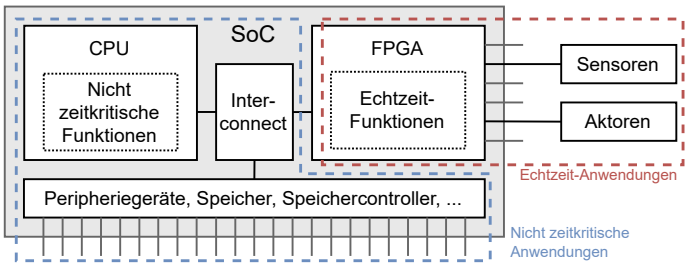


Abbildung 6.2: Umsetzung von FPGA-basierten Echtzeit-Funktionen anhand einer vereinfachten Darstellung der Architektur FPGA-unterstützter SoC

-Komponenten und parallel ausgeführten nicht zeitkritischen Funktionen wird Interferenz zwischen den beiden Systempartitionen vermieden.

Der verfolgte Ansatz ist in Abbildung 6.2 anhand einer abstrakten SoC-Architektur dargestellt, wie sie bspw. in den AMD-Zynq-SoC zu finden ist [39, 193]. Die SoC-Architektur umfasst einen oder mehrere CPU-Cluster, eine FPGA-Komponente sowie verschiedene weitere Komponenten wie Peripheriekomponenten, lokale Speicher und Speichercontroller, über welche externe Speicher (häufig DDR-RAM) angebunden wird. Die Komponenten des SoC sind über ein Interconnect verbunden, welches die Ansteuerung der Peripheriekomponenten und Zugriff auf Speicher durch die CPU und FPGA-basierte Logikschaltungen ermöglicht. Externe Schnittstellen des SoC werden zum einen von Peripheriekomponenten (*PS MIO Pins* in [39, 193]) und Speichercontrollern bereitgestellt, zum anderen ist auch die FPGA-Komponente direkt mit externen Schnittstellen des SoC verbunden (*PL Pins* in [39, 193]). Somit ist es möglich, externe Signale direkt an im FPGA instanziierte Logik anzubinden.

Um Interferenzeffekte zwischen Kernen der CPU-Cluster, an der Interconnect-Infrastruktur des SoC sowie an internen und extern angebundenen Speichern zu umgehen, können Datenverarbeitungsfunktionen, an welche harte Echtzeitanforderungen gestellt sind, in Form von Logikschaltungen im FPGA des SoC umgesetzt werden. Die Anbindung von Sensoren und Aktoren bzw. für die Echtzeitfunktion relevanten Kommunikationskanälen erfolgt hierbei über die externen Schnittstellen des FPGA (vgl. rot markierter Bereich in Abbildung 6.2). Die Echtzeitanwendung selbst kann im FPGA entweder auf Basis eines Soft-Core-Prozessors mit entsprechender Software oder in Form einer Logikschaltung umgesetzt sein.

Umfasst die Echtzeitanwendung Funktionen, welche auf weitere Komponenten des SoC zugreifen (vgl. blau markierter Bereich in Abbildung 6.2), ist zu beachten, dass bei diesen Zugriffen Interferenz mit parallel, z.B. auf den CPU-Kernen, ausgeführten Anwendungen

auftreten kann. Dies betrifft z.B. Anwendungen, in welchen erfasste Sensordaten neben der Echtzeitverarbeitung auch im RAM gespeichert werden sollen, um sie auf der CPU des SoC softwarebasiert weiterzuverarbeiten oder weiterzuleiten. In solchen Fällen muss bei der Umsetzung der Echtzeitanwendung sichergestellt werden, dass die Echtzeitfunktionen von den potenziell von Interferenz betroffenen Funktionen derart entkoppelt sind, dass Verzögerungen bei der Umsetzung letzterer keine Auswirkungen auf die Einhaltung von Echtzeitgarantien haben.

Auf den CPU-Clustern des SoC können parallel zeitunkritische Funktionen implementiert werden. Im Hinblick auf das Echtzeitverhalten der im FPGA implementierten Komponente können diese grundsätzlich uneingeschränkt auf Peripheriekomponenten, lokalen sowie externen Speicher zugreifen. Es muss lediglich sichergestellt werden, dass die Betriebsfähigkeit der FPGA-basierten Funktionen erhalten bleibt.

6.3 Hypervisor-Integration und Speicherisolation FPGA-basierter Funktionen

Zur Integration FPGA-basierter Funktionen in Hypervisor-partitionierten Systemen wird im Folgenden eine Architektur vorgestellt, welche den im SoC integrierten FPGA partitioniert und diese FPGA-Partitionen in den Kontext von Hypervisor-Partitionen einbindet. Dies ermöglicht die Umsetzung von Anwendungen in Hypervisor-Partitionen, welche neben softwarebasierten Funktionen auch FPGA-basierte Anwendungslogik umfassen. Der Fokus der Architektur liegt hierbei auf der Sicherstellung der räumlichen Isolation, der Minimierung von Einschränkungen hinsichtlich der umsetzbaren Anwendungen und der modularen Austauschbarkeit von Anwendungen zur Laufzeit.

Die im Folgenden vorgestellte Architektur wurde im Kontext einer Edge-Einheit für ein hochflexibles Produktionssystemkonzept angewandt und in [SKM+23] publiziert. Ihre Umsetzung auf dem Zynq UltraScale+ MPSoC [39] erfolgte in der betreuten Abschlussarbeit [Kre21].

6.3.1 Anforderungen

Etablierte Hypervisors ermöglichen die isolierte Ausführung von Software innerhalb mehrerer logischer Partitionen auf geteilt genutzter Ausführungshardware. Die im Folgenden vorgestellte Architektur soll diese Partitionierung auf FPGA-Komponenten in SoC-Plattformen ausweiten, sodass neben Softwarekomponenten auch FPGA-basierte Funktionen (im Folgenden als *FPGA-Anwendungslogik* bezeichnet) innerhalb einer Partition genutzt werden können.

Die Architektur wurde auf Basis der folgenden Anforderungen entwickelt:

- Die Architektur soll die gleichzeitige Ausführung mehrerer FPGA-unterstützter Anwendungen auf einem SoC ermöglichen. Analog zur gleichzeitigen Ausführung von Software verschiedener Partitionen auf der CPU soll auch die gleichzeitige Ausführung von FPGA-Anwendungslogik mehrerer Partitionen möglich sein. Im Hinblick auf den in Abschnitt 6.1.1 vorgestellten Anwendungsfall ermöglicht dies die Umsetzung von Datenverarbeitungsfunktionen verschiedener Kinematiken auf einer Recheneinheit.
- Die Architektur soll zu möglichst geringen Einschränkungen bei der Umsetzung von FPGA-Anwendungslogik führen. Hierzu soll die Anbindung dieser an die weiteren Systemkomponenten über eine möglichst generische Schnittstelle erfolgen. Diese soll insbesondere das Auslösen von Interrupts an die CPU sowie das Durchführen von DMA-Transaktionen aus der Anwendungslogik heraus ermöglichen.
- Die Architektur soll die räumliche Partitionierung sicherstellen. Hierzu soll verhindert werden, dass über FPGA-basierte Anwendungslogik Zugriff auf Speicherbereiche oder Systemkomponenten erlangt werden kann, welche der Partition nicht zugewiesen sind.
- Die Architektur soll den modularen Austausch einzelner Partitionen zur Laufzeit unterstützen. Sie soll das Laden und Entladen von Anwendungen ermöglichen, ohne die Ausführung parallel auf der Plattform aktiver Anwendungen zu beeinträchtigen. Diese Anforderung soll die Anwendbarkeit in rekonfigurierbaren, modularen Systemen sicherstellen, in welchen je nach Teilsystemzustand unterschiedliche Anwendungen ausgeführt werden müssen (vgl. bspw. Abschnitt 6.1.1).

6.3.2 Systemübersicht

Im Folgenden wird ein Überblick über die Architektur zur Integration von FPGA-Komponenten in Hypervisor-partitionierte SoC gegeben. Hierzu wird zunächst die Partitionierung des SoC dargestellt. Im nachfolgenden Abschnitt wird anschließend die Umsetzung der räumlichen FPGA-Partitionierung beschrieben.

Abbildung 6.3 stellt die Partitionierung des SoC dar. Softwareseitig erfolgt die Partitionierung mittels eines Hypervisors, welcher die Zuweisung von CPU-Ressourcen, Speicher und Peripheriekomponenten zu Partitionen durchsetzt und die Ausführung der Gastanwendungen und -OS kontrolliert.

Um die Integration von FPGA-Anwendungslogik zu ermöglichen, wird der FPGA mittels einer *Shell* partitioniert. Die Shell ist ein statisches FPGA-Design, welches PRR definiert, in welchen die anwendungsspezifischen FPGA-Funktionen umgesetzt werden können.

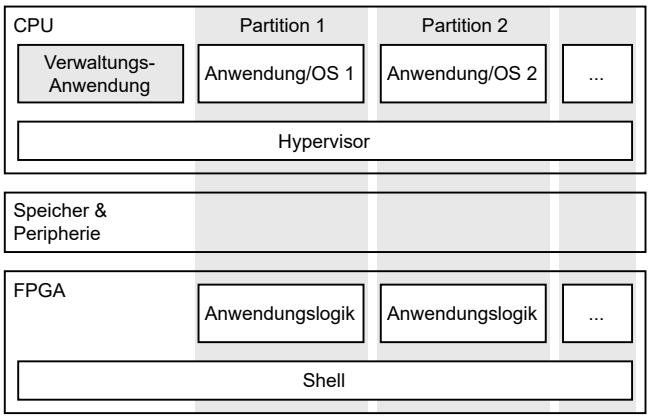


Abbildung 6.3: Partitionierung des SoC (nach [SKM+23])

Durch die Anwendung von DPR kann Anwendungslogik in einer PRR zur Laufzeit ausgetauscht werden, ohne die Ausführung der Anwendungslogik in anderen Regionen zu beeinflussen.

Neben den PRR selbst umfasst die Shell Logik zur Anbindung der PRR an weitere Komponenten des SoC. Hierbei ist in der Shell Logik zur Umsetzung der räumlichen Partitionierung implementiert. Sie stellt sicher, dass von einer PRR ausgehende Speicherzugriffe und Interrupts eindeutig identifiziert werden können, sodass Interrupts der zugehörigen Partition auf der CPU zugeordnet werden können und Zugriffe auf Speicherbereiche und Peripheriekomponenten auf die der Partition zugewiesenen Ressourcen beschränkt sind. Die Schnittstelle zur PRR ist in Abschnitt 6.3.3 detaillierter beschrieben, die Umsetzung der Speicherisolation in Abschnitt 6.3.4.

Zuletzt umfasst die Architektur eine Verwaltungsanwendung, welche in einer privilegierten Hypervisorpartition ausgeführt wird und das Laden und Entladen von Anwendungen kontrolliert. Sie koordiniert das Starten und Stoppen von Hypervisor-Partitionen und das Laden und Entladen der zugehörigen, FPGA-basierten Anwendungslogik mittels DPR.

6.3.3 Schnittstelle der rekonfigurierbaren FPGA-Regionen

Wie im vorherigen Abschnitt beschrieben, setzt die Shell die Partitionierung des FPGA mittels DPR um. Hierzu definiert sie Ort und Größe rekonfigurierbarer FPGA-Regionen sowie deren Schnittstelle (im Folgenden *PRR-Schnittstelle* genannt) hin zu weiteren SoC-Komponenten.

Um möglichst vielfältige Anwendungslogik zu unterstützen, umfasst die PRR-Schnittstelle

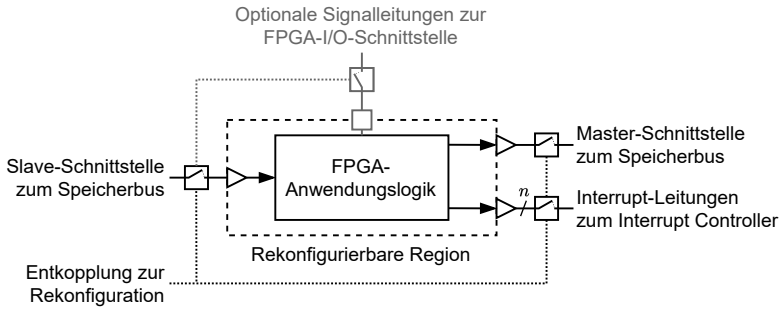


Abbildung 6.4: Schnittstellen der rekonfigurierbaren FPGA-Region

eine Master- und eine Slave-Schnittstelle zur Speicherinfrastruktur sowie Interruptleitungen (vgl. Abbildung 6.4). Über die Master-Schnittstelle kann die FPGA-Anwendungslogik aktiv auf Speicher und Peripheriekomponenten zugreifen (DMA). Über die Slave-Schnittstelle kann sie Register und Speicher zum Zugriff durch andere Systemkomponenten bereitstellen. Über die Interruptleitungen können Interrupts an die CPU ausgelöst werden. Sie sind durch die Shell mit dem Interrupt Controller des SoC verbunden, welcher diese, durch den Hypervisor konfiguriert, an die zugehörige Partitionssoftware weiterleitet. Bei Bedarf kann weiterhin eine Schnittstelle für Signalleitungen zur direkten Anbindung SoC-externer Komponenten umgesetzt werden. Diese Schnittstelle der PRR wird durch die Shell direkt mit der aus dem FPGA erreichbaren I/O-Schnittstelle des SoC verbunden.

Um die Rekonfiguration zur Laufzeit zu ermöglichen, umfasst die Shell Komponenten zur Entkopplung der PRR. Über diese wird die PRR während der partiellen Rekonfiguration von der umgebenden Shell getrennt, um die Weiterverarbeitung ungültiger Signale zu verhindern. Zusätzlich können als Teil der Shell an jeder Master-Schnittstelle Module zur Fehlerisolation angefügt werden (z.B. AXISAFETY [175] in AXI-Bus-basierten SoC). Diese fangen fehlerhaftes Verhalten der Anwendungslogik ab und können so weitreichende Störungen durch inkorrekte Implementierungen des Speicherbusprotokolls in der Anwendungslogik verhindern.

6.3.4 Umsetzung der Speicherisolation

Wie in Abschnitt 2.4.4.2 beschrieben, wird Speicherpartitionierung von Hypervisors auf Plattformen mit Virtualisierungsunterstützung meist mittels der MMU erreicht. Die MMU wird vom Hypervisor so konfiguriert, dass sie eine Adressübersetzung von VM-physischen zu den der Partition zugewiesenen Host-physischen Adressen durchführt. Somit erhält die Partition nur Zugriff auf Adressbereiche, welche ihr per Konfiguration zugeteilt sind.

Zur Durchsetzung der Speicherisolation für FPGA-Anwendungslogik muss analog sichergestellt werden, dass von der Master-Schnittstelle einer PRR aus nur die der jeweiligen Partition zugewiesenen Adressbereiche erreichbar sind. Gleichzeitig muss beachtet werden, dass die Partitions-Software VM-physische Adressierung nutzt, während FPGA-basierte Logik ohne weitere Maßnahmen mit Host-physischen Speicheradressen arbeitet. Um die Anwendungsentwicklung zu vereinfachen, ist es daher vorteilhaft, die von der MMU umgesetzte Adressübersetzung von VM-physischen zu Host-physischen Adressen auch auf Speicherzugriffe der FPGA-basierten Anwendungslogik anzuwenden. Beide Aspekte können durch Nutzung der IOMMU (bzw. SMMU) des SoC erreicht werden. Analog zur MMU ermöglicht diese Adressübersetzungen für Speicherzugriffe CPU-externer Komponenten (vgl. Abschnitt 2.4.4.3).

Die IOMMU des SoC unterscheidet verschiedene Master-Komponenten der Speicherinfrastruktur anhand einer ID, welche als Teil jeder Speichertransaktion angegeben wird. Diese ID ist für alle Master-Komponenten im statischen Teil des SoC, dem *Processing System*, eindeutig definiert. Für Speicherzugriffe aus dem FPGA werden an der Schnittstelle zum Processing System IDs generiert, welche sich aus einem statischen Teil und einem dynamischen Teil zusammensetzen. Der dynamische Teil kann hierbei von der im FPGA instanziierten Master-Komponente definiert werden. [194]

Abbildung 6.5 zeigt die zur Umsetzung der Speicherpartitionierung relevanten Teile der Speicherinfrastruktur der Shell und des Processing System. Um eine eindeutige Identifikation der aus der FPGA-Anwendungslogik heraus durchgeführten Speicherzugriffe zu ermöglichen, ist der Master-Schnittstelle jeder rekonfigurierbaren Region eine Komponente nachgeschaltet, welche diese ID auf einen eindeutigen, bekannten Wert setzt. Dies erfolgt unabhängig vom Verhalten der Anwendungslogik, wodurch Manipulation der ID durch diese verhindert wird. Die resultierenden, für jede PRR eindeutigen IDs ermöglichen die Unterscheidung der Speicherzugriffe aus verschiedenen PRR an der IOMMU. Unter Nutzung dieser IDs wird die IOMMU so konfiguriert, dass sie für Speicherzugriffe aus PRR die gleichen Adressübersetzungen durchführt, welche für die zugehörige Partitionssoftware durch die MMU erfolgen[1]. Somit kann bei der Entwicklung von FPGA-Anwendungslogik das gleiche Speicherlayout genutzt werden wie bei der Entwicklung der Software innerhalb der zugehörigen Hypervisor-Partition[2].

