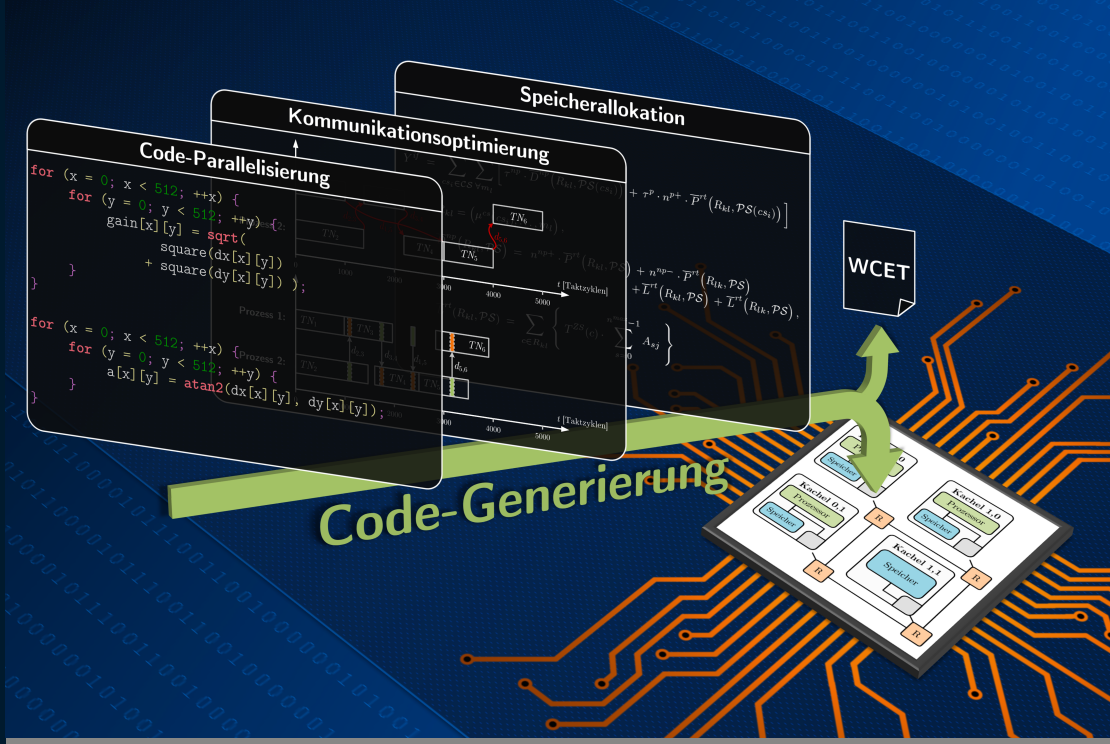


Simon Thomas Reder

Compileroptimierung und parallele Code-Generierung für zeitkritische eingebettete Multiprozessorsysteme

Compileroptimierung und parallele Code-Generierung für zeitkritische eingebettete Multiprozessorsysteme

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

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

DISSERTATION

von

M. Sc. Simon Thomas Reder

geb. in Konstanz

Tag der mündlichen Prüfung:
27. August 2020

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

Compileroptimierung und parallele Code-Generierung für zeitkritische eingebettete Multiprozessorsysteme

1. Auflage: September 2020

Zusammenfassung

Durch den voranschreitenden Trend der Digitalisierung gewinnen intelligente digitale Systeme in vielen Bereichen des täglichen Lebens zunehmend an Bedeutung. Dies betrifft insbesondere auch den Bereich sicherheitskritischer Echtzeitsysteme, für deren sicheren Betrieb die Echtzeitfähigkeit nachgewiesen werden muss. Im Gegensatz zu Anwendungsfeldern ohne diese Anforderung sind effiziente Mehrkernprozessoren in Echtzeitsystemen derzeit noch kaum verbreitet. Der Hauptgrund für die bisherige Dominanz der Einzelkernprozessoren sind fehlende Methoden und Werkzeuge, um parallele Echtzeit-Software für Mehrkernprozessoren zu entwickeln und selbst im ungünstigsten Fall noch eine maximale Ausführungszeit (englisch *Worst Case Execution Time*, kurz WCET) garantieren zu können. In diesem Kontext besteht eines der wesentlichen Probleme darin, dass sich parallel ablaufende Software-Routinen im Hinblick auf ihre Laufzeit gegenseitig beeinflussen können. In Mehrkernprozessoren geschieht dies vor allem bei gleichzeitigen Zugriffen mehrerer Kerne auf eine gemeinsam genutzte Hardware-Ressource. Geeignete Methoden, um den Einfluss dieser als *Interferenz* bezeichneten Effekte bei der Software-Entwicklung präzise vorherzusagen und sichere Garantien für die Ausführungszeit abzuleiten, sind Gegenstand aktueller Forschung. Gleiches gilt für Software-Werkzeuge zur automatischen Parallelisierung, die auf harte Echtzeitanwendungen spezialisiert sind.

Diese Arbeit zielt darauf ab, die Anwendbarkeit von Mehrkernprozessoren in Echtzeitsystemen durch Beiträge in den Bereichen der automatischen Software-Parallelisierung, Code-Optimierung und Hardware-Modellierung signifikant zu verbessern. Als Bestandteil einer Werkzeugkette zur automatischen Parallelisierung von sequentieller Echtzeit-Software wird in dieser Arbeit ein Compiler-Werkzeug zur WCET-optimierten parallelen Code-Generierung und ein zugehöriges paralleles Programmiermodell vorgestellt. Hierdurch können – weitgehend ohne Zutun eines Endanwenders – gut vorhersagbare parallele Programme erzeugt werden. Durch das Programmiermodell wird dabei sichergestellt, dass die Ausführungszeit, einschließlich der Interferenzeffekte, mit Hilfe einer statischen WCET-Analyse sicher nach oben abgeschätzt werden kann. Als Teil der Code-Generierung stellt die vorliegende Arbeit zwei Optimierungsmethoden vor, die zum einen den Kommunikations- und Synchronisationsaufwand zwischen den Prozessorkernen reduzieren und zum anderen die optimierte Allokation verteilter Speicher in heterogenen Speicherhierarchien ermöglichen. Erstere ist auf des