[1] Im Gegensatz zur MMU beschränkt sich die IOMMU auf die Übersetzung zwischen VM-physischen und Host-physischen Adressen.

[2] Das aus Sicht der FPGA-basierten Anwendungslogik nutzbare Speicherlayout entspricht dem VM-physischen Speicherlayout auf Softwareseite (vgl. Abbildung 2.10). Ein in der VM ausgeführtes Betriebssystem kann eine zweite Adressübersetzungsstufe von VM-virtuellen auf VM-physische Adressen konfigurieren. In diesem Fall liegt es in der Verantwortung des Anwendungsentwicklers, diese zweite Adressübersetzung zu berücksichtigen. Dies entspricht der Situation in nichtvirtualisierten Systemen.

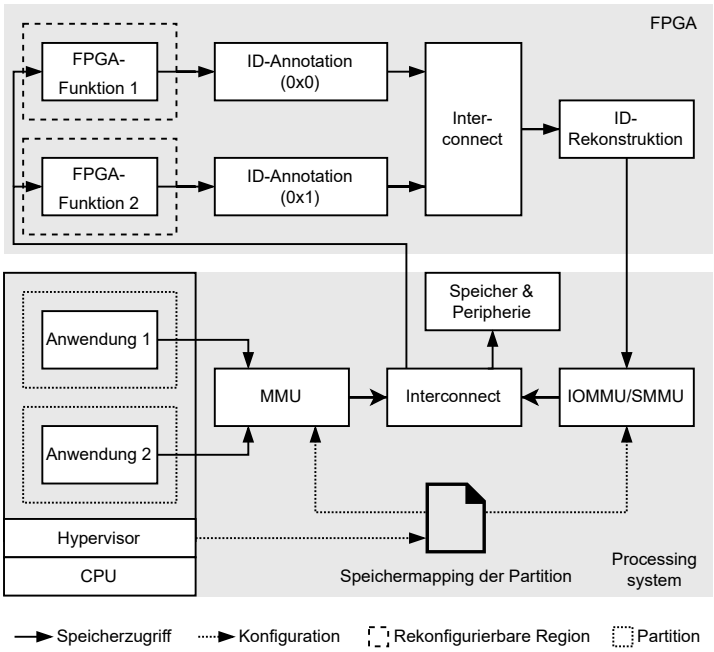


Abbildung 6.5: Speicherinfrastruktur zur einheitlichen Speicherpartitionierung von Hypervisor-Partitionen und FPGA-Anwendungslogik (nach [SKM+23])

6.4 Adaptive Anbindung externer Komponenten

Während der in Abschnitt 6.2 beschriebene Ansatz zur FPGA-basierten Umsetzung von Echtzeitfunktionen in vielen Anwendungen ausreicht, stößt er in hochdynamischen Umgebungen an seine Grenzen. Sollen wechselnde Sensoren und Aktoren angebunden werden, kann dies die Umsetzung verschiedener Kommunikationsprotokolle erfordern, welche sich sowohl hinsichtlich der logischen als auch physischen Schnittstelle unterscheiden.

Die Nutzung des FPGA zur Anbindung von Sensoren und Aktoren bietet hier das Potenzial, austauschbare Kommunikationslogik effizient umzusetzen. Das im Folgenden vorgestellte Konzept schöpft dieses Potenzial aus, um die flexible Anbindung von Sensoren und Aktoren über verschiedene Kommunikationsprotokolle zu ermöglichen.

Das in diesem Kapitel beschriebene Konzept wurde in [SKM+22] im Kontext einer Einheit zur Sensorintegration für hochflexible Produktionssysteme veröffentlicht. Die Umsetzung auf Basis eines Zynq 7000 SoC [193] erfolgte in der betreuten Abschlussarbeit [Kar21]. Der in Abschnitt 6.4.2.3 beschriebene Ansatz zur unabhängigen Rekonfiguration mehrerer parallel genutzter Kommunikationsprotokolle geht über das bereits Veröffentlichte hinaus.

6.4.1 Motivation und Anforderungen

Die in Abschnitt 6.2 beschriebene Umsetzung von Echtzeitfunktionen basiert auf der direkten Anbindung externer Komponenten an den FPGA. Zur Umsetzung der von Sensoren und Aktoren genutzten Kommunikationsprotokolle ist hierbei in den meisten Fällen zusätzliche Hardware erforderlich. Während Protokollkomponenten des Data Link Layers (DLL, vgl. ISO/OSI-Referenzmodell [195]) und höherer Schichten im FPGA umgesetzt werden können, müssen Komponenten des Physical Layer (PHY) zur Pegelwandlung und ggf. Terminierung der Kommunikationsleitungen als externe Komponenten (sog. *Transceiver*) an das SoC angebunden werden.

Sowohl die Umsetzung von Kommunikationslogik im FPGA als auch Anbindung externer Transceiver ist ein etabliertes Vorgehen bei der Umsetzung von Kommunikationsschnittstellen. Daher ist anzunehmen, dass die in Abschnitt 6.2 beschriebene Architektur für viele eingebettete Systeme realisierbar ist, in denen über feste Kommunikationsprotokolle mit externen Komponenten interagiert werden soll. Erfordert der Anwendungsfall die Kommunikation mit wechselnden Komponenten unter Nutzung unterschiedlicher Protokolle, stößt der vorgestellte Ansatz jedoch an seine Grenzen. Während der FPGA das Potenzial zur Rekonfiguration bietet, ist bei der Umstellung des Kommunikationsprotokolls oft ein Austausch der SoC-externen Transceiver notwendig.

Ein Beispiel für einen solchen Anwendungsfall stellt das in Abschnitt 6.1.1 vorgestellte Produktionssystem dar. Durch die Möglichkeit zum Wechsel der Endeffektoren wird eine Vielzahl unterschiedlicher Sensoren und Aktoren an die Kinematik angebunden, welche von der Steuerungsinfrastruktur ausgewertet bzw. angesteuert werden müssen. Die Schnittstelle zwischen Endeffektor und Kinematik ist hierbei fest und soll über die Lebenszeit der Kinematik mit verschiedensten Endeffektoren kompatibel sein. Gleichzeitig ist die Anzahl verfügbarer Signalleitungen beschränkt, um die Größe und damit das Gewicht des zur Anbindung des Endeffektors genutzten Koppelmoduls gering zu halten.

Während die Spezifikation eines einheitlichen Kommunikationsprotokolls eine mögliche Lösung dieses Problems darstellt, ist sie auch mit Nachteilen verbunden. So ist die Auswahl an integrierbaren Sensoren und Aktoren auf solche beschränkt, welche das geforderte Protokoll unterstützen. Alternativ müssen Protokollwandlerkomponenten eingebracht werden, welche Kosten und Gewicht des Endeffektors erhöhen.

Aus diesem Grund wird im Folgenden ein alternatives Konzept zur flexiblen Anbindung solcher externen Komponenten über eine rekonfigurierbare Kommunikationsschnittstelle vorgestellt. An dieses sind folgende Anforderungen gestellt:

- Es soll die Anbindung externer Komponenten unter Nutzung verschiedener Kommunikationsprotokolle unterstützen. Die Kommunikation soll hierbei über eine

begrenzte, möglichst geringe Anzahl an Signalleitungen erfolgen.

- Es soll die Umsetzung des in Abschnitt 6.2 beschriebenen Ansatzes ermöglichen, Datenverarbeitungsfunktionen mit harten Echtzeitanforderungen FPGA-basiert zu implementieren.
- Die für den Protokollwechsel benötigte Zeit soll geringer sein als die zum Austausch der angebundenen Komponenten benötigte Zeit. Zur Orientierung wird das betrachtete Produktionssystemkonzept herangezogen. Im Kontext dieses Systems kann von einer Dauer im Bereich einiger Sekunden ausgegangen werden.

6.4.2 Konzept und Systemarchitektur

Das in diesem Abschnitt vorgestellte Konzept zielt darauf ab, verschiedene Sensoren und Aktoren (im Folgenden als Komponenten bezeichnet) über eine feste physische Kommunikationsschnittstelle mit geringer Leitungszahl an ein SoC anzubinden.

Verschiedene Kommunikationsprotokolle bzw. -protokollstapel unterscheiden sich hinsichtlich physischer und logischer Eigenschaften. Die physischen Eigenschaften sind hierbei im PHY-Layer des Protokollstapels definiert und umfassen bspw. die Anzahl genutzter Signalleitungen, Spannungslevel bzw. Stromstärken, Leitungsterminierung und Signalformen. Logische Eigenschaften sind unabhängig von der physischen Repräsentation der Information. Für einzelne Kommunikationsverbindungen umfassen sie bspw. Mechanismen zur Erkennung und Korrektur von Übertragungsfehlern und zur Arbitrierung des Zugriffs bei geteilten Kommunikationsmedien. Diese sind im Data Link Layer des Protokollstapels definiert. In komplexeren Netzwerken umfassen sie weiterhin Mechanismen zur Wegfindung, Stauvermeidung und Sicherstellung zuverlässiger Kommunikation (vgl. Schichten 3 und 4 des ISO/OSI-Referenzmodells [195]).

Um verschiedene Protokollstapel zu unterstützen, müssen sowohl verschiedene PHY-Layer-Implementierungen als auch verschiedene Implementierungen der höheren Protokollschichten unterstützt werden. Eine Architektur, welche dies unter Nutzung einer begrenzten Anzahl an Signalleitungen erreicht, ist in Abbildung 6.6 dargestellt.

6.4.2.1 Aufbau der Architektur

Die Architektur basiert auf einem SoC, welches neben CPU-Kernen auch einen FPGA umfasst. Dieser wird zur Umsetzung der Protokolllogik genutzt und kann weiterhin Funktionen zur Echtzeitdatenverarbeitung umfassen. Die Protokolllogik und die Echtzeit-Funktionen werden hierbei in einer rekonfigurierbaren FPGA-Region platziert, sodass sie beim Wechsel des Kommunikationsprotokolls bzw. der Sensor-/Aktorkomponente mittels

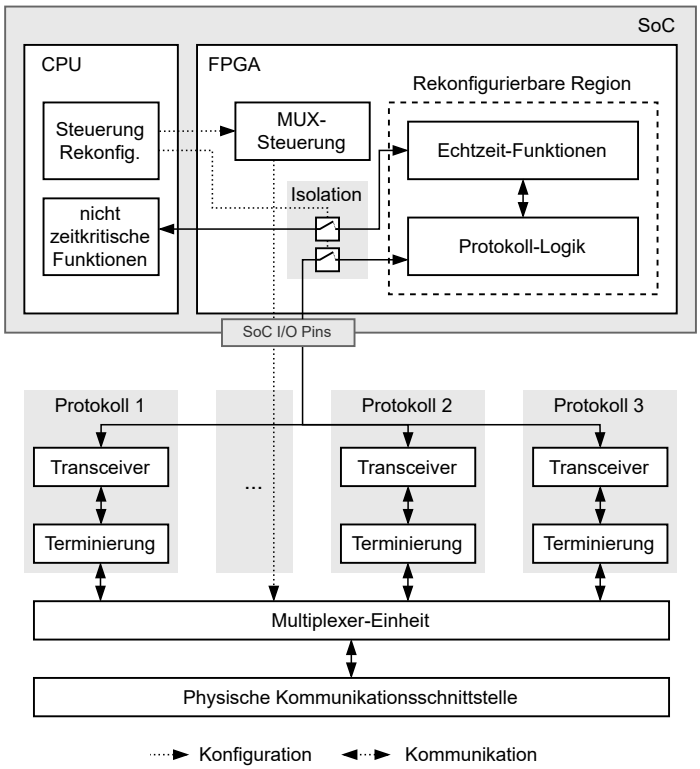


Abbildung 6.6: FPGA-basierte Anbindung unterschiedlicher Kommunikationsprotokolle über eine statische Schnittstelle (aufbauend auf [SKM+22])

DPR ausgetauscht werden können. Werden separate rekonfigurierbare Regionen definiert, können Protokoll- und Anwendungslogik unabhängig voneinander ausgetauscht werden.

Da die zur Umsetzung der physischen Protokolleigenschaften notwendigen Komponenten in der Regel nicht im SoC integriert sind, wird für jedes zu unterstützende Protokoll eine entsprechende *PHY-Schaltung* ans SoC angebunden. Diese umfasst den Transceiver sowie die ggf. benötigte Schaltung zur Leitungsterminierung. Die Anbindung der Transceiver erfolgt hierbei über I/O-Pins des SoC, welche direkt an den FPGA angebunden sind.

Die Anbindung der protokollspezifischen PHY-Schaltungen an die Sensorschnittstelle erfolgt über eine Multiplexer-Einheit. Sie ist so ausgeführt, dass jede Signalleitung jeder PHY-Schaltung mit einer beliebigen Leitung der physischen Kommunikationsschnittstelle verbunden werden kann. Die Multiplexer-Einheit ermöglicht hierbei bidirektionale Kommunikation; sie kann bspw. durch Analogmultiplexer realisiert werden.

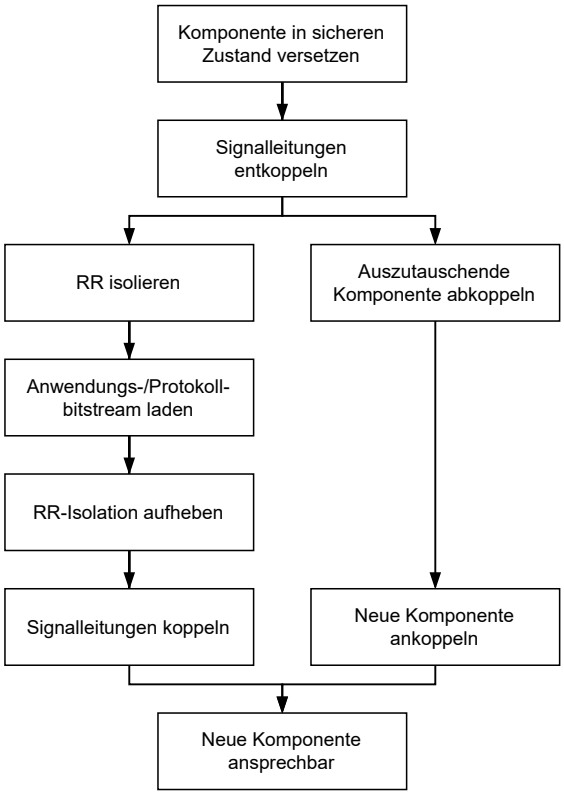


Abbildung 6.7: Ablauf der Protokollrekonfiguration

Die Ansteuerung der Multiplexer und somit die Auswahl der Schnittstellenbelegung erfolgt über Steuersignale, welche im statischen Teil des FPGA generiert werden. Die Schnittstellenbelegung wird hierbei von einer Steuerungsanwendung auf der CPU des SoC vorgegeben, welche den Wechsel zwischen Kommunikationsprotokollen kontrolliert.

6.4.2.2 Ablauf der Protokollrekonfiguration

Abbildung 6.7 zeigt den Ablauf der Protokollrekonfiguration beim Wechsel der angebundenen Komponente.

Vor Beginn des Komponententauschs muss sichergestellt werden, dass sich die angebundene Komponente in einem sicheren Zustand befindet. Ist dies erreicht, werden die bisher genutzten Signalleitungen der physischen Schnittstelle durch Konfiguration der Multiplexer-Einheit von der PHY-Schaltung entkoppelt. Anschließend kann der me-

chanische Entkopplungsvorgang der Komponente beginnen und eine neue Komponente gekoppelt werden.

Parallel zum physischen Komponentenwechsel beginnt die Rekonfiguration des FPGA: Zunächst wird die PRR isoliert, um Störsignale an den Transceivern und SoC-internen Schnittstellen während der Rekonfiguration zu verhindern. Daraufhin wird die neue Anwendungs- und Protokolllogik in die PRR geladen und die Isolation der PRR aufgehoben. Im Anschluss wird die Multiplexer-Einheit mit der für das neue Protokoll benötigten Belegung der physischen Schnittstelle konfiguriert.

Sobald dies erfolgt und die neue Komponente gekoppelt ist, ist die Rekonfiguration abgeschlossen. Die neue Komponente kann durch die Anwendungslogik genutzt werden.

6.4.2.3 Modularität und Skalierbarkeit

Wird das in den vorangegangenen Abschnitten vorgestellte Konzept wie in Abbildung 6.6 dargestellt implementiert, ermöglicht es neben der sequenziellen Umsetzung einzelner Kommunikationsprotokolle auch die parallele Umsetzung verschiedener Protokolle auf unterschiedlichen Signalleitungen der physischen Schnittstelle. Wird die PRR des FPGA ausreichend groß gewählt, können verschiedene Protokolllogik-Blöcke parallel implementiert werden, welche wiederum parallel verschiedene PHY-Schaltungen nutzen. Begrenzt ist die Anzahl der gleichzeitig nutzbaren Protokolle in diesem Fall durch die Anzahl verfügbarer, zu den Protokollen kompatiblen Signalleitungen an der physischen Schnittstelle. In diesem Fall erfordert der Austausch eines Kommunikationsprotokolls die Rekonfiguration der gesamten PRR und unterbricht somit die Ausführung von Echtzeit-Funktionen.

Um die Unterbrechung der Anwendungslogik zu umgehen, können separate PRR für Protokoll- und Anwendungslogik genutzt werden. Dies erlaubt den unabhängigen Austausch von Echtzeit-Funktionen und Protokolllogik. So kann auch die Rekonfigurationsdauer reduziert werden, wenn nur eine der beiden Elemente ausgetauscht werden muss. Dies kann in Anwendungsfällen vorteilhaft sein, in denen sequenziell unterschiedliche Echtzeit-Funktionen ausgeführt werden sollen, welche die gleiche Kommunikationsschnittstelle nutzen. Bei paralleler Nutzung mehrerer Protokolle bleibt jedoch das Problem kurzer Verbindungsunterbrechungen bei Austausch eines Protokolls bestehen.

Um den vollständig unabhängigen Austausch parallel genutzter Kommunikationsprotokolle zu ermöglichen, kann das in Abbildung 6.6 dargestellte FPGA-Design um zusätzliche PRR erweitert werden. Wird jedes Protokoll in einer eigenen PRR umgesetzt, können diese unabhängig voneinander rekonfiguriert werden. Die Anbindung der PRR an die zugehörigen PHY-Schaltungen kann hierbei mittels einer im statischen Teil des FPGA implementierten Multiplexer-Logik erreicht werden. Im Kontext des in Abschnitt 6.4.1 beschriebenen Anwendungsfalls könnten so auch mehrere Endeffektorschnittstellen durch

eine Steuerungseinheit bedient werden. Soll dabei an mehreren Schnittstellen gleichzeitig das gleiche Kommunikationsprotokoll genutzt werden, müssen mehrere PHY-Schaltungen dieses Protokolls im System verfügbar sein.

Um beim langfristigen Einsatz einer auf dem vorgestellten Konzept basierenden Steuerungseinheit Unterstützung für zusätzliche Kommunikationsprotokolle nachrüsten zu können, können PHY-Schaltungen mittels Erweiterungskarten realisiert werden.

6.4.3 Prototypische Umsetzung

Das in Abschnitt 6.4.2 vorgestellte Konzept wurde im Rahmen der Abschlussarbeit [Kar21] prototypisch umgesetzt, um seine Anwendbarkeit zur adaptiven Sensoranbindung im Kontext hochflexibler Produktionssysteme (vgl. Abschnitte 6.1.1 und 6.4.1) zu evaluieren. Der Ausarbeitung können technische Details der Implementierung entnommen werden. Im Folgenden wird ein Überblick gegeben.

Auf Basis einer Analyse häufig genutzter Kommunikationsprotokolle im industriellen Umfeld [196] wurden die Protokolle Ethernet, CAN und RS-485 sowie Unterstützung für einfache digitale I/O-Signale (im Folgenden als digitale I/O bezeichnet) integriert.

Die Implementierung basiert auf dem Avnet Zedboard [197], einem Entwicklungsboard für den ARM Zynq-7020 SoC [193], welches durch eine PHY-Erweiterungsplatine ergänzt wurde. Analog zu Abbildung 6.6 wurde die Protokoll-Logik im FPGA des SoC implementiert. Die Transceiver, Terminierung und die Multiplexer-Einheit wurden auf der Erweiterungsplatine umgesetzt.

Aufgrund der beschränkten Bandbreite des verwendeten Analog-Multiplexers musste die Ethernet-Schnittstelle separat ausgeführt werden. Sie ist somit dauerhaft auf festen Pins der physischen Sensorschnittstelle verfügbar. Dementsprechend ist die Implementierung der Ethernet-Protokolllogik als Teil des statischen FPGA-Designs umgesetzt, während für CAN, RS-485 und digitale I/O partielle Designs zur Integration in die PRR implementiert wurden.

Anstelle einer Implementierung der Anwendungslogik im FPGA wurde der FPGA nur zur Umsetzung der Protokolllogik bis zum Data-Link-Layer bzw. der MAC-Logik (Media-Access-Control-Logik) genutzt. Höhere Protokollschichten sowie Anwendungen wurden auf der CPU des SoC in Software unter Nutzung des Linux OS umgesetzt. Hierdurch konnten bestehende Gerätetreiber zur Interaktion mit der Protokolllogik in der PRR genutzt werden. Zur Aktivierung und Parametrisierung der nach jeder Rekonfiguration benötigten Treiber wurden *Device Tree Overlays (DTO)* genutzt. Diese Implementierung ist ausreichend, um die Protokollrekonfiguration zu untersuchen.

Tabelle 6.1: Mittlere und maximale Rekonfigurationslatenzen beim Protokollwechsel

Rekonfigurations-schritt	**Latenz [µs]**					
	CAN		**RS-485**		**Digital I/O**	
	Mw.	***Max.***	***Mw.***	***Max.***	***Mw.***	***Max.***
Leitungen entkoppeln	10.3	11.8	10.3	13.9	10.4	84.5
DTO entfernen	96 639	166 229	2050	81 563	937	101 158
PRR isolieren	16.4	28.1	20.2	447	14.8	98.4
Bitstream laden	132 773	134 330	133 337	136 270	132 979	135 182
PRR-Isolation aufheben	2.70	3.08	2.69	4.44	2.75	59.6
DTO laden	4512	6925	29 961	33 957	4115	4997
Leitungen koppeln	16.4	18.4	16.3	78.0	16.1	64.2
Gesamtlatenz	233 970	307 545	165 397	252 333	138 075	241 644

6.4.4 Evaluation

Zur Evaluation des Konzepts wurden die in Abschnitt 6.4.3 beschriebene Implementierung hinsichtlich der Rekonfigurationszeiten und des FPGA-Ressourcenverbrauchs untersucht. Weiterhin wurden Funktionstests der umgesetzten Protokolle durchgeführt.

Der Funktionstest erfolgte durch Anbindung eines Einplatinenrechners mit CAN- und RS-485-Schnittstelle. Mittels in [Kar21] entwickelter Testanwendungen auf dem Einplatinenrechner und dem SoC konnte die fehlerfreie Kommunikation über beide Schnittstellen festgestellt werden. Die genutzten Übertragungsraten betrugen hierbei 500 kbit/s für CAN bzw. 115.2 kBd für RS-485. Weiterhin konnte die CAN-Kommunikation unter Nutzung eines industriellen Sensors getestet mit einer Datenrate von 1 Mbit/s nachgewiesen werden.

Die Evaluation der Rekonfigurationslatenzen erfolgte durch Erweiterung der Steuerungsanwendung um eine Zeitmessung. Für jedes Protokoll wurden 1000 Messungen durchgeführt. Tabelle 6.1 zeigt die Zusammensetzung der gemessenen Rekonfigurationslatenzen für das CAN-Protokoll, das RS-485-Protokoll und die Digital-I/O-Umsetzung. Es ist erkennbar, dass die Rekonfigurationslatenz mit maximal 308 ms deutlich unter den geforderten Reaktionslatenzen liegt. Weiterhin ist erkennbar, dass in den meisten Fällen ein großer Teil der Rekonfigurationslatenz auf das Laden und Entladen des Device Tree Overlays entfällt. Wird die Anwendung vollständig im FPGA implementiert, entfällt dieser Aspekt. Im Gegenzug ist bei Umsetzung der Anwendung im FPGA ein höherer Ressourcenbedarf der PRR und eine längere Dauer zum Laden des Bitstreams zu erwarten.

Die von der Implementierung benötigten FPGA-Ressourcen sind in Tabelle 6.2 dargestellt. Das statische Design nimmt hierbei ca. 30 % der Lookup Tables und 20 % der Register

Tabelle 6.2: FPGA-Ressourcenbedarf auf dem Zynq-7020 (nach [SKM+22])

Design	LUT		Register		Multiplexer	
Statisch	15 687	(29.7 %)	21 098	(20.0 %)	30	(< 0.1 %)
CAN PRR	1040	(2.0 %)	917	(0.9 %)	12	(< 0.1 %)
RS-485 PRR	728	(1.4 %)	622	(0.6 %)	1	(< 0.1 %)
Digital I/O PRR	512	(1.0 %)	405	(0.4 %)	0	(0 %)

des FPGA in Anspruch, der Ressourcenbedarf der Protokolllogik in der PRR ist mit bis zu 2 % gering. Hierbei zeigt die CAN-Logik den höchsten Ressourcenbedarf, die Digitale-IO-Implementierung den niedrigsten. Es ist jedoch zu beachten, dass die PRR anhand des Ressourcenbedarfs des größten partiellen Designs dimensioniert werden muss.

6.5 Diskussion und Anwendbarkeit

Zunächst erfolgt eine Diskussion der in diesem Kapitel vorgestellten Konzepte. Basierend darauf wird die aus der gemeinsamen Betrachtung der Konzepte resultierende Gesamtarchitektur vorgestellt und hinsichtlich ihrer Anwendbarkeit zur Umsetzung von Echtzeitanwendungen in Hypervisor-partitionierten Systemen diskutiert.

6.5.1 FPGA-Partitionierung

Das in Abschnitt 6.3 vorgestellte Konzept ermöglicht die Umsetzung von Anwendungslogik im FPGA in mittels Hypervisors partitionierten Systemen. Es stellt die vom Hypervisor für Software umgesetzte räumliche Partitionierung auch für FPGA-basierte Anwendungslogik sicher und erlaubt die parallele Nutzung des FPGA aus verschiedenen Partitionen heraus.

Im Hinblick auf die in Abschnitt 6.3.1 beschriebenen Anforderungen ermöglicht es durch den Einsatz von DPR, FPGA-basierte Anwendungslogik zur Laufzeit modular auszutauschen. Da die Anwendungslogik als partielles Design innerhalb von PRR umgesetzt wird, läuft parallel aktive Anwendungslogik anderer Partitionen auch während der Rekonfiguration einer PRR weiter. Bei Nutzung eines geeigneten Hypervisors ist so das dynamische Laden und Entladen von Hypervisor-Partitionen samt zugehöriger Anwendungslogik im FPGA zur Laufzeit möglich.

Beschränkt ist dieser modulare Austausch von FPGA-Anwendungslogik durch die begrenzten FPGA-Ressourcen jeder PRR. Die PRR müssen bei im Rahmen der Systementwicklung so dimensioniert werden, dass ausreichend FPGA-Ressourcen zur Umsetzung

der darin zu platzierenden Anwendungslogik zur Verfügung stehen. Sollen nachträglich Funktionen mit höherem Ressourcenbedarf integriert werden, erfordert dies eine Anpassung der Shell und somit eine vollständige Rekonfiguration des FPGA. Dies ist ebenfalls zur Laufzeit möglich, führt jedoch zur Unterbrechung und Rücksetzung der auf dem FPGA aktiven Anwendungslogik.

In Hinblick auf die Umsetzung von Echtzeitanwendungen mit geringer Reaktionslatenz ist die statische Partitionierung des FPGA vorteilhaft. Hierdurch ist die FPGA-Anwendungslogik dauerhaft aktiv und kann direkt auf eingehende Ereignisse reagieren, auch wenn die zugehörige Partitionssoftware auf der CPU gerade nicht ausgeführt wird. Ebenfalls vorteilhaft ist hierzu die generische Schnittstelle der PRR, deren DMA-Funktionalität den direkten Zugriff auf Speicher und Peripheriegeräte des SoC ohne Unterstützung der CPU ermöglicht.

Die geforderte Durchsetzung räumlicher Isolation setzt die vorgestellte Architektur durch das Zusammenspiel von Shell, Hypervisor und IOMMU um. Hierdurch ist sichergestellt, dass FPGA-basierte Anwendungslogik nur auf Speicherbereiche zugreifen kann, welche der zugehörigen Hypervisor-Partition zugewiesen sind. Durch die Nutzung von Modulen zur Fehlerisolation kann weiterhin vermieden werden, dass Anwendungslogik durch inkorrekte Implementierung des Speicherbusprotokolls zum Ausfall weiterer Komponenten führt.

Die implementierten Isolationsmechanismen verhindern somit die direkte Beeinflussung der FPGA-Anwendungslogik anderer Partitionen, z.B. durch Manipulation der von dieser bereitgestellten Register, sowie den direkten Zugriff auf Daten anderer Partitionen. Dennoch kann die gegenseitige Störung von FPGA-basierten Funktionen nicht vollständig ausgeschlossen werden. Zum einen können am Speicherbus durch intensive Nutzung von SoC-Komponenten Interferenzeffekte zwischen Anwendungslogik verschiedener Partitionen bzw. zwischen FPGA- und CPU-basierten Funktionen auftreten. Da der Speicherbus nach dem in Abschnitt 6.2 beschriebenen Ansatz jedoch nicht Teil der Echtzeitdomäne ist, wird dies als akzeptabel betrachtet. Sollte eine Vermeidung dieser Interferenz nötig sein, kann dies durch Einbringen von Komponenten zur Beschränkung der Speicherzugriffsrate erreicht werden (vgl. Abschnitt 3.3.2.2). Zum anderen sind durch die geteilte Nutzung des FPGA Seitenkanalangriffe und Denial-of-Service-Attacken denkbar. So wurde bereits gezeigt, dass FPGAs durch Angriffe auf die Spannungsversorgung mittels darauf ausgerichteter Logikschaltungen im FPGA zum Absturz gebracht werden können [169]. Die Erkennung solcher Schaltungen ist Gegenstand aktueller Forschung [198].

6.5.2 Adaptive Anbindung externer Komponenten

Das in Abschnitt 6.4 vorgestellte Konzept ermöglicht die direkte Anbindung wechselnder externer Komponenten an den FPGA, wobei geteilt genutzte Kommunikationsleitungen

und die Umsetzung verschiedener Kommunikationsprotokolle ermöglicht werden. Hierbei reduziert es den Umfang benötigter externer Hardwarekomponenten, indem Protokolllogik im FPGA implementiert wird.

Anhand einer prototypischen Implementierung des Konzepts konnte seine Anwendbarkeit für die Kommunikationsprotokolle CAN- und RS-485 gezeigt werden, welche die Basis für verschiedene industrielle Kommunikationsprotokolle darstellen. Beispiele hierfür umfassen die Protokolle CANopen, DeviceNet, Modbus-RTU und CC-Link.

Während das vorgestellte Konzept grundsätzlich auf viele drahtgebundene Kommunikationsprotokolle angewandt werden kann, ist die praktische Anwendbarkeit durch verschiedene Aspekte eingeschränkt, welche bei der Integration von Protokollen beachtet werden müssen. Eine solche Einschränkung betrifft die Bandbreite der Kommunikationssignale. So beschränkt die Multiplexer-Einheit die Auswahl geeigneter Protokolle auf solche, deren höchste Frequenzanteile unterhalb der Grenzfrequenz der Multiplexerkomponente liegen. In der prototypischen Implementierung lag diese bei ca. 28 MHz. Für Kommunikationsprotokolle, die höhere Signalfrequenzen nutzen, müssen somit separate Signalleitungen vorgesehen werden. Eine weitere Einschränkung stellen die Anforderungen der Protokolle an die genutzten Signalleitungen dar. Werden feste Signalleitungen zur Übertragung verschiedener Protokolle genutzt, müssen diese, bspw. hinsichtlich Leitungsimpedanz und Schirmung, mit allen darüber genutzten Protokollen kompatibel sein. Ist dies nicht der Fall, müssen separate Leitungen genutzt werden.

In einfacheren Anwendungsfällen, in welchen das Multiplexing von Kommunikationsprotokollen über feste Signalleitungen nicht erforderlich ist, entfallen diese Einschränkungen. In diesem Fall können separate, auch drahtlose, Verbindungen zu den anzubindenden Komponenten aufgebaut werden. Von der Rekonfigurierbarkeit im FPGA profitieren auch diese Szenarien: Solange zu jedem Zeitpunkt nur eine Teilmenge der physisch verfügbaren Schnittstellen parallel genutzt werden soll, können durch das bedarfsgerechte Laden der entsprechenden Protokolllogik gegenüber dauerhaft aktiver Logikimplementierungen FPGA-Ressourcen eingespart werden.