parallele Programmiermodell abgestimmt und erlaubt die optimierte Platzierung von Kommunikations- und Synchronisationsoperationen sowie das Entfernen redundanter Synchronisation auf einer feingranularen Ebene. Die Optimierung der Speicherallokation ergänzt den Ansatz um ein formales Optimierungsmodell zur Zuweisung der Datenfelder eines generierten Programms zu den Speicherbereichen der Zielplattform. Das Modell bezieht dabei sowohl die Kosten für Interferenzeffekte als auch die Speicherhierarchie von Zielplattformen mit verteilten und heterogenen Speichern mit ein. Um die Schritte zur Generierung, Optimierung und Analyse von paralleler Echtzeit-Software weitgehend plattformunabhängig aufbauen zu können, beinhaltet die vorliegende Arbeit außerdem einen Ansatz zur generischen Modellierung von Mehrkernprozessorarchitekturen. Dieser erlaubt es, die Zielplattform mit Hilfe einer entsprechend erweiterten Architekturbeschreibungssprache (ADL) zu beschreiben, wodurch sich die darauf aufbauenden Entwicklungswerkzeuge mit überschaubarem Aufwand auf ein breites Spektrum von Hardware-Plattformen anwenden lassen.

Mit dieser neuartigen Kombination erweitert die vorliegende Arbeit den Stand der Technik um einige wesentliche Bausteine, die die weitgehend automatisierte Parallelisierung von Echtzeit-Software ohne stark einschränkende Annahmen bezüglich der Struktur des Eingabeprogramms ermöglichen. Zu den weiteren Neuerungen dieser Arbeit zählen die Plattformunabhängigkeit bei der WCET-optimierten Software-Parallelisierung und die Berücksichtigung von Interferenzeffekten bei der Speicherallokation in Echtzeitsystemen.

Die experimentelle Evaluation der vorgestellten Methoden und deren prototypischer Umsetzung zeigt, dass die WCET aller betrachteten Testanwendungen von der Parallelisierung profitieren kann. Auf einer Plattform mit vier Prozessorkernen konnte z. B. die WCET einer Anwendung aus dem Bereich der Bildverarbeitung (konkret eine Implementierung der Hough-Transformation) durch die Parallelisierung im Vergleich zum sequentiellen Eingabeprogramm um Faktor 3,21 verbessert werden. Auch die Optimierungsansätze für Kommunikation und Speicherallokation führen größtenteils zu einer deutlichen Verbesserung der WCET. So konnten die durch Interferenzen verursachten Kosten im Zuge der Speicherallokation z. B. um bis zu 49 % reduziert werden.

Insgesamt haben die Ergebnisse dieser Arbeit damit das Potential, die effiziente und kostengünstige Nutzung von Mehrkernprozessoren im Bereich harter Echtzeitsysteme wesentlich voranzutreiben.

Abstract

Driven by the ongoing trend of digitalization, intelligent digital systems become increasingly important in various areas of daily life. That includes the field of safety-critical real-time systems, which must satisfy real-time constraints to ensure their safe operation. In contrast to application domains without these requirements, efficient multi-core processors are still uncommon in state-of-the-art real-time systems. The main reasons for the dominance of single-core processors are missing methods and tools to develop and analyze parallel real-time software for multi-core processors. In particular, specialized analysis tools are needed to provide guarantees for the Worst-Case Execution Time (WCET) of parallel software. The major problem in that context is that parallelly executed software routines may affect each other's execution times. In multi-core processors, this happens, e.g., when multiple processor cores concurrently access the same shared hardware resource. Suitable methods to precisely predict the impact of these so-called *interference effects* during design time and derive guarantees for the execution time are subject to ongoing research. That is equally true for software-tools to automatically parallelize programs, which are subject to hard real-time constraints.

The goal of this work is to significantly improve the applicability of multi-core processors in real-time systems by contributing to the state-of-the-art in automated software parallelization, code optimization, and hardware modeling. As part of a tool-chain to automatically parallelize sequential real-time software, this work introduces a compiler tool for WCET-aware parallel code generation and a corresponding parallel programming model. Based on this combination, well-predictable parallel programs can be generated, mostly without requiring end-user interventions. In the process, the programming model ensures that software execution times, including interference effects, can be tightly bounded using a static WCET analysis. As part of the code generation, this work presents two optimization methods that, on the one hand, reduce communication and synchronization overhead and, on the other hand, optimize the allocation of memories within heterogeneous memory hierarchies. The former is adjusted to the programming model and allows for the optimized placement of communication and synchronization operations with a fine granularity. Moreover, redundant synchronization can be detected and removed in the process. The memory allocation scheme supplements the approach with a formal optimization model

for mapping the data fields of the application onto the memory segments of the target platform. This model explicitly accounts for the costs of interference effects as well as the memory hierarchy of multi-core processors with distributed and heterogeneous memories. To keep the steps of generating, optimizing, and analyzing the code independent of the target platform, this work includes a generic model for multi-core processor architectures. The latter describes the target platform with an architecture description language (ADL) and allows the corresponding development tools to be applied to a wide range of target platforms without much effort.

With this novel combination, the present work extends the state-of-the-art by several essential building blocks that enable a mostly automated parallelization of real-time software without strong assumptions regarding the structure of the input program. Further novelties of this work are the platform-independence in WCET-aware code parallelization and the inclusion of interference while allocating memories for real-time systems.

The experimental evaluation of the presented methods and their implementation shows that the WCET of all considered test applications can benefit from parallelization. For a platform with four cores, the parallelization method, e.g., improved the WCET of an application from the image-processing domain (more precisely an implementation of the Hough Transformation) by factor 3.21 compared to the sequential input program. The two optimization methods for communication and memory allocation result in significant improvements of the WCET in the vast majority of cases as well. The memory allocation, for instance, reduced the costs caused by interference by up to 49 %.

Overall, the results of this work have the potential to significantly facilitate the efficient and cost-effective use of multi-core processors in the domain of hard real-time systems.