6.5.3 Gemeinsame Betrachtung

Die in diesem Kapitel vorgestellten Konzepte dienen der Umsetzung von Echtzeit-Funktionalität auf Hypervisor-partitionierten SoC unter Nutzung der im SoC enthaltenen FPGA-Komponente. Wie eingehend motiviert (vgl. Abschnitt 6.1) ist die Umsetzung von Echtzeit-Anwendungslogik im FPGA aufgrund des erreichbaren zeitlichen Determinismus und der Erreichbarkeit geringer Reaktionslatenzen durch Parallelisierung attraktiv. Aus diesem Grund sieht der in Abschnitt 6.2 beschriebene Ansatz die Implementierung solcher Funktionen im FPGA vor. Um Interferenz mit weiteren SoC-Komponenten, insbeson-

dere den CPU-Kernen, zu vermeiden, ist für Echtzeitaufgaben die direkte Anbindung externer Komponenten an den FPGA vorgesehen. Gleichzeitig kann der Zugriff der FPGA-basierten Anwendung auf weitere SoC-Komponenten notwendig sein. Dies ist z.B. der Fall, wenn die erfassten Daten zur zeitunkritischen Weiterverarbeitung an die CPU übergeben werden sollen oder auf Peripheriegeräte des SoC zugegriffen werden soll. Ist ein solcher Zugriff auf SoC-Komponenten nötig, sind in Hypervisor-partitionierten Systemen zusätzliche Maßnahmen erforderlich, um die durch den Hypervisor umgesetzte räumliche Partitionierung des Speichers und der Peripheriekomponenten nicht zu verletzen.

Das in Abschnitt 6.3 vorgestellte Konzept löst dieses Problem, indem es eine räumliche Partitionierung von FPGA-basierten Funktionen unter Nutzung der Speicherschutzeinheiten des SoC umsetzt. Gleichzeitig partitioniert es den FPGA selbst und ermöglicht somit den modularen Einsatz von Anwendungslogik auf geteilter FPGA-Hardware. Die FPGA-basierten Anwendungsfunktionen können wiederum Hypervisor-Partitionen zugewiesen werden, wodurch letztendlich Anwendungsmodule, bestehend aus CPU-basierter Software und FPGA-basierten Logikschaltungen, modular auf dem SoC eingesetzt werden können.

Während die Anbindung der externen Komponenten an den FPGA mittels dedizierter ICs (integrierte Schaltkreise, engl. integrated circuit) zur Umsetzung der Protokolllogik und Pegelwandlung erfolgen kann, ergibt sich durch die Nutzung des FPGA Optimierungspotenzial. Gleichzeitig kommt die etablierte Anbindung von Sensorik und Aktoren mittels individueller Signalleitungen in hochflexiblen Anwendungsfällen (vgl. Abschnitt 6.1.1) an ihre Grenzen. Um dieses Potenzial zu nutzen und den Anforderungen hochflexibler eingebetteter Systeme Rechnung zu tragen, ermöglicht das in Abschnitt 6.4 beschriebene Konzept die effiziente Anbindung verschiedener externer Komponenten an den FPGA.

In Kombination ergeben die vorgestellten Konzepte eine SoC-basierte HW/SW-Architektur, welche in Hypervisor-partitionierten SoC die Echtzeitkommunikation mit externen Komponenten sowie die Echtzeitverarbeitung der erhaltenen Daten durch Umsetzung im FPGA ermöglicht. Im Folgenden wird die hierzu nötige Integration der Konzepte weiter erläutert.

6.5.3.1 Integration der vorgestellten Konzepte

Die Integration der in diesem Kapitel vorgestellten Konzepte ist in Abbildung 6.8 visualisiert. Die hardwareseitige Grundlage der Architektur bildet der in Abbildung 6.6 dargestellte Aufbau eines SoC mit externen Schaltungen für die Anbindung verschiedener Komponenten über (ggf. wechselnde) Kommunikationsprotokolle. Softwareseitige Grundlage ist die in Abschnitt 6.3 vorgestellte Architektur, welche die gemeinsame Partitionierung von Prozessor, Speicher, Peripheriegeräten und FPGA umsetzt. Sie ermöglicht die effiziente Integration verschiedener Datenverarbeitungsfunktionen auf einer Rechen-

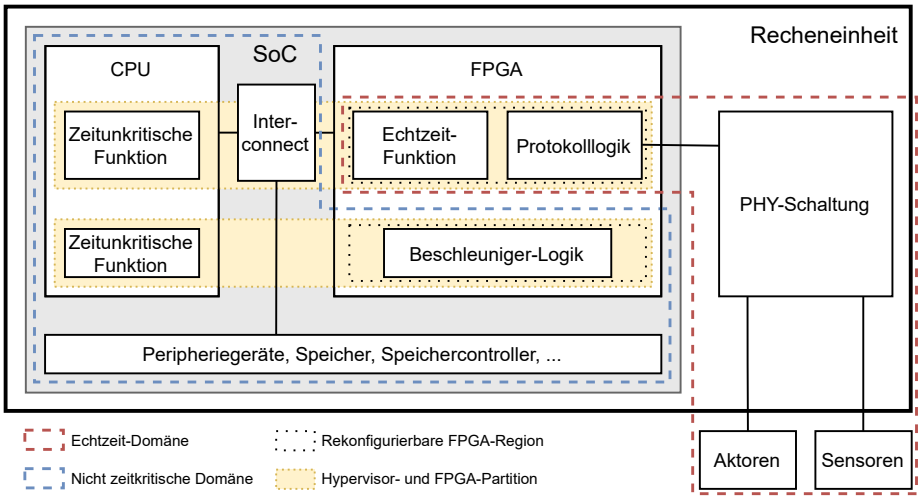


Abbildung 6.8: Architektur einer Recheneinheit zur FPGA-basierten Umsetzung von Echtzeitfunktionen im Hypervisor-partitionierten SoC.

plattform und unterstützt, wo nötig, die Integration FPGA-basierter Anwendungslogik, wie bspw. Hardwarebeschleuniger.

Zur Umsetzung von Echtzeitfunktionen im FPGA wird die zugehörige PRR so konfiguriert, dass der direkte Zugriff auf die I/O-Schnittstelle zu den externen PHY-Komponenten möglich ist. Weiterhin wird, wie in Abschnitt 6.4.2 beschrieben, auch die Protokolllogik in der PRR platziert. Wenn das in Abschnitt 6.4.2 beschriebene Multiplexing von Kommunikationsleitungen erforderlich ist (nicht in Abbildung 6.8 dargestellt), bietet sich die Umsetzung der Multiplexeransteuerung durch eine privilegierte Steuerungssoftware auf der CPU unter Nutzung der GPIO-Komponente des SoC an.

6.5.3.2 Implementierungsalternativen der Echtzeitfunktion

Die Implementierung der Echtzeitfunktion in Form digitaler Logik im FPGA ermöglicht in vielen Fällen eine hocheffiziente Umsetzung der Datenverarbeitungsalgorithmen, ist jedoch – aufgrund der fehlenden Abstraktionen durch darunterliegende Prozessor-, Treiber- und Betriebssystemkomponenten – häufig deutlich komplexer und zeitintensiver als ihre Umsetzung in Software. Zwei Ansätze, um diese Komplexität zu reduzieren, sind die Nutzung von Soft-Prozessoren und High-Level-Synthese. Soft-Prozessoren sind Prozessoren, welche innerhalb eines FPGA realisiert sind. Durch die Instanziierung eines solchen Prozessors samt benötigtem Speicher in der PRR kann die Echtzeitfunktion als Softwareanwendung entwickelt und in der PRR ausgeführt werden. Durch Anbindung

an die Protokolllogik in der gleichen PRR ist die Nutzung der Kommunikationsschnittstellen durch die Echtzeitanwendung möglich. Die Umsetzung der Echtzeitanwendung und ggf. von Teilen der Protokolllogik auf Soft-Prozessoren kann zu einer Reduktion des Implementierungsaufwands und zu höherer Flexibilität des Echtzeitsystems führen. Eine Alternative bietet die Anwendung von High-Level-Synthese. Diese vereinfacht die Entwicklung FPGA-basierter Anwendungslogik durch die automatisierte Transformation von abstrakten Verhaltensbeschreibungen – meist Programmiersprachen wie C und C++ – in Hardwarebeschreibungen.

6.5.3.3 Erreichbares Echtzeitverhalten

Die in diesem Kapitel vorgestellten Konzepte haben das Ziel, die Implementierung von Funktionen zu ermöglichen, an welche harte Echtzeitanforderungen gestellt sind. Im Folgenden werden die Erreichbarkeit dieser Echtzeiteigenschaften und die hierfür erforderlichen Bedingungen diskutiert.

Die vorgestellte Architektur sieht die Umsetzung der Echtzeitfunktion in Form digitaler Schaltungen im FPGA vor. Dies dient der Vermeidung von Interferenzeffekten, welche durch die geteilte Nutzung von SoC-Komponenten, insbesondere der Speicherinfrastruktur und von Peripheriekomponenten, entstehen können. Das gleiche Ziel wird durch die direkte Anbindung der externen Komponenten, welche an der Erfüllung der Echtzeitfunktion auf Systemebene beteiligt sind, an den FPGA verfolgt.

Die Architektur basiert auf der Annahme, dass durch Umsetzung der Echtzeitfunktion auf dem FPGA Störungen vermieden werden können, welche das Zeitverhalten der Echtzeitfunktion beeinflussen und so das Einhalten von Echtzeitanforderungen gefährden. Davon kann unter folgenden Voraussetzungen ausgegangen werden:

Bedingung 1: Die im FPGA implementierte Anwendungslogik muss echtzeitfähig sein.

Bedingung 2: Die im FPGA implementierte Protokolllogik muss echtzeitfähig sein.

Bedingung 3: An das SoC angebundene und zur Erfüllung der Echtzeitfunktion genutzte externe Komponenten müssen echtzeitfähig sein.

Bedingung 4: Der korrekte Betrieb der im FPGA implementierten Anwendungs- und Protokolllogik sowie deren Kontrolle über die FPGA-I/O-Schnittstelle muss sichergestellt sein.

Der Begriff der *Echtzeitfähigkeit* bezeichnet hierbei die Eigenschaft eines Subsystems, mit einer ausreichenden Wahrscheinlichkeit innerhalb einer ausreichenden Zeit auf eine Eingabe mit einer Ausgabe zu reagieren, sodass durch Verkettung der Subsysteme die Echtzeitanforderungen an das Gesamtsystem hinsichtlich dessen Reaktionslatenz und der Wahrscheinlichkeit der Einhaltung dieser Reaktionslatenz erfüllt werden können.

Die in Bedingung 1 geforderte Echtzeitfähigkeit der Echtzeit-Anwendungslogik ist vom Anwendungsentwickler zu erfüllen. Hierzu ist insbesondere sicherzustellen, dass die Reaktion der Echtzeit-Anwendung nicht durch das Warten auf nicht echtzeitfähige Systemkomponenten über die Deadline hinaus verzögert wird. Wie in Abschnitt 6.2 beschrieben, wird insbesondere für SoC-Komponenten außerhalb des FPGA kein Echtzeitverhalten angenommen. Zugriffe auf diese Komponenten müssen daher von der Echtzeitlogik entkoppelt sein. Gleiches gilt für die im FPGA implementierte Protokolllogik (Bedingung 2), welche ebenfalls vom Anwendungsentwickler bereitgestellt wird.

Die in Bedingung 3 geforderte Echtzeitfähigkeit externer Komponenten betrifft die zur Umsetzung der Kommunikation genutzten SoC-externen Schaltungen. Dies umfasst insbesondere die Transceiver und ggf. Multiplexer, welche innerhalb der Systemgrenze der Recheneinheit liegen. Betrachtet man die Echtzeitfähigkeit auf Gesamtsystemebene (vgl. rote Markierung in Abbildungen 6.2 und 6.8), muss ebenfalls die Echtzeitfähigkeit der über die Kommunikationsleitungen angebundenen Komponenten (Sensoren und Aktoren) gefordert werden. Von der Echtzeitfähigkeit der Transceiver und Multiplexer kann in den meisten Fällen ausgegangen werden. Maximale Signallaufzeiten und Umschaltzeiten sind in den Datenblättern der genutzten Komponenten angegeben.

Der korrekte Betrieb der Anwendungs- und Protokolllogik sowie deren Zugriff auf die FPGA-I/O-Schnittstelle (Bedingung 4) ist erforderlich, da eine Unterbrechung oder Beeinträchtigung dieser Komponenten zu einem Ausfall oder einer verspäteten Reaktion der Echtzeitanwendung führen kann. Aus diesem Grund muss sichergestellt werden, dass

- die im FPGA implementierte Logik mit der vom Anwendungsentwickler spezifizierten Taktfrequenz ausgeführt wird,
- es nicht zu Unterbrechungen im Ablauf der implementierten Logikschaltung kommt und
- der Kommunikationspfad zwischen Protokolllogik und den PHY-Schaltungen nutzbar ist.

Werden Teile der Logikschaltung mit verringerter oder überhöhter Taktfrequenz betrieben, kann dies zu verspäteter Ausgabe oder fehlerhaftem Verhalten führen. Um die rechtzeitige und korrekte Reaktion des Echtzeitsystems sicherzustellen, müssen unerwartete Frequenzänderungen der die Logik treibenden Taktsignale ausgeschlossen werden. Solche Änderungen können bspw. im Rahmen von Schutzmechanismen zur Verhinderung thermischer Überlastung des SoC implementiert sein oder durch Einflussnahme auf die Konfiguration von Taktquellen im System entstehen.

Werden Taktquellen vollständig deaktiviert bzw. von der implementierten Schaltung getrennt, ist der Ablauf taktgesteuerter Logik unterbrochen. Auch die Beeinflussung der Spannungsversorgung kann zu Unterbrechungen führen. Dies kann bspw. durch die

Aktivierung von Energiesparmodi *(Power Gating)*, durch eine Fehlkonfiguration der Spannungsversorgungsinfrastruktur des SoC oder durch das Verhalten der im FPGA ausgeführten Logik erfolgen (vgl. die in Abschnitt 6.5.1 diskutierten Angriffe auf die Spannungsversorgung). Zuletzt führt auch die Rekonfiguration des FPGA zu einer Unterbrechung der Logikausführung in den von der Rekonfiguration betroffenen FPGA-Regionen und darf daher nur durchgeführt werden, wenn es im Hinblick auf das Echtzeitsystem zulässig ist.

Zuletzt muss sichergestellt werden, dass der Kommunikationspfad zwischen der im FPGA implementierten Protokolllogik und den Transceivern nutzbar ist. Dieser umfasst insbesondere die vom FPGA genutzten I/O-Schnittstellen an der Grenze des SoC. Auch deren Funktion kann durch Power Gating beeinträchtigt werden. Weiterhin umfasst der Kommunikationspfad die Isolationskomponenten, welche die zur Implementierung genutzten PRR während der Rekonfiguration abschotten. Hier ist sicherzustellen, dass eine solche Abschottung nur während einer beabsichtigten Rekonfiguration stattfindet.

Läuft der FPGA somit unabhängig vom Verhalten potenziell störender SoC-Komponenten und sind die Komponenten entlang der Echtzeit-Datenverarbeitungskette echtzeitfähig implementiert und wie vom Entwickler vorgesehen betriebsbereit, kann von einer Echtzeitfähigkeit des Gesamtsystems ausgegangen werden.

6.5.4 Anwendbarkeit im Kontext hochflexibler Produktionssysteme

Im Folgenden wird die Anwendbarkeit der in Abschnitt 6.5.3 vorgestellten Architektur im Kontext hochflexibler Produktionssysteme diskutiert. Dies erfolgt am Beispiel einer Recheneinheit zur Umsetzung von Datenverarbeitungsaufgaben im in Abschnitt 6.1.1 beschriebenen *Wertstromkinematik*-Konzept. Die Anwendung der einzelnen, in Abschnitt 6.3 und Abschnitt 6.4 beschriebenen Konzepte erfolgte im Kontext dieses Produktionssystemkonzepts in den eigenen Veröffentlichungen [SKM+23] bzw. [SKM+22].

Wie in Abschnitt 6.1.1 beschrieben, basiert das Wertstromkinematik-Produktionssystem auf der automatischen Rekonfiguration universell nutzbarer Kinematiken, um verschiedene Produktionsprozesse umsetzen zu können. Die Rekonfiguration erfolgt in Form eines Wechsels des Endeffektors, welcher benötigte Werkzeuge und Sensoren umfasst, und ggf. der Kopplung bzw. Entkopplung von Kinematiken.

Die mit verschiedenen Werkzeugen und somit Fertigungsprozessen, Sensoren und Aktoren zusammenhängenden Datenverarbeitungsaufgaben müssen durch Recheneinheiten im Produktionssystem erbracht werden. Im Folgenden wird diskutiert, in wie weit sich die in Abschnitt 6.5.3 vorgestellte Architektur zur Erfüllung der in Abschnitt 6.1.1 beschriebenen Anforderungen an Recheneinheiten im WSK-System eignen.