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Instituts für Technologie (KIT). In einigen spannenden Jahren als wissenschaftlicher Mitarbeiter hatte ich die Gelegenheit, Einblicke in interessante Forschungsthemen und die Hochschullehre zu erhalten. Während der Arbeit am ITIV und der Mitwirkung an Forschungsprojekten durfte ich zahlreiche Kollegen und Projektpartner kennenlernen, bei denen ich mich an dieser Stelle für den stets konstruktiven Austausch und die kollegiale Zusammenarbeit bedanken möchte.

Besonderen Dank möchte ich Prof. Jürgen Becker für die Möglichkeit zur Forschung an dem hier präsentierten Promotionsthema, die hervorragende Betreuung und die Übernahme des Hauptreferats bei dieser Dissertation aussprechen. Bedanken möchte ich mich auch für das entgegengebrachte Vertrauen, das mir ermöglicht hat, Verantwortung in Forschungsprojekten, Projektanträgen und verschiedenen anderen Bereichen des Instituts zu übernehmen.

Weiterhin möchte ich mich bei Prof. Theo Ungerer für die Übernahme des Korreferats und die Begutachtung der Arbeit bedanken. Auch den Professoren Trommer, Ulusoy und Perić danke ich für ihre Mitwirkung am Prüfungsausschuss.

Diese Arbeit wäre nicht möglich gewesen ohne das motivierende, kreative und humorvolle Arbeitsumfeld am ITIV. Dafür möchte ich all den einzigartigen Freunden und Kollegen danken, die mich in den vergangenen Jahren am ITIV begleitet haben. Sie sorgten stets für einen abwechslungsreichen Arbeitsalltag und hatten ein offenes Ohr für Diskussionen und neue Ideen. Ich möchte mich besonders bei meinen langjährigen Wegbegleitern & Bürokollegen Christoph Roth und Harald Bucher, aber auch bei den vielen anderen Kollegen (Steffen, Timo, Hannes, Lidia, Fabian, Florian, Augusto, …), einschließlich derer, die das ITIV vor mir verlassen haben (Oliver, Michael, Falco, Lukas, Timo, …), für die gemeinsame Zeit bedanken. Auch abseits der Arbeit – sei es bei den obligatorischen Kaffee- und Mittagspausen, dem Bau von Doktorhüten oder den vielfältigen Freizeitaktivitäten – wurde es mit den Kollegen niemals langweilig. Dafür danke ich besonders den Organisatoren (Steffen, Lidia, Fabian, …) der zahlreichen Kino- & Restaurantbesuche, Grillabende und anderer Aktivitäten.

Weiterhin möchte ich allen danken, die durch Korrekturen, Anregungen, konstruktive Kritik oder in anderer Weise zum Gelingen dieser Arbeit beigetragen haben. Nicht unerwähnt bleiben sollen dabei auch alle Studenten, die als HiWis, Bacheloranden oder Masteranden am ITIV wichtige Vorarbeiten geleistet haben.

Nicht zuletzt möchte ich auch meiner Familie für ihre kontinuierliche Unterstützung einen ganz besonderen Dank aussprechen. Vor allem meinen Eltern Edith und Wolfgang danke ich dafür, dass sie mich in Ausbildung und Studium unterstützt und mir dadurch auch den Weg zur Promotion geebnet haben.

Karlsruhe, im September 2020
Simon Reder

Inhaltsverzeichnis

Kapitel 1.

Einleitung

1.1. Motivation

Die *Digitalisierung* zählt zu den wesentlichen Trends der heutigen Zeit und betrifft große Teile der Wirtschaft, Mobilität, Infrastruktur und des gesellschaftlichen Lebens. Durch den anhaltenden technologischen Fortschritt eröffnen digitale rechnergestützte Systemen neue Anwendungsfelder wie das autonome und teilautonome Fahren, digital vernetzte und hoch automatisierte Produktionsanlagen (unter dem Schlagwort *Industrie 4.0*), intelligente Städte (englisch *Smart Cities*), intelligente Stromnetze (englisch *Smart Grids*) oder das *Internet der Dinge* (englisch *Internet of Things*, kurz IoT). Die dabei eingesetzten digitalen Technologien werden aufgrund ihrer intelligenten Interaktion mit Mensch und Umwelt oft auch als *„smart“* bezeichnet. In den vielfältigen Anwendungen dieser „*smarten*“ Zukunftstechnologien liegt Schätzungen zufolge ein erhebliches globales Marktpotential, das auf ca. 900 Mrd. US-Dollar beziffert werden kann [10]. Auch im Hinblick auf gesellschaftliche Herausforderungen wie den globalen Klimawandel wird von einigen dieser Technologien für die Zukunft ein wesentlicher Beitrag hin zu mehr Nachhaltigkeit erhofft. So könnte z. B. durch die Digitalisierung des Mobilitätssektors mit Technologien wie vernetzter Mobilität oder intelligenter Verkehrsführung eine deutliche Reduktion der Treibhausgas-Emissionen erreicht werden [62]. Für die nahe und mittlere Zukunft ist daher von einem anhaltend starken Wachstum im Bereich der Digitalisierung und allen damit verbundenen Schlüsseltechnologien auszugehen. Die Forschung spielt dabei sowohl bei den Basistechnologien für digitale Hardware-/Software-Systeme als auch den zugehörigen Anwendungsfeldern eine maßgebliche Rolle als Treiber zukünftiger Innovationen.

Zu den wesentlichen Schlüsselfaktoren der Digitalisierung zählen gemäß [99] u. a. die eingesetzten *intelligenten Algorithmen*, einschließlich der *künstlichen Intelligenz*, sowie die globale Vernetzung von Geräten und Maschinen aller Art im *Internet der Dinge*. Gerade im Internet der Dinge findet die Digitalisierung nicht nur in Rechenzentren und den digitalen Endgeräten der Verbraucher statt,

sondern betrifft in zunehmendem Maße auch technische Systeme, die in ständiger Interaktion mit ihrer physikalischen Umwelt stehen. In solchen Anwendungsfeldern, z. B. dem autonomen Fahren, sind digitale Rechnersysteme meist in einen größeren technischen Kontext eingebettet, wo sie physikalische Prozesse überwachen, steuern und regeln. Man spricht bei solchen Computersystemen deshalb auch von *eingebetteten Systemen*. Eingebettete Systeme befinden sich häufig an der Schnittstelle zwischen der digitalen „Cyber"-Welt und der physikalischen Welt, weshalb sich hier auch der Begriff *Cyber-Physikalisches System* (CPS) etabliert hat (siehe z. B. [66]).

1.1.1. Cyber-Physikalische Systeme

Im Gegensatz zu Computersystemen in Rechenzentren oder im Endanwenderbereich bringt die Interaktion mit physikalischen Prozessen beim Entwurf der Hardware und Software von Cyber-Physikalischen Systemen eine ganze Reihe zusätzlicher Anforderungen mit sich (siehe z. B. [66]). Cyber-Physikalische Systeme befinden sich häufig in einem *sicherheitskritischen* Umfeld, wo eventuelle Fehlfunktionen der Hardware oder Software zu gravierenden Schäden führen oder gar Menschenleben gefährden können. Dies ist beispielsweise im Bereich des autonomen Fahrens der Fall, wo es durch Fehlfunktionen von (teil-)autonom fahrenden Fahrzeugen bereits zu Unfällen mit tragischem Ausgang kam. Ein prominentes Beispiel ist der Unfall eines autonomen Fahrzeugs der US-amerikanischen Firma Uber Inc. aus dem Jahr 2018, bei dem ein Fußgänger von den Algorithmen nicht korrekt erkannt wurde (siehe [60]). Autonom fahrende Fahrzeuge sind nur eines von vielen Anwendungsfeldern, in denen Cyber-Physikalische Systeme sicherheitskritische Aufgaben wahrnehmen. Weitere Beispiele finden sich u. a. in der Luftfahrt oder der Steuerung von Infrastruktureinrichtungen wie z. B. der Stromnetze, wo Fehlfunktionen der digitalen Steuerungssysteme ebenfalls gravierende Auswirkungen haben können. Um Unfälle zu vermeiden und die gesellschaftliche Akzeptanz intelligenter Systeme sicherzustellen, ist die Berücksichtigung von Sicherheitsaspekten bei der Entwicklung und Verifikation eines CPS unerlässlich. Dies wird umso bedeutender, wenn der Automatisierungsgrad und die Anzahl der Cyber-Physikalischen Systeme in sicherheitskritischen Bereichen im Zuge der Digitalisierung weiter steigen.

Grundsätzlich unterscheidet man bei der Bewertung der Sicherheit eines CPS zwischen funktionaler Sicherheit im Sinne der Vermeidung von Systemausfällen/Fehlfunktionen (englisch *Safety*) und der Absicherung gegen gezielte Cyber-Attacken (englisch *Security*). Während grundsätzlich beide Aspekte von entscheidender Bedeutung für den sicheren Betrieb eines CPS im kritischen Umfeld sind, wird im Rahmen dieser Arbeit nur die funktionale Sicherheit

(Safety) näher betrachtet. Ein wichtiger Bestandteil funktionaler Sicherheit und ein Hauptfokus dieser Arbeit ist die *Echtzeitfähigkeit* von Cyber-Physikalischen Systemen. Unter Echtzeitfähigkeit versteht man die Anforderung, dass ein CPS aufgrund der Interaktion mit physikalischen Prozessen stets in einem vorgegebenen Zeitrahmen reagieren können muss. Während es z. B. bei Unterhaltungs- und Verbraucherelektronik unkritisch ist, wenn der Benutzer etwas länger auf die Reaktion einer graphischen Benutzeroberfläche warten muss, kann eine verzögerte Reaktion bei sicherheitskritischen CPS gravierende Schäden verursachen. So könnte beispielsweise ein Airbag im Automobil zu spät ausgelöst werden oder ein durch digitale Regler stabilisiertes System aufgrund zu langer Reaktionszeiten instabil werden. Für sicherheitskritische CPS ist es daher unerlässlich, das Zeitverhalten der eingesetzten Kombination aus Software und Hardware detailliert zu analysieren, um Garantien hinsichtlich der Reaktionszeiten geben zu können. In vielen Anwendungsgebieten gelten entsprechende Sicherheitsstandards, die u. a. einen Nachweis der Echtzeitanforderungen vorschreiben. Ein Beispiel aus dem Bereich der Luftfahrt ist der DO-178C Standard, der die Grundlage für die Zertifizierung von sicherheitskritischen Software-Systemen durch die Luftfahrtbehörden bildet [74; 118].

1.1.2. Echtzeitanforderungen

Eine wichtige Größe für die Verifikation von Echtzeitanforderungen ist die maximale Ausführungszeit einer Software-Routine im ungünstigsten Ausführungsfall (englisch *Worst Case Execution Time*, kurz WCET). Die Herangehensweisen bei der Bestimmung bzw. Schätzung der WCET unterscheiden sich jedoch, je nachdem ob *harte* oder *weiche* Echtzeitsysteme untersucht werden [20]. Bei ersteren kann eine Überschreitung der maximalen Reaktionszeit in ihrer Auswirkung dem Versagen des Systems gleichkommen. Es muss daher i. A. bereits zur Entwurfszeit nachgewiesen werden, dass die WCET der relevanten Programmteile garantiert unterhalb der geforderten Reaktionszeit liegt. Bei weichen Echtzeitanforderungen muss das System hingegen nur im Mittel schnell genug reagieren. Vereinzelte Zeitüberschreitungen sind entweder unkritisch oder ihre Auswirkungen können durch geeignete Maßnahmen zur Laufzeit abgemildert werden.