Zunächst wird der Aspekt der effizienten Umsetzung der Datenverarbeitungsaufgaben im Gesamtsystem betrachtet. Effiziente Datenverarbeitung im Gesamtsystem wird erreicht, wenn alle Verarbeitungsaufgaben in der geforderten Zeit mit einer möglichst geringen Anzahl an Recheneinheiten erfüllt werden können, da somit Kosten für die Anschaffung und den Betrieb der Recheneinheiten minimiert werden. Die Grundlage hierfür wird in der vorgestellten Architektur durch die Partitionierung der Recheneinheit gelegt, welche vom Hypervisor umgesetzt wird. Sie ermöglicht die parallele Ausführung verschiedener Anwendungen auf einer Recheneinheit und stellt die unabhängige Verwaltbarkeit der Anwendungen sicher. Durch die unabhängige Verwaltbarkeit können Datenverarbeitungsanwendungen bei der Rekonfiguration einer Kinematik unabhängig von parallel ausgeführten Anwendungen ausgetauscht werden. Dies ist auch möglich, wenn die Anwendung FPGA-basierte Anwendungslogik umfasst.

Wie in Abschnitt 6.1.1 beschrieben, unterscheiden sich Datenverarbeitungsaufgaben in Produktionssystemen stark hinsichtlich ihrer Rechenintensität. Sollen rechenintensive Aufgaben umgesetzt werden, ermöglicht die FPGA-Integration die Nutzung anwendungsspezifischer Hardwarebeschleuniger. Für gut parallelisierbare Algorithmen wie die Inferenz in Machine-Learning-Anwendungen oder Bildverarbeitungsaufgaben kann dies die Verarbeitungsgeschwindigkeit gegenüber einer rein CPU-basierten Umsetzung deutlich erhöhen. Weiterhin entlastet die Auslagerung von Datenverarbeitung in den FPGA die CPU, wodurch diese für weitere Anwendungen zur Verfügung steht. Für wenig rechenintensive und kontrollflusslastige Anwendungen bietet sich die Ausführung direkt auf der CPU an, da CPU-Ressourcen in feinerer Granularität zwischen Partitionen geteilt werden können.

Im Hinblick auf die Echtzeitanforderungen der Datenverarbeitungsaufgaben können unkritische Aufgaben grundsätzlich an beliebiger Stelle im Datenverarbeitungsnetzwerk des Produktionssystems umgesetzt werden. Stehen mehrere Recheneinheiten zur Verfügung, kann die Platzierung dieser Aufgaben bspw. nach Kriterien der Verfügbarkeit geeigneter Rechenressourcen sowie der Auslastung des Netzwerks zwischen Datenquelle und Recheneinheit bestimmt werden. Im Gegensatz dazu erfordern Datenverarbeitungsaufgaben mit harten Echtzeitanforderungen die Berücksichtigung möglicher Verzögerungen durch die geteilte Nutzung von Netzwerk- und Rechenressourcen. Sie benötigen einen echtzeitfähigen Übertragungspfad von der Datenquelle über eine Recheneinheit zur Datensenke. Die vorgestellte Architektur unterstützt dies durch die Möglichkeit der direkten Anbindung der entsprechenden Komponenten an den FPGA der Recheneinheit und der Umsetzung der Datenverarbeitungsaufgabe in diesem. Unter Berücksichtigung der in Abschnitt 6.5.3.3 beschriebenen Randbedingungen können so harte Echtzeitanforderungen erfüllt und geringe Reaktionslatenzen erreicht werden.

Im Kontext von WSK-Produktionssystemen sind Datenquelle- und -senke häufig Sensoren bzw. Aktoren des Endeffektors. Da dieser automatisiert ausgetauscht wird, sieht das

WSK-Konzept eine feste elektrische Schnittstelle zwischen Endeffektor und Kinematik vor, welche in Form von Schnellwechselsystemen umgesetzt wird [MBS+23]. Wie in Abschnitt 6.4.1 motiviert, kann hierbei der automatisierte Wechsel der auf den Signalleitungen genutzten Kommunikationsprotokolle vorteilhaft sein. Dies wird durch die vorgestellte Architektur, wie in Abschnitt 6.4.2 beschrieben, ermöglicht. Es erfordert jedoch die direkte Anbindung der zu nutzenden Signalleitungen an die Multiplexer-Einheit der Recheneinheit und somit räumliche Nähe zur Maschine. Eine Recheneinheit kann hierbei auch zur Anbindung von Sensoren bzw. Aktoren von Endeffektoren mehrerer Kinematiken genutzt werden.

Wie in Abschnitt 6.4 beschrieben, konnte die Anwendbarkeit des Konzepts u.a. für die Protokolle CAN und RS-485 im Laboraufbau gezeigt werden. Diese bilden, wie in Abschnitt 6.5.2 beschrieben, die Grundlage einiger etablierter industrieller Kommunikationsprotokolle. Da das reale Produktionsumfeld ein deutlich anspruchsvolleres Anwendungsumfeld darstellt, insbesondere hinsichtlich der genutzten Leitungslängen sowie elektromagnetischer Störeinflüsse, ist der Funktionsnachweis in der Einsatzumgebung noch zu erbringen. Die Bestimmung der erreichbaren Einsparungen durch Nutzung von Protokollmultiplexing ist daher im Rahmen einer Praktikabilitätsuntersuchung mit verschiedenen weiteren Protokollen Gegenstand zukünftiger Forschung.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die vorliegende Arbeit widmet sich der Interferenzbeherrschung auf Hypervisor-verwalteten MPSoC-Plattformen, um die effiziente Umsetzung von Mixed-Criticality-Systemen mit gemischten Echtzeitanforderungen auf diesen Plattformen zu ermöglichen. Die vorgestellten Konzepte und Architekturen zielen hierzu auf die Begrenzung der Interferenzauswirkungen bzw. die Vermeidung von Interferenzkanälen auf diesen Plattformen ab, während gleichzeitig die Optimierung der Ausnutzung paralleler Rechenkapazität angestrebt wird.

Die Konzepte zur Interferenzbeherrschung setzen softwareseitig auf Ebene des Hypervisors an. Dies erlaubt die Anwendung in partitionierten Systemen, in welchen sowohl Bare-Metal-Anwendungen als auch komplexere Anwendungsstrukturen auf Basis von Betriebssystemen integriert sind. Somit wird dem Bedarf nach der Integration verschiedener Anwendungen auf leistungsfähige Plattformen Rechnung getragen.

Um die Interferenzproblematik in MCP zu demonstrieren, wurden in Kapitel 4 die Auswirkungen von Interferenzeffekten auf die Reaktionszeit von Software in eingebetteten Systemen untersucht. Dies erfolgte am Beispiel eine Steuereinheit zur Nachrüstung von Industrie-4.0-Funktionen in Produktionsanlagen. Im Rahmen der Untersuchung wurden Ende-zu-Ende-Messungen der Reaktionszeit einer minimalen Evaluationsanwendung durchgeführt, welche in unterschiedlichen Basissoftwarearchitekturen implementiert und in verschiedenen Lastszenarien betrieben wurde. Hierbei wurde festgestellt, dass beim Einsatz des statisch partitionierenden Jailhouse-Hypervisors ohne weitere Maßnahmen zur Interferenzbeherrschung in den untersuchten Szenarien keine zeitliche Isolation zwischen den Hypervisorpartitionen besteht. Es zeigte sich eine starke Abhängigkeit der maximal gemessenen Reaktionslatenzen der Evaluationsanwendung vom Verhalten in einer separaten Partition parallel ausgeführter Software. Die maximale gemessene Reaktionslatenz stieg bei Belastung der parallel aktiven Partition um 151 % bis 255 % gegenüber dem unbelasteten Fall an, die mittlere Reaktionslatenz um 0.7 %. Durch den Vergleich mit unpartitionierten Architekturen konnte festgestellt werden, dass die untersuchte Partitionierung der Hardwareplattform durch den Hypervisor zwar zur Umsetzung von Anwendungen mit weichen Echtzeitanforderungen vorteilhaft sein kann, für die effizi-

ente Umsetzung von Anwendungen mit harten Echtzeitanforderungen jedoch weitere Maßnahmen zur Interferenzbeherrschung nötig sind.

In den Kapiteln 5 und 6 wurden hierfür geeignete Konzepte vorgestellt. Sie sehen die Implementierung der Echtzeit-Funktionen in unterschiedlichen Komponenten des MPSoC vor und sind somit für unterschiedliche SoC-Hardwarearchitekturen anwendbar.

Um Interferenzeffekte zwischen Anwendungen verschiedener Hypervisor-Partitionen auf MCP-Komponenten im MPSoC zu beherrschen und gleichzeitig die Dauer paralleler Softwareausführung zu optimieren, wurde in Kapitel 5 ein reaktiver Ansatz zum Scheduling von Systemen gemischter Echtzeitkritikalität auf Hypervisor-partitionierten MCP vorgestellt. Durch die kontinuierliche Überwachung des Ausführungsfortschritts der kritischen Anwendung durch eine Monitoring-Komponente werden frühzeitig Zustände erkannt, in welchen die Deadline von Echtzeit-Tasks aufgrund von Interferenzeffekten nicht eingehalten werden kann. Durch den rechtzeitigen Eingriff in den vom Hypervisor umgesetzten Schedule wird das Eintreten dieser kritischen Zustände verhindert. Gleichzeitig wird die Dauer paralleler Ausführung von Echtzeit-Tasks und unkritischen Anwendungen auf dem MCP optimiert. Das Monitoring erfolgt hierbei auf Basis von Ausführungs-Trace-Daten, welche Informationen über die ausgeführte Instruktion und somit den Programmfortschritt liefern. Der ereignisbasierte Charakter von Trace-Daten erlaubt überdies die effiziente Umsetzung des Monitors.

Das Konzept wurde unter Nutzung eines eingebetteten Hypervisors implementiert und hinsichtlich seiner Wirksamkeit evaluiert. Weiterhin wurde der Einfluss verschiedener Systemparameter auf die erreichbare Systemeffizienz untersucht und mit einem konservativen Scheduling-Ansatz verglichen. Um kontrollierbare Interferenzszenarien untersuchen zu können, wurden hierzu parametrisierbare, synthetische Evaluationsanwendungen genutzt. Die Evaluation zeigt, dass der vorgestellte Ansatz die Einhaltung der Deadline der Echtzeit-Anwendungen in allen untersuchten Systemkonfigurationen und Interferenzszenarien sicherstellen konnte und dabei eine deutliche Erhöhung des Zeitanteils paralleler Taskausführung gegenüber dem Referenzansatz ermöglichte (vgl. Abschnitt 5.4.3.2). Weiterhin konnte festgestellt werden, dass der hierdurch erreichbare Datendurchsatz stark von der Intensität der zwischen den Tasks auftretenden Interferenzeffekte abhängt. Während bei geringer Interferenz ein höherer Datendurchsatz erreicht wurde, lag der erreichte Durchsatz bei sehr starker Interferenz unter dem des Referenzansatzes. In diesem Aspekt konnte der in der Literatur bereits identifizierte Effekt bestätigt werden, dass die erreichbare Systemperformance bei paralleler Ausführung von Tasks bei sehr starker Interferenz geringer ausfallen kann als bei sequenzieller Ausführung (vgl. Abschnitt 5.4.4.1).

Als Alternative zur MCP-basierten Implementierung der Echtzeitfunktion wurde in Kapitel 6 eine Architektur vorgestellt, welche eine Partitionierung von FPGA-Ressourcen heterogener MPSoC durchführt, diese in den Partitionskontext eines Hypervisors ein-

bindet und für die Umsetzung von Echtzeit-Funktionen nutzbar macht. Die Architektur ermöglicht sowohl die Integration anwendungsspezifischer, FPGA-basierter Hardwarebeschleuniger als auch die FPGA-basierte Umsetzung von Echtzeit-Funktionen unter Umgehung von Interferenzkanälen mit anderen Komponenten des SoC. Durch die Nutzung dynamischer partieller Rekonfiguration ermöglicht sie den modularen Austausch von Anwendungssoftware in Hypervisorpartitionen samt zugehöriger Anwendungslogik im FPGA. Zur Umsetzung von Echtzeitfunktionen besteht die Möglichkeit, externe Komponenten, wie Sensoren und Aktoren, direkt an die FPGA-basierte Anwendungslogik anzubinden. Ein Mechanismus zur flexiblen Rekonfiguration der hierzu genutzten Kommunikationsprotokolle ermöglicht die Anwendung der Architektur in hochflexiblen Einsatzumgebungen. Dies wird am Beispiel zukünftiger Produktionssystemkonzepte dargestellt.

Die in dieser Arbeit vorgestellten Konzepte und Architekturen umgehen bzw. begrenzen die Auswirkungen von Interferenz zwischen parallel ausgeführten Anwendungen in Hypervisor-verwalteten MPSoC. Durch reaktives Scheduling auf MCP-Komponenten und die Option, Echtzeitfunktionen in programmierbare Logik auszulagern, bieten sie die Möglichkeit, die parallele Nutzung von Recheneinheiten in Hypervisor-partitionierten MPSoC zu verbessern und gleichzeitig die Einhaltung von Echtzeitanforderungen und der Partitionierung sicherzustellen. Die Arbeit bildet somit eine Grundlage zur effizienten Integration von Anwendungen mit gemischten Echtzeitanforderungen in Hypervisor-partitionierten heterogenen MPSoC und leistet somit einen Beitrag zur effizienteren Umsetzung von Mixed-Criticality-Anwendungen in heterogenen eingebetteten Systemen.

7.2 Ausblick

Der Fokus der vorliegenden Arbeit lag auf der Entwicklung neuer Konzepte und Architekturen zur Interferenzbeherrschung in Hypervisor-verwalteten MPSoC, um die effiziente Integration von Anwendungen mit unterschiedlichen Echtzeitanforderungen zu ermöglichen. Aus diesem Grund erfolgte die Umsetzung und Evaluation der Konzepte unter Laborbedingungen.

So erfolgte die Evaluation des in Kapitel 5 vorgestellten Ansatzes mittels synthetischer Anwendungen zur kontrollierten Erzeugung von Interferenzeffekten. Während dies zur Untersuchung der Wirksamkeit des Ansatzes unter Extrembedingungen nötig ist und die Untersuchung grundlegender Zusammenhänge ermöglicht, sind die erhaltenen Werte nur bedingt auf reale Anwendungen übertragbar. Die Anwendung des vorgestellten Konzepts auf industrielle Anwendungen birgt daher vielversprechendes Erkenntnispotenzial. Auch hinsichtlich der vom Monitor und Execution Controller benötigten Ressourcen sind weitere Optimierungen möglich. Durch die relativ geringe Rechenlast des Controllers zur Verarbeitung von Trace-Daten ist eine Umsetzung als Task innerhalb eines Echt-

zeitbetriebssystems vorteilhaft, da dies die parallele Ausführung weiterer Systemtasks ermöglicht. Hierbei ist jedoch sicherzustellen, dass die somit auftretende Erhöhung der Reaktionslatenz des Execution Controllers nach oben beschränkt ist und bei der Parametrisierung des Systems berücksichtigt wird. Sofern das MPSoC einen FPGA umfasst, ist die Umsetzung des Execution Controllers auch in diesem möglich. Wie in Abschnitt 5.5.1.4 beschrieben, kann hierdurch die vollständige Entkopplung des Tracedatenpfades vom Anwendungsdatenpfad erreicht werden.

Auch im Hinblick auf das in Kapitel 6 vorgestellte Konzept zum Multiplexing verschiedener Kommunikationsprotokolle beschränkt sich die Untersuchung bisher auf Laboraufbauten sowie auf eine begrenzte Menge an Protokollen und Protokollparametern. Um das Potenzial des Konzepts im angestrebten Einsatzumfeld zu bestimmen, ist eine systematische Untersuchung weiterer industriell genutzter Protokolle zu empfehlen und die Anwendbarkeit in Umgebungen mit erhöhten Störeinflüssen zu prüfen. Das Aufbauen einer Bibliothek der hierbei als kompatibel identifizierten Kommunikationsprotokolle kann die praktische Umsetzung des Protokollmultiplexing-Konzepts vereinfachen.

Im Rahmen der Anwendung der in dieser Arbeit vorgestellten Konzepte müssen verschiedene Entwurfsentscheidungen getroffen und Konfigurationen erstellt werden. Die (teilweise) Automatisierung dieser Schritte kann den hierdurch bedingten Entwicklungsaufwand deutlich reduzieren und die Attraktivität der Anwendung der Konzepte erhöhen. Im Hinblick auf den in Kapitel 5 vorgestellten Trace-Monitoring-Ansatz stellt die Auswahl der zu überwachenden CBI eine für die Performanz des Konzepts zentrale Entwurfsentscheidung dar. Eine automatisierte Analyse von Echtzeit-Tasks hinsichtlich CBI mit großem Einfluss auf die verbleibende Ausführungszeit kann den manuellen Integrationsaufwand deutlich reduzieren. In ähnlicher Weise würde das in Kapitel 6 vorgestellte Konzept von einer Möglichkeit zur automatischen Ableitung der PRR-Parameter anhand einer Liste zu unterstützender FPGA-Funktionen profitieren. Die automatische Ableitung der HDL-Projekte auf Basis dieser Informationen wurde in [Kre21] bereits grundlegend umgesetzt.