Ein naheliegender Ansatz zur approximativen Bestimmung der WCET ist die gezielte Zeitmessung auf einem prototypischen System mit möglichst ungünstigen Eingabedaten. Durch hinreichend häufiges Wiederholen in verschiedenen Ausführungsszenarien kann dann der Versuch unternommen werden, einen möglichst ungünstigsten Ausführungsfall zu erzielen. Im Allgemeinen lässt sich jedoch

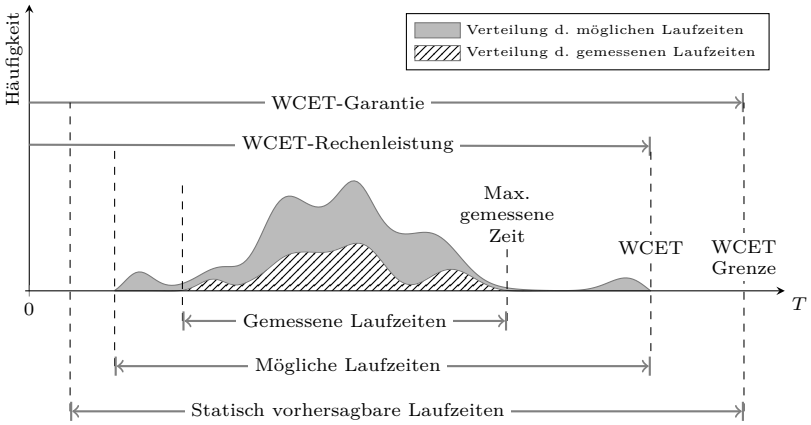


Abbildung 1.1.: Beispielhafte Verteilung der Laufzeit T eines Programms (vgl. [116])

nicht sicher nachweisen, dass bei einer solchen Suche auch tatsächlich der Fall mit der maximal möglichen Ausführungszeit gefunden wurde. Abbildung 1.1 veranschaulicht dies anhand zweier beispielhafter Häufigkeitsverteilungen für die gemessenen und die grundsätzlich möglichen Ausführungszeiten eines Programms (vgl. [116]). Die grau hinterlegte Kurve repräsentiert die tatsächliche Verteilung der Laufzeiten, wenn alle potentiell möglichen Ausführungsszenarien berücksichtigt werden. Bei komplexeren Programmen ist es in der Praxis jedoch nicht möglich, all diese Szenarien vollständig aufzulisten und durch Zeitmessungen zu bewerten. Im Allgemeinen kann durch Messungen deshalb nur eine kleine Teilmenge der Häufigkeitsverteilung (schraffierte Kurve in Abbildung 1.1) bestimmt werden, während die tatsächliche Verteilung unbekannt bleibt. Die maximal gemessene Laufzeit in einer Messreihe kann deshalb prinzipiell auch beliebig unterhalb der tatsächlichen WCET liegen. Aus diesem Grund ist die Methode der Zeitmessung meist nicht geeignet, um eine aussagekräftige WCET für *harte* Echtzeitsysteme zu ermitteln.

Die nötigen Garantien für harte Echtzeitsysteme lassen sich im Allgemeinen nur durch die statische Bestimmung eines sicheren oberen Grenzwertes für die Ausführungszeit gewährleisten. In der Praxis wird ein solcher Wert mit Hilfe eines WCET-Analysewerkzeugs anhand des kompilierten Maschinencodes ermittelt. Auch statische Analysen liefern jedoch nicht die tatsächliche WCET, sondern eine meist deutlich höhere obere Grenze (siehe WCET-Grenze in Abbildung 1.1). Ein Grund für diese Überschätzung ist oft das Fehlen von *a priori* Informationen

über den späteren Laufzeit-Zustand von Hardware und Software. So kann z. B. die Ausführungszeit eines Programmteils durch Hardware-Funktionen wie Caches oder adaptive Sprungvorhersagen von den zuvor ausgeführten Instruktionen abhängen. Um auf der sicheren Seite zu bleiben, muss eine statische WCET-Analyse in solchen Fällen pessimistisch vorgehen und im Zweifel z. B. annehmen, dass langsamere Hauptspeicherzugriffe nicht durch die Caches abgefangen werden können. Die zentrale Herausforderung bei der statischen WCET-Berechnung besteht deshalb darin, die pessimistische Überschätzung durch solche Annahmen zu minimieren und gleichzeitig garantiert oberhalb der realen WCET zu bleiben.

1.1.3. Mehrkernprozessoren

Neben den beschriebenen Sicherheitsaspekten führt die Digitalisierung auch in anderen Bereichen zu steigenden Anforderungen an zukünftige Generationen von eingebetteten Systemen. Mit dem zunehmenden Umfang an intelligenter Funktionalität sowie dem verstärkten Einsatz von künstlicher Intelligenz steigt auch der Bedarf an Rechenleistung signifikant an. Gleichzeitig müssen die Systeme in vielen Anwendungsbereichen energieeffizient arbeiten, um die Mikrochips z.B. noch mit akzeptablem Aufwand kühlen zu können und/oder die Laufzeit von batteriebetriebenen Geräten zu verbessern.

Während die Rechenleistung von Mikroprozessoren lange Zeit durch Anheben der Taktfrequenzen erhöht werden konnte, stößt dieses Vorgehen inzwischen an physikalische Grenzen. Da bei Schaltvorgängen innerhalb der Mikroprozessoren elektrische Kapazitäten umgeladen werden müssen, begrenzt die Dauer des Umladevorgangs die maximal erreichbare Schaltfrequenz (siehe z. B. [65]). Bis zu einem gewissen Grad kann die Reduktion der Kapazität durch Verkleinerung der Strukturgrößen oder die Beschleunigung der Umladevorgänge durch höhere Versorgungsspannungen Abhilfe schaffen. Für Ersteres ist jedoch absehbar, dass die derzeit in der Halbleiterproduktion eingesetzten Strukturgrößen von bereits unter 10 nm nicht mehr sehr viel stärker miniaturisiert werden können. Höhere Versorgungsspannungen steigern dagegen die Verlustleistung, sodass der Energieverbrauch und der zur Kühlung erforderliche Aufwand schnell zu limitierenden Faktoren werden. Um die nötige Rechenleistung dennoch bereitstellen zu können, setzen moderne Rechnerarchitekturen auf mehrere integrierte Prozessorkerne (im englischen auch *Central Processing Unit*, kurz CPU). Solche *Multiprozessorsysteme*, auch *Mehrkernprozessoren* genannt, werden oft durch weitere spezialisierte Beschleunigereinheiten wie Grafikprozessoren (englisch *Graphics Processing Units*, kurz GPUs) oder rekonfigurierbare Logik in Form von *Field Programmable Gate Arrays* (FPGAs) ergänzt. Hierdurch kann die Effizienz bestimmter Berechnungen weiter verbessert werden und man spricht

aufgrund der Heterogenität der Recheneinheiten in diesem Fall auch von *heterogenen Mehrkernprozessoren*. Ein großer Vorteil solcher Systeme besteht darin, dass sie ein deutlich verbessertes Verhältnis von Rechenleistung zu elektrischer Verlustleistung erreichen können. Ein entscheidender Nachteil ist jedoch, dass die Programmierung auf Software-Seite durch die Parallelität und Heterogenität ungleich komplexer ist als bei Systemen mit nur einem Kern. Sobald ein einzelner Kern nicht mehr ausreichend Rechenleistung bietet, ist deshalb ein erheblicher Mehraufwand in der Software-Entwicklung erforderlich. In vielen Bereichen spielt diese Problematik schon seit geraumer Zeit eine Rolle, was z. B. in dem vielzitierten Artikel von Herb Sutter *„The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software“* [107] aus dem Jahr 2005 zum Ausdruck kommt.

In zeitkritischen Anwendungen gewinnen Mehrkernprozessoren trotz des Bedarfs an Rechenleistung nur langsam an Bedeutung. Ein wesentlicher Grund dafür ist, dass die erwähnten Schwierigkeiten beim Einsatz mehrerer Prozessorkerne durch die zusätzlichen Echtzeitanforderungen nochmals deutlich verschärft werden. Die höhere Komplexität von Mehrkernprozessoren führt hier nicht nur zu einem steigenden Software-Entwicklungsaufwand, sondern auch zu weiteren Herausforderungen bei der Verifikation der Echtzeitfähigkeit. Hinzu kommt, dass sowohl typische parallele Programmiermodelle als auch die meisten gängigen Mehrkernrechnerarchitekturen nicht auf das Einhalten von garantierten Reaktionszeiten optimiert sind. Hier wird zumeist auf eine gute durchschnittliche Rechenleistung abgezielt, während die Vorhersagbarkeit des exakten Zeitverhaltens weniger relevant ist.

Ein grundsätzliches Problem für die zeitliche Vorhersagbarkeit von Mehrkern-Rechnerarchitekturen sind Hardware-Ressourcen wie Speicher oder Bussysteme, die von mehreren Kernen gemeinsam genutzt werden [53; 72]. Zeitgleiche Zugriffe von unterschiedlichen Kernen auf eine solche Ressource können von der Hardware meist nicht parallel abgearbeitet werden, sodass einer oder mehrere Zugriffe auf ihre Bearbeitung warten müssen. Im durchschnittlichen Fall wird typischerweise ein Großteil der Zugriffe durch Caches abgefangen, sodass die mittlere Auslastung der Speicherressourcen hinreichend niedrig bleibt. In Echtzeitsystemen ist jedoch der ungünstigste Fall ausschlaggebend, und dort kann es zu einer niedrigeren Effizienz der Caches und einer ungünstigen zeitlichen Verteilung der Zugriffe auf gemeinsam genutzte Ressourcen kommen. Für die Berechnung der WCET spielt die daraus resultierende gegenseitige Beeinflussung der Zugriffszeiten zwischen den Prozessorkernen – auch *Interferenz* genannt – deshalb eine erheblich größere Rolle. Ohne detaillierte Informationen über die Anzahl und zeitliche Verteilung der Zugriffe auf gemeinsam genutzte Ressourcen muss ein WCET-Analysewerkzeug hier pessimistische Annahmen treffen, die wiederum zu einer starken Überschätzung der tatsächlichen WCET führen können. Ansätze

zur Lösung dieses Problems sind Gegenstand aktueller Forschung (z. B. [78]) und es gibt im Gegensatz zu Einzelkernprozessoren noch keine kommerzielle Lösung zur statischen WCET-Analyse auf Mehrkernprozessoren.

Zusammenfassend lässt sich sagen, dass einzelne Prozessorkerne den zunehmenden Bedarf an Rechenleistung in eingebetteten Systemen bei gleichzeitig hoher Energieeffizienz voraussichtlich nicht mehr lange decken können. Dem flächendeckenden Einsatz von Mehrkernprozessoren in Cyber-Physikalischen Systemen stehen jedoch noch verschiedene Hindernisse entgegen. Die erhöhte Komplexität paralleler Software erschwert die Programmierung und kann die Produktivität bei der Software-Entwicklung drastisch reduzieren. Bei harten Echtzeitanwendungen fehlt es zudem an Analysewerkzeugen, die Interferenzeffekte berücksichtigen und präzise WCET-Grenzen für parallele Programme bestimmen können. Um den Anforderungen an zukünftige eingebettete Systeme gerecht zu werden, müssen die genannten Nachteile z.B. durch neuartige Software-Werkzeuge zur Entwurfsautomatisierung und zur Verifikation des Zeitverhaltens behoben werden.

1.2. Zielsetzung und Beiträge dieser Arbeit

Im Kontext der oben beschriebenen Problemstellungen besteht das Ziel dieser Arbeit darin, wesentliche Beiträge zur Entwurfsautomatisierung für parallele Echtzeit-Software beizusteuern. Dadurch soll der Entwicklungsaufwand idealerweise auf ein mit sequentieller Software vergleichbares Niveau gebracht werden. Der Schwerpunkt dieser Arbeit liegt vor allem auf der Entwicklung neuer Methoden und prototypischer Software-Werkzeuge im Bereich der *automatischen Software-Parallelisierung* für Mehrkernprozessoren in zeitkritischen Cyber-Physikalischen Systemen. Der Stand der Technik soll dabei um eine Methode zur automatisierten Transformation existierender sequentieller Echtzeitprogramme in eine parallele Entsprechung erweitert werden, ohne für das Eingabeprogramm ein spezialisiertes Programmiermodell vorauszusetzen. Software-Entwickler können sich damit weiterhin auf die Entwicklung von sequentieller Echtzeit-Software konzentrieren, während der aufwändige Prozess zur Anpassung des Programms an Mehrkern-Plattformen größtenteils durch ein Compiler-Werkzeug bewältigt werden kann. Um eine präzise statische WCET-Analyse zu ermöglichen, muss das generierte parallele Programm während der Transformation auf zeitliche Vorhersagbarkeit und niedrige Interferenz-Kosten optimiert werden. Aus diesem Grund können Ansätze, die nicht auf Echtzeitsysteme spezialisiert sind, nur bedingt auf die vorliegende Problemstellung übertragen werden. Für harte Echtzeit-Software sind deshalb neuartige Lösungsansätze erforderlich.