Im Hinblick auf den in Kapitel 6 beschriebenen Anwendungsfall des *Wertstromkinematik*-Produktionssystems können darüber hinausgehende Tools die zu unterstützenden Allokationen von Datenverarbeitungsfunktionen auf im Netzwerk vorhandene Recheneinheiten bestimmen und somit den benötigten Input zur Ableitung der Shell-Designs der einzelnen Recheneinheiten liefern.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADAS	Advanced Driver Assistance Systems
AMP	Asymmetric Multiprocessing
API	Programmierschnittstelle (engl. Application Programming Interface)
APU	Anwendungsprozessoreinheit (engl. Application Processing Unit)
ASIC	Anwendungsspezifischer integrierter Schaltkreis (Application-specific Integrated Circuit)
ASIL	Automotive Safety Integrity Level
AXI	Advaned eXtensible Interface
BET	Best-Effort-Task (nicht echtzeitkritischer Task)
BRAM	Block RAM
CAN	Controller Area Network
CAT	Cache Allocation Technology
CBI	Bedingte Verzweigungsinstruktion (engl. Conditional Branch Instruction)
CFG	Kontrollflussgraph (engl. Control Flow Graph)
COTS	Commercial-off-the-shelf
CPPS	Cyber-physisches Produktionssystem
CPS	Cyber-physisches System
CPU	Hauptprozessor (engl. Central Processing Unit)
DAL	Design Assurance Level
DDR	Double Data Rate
DLL	Data Link Layer
DMA	Direct Memory Access
DPR	Dynamische partielle Rekonfiguration
DRAM	Dynamic RAM
DSP	Digitaler Signalprozessor
DTO	Device Tree Overlay

EC	Execution Controller
EE-Architektur	Elektrische/Elektronische Architektur
EL	Evaluationslogik
ET	Echtzeit-Task
ETF	Embedded Trace FIFO
ETM	Embedded Trace Macrocell
FIFO	First in, first out
FPGA	Field-Programmable Gate Array
GPIO	General-Purpose Input/Output
GPOS	General-Purpose Operating System
GPU	Grafikprozessor (engl. graphics processing unit)
HDL	Hardwarebeschreibungssprache (Hardware Description Language)
I/O	Eingabe/Ausgabe (engl. Input/Output)
IC	Interconnect
IMA	Integrated Modular Avionics
IOMMU	Input/Output Memory Management Unit
IPC	Inter-Partitions-Kommunikation (engl. Inter-partition Communication)
IPI	Inter-Partitions-Interrupt
IPI	Inter-Prozessor-Interrupt
ISA	Befehlssatzarchitektur (engl. Instruction Set Architecture)
isWCET	Interference-sensitive WCET
LB	Logikblock
LLC	Last-Level Cache
LRU	Least Recently Used
LTE	Long-Term Evolution
LUT	Lookup Table
MAC	Media Access Control
MAF	Major (Time) Frame
MCP	Multicore-Prozessor
MCS	System gemischter Kritikalität (engl. Mixed-Criticality System)
MES	Manufacturing Execution System
MIO	Multiplexed I/O

MMU	Memory Management Unit
MPAM	Memory System Resource Partitioning and Monitoring
MPSoC	Mehrprozessor-Einchipsystem (engl. Multi-Processor System-on-Chip)
NUMA	Non-Uniform Memory Access
OS	Betriebssystem (engl. Operating System)
PCI	Peripheral Component Interconnect
PHY	Physical Layer
PL	Programmable Logic
PREM	Predictable Execution Model
PRR	Partiell rekonfigurierbare Region
PS	Processing System
RAM	Random Access Memory
RDT	Resource-Direktor-Technik
RPU	Echtzeitprozessoreinheit (engl. Real-time Processing Unit)
RTOS	Echtzeitbetriebssystem (engl. Real-Time Operating System)
SIL	Safety Integrity Level
SMMU	System Memory Management Unit
SoC	Einchipsystem (engl. System-on-Chip)
SR-IOV	Single-Root-I/O-Virtualization
SRAM	Static RAM
TCB	Trusted Computing Base
TCM	Tightly-Coupled Memory
TTC	Triple Timer Counter
UMA	Uniform Memory Access
UMTS	Universal Mobile Telecommunications System
vCPU	Virtueller CPU-Kern
VM	Virtual Machine
VMM	Virtual Machine Monitor
WCET	Worst-Case Execution Time
WSK	Wertstromkinematik

Literatur- und Quellennachweise

[1] *Zukunftsbild „Industrie 4.0“.* Technischer Bericht, Bundesministerium für Bildung und Forschung, Bonn, 2015.

[2] S. Solaimani, W. Keijzer-Broers und H. Bouwman: *What we do – and don't – know about the Smart Home: An analysis of the Smart Home literature.* Indoor and Built Environment, 24(3):370–383, 2015.

[3] C. Harrison, B. Eckman, R. Hamilton, P. Hartswick, J. Kalagnanam, J. Paraszczak und P. Williams: *Foundations for Smarter Cities.* IBM Journal of Research and Development, 54(4), 2010.

[4] E. Geisberger und M. Broy (Herausgeber): *agendaCPS.* Springer Berlin Heidelberg, 2012.

[5] E. A. Lee und S. A. Seshia: *Introduction to Embedded Systems - A Cyber-Physical Systems Approach.* MIT Press, 2. Auflage, 2017.

[6] L. Monostori, B. Kádár, T. Bauernhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn und K. Ueda: *Cyber-physical systems in manufacturing.* CIRP Annals, 65(2):621–641, 2016.

[7] C.-Y. Chan: *Fundamentals of Crash Sensing in Automotive Air Bag Systems.* SAE International, Januar 2000.

[8] B. Kasper: *Industrie 4.0: Technologieentwicklung und sicherheitstechnische Bewertung von Anwendungsszenarien.* Technischer Bericht, Bundesanstalt für Arbeitsschutz und Arbeitsmedizin, 2019.

[9] H. Diess: *Levers to unleash value*, Januar 2020. Volkswagen Group Presentation.

[10] C. Märtin: *Post-Dennard Scaling and the final Years of Moore's Law.* Technischer Bericht, Hochschule Augsburg University of Applied Sciences, 2014.

[11] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous und A. LeBlanc: *Design of ion-implanted MOSFET's with very small physical dimensions.* IEEE Journal of Solid-State Circuits, 9(5):256–268, 1974.

[12] O. Burkacky, J. Deichmann und J. P. Stein: *Automotive software and electronics 2030: mapping the sector's future landscape.* McKinsey & Company, 2019.

[13] D. Siepmann: *Industrie 4.0 – Fünf zentrale Paradigmen.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[14] R. I. Davis und L. Cucu-Grosjean: *A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems.* LITES: Leibniz Transactions on Embedded Systems, Mai 2019.

[15] K. Berns, A. Köpper und B. Schürmann: *Technische Grundlagen Eingebetteter Systeme.* Springer Vieweg Wiesbaden, September 2019.

[16] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega und F. J. Cazorla: *Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study.* In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, Seiten 241–248, 2013.

[17] 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.* dpunkt.verlag, Mai 2011.

[18] M. Rausand: *Reliability of Safety-Critical Systems: Theory and Application.* John Wiley & Sons Inc, Hoboken, New Jersey, 2014.

[19] International Electrotechnical Commission (IEC): *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*, April 2010. Parts 1-7.

[20] Society of Automotive Engineers: *ARP4754A: Guidelines for Development of Civil Aircraft and Systems*, Dezember 2010.

[21] International Organization for Standardization: *ISO 26262: Road vehicles - Functional safety*, Dezember 2018. Parts 1-12.

[22] M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench und J. Nowotsch: *Mixed-Criticality Embedded Systems – A Balance Ensuring Partitioning and Performance.* In: *2015 Euromicro Conference on Digital System Design*, Seiten 453–461, 2015.

[23] R. Ernst und M. Di Natale: *Mixed Criticality Systems—A History of Misconceptions?* IEEE Design & Test, 33(5):65–74, 2016.

[24] P. Graydon und I. Bate: *Safety assurance driven problem formulation for mixed-criticality scheduling.* Proc. WMC, RTSS, Seiten 19–24, 2013.

[25] P. Axer, J. Diemer, M. Negrean, M. Sebastian, S. Schliecker und R. Ernst: *Mastering MPSoCs for Mixed-critical Applications.* Information and Media Technologies, 6(4):1027–1052, 2011.

[26] D. Kleidermacher und M. Kleidermacher: *Embedded systems security: practical methods for safe and secure software and systems development.* Elsevier, 2012.

[27] A. Avizienis, J.-C. Laprie, B. Randell und C. Landwehr: *Basic concepts and taxonomy of dependable and secure computing.* IEEE Transactions on Dependable and Secure Computing, 1(1):11–33, 2004.

[28] J. L. Hennessy und D. A. Patterson: *Computer architecture: a quantitative approach.* Morgan Kaufmann/Elsevier, Waltham, MA, 5 Auflage, 2012.

[29] U. Brinkschulte und T. Ungerer: *Grundlegende Prozessortechniken*, Seiten 17–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[30] Schiffmann, Wolfram: *Prozessorarchitektur.* In: *Technische Informatik 2*, Seiten 119–135. Springer-Verlag, Berlin/Heidelberg, 2005.

[31] *big.LITTLE - Arm.* https://www.arm.com/technologies/big-little. Zugriff: 2023-08-05.

[32] P. Greenhalgh: *Big.LITTLE Processing with ARM Cortex™-A15 & Cortex-A7.* Technischer Bericht, ARM, September 2011.

[33] J. L. Hennessy und D. A. Patterson: *Computer architecture: a quantitative approach.* Morgan Kaufmann/Elsevier, Cambridge, MA, 6 Auflage, 2019.

[34] J. J. Rodríguez Andina, E. de la Torre-Arnanz und M. D. Valdes Peña: *FPGAs: Fundamentals, advanced features, and applications in industrial electronics.* CRC Press, Taylor & Francis Group, Boca Raton, 2017.

[35] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide.* Technischer Bericht, Intel, Juni 2023.

[36] *Arm CoreSight Architecture Specification v3.0.* Technischer Bericht, Arm Ltd., Februar 2022.

[37] *AURIX Trace Overview and Use-Cases.* Technischer Bericht, iSystem AG, März 2019.

[38] *The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface, Version 3.0.* Standard, IEEE - Industry Standards and Technology Organization (IEEE-ISTO), Juni 2012.

[39] Xilinx: *Zynq UltraScale+ Device - Technical Reference Manual*, Dezember 2020.

[40] A. Rajabzadeh und S. G. Miremadi: *CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing.* Microelectronics Reliability, 46(5):959–972, 2006.

[41] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla und P. Martín-Holgado: *Online Error Detection Through Trace Infrastructure in ARM*

Microprocessors. IEEE Transactions on Nuclear Science, 66(7):1457–1464, 2019.

[42] M. Fazeli, R. Farivar und S. Miremadi: *A software-based concurrent error detection technique for power PC processor-based embedded systems.* In: *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, Seiten 266–274, Oktober 2005.

[43] A. Hoppe, J. Becker und F. L. Kastensmidt: *Fine Grained Control Flow Checking with Dedicated FPGA Monitors.* In: *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, Seiten 219–224, 2020.

[44] A. Wankler Hoppe: *Real-time trace decoding and monitoring for safety and security in embedded systems.* Dissertation, Karlsruher Institut für Technologie (KIT), 2022.

[45] *Understanding Embedded Trace.* Whitepaper, Accemic Technologies GmbH, September 2020.

[46] J. Rushby: *Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance.* Technischer Bericht, März 2000. NASA Contractor Report NASA/CR-1999-209347.

[47] R. Obermaisser und D. Weber: *Architectures for mixed-criticality systems based on networked multi-core chips.* In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA).* IEEE, 2014.

[48] R. Fuchsen: *How to address certification for multi-core based IMA platforms: Current status and potential solutions.* In: *29th Digital Avionics Systems Conference*, 2010.

[49] M. Portnoy: *Virtualization essentials.* John Wiley & Sons Inc, 2012.

[50] G. J. Popek und R. P. Goldberg: *Formal Requirements for Virtualizable Third Generation Architectures.* Communications of the ACM, 17(7):412–421, Juli 1974.

[51] M. Cinque, D. Cotroneo, L. De Simone und S. Rosiello: *Virtualizing mixed-criticality systems: A survey on industrial trends and issues.* Future Generation Computer Systems, 129:315–330, 2022.

[52] R. P. Goldberg: *Architectural principles for virtual computer systems.* Dissertation, Harvard University Cambridge, MA, 1973.

[53] G. Heiser: *The Role of Virtualization in Embedded Systems.* In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, Seite 11–16, New York, NY, USA, 2008. Association for Computing Machinery.

[54] VMWare, Inc.: *Understanding Full Virtualization, Paravirtualization, and Hardware Assist.* Technischer Bericht, 2007.

[55] Airlines Electronic Engineering Committee (AEEC): *Avionics Application Software*

Standard Interface Part 1 – Required Services (ARINC specification 653P1-3), November 2010.

[56] P. J. Prisaznuk: *ARINC 653 role in Integrated Modular Avionics (IMA)*. In: *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008.

[57] H. Theiling: *PikeOS and Time-Triggering*, 2013.

[58] R. Fuchsen: *Using Multi-Core Platforms in Safety-Critical Environments*, 2019.

[59] *XEN Project Schedulers*. https://wiki.xenproject.org/wiki/Xen_Project_Schedulers. Zugriff: 2023-08-02.

[60] A. Crespo, I. Ripoll und M. Masmano: *Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach*. In: *2010 European Dependable Computing Conference*, Seiten 67–72, 2010.

[61] Green Hills Software: *Tech Note: Embedded Hypervisors Fall Short of INTEGRITY-178 tuMP for Security and Multicore Features*, 2023.

[62] S. H. VanderLeest: *ARINC 653 hypervisor*. In: *29th Digital Avionics Systems Conference*, 2010.

[63] J. Strosnider, J. Lehoczky und L. Sha: *The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*. IEEE Transactions on Computers, 44(1):73–91, 1995.

[64] S. Xi, J. Wilson, C. Lu und C. Gill: *RT-Xen: Towards real-time hypervisor scheduling in Xen*. In: *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, Seiten 39–48, 2011.

[65] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky und I. Lee: *Real-time multi-core virtual machine scheduling in Xen*. In: *2014 International Conference on Embedded Software (EMSOFT)*, 2014.

[66] R. Ramsauer, J. Kiszka, D. Lohmann und W. Mauerer: *Look Mum, no VM Exits! (Almost)*, 2017.

[67] C. A. Waldspurger: *Memory Resource Management in VMware ESX Server*. In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, Seite 181–194, USA, 2002. USENIX Association.

[68] *Jailhouse Source Code*. https://github.com/siemens/jailhouse. Zugriff: 2023-08-03.

[69] *QEMU*. https://www.qemu.org/. Zugriff: 2023-08-03.

[70] *Inter-VM Shared Memory device*. https://www.qemu.org/docs/master/system/devices/ivshmem.html. Zugriff: 2023-08-03.

[71] SYSGO GmbH: *PikeOS RTOS & Hypervisor*. https://www.sysgo.com/pikeos.

Zugriff: 2023-08-03.

[72] SYSGO GmbH: *PikeOS 5.1 - Certified RTOS with Hypervisor Functionality*. https://www.sysgo.com/fileadmin/user_upload/data/flyers_brochures/SYSGO_PikeOS_Product_Overview.pdf, November 2022. Product Overview.

[73] A. Motzkus und M. Oezer: *PikeOS Safe Real-Time Scheduling*, 2016.

[74] *XEN Project*. https://xenproject.org/. Zugriff: 2023-08-03.

[75] *XEN Project Software Overview*. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Zugriff: 2023-08-03.

[76] Automotive Grade Linux (AGL) Virtualization Expert Group (EG-VIRT): *The Automotive Grade Linux Software Defined Connected Car Architecture*. Technischer Bericht, The Linux Foundation, Juni 2018.

[77] A. Burns und R. I. Davis: *Mixed Criticality Systems - A Review (13th Edition, February 2022)*. Februar 2022.

[78] J. Nowotsch und M. Paulitsch: *Leveraging Multi-core Computing Architectures in Avionics*. In: *2012 Ninth European Dependable Computing Conference*, Seiten 132–143, 2012.

[79] 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*, Seiten 109–118, 2014.

[80] D. Dasari, B. Akesson, V. Nelis, M. A. Awan und S. M. Petters: *Identifying the sources of unpredictability in COTS-based multicore systems*. In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, Seiten 39–48, Porto, Juni 2013. IEEE.

[81] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters und H. Theiling: *Multicore In Real-Time Systems – Temporal Isolation Challenges Due To Shared Resources*. Technischer Bericht, 2013.

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

[83] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez und M. Gatti: *Deterministic platform software for hard real-time systems using multi-core COTS*. In: *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 2015.

[84] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer und R. I. Davis: *A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems*. ACM

Computing Surveys, 52(3), Juni 2019.

[85] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich und R. Pellizzoni: *A Survey on Cache Management Mechanisms for Real-Time Embedded Systems.* ACM Computing Surveys, 48(2):32:1–32:36, November 2015.