Im Gesamtkontext der automatischen Software-Parallelisierung zielt die vorliegende Arbeit insbesondere auf die folgenden Forschungsschwerpunkte ab:

1. Die Reduktion des Entwicklungsaufwands für parallele Echtzeitprogramme durch *Entwurfsautomatisierung*.
2. Die *feingranulare Parallelisierung allgemeiner Programme*, einschließlich iterativer Algorithmen.
3. Die *Compiler-gestützte Optimierung paralleler Echtzeitprogramme* mit dem Ziel, den WCET-Grenzwert des parallelen Programms zu minimieren.
4. Die Realisierung *plattformunabhängiger* Entwicklungswerkzeuge für harte Echtzeit-Software.
5. Die Schaffung der Grundlagen für eine *akkurate statische Analyse von Interferenzeffekten* ohne Interferenz-freie Hardware vorauszusetzen.

Nach bisherigem Stand der Literatur wurde noch keine Lösung publiziert, die all diese Aspekte adäquat adressiert. Dies zeigt sich u. a. in dem Kurzüberblick aus Tabelle 1.1 über den gegenwärtigen Stand der Technik im Bereich der WCET-optimierten automatischen Parallelisierung. Einige der darin gelisteten Ansätze vereinfachen das Problem, indem sie auf die Parallelisierung von Schleifen im Programm verzichten, sodass eine azyklische Darstellung des Eingabeprogramms verwendet werden kann. In Tabelle 1.1 sind diese Ansätze durch die fehlende Unterstützung für die Parallelisierung iterativer Programme gekennzeichnet. Zusätzlich wird meist auch Interferenz-Freiheit vorausgesetzt, wozu entweder spezialisierte Hardware oder eine Reduktion der Parallelität erforderlich ist. Das Kriterium der Plattformunabhängigkeit ist ebenfalls nur für einen Teil der existierenden Software-Werkzeuge erfüllt. Alle gelisteten Ansätze, die keiner der genannten Einschränkungen unterliegen, setzten hingegen nur auf teilautomatisierte Parallelisierung (niedriger Automatisierungsgrad in Tabelle 1.1). In diesen Fällen sind umfangreiche manuelle Arbeiten durch einen Software-Entwickler erforderlich, die wiederum die Produktivität bei der Software-Entwicklung beeinträchtigen. Insgesamt fehlt es damit an einem Ansatz, der einen hohen Automatisierungsgrad ohne die genannten Nachteile der übrigen Ansätze ermöglicht (Eine detailliertere Diskussion folgt in Kapitel 3). Die vorliegende Arbeit soll maßgeblich zur Schließung dieser Lücke im Stand der Technik beitragen.

Die im Rahmen dieser Arbeit entstandenen Beiträge sind wesentliche Bestandteile einer integrierten Werkzeugkette zur automatischen Parallelisierung und statischen WCET-Analyse für Echtzeit-Software auf Mehrkernprozessoren. Diese Werkzeugkette wurde im Rahmen des von der Europäischen Union (EU) über das

Ansatz	Automatisierung	Iterative Programme	Plattformunabhängig	Interferenzen erlaubt
Panić et al. [82]	Hoch	Nein	Ja	Nein
Matějka et al. [73]	Hoch	Nein	Nein	Nein
Pagetti et al. [81]	Hoch	Nein	Nein	Nein
Didier et al. [24]	Hoch	Nein	Nein	Ja
Ungerer, Jahr, Frieb et al. [113; 48; 33]	Niedrig	Ja	Ja	Ja
Stegmeier et al. [103]	Niedrig	Ja	Ja	Ja

Tabelle 1.1.: Kurzüberblick zum Stand der Technik im Bereich der automatisierten und teilautomatisierten Parallelisierung von Echtzeit-Software

„Horizon 2020"-Programm[1] geförderten Projekts „ARGO" entwickelt (siehe z. B. [DPA+17]). Ihr Hauptziel ist die weitgehende Automatisierung aller Schritte, die bei der Erzeugung von Echtzeitanwendungen für Mehrkernprozessoren zusätzlich zu den herkömmlichen Entwicklungsprozessen für Einzelkern-Software anfallen. Das primäre Zielkriterium besteht dabei in der Minimierung der durch statische Analysen ermittelten WCET-Grenze für das erzeugte parallele Programm. Als Nebenkriterium soll auch die Leistungsfähigkeit im durchschnittlichen Fall nicht übermäßig beeinträchtigt werden.

In diesem Gesamtkontext fokussiert sich die vorliegende Arbeit auf die *Transformation von sequentiellem zu parallelem Programmcode*, die *Compiler-basierte Optimierung paralleler Programme* hinsichtlich Kommunikation, Synchronisation und Speichernutzung sowie die *generische Modellierung des Zeitverhaltens in Mehrkernprozessoren*. Sie stellt damit Lösungen für einige entscheidende Teilprobleme bei der automatischen Parallelisierung und Optimierung von Echtzeitprogrammen für Mehrkernprozessoren bereit. Die konkreten Beiträge dieser Arbeit beinhalten dabei insbesondere:

- Ein *Compiler-Werkzeug zur WCET-optimierten parallelen Code-Generierung* aus sequentiellen Eingabeprogrammen.
- Ein *paralleles Programmiermodell für harte Echtzeitanwendungen* auf Mehrkernprozessoren, die nicht frei von Interferenzen sind.
- Methoden zur *Optimierung von Kommunikation und Synchronisation* für hybride Kommunikationsmodelle mit Nachrichtenübertragung und gemeinsam genutzten Speichern.

[1] `https://ec.europa.eu/programmes/horizon2020/`

- Ein WCET-basiertes *Optimierungsverfahren zur effizienten Speicherallokation* in komplexen verteilten Speicherhierarchien.
- Ein *generisches Modell für das Zeitverhalten von Mehrkernprozessoren* kombiniert mit einer erweiterten *Architekturbeschreibungssprache* (englisch *Architecture Description Language*, kurz ADL).