[86] V. Suhendra und T. Mitra: *Exploring locking & partitioning for predictable shared caches on multi-cores.* In: *Proceedings of the 45th annual Design Automation Conference*, DAC '08, Seiten 300–303, New York, NY, USA, Juni 2008. Association for Computing Machinery.

[87] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo und R. Pellizzoni: *Real-time cache management framework for multi-core architectures.* In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 45–54, April 2013.

[88] S. Mittal: *A Survey of Techniques for Cache Partitioning in Multicore Processors.* ACM Computing Surveys, 50(2):27:1–27:39, Mai 2017.

[89] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal und R. Iyer: *Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family.* In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Seiten 657–668, März 2016.

[90] R. E. Kessler und M. D. Hill: *Page placement algorithms for large real-indexed caches.* ACM Transactions on Computer Systems, 10(4):338–359, November 1992.

[91] N. Guan, M. Stigge, W. Yi und G. Yu: *Cache-aware scheduling and analysis for multicores.* In: *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, Seiten 245–254, New York, NY, USA, Oktober 2009. Association for Computing Machinery.

[92] H. Kim, A. Kandhalu und R. Rajkumar: *A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems.* In: *2013 25th Euromicro Conference on Real-Time Systems*, Seiten 80–89, Juli 2013.

[93] B. C. Ward, J. L. Herman, C. J. Kenna und J. H. Anderson: *Making Shared Caches More Predictable on Multicore Platforms.* In: *2013 25th Euromicro Conference on Real-Time Systems*, Seiten 157–167, Juli 2013.

[94] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson und F. D. Smith: *Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning.* Real-Time Systems, 53(5):709–759, September 2017.

[95] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo und X. Li: *A Simple Cache Partitioning Approach in a Virtualized Environment.* In: *2009 IEEE International*

Symposium on Parallel and Distributed Processing with Applications, Seiten 519–524, August 2009.

[96] D. Kim, H. Kim und J. Huh: *vCache: Providing a Transparent View of the LLC in Virtualized Environments.* IEEE Computer Architecture Letters, 13(2):109–112, Juli 2014.

[97] H. Kim und R. R. Rajkumar: *Real-time cache management for multi-core virtualization.* In: *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, New York, NY, USA, Oktober 2016. Association for Computing Machinery.

[98] P. Modica, A. Biondi, G. Buttazzo und A. Patel: *Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms.* In: *2018 IEEE International Conference on Industrial Technology (ICIT)*, Seiten 1651–1657, Februar 2018.

[99] Y. Lim und H. Kim: *Cache-Aware Real-Time Virtualization for Clustered Multi-Core Platforms.* IEEE Access, 7:128628–128640, 2019.

[100] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente und M. Bertogna: *Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems.* In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[101] G. Schwäricke, T. Kloda, G. Gracioli, M. Bertogna und M. Caccamo: *Fixed-Priority Memory-Centric Scheduler for COTS-Based Multiprocessors.* 2020.

[102] W. J. Bolosky: *Software Coherence in Multiprocessor Memory Systems.* Dissertation, University of Rochester, Computer Science Department, Mai 1993.

[103] D. Dasari, B. Akesson, V. Nelis, M. A. Awan und S. M. Petters: *Identifying the sources of unpredictability in COTS-based multicore systems.* In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, Seiten 39–48, Porto, Juni 2013.

[104] H. Yun, R. Mancuso, Z.-P. Wu und R. Pellizzoni: *PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms.* In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 155–166, April 2014.

[105] X. Pan, Y. J. Gownivaripalli und F. Mueller: *TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring.* In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Seiten 363–372, Mai 2016.

[106] A. Schranzhofer, J.-J. Chen und L. Thiele: *Timing Predictability on Multi-Processor Systems with Shared Resources.* Embedded Systems Week – Workshop on Reconci-

ling Performance with Predictability, 2009.

[107] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele und M. Caccamo: *Worst-case response time analysis of resource access models in multi-core systems.* In: *Proceedings of the 47th Design Automation Conference*, DAC '10, Seiten 332–337, New York, NY, USA, Juni 2010. Association for Computing Machinery.

[108] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti und W. Puffitsch: *Predictable Flight Management System Implementation on a Multicore Processor.* In: *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France, Februar 2014.

[109] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo und R. Kegley: *A Predictable Execution Model for COTS-Based Embedded Systems.* In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Seiten 269–279, 2011.

[110] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo und L. Sha: *Real-Time Control of I/O COTS Peripherals for Embedded Systems.* In: *2009 30th IEEE Real-Time Systems Symposium*, Seiten 193–203, Dezember 2009.

[111] G. Yao, R. Pellizzoni, S. Bak, E. Betti und M. Caccamo: *Memory-centric scheduling for multicore hard real-time systems.* Real-Time Systems, 48(6):681–715, November 2012.

[112] G. Yao, R. Pellizzoni, S. Bak, H. Yun und M. Caccamo: *Global Real-Time Memory-Centric Scheduling for Multicore Systems.* IEEE Transactions on Computers, 65(9):2739–2751, September 2016.

[113] T. Thilakasiri und M. Becker: *Methods to Realize Preemption in Phased Execution Models.* ACM Transactions on Embedded Computing Systems, 22(5s), Oktober 2023.

[114] I. Senoussaoui, H.-E. Zahaf, G. Lipari und K. M. Benhaoua: *Contention-free scheduling of PREM tasks on partitioned multicore platforms.* In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2022.

[115] I. Senoussaoui, M. K. Benhaoua, H.-E. Zahaf und G. Lipari: *Toward memory-centric scheduling for PREM task on multicore platforms, when processor assignments are specified.* In: *2022 3rd International Conference on Embedded & Distributed Systems (EDiS)*, Seiten 11–15, November 2022.

[116] J. Arora, S. A. Rashid, C. Maia und E. Tovar: *Analyzing Fixed Task Priority Based Memory Centric Scheduler for the 3-Phase Task Model.* In: *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Seiten 51–60, August 2022.

[117] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo und L. Sha: *MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms.* In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 55–64, April 2013.

[118] 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: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. IEEE Conference Publications, 2014.

[119] F. Bellosa: *Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment.* Technischer Bericht TR-I4-97-02, University of Erlangen, Germany, Juli 1997.

[120] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer und U. Schlichtmann: *Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores.* In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, Seiten 133–145, 2022.

[121] J. Kim, P. Shin, S. Noh, D. Ham und S. Hong: *Reducing Memory Interference Latency of Safety-Critical Applications via Memory Request Throttling and Linux Cgroup.* In: *2018 31st IEEE International System-on-Chip Conference (SOCC)*, Seiten 215–220, September 2018.

[122] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo und S. Lui: *Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality.* 2012.

[123] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo und L. Sha: *MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms Heechul.* In: *IEEE Transactions on Computers*, 2015.

[124] J. Nowotsch und M. Paulitsch: *Quality of service capabilities for hard real-time applications on multi-core processors.* In: *Proceedings of the 21st International conference on Real-Time Networks and Systems*, RTNS '13, Seiten 151–160, New York, NY, USA, Oktober 2013. Association for Computing Machinery.

[125] H. Yun, W. Ali, S. Gondi und S. Biswas: *BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms.* IEEE Transactions on Computers, 66(7):1247–1252, Juli 2017.

[126] M. Xu, R. Gifford und L. T. X. Phan: *Holistic multi-resource allocation for multicore real-time virtualization.* In: *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, Juni 2019. Association for Computing Machinery.

[127] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig und M. Paulitsch: *Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study.* Seiten 22 pages, 768620 bytes, 2017.

[128] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo und R. Mancuso: *MemPol: Policing Core Memory Bandwidth from Outside of the Cores.* In: *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 235–248, Mai 2023.

[129] *Arm Architecture Reference Manual Supplement: Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture.* Technischer Bericht, Arm Ltd., November 2022.

[130] M. Zini, D. Casini und A. Biondi: *Analyzing Arm's MPAM From the Perspective of Time Predictability.* IEEE Transactions on Computers, 72(1):168–182, Januar 2023.

[131] *Intel® Resource-Director-Technik (Intel® RDT).* https://www.intel.com/content/www/de/de/architecture-and-technology/resource-director-technology.html. Zugriff: 2024-01-07.

[132] P. Sohal, M. Bechtel, R. Mancuso, H. Yun und O. Krieger: *A Closer Look at Intel Resource Director Technology (RDT).* In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS '22, Seiten 127–139, New York, NY, USA, Juni 2022. Association for Computing Machinery.

[133] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, M. Roy und F. Vargas: *Monitoring On-line Timing Information to Support Mixed-Critical Workloads.* In: *IEEE Real-Time Systems Symposium 2013*, Seiten 9–10, Vancouver, Canada, Dezember 2013.

[134] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy und C. Rochange: *Run-Time control to increase task parallelism in mixed-Critical systems.* In: *Proceedings - Euromicro Conference on Real-Time Systems*, Seiten 119–128, 2014.

[135] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal und D. G. Pérez: *Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems.* In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, Seiten 139–148, New York, NY, USA, Oktober 2014. Association for Computing Machinery.

[136] A. Kritikakou, T. Marty und M. Roy: *DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems.* ACM Transactions on Design Automation of Electronic Systems, 23(2):13:1–13:26, Oktober 2017.

[137] J. Freitag, S. Uhrig und T. Ungerer: *Virtual Timing Isolation for Mixed-Criticality*

Systems. In: S. Altmeyer (Herausgeber): *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, Band 106 der Reihe *Leibniz International Proceedings in Informatics (LIPIcs)*, Seiten 13:1–13:23, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[138] J. Freitag und S. Uhrig: *Dynamic interference quantification for multicore processors.* In: *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017.

[139] J. Freitag und S. Uhrig: *Closed Loop Controller for Multicore Real-Time Systems.* In: M. Berekovic, R. Buchty, H. Hamann, D. Koch und T. Pionteck (Herausgeber): *Architecture of Computing Systems – ARCS 2018*, Lecture Notes in Computer Science, Seiten 45–56, Cham, 2018. Springer International Publishing.

[140] A. Crespo, P. Balbastre, J. Simo, J. Coronel, D. Gracia-Perez und P. Bonnot: *Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems.* IEEE Access, 6:50627–50640, 2018.

[141] A. Crespo, A. Soriano, P. Balbastre, J. Coronel, D. G. Perez und P. Bonnot: *Hypervisor Feedback Control of Mixed-Criticality Systems: the XtratuM Approach.* In: *Proceedings of OSPERT 2017, the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Seiten 35–40, 2017.

[142] A. Saeed, D. Hoornaert, D. Dasari, D. Ziegenbein, D. Mueller-Gritschneder, U. Schlichtmann, A. Gerstlauer und R. Mancuso: *Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs.* 2023.

[143] A. Hoppe, J. Becker und F. L. Kastensmidt: *High-speed Hardware Accelerator for Trace Decoding in Real-Time Program Monitoring.* In: *2021 IEEE 12th Latin America Symposium on Circuits and System (LASCAS)*, 2021.

[144] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-García, A. Lindoso und L. Entrena: *Online Test of Control Flow Errors: A New Debug Interface-Based Approach.* IEEE Transactions on Computers, 65(6):1846–1855, Juni 2016.

[145] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla und P. Martín-Holgado: *Online Error Detection Through Trace Infrastructure in ARM Microprocessors.* IEEE Transactions on Nuclear Science, 66(7):1457–1464, Juli 2019.

[146] A. Hoppe, F. L. Kastensmidt und J. Becker: *Investigating real-time control-flow error detection in hardware: How fast can we detect errors and take action?* Microelectronics Reliability, 126:114264, November 2021.

[147] A. W. Hoppe, F. L. Kastensmidt und J. Becker: *Control Flow Analysis for Embedded Multi-core Hybrid Systems.* In: N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos und P. C. Diniz (Herausgeber): *Applied Reconfigurable Computing.*

Architectures, Tools, and Applications, Lecture Notes in Computer Science, Seiten 485–496, Cham, 2018. Springer International Publishing.

[148] D. Kuzhiyelil, P. Zieris, M. Kadar, S. Tverdyshev und G. Fohler: *Towards Transparent Control-Flow Integrity in Safety-Critical Systems.* In: W. Susilo, R. H. Deng, F. Guo, Y. Li und R. Intan (Herausgeber): *Information Security*, Lecture Notes in Computer Science, Seiten 290–311, Cham, 2020. Springer International Publishing.

[149] M. Kadar, G. Fohler, D. Kuzhiyelil und P. Gorski: *Safety-Aware Integration of Hardware-Assisted Program Tracing in Mixed-Criticality Systems for Security Monitoring.* In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Seiten 292–305, Mai 2021.

[150] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapôtre und G. Gogniat: *ARMHEx: A hardware extension for DIFT on ARM-based SoCs.* In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, September 2017.

[151] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, A. K. Biswas, V. Lapôtre und G. Gogniat: *A small and adaptive coprocessor for information flow tracking in ARM SoCs.* In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dezember 2018.

[152] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapôtre, G. Gogniat und A. K. Biswas: *A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components.* In: *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, Seiten 92–97, Dezember 2018.

[153] Y. Lee, I. Heo, D. Hwang, K. Kim und Y. Paek: *Towards a practical solution to detect code reuse attacks on ARM mobile devices.* In: *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15, New York, NY, USA, Juni 2015. Association for Computing Machinery.

[154] A. Vaishnav, K. D. Pham und D. Koch: *A Survey on FPGA Virtualization.* In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Seiten 131–1317, August 2018.

[155] Q. Ijaz, E.-B. Bourennane, A. K. Bashir und H. Asghar: *Revisiting the High-Performance Reconfigurable Computing for Future Datacenters.* Future Internet, 12(4):64, April 2020.

[156] M. H. Quraishi, E. B. Tavakoli und F. Ren: *A Survey of System Architectures and Techniques for FPGA Virtualization.* In: *IEEE Transactions on Parallel and Distributed Systems*, Band 32, Seiten 2216–2230, 2021.

[157] R. Skhiri, V. Fresse, J. P. Jamont, B. Suffran und J. Malek: *From FPGA to Support*

Cloud to Cloud of FPGA: State of the Art. International Journal of Reconfigurable Computing, 2019, Dezember 2019.

[158] C. Wulf, M. Willig und D. Göhringer: *A Survey on Hypervisor-based Virtualization of Embedded Reconfigurable Systems.* In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, Seiten 249–256, August 2021.

[159] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi und B. Kasikci: *A Hypervisor for Shared-Memory FPGA Platforms.* In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 827–844, Lausanne Switzerland, März 2020. ACM.

[160] T. Xia, Y. Tian, J.-C. Prévotet und F. Nouvel: *Ker-ONE: A new hypervisor managing FPGA reconfigurable accelerators.* Journal of Systems Architecture, 98:453–467, September 2019.

[161] T. Xia, J.-C. Prévotet und F. Nouvel: *Hypervisor mechanisms to manage FPGA reconfigurable accelerators.* In: *2016 International Conference on Field-Programmable Technology (FPT)*, Seiten 44–52, Dezember 2016.

[162] AMD: *Dynamic Function eXchange Decoupler LogiCORE IP Product Guide (PG375)*, Mai 2022.

[163] Patrick Kutch: *PCI-SIG SR-IOV Primer*, Januar 2011. Revision 2.5.

[164] S. Chiotakis, S. Pinneterre und M. Paolino: *vFPGAmanager: A Hardware-Software Framework for Optimal FPGA Resources Exploitation in Network Function Virtualization.* In: *2019 European Conference on Networks and Communications (EuCNC)*, Seiten 47–51, Juni 2019.

[165] M. Asiatici, N. George, K. Vipin, S. A. Fahmy und P. Ienne: *Virtualized Execution Runtime for FPGA Accelerators in the Cloud.* IEEE Access, 5:1900–1910, 2017.

[166] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero und G. Buttazzo: *AXI Hyper-Connect: A Predictable, Hypervisor-level Interconnect for Hardware Accelerators in FPGA SoC.* In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*, Juli 2020.

[167] K. D. Pham, K. Paraskevas, A. Vaishnav, A. Attwood, M. Vesper und D. Koch: *ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs.* In: *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*, September 2019.

[168] K. D. Pham, A. Vaishnav, M. Vesper und D. Koch: *ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications.* In: *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, August 2018.

[169] D. R. E. Gnad, F. Oboril und M. B. Tahoori: *Voltage drop-based fault attacks on FPGAs using valid bitstreams.* In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, September 2017.

[170] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen und T. Moscibroda: *The Feniks FPGA Operating System for Cloud Computing.* In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, New York, NY, USA, September 2017. Association for Computing Machinery.

[171] M. Pagani, E. Rossi, A. Biondi, M. Marinoni, G. Lipari und G. Buttazzo: *A Bandwidth Reservation Mechanism for AXI-based Hardware Accelerators on FPGAs.* In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, Stuttgart, Germany, Juli 2019.

[172] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni und G. Buttazzo: *Is Your Bus Arbiter Really Fair? Restoring Fairness in AXI Interconnects for FPGA SoCs.* ACM Transactions on Embedded Computing Systems, 18(5s), Oktober 2019.

[173] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao und D. Burger: *A reconfigurable fabric for accelerating large-scale datacenter services.* ACM SIGARCH Computer Architecture News, 42(3):13–24, Juni 2014.

[174] AMD: *AXI Protocol Checker LogiCORE IP Product Guide (PG101)*, Juni 2023.

[175] *WB2AXIPSP: AXISAFETY.* https://github.com/ZipCPU/wb2axip/blob/master/rtl/axisafety.v, Januar 2024. Zugriff: 2024-01-30.

[176] F. Restuccia, A. Biondi, M. Marinoni und G. Buttazzo: *Safely Preventing Unbounded Delays During Bus Transactions in FPGA-based SoC.* In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Seiten 129–137, Mai 2020.

[177] J. Freitag: *Virtual Timing Isolation Safety-Net for Multicore Processors.* Dissertation, Universität Augsburg, Augsburg, März 2020.

[178] J. P. A. Ribeiro: *A TrustZone-assisted hypervisor supporting dynamic partial reconfiguration.* Masterarbeit, University of Minho, Oktober 2018.

[179] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy und D. L. Maskell: *Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform.* Journal of Signal Processing Systems, 77(1):61–76, Oktober 2014.

[180] S. K. Everton, M. Hirsch, P. Stravroulakis, R. K. Leach und A. T. Clare: *Review of*

in-situ process monitoring and in-situ metrology for metal additive manufacturing. Materials & Design, 95:431–445, April 2016.

[181] R. K. Mobley: *An introduction to predictive maintenance*. Butterworth-Heinemann, Amsterdam [u.a.], 2. ed Auflage, 2002.

[182] Lemaker: *Banana Pro & Pi User Manual V1.0*, April 2015.

[183] *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. https://www.freertos.org/. Zugriff: 2024-01-12.

[184] *The Real Time Linux collaborative project [Wiki]*. https://wiki.linuxfoundation.org/realtime/start. Zugriff: 2024-01-16.

[185] *stress - tool to impose load on and stress test a computer system*. https://github.com/resurrecting-open-source-projects/stress. Zugriff: 2024-01-18.

[186] J. L. Hintze und R. D. Nelson: *Violin Plots: A Box Plot-Density Trace Synergism*. The American Statistician, 52(2):181–184, Mai 1998.

[187] B. Andersson: *Verifying Timing in Undocumented Multicore Processors*. https://insights.sei.cmu.edu/blog/verifying-timing-in-undocumented-multicore-processors/, August 2020. Zugriff: 2023-08-20.

[188] Xilinx: *ZCU102 Evaluation Board - User Guide*, Februar 2023.

[189] V. Suhendra und T. Mitra: *Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores*. In: *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, Seite 300–303, New York, NY, USA, 2008. Association for Computing Machinery.

[190] S. J. Hu: *Evolving Paradigms of Manufacturing: From Mass Production to Mass Customization and Personalization*. Procedia CIRP, 7:3–8, Januar 2013.

[191] *Wertstromkinematik - Innovative, wandlungsfähige Produktion der Zukunft*. http://wertstromkinematik.de/, März 2021. Zugriff: 2024-01-26.

[192] E. Mühlbeier, P. Gönnheimer, L. Hausmann und J. Fleischer: *Value Stream Kinematics*. In: B.-A. Behrens, A. Brosius, W. Hintze, S. Ihlenfeldt und J. P. Wulfsberg (Herausgeber): *Production at the leading edge of technology*, Lecture Notes in Production Engineering, Seiten 409–418, Berlin, Heidelberg, 2021. Springer.

[193] AMD: *Zynq 7000 SoC Technical Reference Manual*, Juni 2023.

[194] *Xen and PL Masters - Xilinx Wiki*. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842066/Xen+and+PL+Masters, April 2020. Zugriff: 2024-01-29.

[195] *Information Technology – Open Systems Interconnection – Basic Reference Model*.

Standard, International Telecommunications Union, 1994.

[196] HMS Networks: *Continued growth for industrial networks despite pandemic.* https://www.hms-networks.com/news-and-insights/news-from-hms/2021/03/31/continued-growth-for-industrial-networks-despite-pandemic, März 2021. Zugriff: 2024-01-30.

[197] Avnet: *ZedBoard Product Brief*, 2022.

[198] J. Chaudhuri und K. Chakrabarty: *Criticality Analysis of Ring Oscillators in FPGA Bitstreams.* In: *2023 IEEE European Test Symposium (ETS)*, Mai 2023.

Betreute studentische Arbeiten

[Bal17] L. M. Balzer: *Entwicklung einer Hardwareerweiterung für sicherheitskritische heterogene MPSoC-Architekturen unter Verwendung dynamisch partieller Rekonfiguration.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2017.

[Del20] C. V. Del Rio Ortiz: *Entwicklung eines Ressourcenverwaltungsmoduls zum Gastsystemmanagement für den XEN-Hypervisor.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2020.

[Dud22] K. Dudzik: *Hierarchisches Scheduling zur Realisierung von Logical Execution Time in RTOS- und Hypervisor-basierten Eingebetteten Systemen.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Fuc23] V. Fuchs: *Entwurf eines Synchronisationsframeworks und Anwendung zur Entwicklung eines Synchronisationsmechanismus im Umfeld von Werkzeugmaschinen.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2023.

[Gra22] J. Graff: *Hypervisor-level Schedule Synchronization in Networked Embedded Systems to Realize System-level Logical Execution Time.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Gut22] A. Gutermann: *Analyse und Erweiterung der Thread Bibliothek zur Verbesserung der Echtzeitfähigkeit von OpenJ9.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Hir22] S. Hirsch: *Umsetzung von System-Level Logical Execution Time für verteilte RTEMS-Instanzen.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Hof20] M. Hoffmann: *Erweiterung eines Magnetfeld-Sensor-Prüfstandes um eine zweite zu messende Achse.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2020.

[Hot18] T. Hotfilter: *Design and Implementation of a Concept for Remote I/O Peripheral Access after Dynamic Migration.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2018.

[Ibn22] K. Ibn Bari: *Development and Implementation of a Flexible Safety-Strategy for a Robot-based Handling Module.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Ism22] A. Ismail: *Echtzeitoptimierung der Signalverarbeitung einer hochdynamischen Lasermaterialbearbeitungsanlage.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Jia18] G. Jiang: *Untersuchung von High-Level-Synthese zur Entwicklung eines Kryptographiemoduls für rekonfigurierbare Hardwareplattformen.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2018.

[Kar21] C. Karle: *Entwurf einer Architektur zum Multiplexing verschiedener Kommunikationsprotokolle unter Nutzung von FPGAs.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2021.

[Kec23] F. Keck: *Pattern-basierte Implementierung von Software-Redundanz für sicherheitskritische Echtzeitsysteme.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2023.

[Kep22] C. Kepes: *Umsetzung von Logical Execution Time in FreeRTOS als Basis für Zeitdeterminismus in verteilten Systemen.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2022.

[Kle17] M. Kleinert: *Untersuchung von Störeinflüssen auf phasenbasierte Algorithmen bei der videobasierten Schwingungsanalyse im industriellen Umfeld.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2017.

[Kre18] M. Kreutzer: *Entwicklung eines Monitoring-Systems für Hypervisor-basierte Multiprozessor SoCs.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2018.

[Kre21] M. Kreutzer: *Architecture for the Integration of Reconfigurable Hardware Accelerators in Hypervisor-based Edge Devices.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2021.

[Leh18] N. Lehrke: *Entwicklung eines Systems zur Fahrzeuglokalisierung durch hardwaregestützte Bildverarbeitung und Datenfusion mit V2V-Positionsdaten.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2018.

[Mat21] M. Matt: *Entwurf einer Hardwarearchitektur einer 5G-NR Kommunikation für verfahrenstechnische Messgeräte.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2021.

[Mat23] J. Matthies: *Entwicklung eines Frameworks zur transparenten Einbringung von Testdaten in eingebettete Messsysteme.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2023.

[Met20] H. Metzger: *Edge in der industriellen Automatisierung - Aufbereitung und Bereitstellung von Feldgerätedaten.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2020.

[Rak20] E. Rakipaj: *Automatisiertes Job-Deployment in rekonfigurierbaren Industrie-4.0-Produktionsystemen am Beispiel additiver Fertigung.* Bachelorarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2020.

[Sza18] T. Szabo: *Überwachung von Embedded Hypervisors anhand von CPU-Trace-Informationen.* Masterarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2018.

Publikationen

[SBFB21] F. Schade, D. Barton, J. Fleischer und J. Becker: *Evaluation of a Hypervisor-Based Smart Controller for Industry 4.0 Functions in Manufacturing.* In: *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, Seiten 680–685, Juni 2021.

[SDA+23] F. Schade, T. Dörr, A. Ahlbrecht, V. Janson, U. Durak und J. Becker: *Automatic Deployment of Embedded Real-Time Software Systems to Hypervisor-Managed Platforms.* In: *2023 26th Euromicro Conference on Digital System Design (DSD)*, Seiten 436–443, September 2023.

[SDB22] F. Schade, T. Dörr und J. Becker: *Hypervisor-Based Target Deployment Strategies for Time Predictability in Model-Based Development.* In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, September 2022.

[SKM+22] F. Schade, C. Karle, E. Mühlbeier, P. Gönnheimer, J. Fleischer und J. Becker: *Dynamic Partial Reconfiguration for Adaptive Sensor Integration in Highly Flexible Manufacturing Systems.* Procedia CIRP, 107:1311–1316, Januar 2022.

[SKM+23] F. Schade, M. Kreutzer, E. Mühlbeier, E. Gerlitz, P. Gönnheimer, J. Fleischer und J. Becker: *Modular Hardware/Software Architecture for Edge Units in Highly Flexible Manufacturing Systems.* Procedia CIRP, 120:601–606, Januar 2023.

[SSBT22] F. Schade, T. Sandmann, J. Becker und H. Theiling: *Using Trace Data for Run-Time Optimization of Parallel Execution in Real-Time Multi-Core Systems.* In: *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Seiten 228–234, August 2022.

[BDS+18] F. K. Bapp, T. Dörr, T. Sandmann, F. Schade und J. Becker: *Towards Fail-Operational Systems on Controller Level Using Heterogeneous Multicore SoC Architectures and Hardware Support.* SAE Technical Paper 2018-01-1072, SAE International, Warrendale, PA, April 2018.

[BGS+19] D. Barton, P. Gönnheimer, F. Schade, C. Ehrmann, J. Becker und J. Fleischer: *Modular smart controller for Industry 4.0 functions in machine tools.* Procedia CIRP, 81:1331–1336, Januar 2019.

[BMD+21] J. Becker, L. Masing, T. Dörr, F. Schade, G. Keramidas, C. P. Antonopoulos, M. Mavropoulos, E. Tiganourias, V. Kelefouras, K. Antonopoulos, N. Voros, U. Durak, A. Ahlbrecht, W. Zaeske, C. Panagiotou, D. Karadimas, N. Adler, A. Sailer, R. Weber, T. Wilhelm, F. Oszwald, D. Reinhardt, M. Chamas, A. Bekan, G. Smethurst, F. Siddiqui, R. Khan, V. Garousi, S. Sezer und V. Morales: *XANDAR: X-by-Construction Design framework for Engineering Autonomous & Distributed Real-time Embedded Software Systems.* In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, Seiten 382–383, August 2021.

[DSA+22] T. Dörr, F. Schade, A. Ahlbrecht, W. Zaeske, L. Masing, U. Durak und J. Becker: *A Behavior Specification and Simulation Methodology for Embedded Real-Time Software.* In: *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Seiten 151–159, September 2022.

[DSA23] T. Dörr, F. Schade und A. Ahlbrecht: *Simulation-based development of networked avionics systems using the XANDAR toolchain.* In: *Proceedings of the 4th Summer School on Cyber-Physical Systems and Internet-of-Things*, Nummer Vol. IV, Seiten 279–315. Juli 2023.

[DSB23] T. Dörr, F. Schade und J. Becker: *Pattern-Based Information Flow Control for Safety-Critical On-Chip Systems.* In: J. Guiochet, S. Tonetta und F. Bitsch (Herausgeber): *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, Seiten 181–195, Cham, 2023. Springer Nature Switzerland.

[DSM+22] T. Dörr, F. Schade, L. Masing, J. Becker, G. Keramidas, C. P. Antonopoulos, M. Mavropoulos, V. Kelefouras und N. Voros: *Safety by Construction: Pattern-Based Application of Safety Mechanisms in XANDAR.* In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Seiten 369–370, Juli 2022.

[DSS+19] T. Dörr, T. Sandmann, F. Schade, F. K. Bapp und J. Becker: *Leveraging the Partial Reconfiguration Capability of FPGAs for Processor-Based Fail-Operational Systems.* In: C. Hochberger, B. Nelson, A. Koch, R. Woods und P. Diniz (Herausgeber): *Applied Reconfigurable Computing*, Lecture Notes in Computer Science, Seiten 96–111, Cham, 2019. Springer International Publishing.

[FAO+22] J. Fleischer, A. Albers, J. Ovtcharova, J. Becker, G. Lanza, W. Zhang,

T. Zhang, F. Qiao, Y. Ma, J. Wang, Z. Wu, C. Ehrmann, P. Gönnheimer, M. Behrendt, C. Mandel, T. Stürmlinger, M. Klippert, A. Kimmig, F. Schade, S. Yang, I. Heider, S. Xie, K. Song, J. Peng, P. Goncalves, R. Kampfmann, J. Schlechtendahl, J. Kattner, C. Straub, M. May, Z. Zhu, O. Bai, Y. Lin, Z. Yang, L. Ding und A.-S. Rossol: *Final Report Sino-German Industry 4.0 Factory Automation Platform.* Forschungsbericht, Karlsruher Institut für Technologie (KIT), Karlsruhe, Februar 2022.

[GKM+19] P. Gönnheimer, A. Kimmig, C. Mandel, T. Stürmlinger, S. Yang, F. Schade, C. Ehrmann, B. Klee, M. Behrendt, J. Schlechtendahl, M. Fischer, K. Trautmann, J. Fleischer, G. Lanza, J. Ovtcharova, J. Becker und A. Albers: *Methodical approach for the development of a platform for the configuration and operation of turnkey production systems.* Procedia CIRP, 84:880–885, Januar 2019.

[KSM+21] A. Kimmig, M. Schöck, E. Mühlbeier, F. Oexle und J. Fleischer: *Wertstromkinematik – Produktionssysteme neu gedacht: Interdisziplinäres Forscherteam arbeitet an der Produktionstechnik der Zukunft (Teil 2).* Zeitschrift für wirtschaftlichen Fabrikbetrieb, 116(12):935–939, Dezember 2021.

[LSSS23] V. Lüntzel, F. Schade, M. Sommer und E. Sax: *Modularized Platform for an Embedded Systems Case Study: Concept and Design.* In: *Human Interaction & Emerging Technologies (IHIET 2023): Artificial Intelligence & Future Applications*, Band 111. AHFE Open Acces, 2023.

[MBS+23] E. Mühlbeier, V. Bauer, F. Schade, P. Gönnheimer, J. Becker und J. Fleischer: *Mechatronic Coupling System for Cooperative Manufacturing with Industrial Robots.* Procedia CIRP, 120:744–749, Januar 2023.

[MDS+22] L. Masing, T. Dörr, F. Schade, J. Becker, G. Keramidas, C. P. Antonopoulos, M. Mavropoulos, E. Tiganourias, V. Kelefouras, K. Antonopoulos, N. Voros, U. Durak, A. Ahlbrecht, W. Zaeske, C. Panagiotou, D. Karadimas, N. Adler, A. Sailer, R. Weber, T. Wilhelm, G. Nemeth, F. Siddiqui, R. Khan, V. Garousi, S. Sezer und V. Morales: *XANDAR: Exploiting the X-by-Construction Paradigm in Model-based Development of Safety-critical Systems.* In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, März 2022.

[SKS+22] F. Siddiqui, R. Khan, S. Sezer, K. McLaughlin, L. Masing, T. Dörr, F. Schade, J. Becker, A. Ahlbrecht, W. Zaeske, U. Durak, N. Adler, A. Sailer, R. Weber, T. Wilhelm, G. Nemeth, V. Morales, P. Gomez, G. Keramidas, C. P. Antonopoulos, M. Mavropoulos, V. Kelefouras, K. Antonopoulos, N. Voros, C. Panagiotou und D. Karadimas: *XANDAR: A holistic Cybersecurity En-*

gineering Process for Safety-critical and Cyber-physical Systems. In: *2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring)*, Juni 2022.

[SSJB18] A. Silitonga, F. Schade, G. Jiang und J. Becker: *HLS-Based Performance and Resource Optimization of Cryptographic Modules.* In: *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Seiten 1009–1016, Dezember 2018.

[YYS+19] L. Yigui, S. Youteng, F. Schade, T. Hotfilter, J. Becker, Z. Yuan, O. Zizhou und L. Weiming: *Evaluation of a high-throughput communication link for future automotive ADAS controllers.* Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering, 233(9):2371–2378, August 2019.