Zusammengenommen ermöglichen diese Beiträge eine WCET-optimierte Transformation von sequentiellen Echtzeitprogrammen auf Basis der Programmiersprache „C“ [46] hin zu einer parallelen Entsprechung. Um die Effizienz und statische Vorhersagbarkeit zu erhöhen, werden dabei sowohl verteilte als auch gemeinsam genutzte Speicher unterstützt. Die Optimierung von Speicherallokation und Synchronisationsstruktur eröffnet zudem weiteres Potential für die Minimierung der WCET des erzeugten parallelen Codes. Die ADL in Verbindung mit den zugehörigen Hardware-Modellen stellen dabei sowohl WCET-Metriken für die Optimierungsverfahren als auch detaillierte Informationen für eine statische WCET-Analyse bereit. Auf diese Weise wird eine weitgehend plattformunabhängige Implementierung sowohl der Entwicklungswerkzeuge als auch der WCET-Analyse für Mehrkernprozessoren ermöglicht.

1.3. Aufbau der Arbeit

Der Aufbau dieser Arbeit gliedert sich in neun Kapitel. Im Anschluss an dieses einleitende Kapitel gibt Kapitel 2 einen Überblick über die wesentlichen Grundlagen für das Verständnis der Arbeit. Das schließt insbesondere die Bereiche der Rechnerarchitekturen für Mehrkernprozessoren, der parallelen Programmierung, des Aufbaus von Compiler-Werkzeugen und der darin verwendeten Zwischendarstellungen, sowie einen kurzen Überblick zu statischen WCET-Analysen und Echtzeitbetriebssystemen ein. Das Kapitel führt dabei auch die in dieser Arbeit verwendete Terminologie sowie einige grundlegende Definitionen ein, auf die in den übrigen Kapiteln zurückgegriffen wird.

Das darauffolgende Kapitel 3 gibt einen Überblick über den aktuellen Stand der Technik rund um den Einsatz von Mehrkernprozessoren in Echtzeit-Systemen. Der Schwerpunkt liegt dabei auf Software-Werkzeugen für den Entwurf und insbesondere die automatische Parallelisierung von Echtzeit-Software. Eine für das Verständnis dieser Arbeit besondere Rolle im Stand der Technik nimmt die bereits erwähnte ARGO-Werkzeugkette ein, die den größeren Kontext der Beiträge dieser Arbeit bildet. Die Vorarbeiten aus dem ARGO-Projekt werden dementsprechend in einem etwas ausführlicheren Abschnitt behandelt.

Die Kapitel 4 bis 7 sind der detaillierten Beschreibung der Beiträge dieser Arbeit gewidmet. Den Anfang macht Kapitel 4, in dessen erstem Teil ein statisch

analysierbares Programmiermodell für die automatisch generierten parallelen Echtzeitprogramme definiert wird. Das Programmiermodell bildet die Grundlage für die Compiler-Transformation zur Erzeugung von parallelisiertem C-Code, deren Aufbau im zweiten Teil von Kapitel 4 vorgestellt wird. Die Compiler-Transformation enthält zwei größere Teilschritte zur feingranularen Optimierung des zu erzeugenden Programms, die in Kapitel 4 zunächst eingeführt und hinsichtlich der darin verwendeten Optimierungsmethoden später jeweils in einem eigenen Kapitel im Detail behandelt werden.

Kapitel 5 stellt, ergänzend zu dem Compiler-Werkzeug aus Kapitel 4, ein generisches Plattformmodell für statisch vorhersagbare Mehrkernprozessoren vor. Dieses Modell ermöglicht es, die genannten Compiler-Werkzeuge plattformunabhängig zu gestalten und insbesondere auch das Zeitverhalten der Hardware in Gegenwart von Interferenzeffekten sicher nach oben abschätzen zu können. Es schafft damit eine Grundlage, mit deren Hilfe statische WCET-Analysen für ein breites Spektrum von Mehrkernarchitekturen realisiert werden können. Zur strukturierten Beschreibung von Mehrkernarchitekturen auf Basis des Plattformmodells wird außerdem eine ADL-Erweiterung vorgestellt, die ein breites Spektrum von statisch vorhersagbaren Hardware-Plattformen abbilden kann.

Kapitel 6 befasst sich im Detail mit der Optimierung von Kommunikation und Synchronisation bei der Erzeugung parallelisierter Programme im Rahmen der vorgestellten Compiler-Transformation. Dabei wird u. a. eine Methode vorgestellt, mit deren Hilfe Kommunikations- und Synchronisationsoperationen auf Quellcode-Ebene platziert und redundante Operationen zusammengefasst werden können.

Als weitere Möglichkeit, die erzeugten parallelen Programme zu optimieren, stellt Kapitel 7 eine Methode zur Verwaltung und Allokation von Speichern in Mehrkernprozessoren mit verteilten und heterogenen Speicherhierarchien vor. Dabei liegt der Fokus und eine wesentliche Neuerung im Vergleich zum Stand der Technik unter anderem darin, dass die Auswirkungen von Interferenzeffekten auf die WCET bei der Optimierung explizit modelliert und berücksichtigt werden.

Das darauffolgende Kapitel 8 widmet sich der experimentellen Evaluation und Bewertung der Beiträge dieser Arbeit. Dabei wird zum einen die Anwendbarkeit der vorgestellten Compiler-Transformation auf verschiedene Testprogramme und Zielplattformen demonstriert. Zum anderen werden die in den Kapiteln 6 und 7 eingeführten Optimierungsverfahren hinsichtlich ihrer Effizienz genauer untersucht und bewertet.

Das abschließende Kapitel 9 fasst die Ergebnisse und Schlussfolgerungen der vorliegenden Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Erweiterungen und weiterführende Forschungsarbeiten.

Kapitel 2.

Grundlagen

Dieses Kapitel behandelt die wesentlichen Grundlagen, die zum Verständnis dieser Arbeit benötigt werden. Dabei werden u. a. die grundlegende Terminologie und einige Definitionen, die in den späteren Kapiteln verwendet werden, eingeführt. Das Kapitel gliedert sich in vier Abschnitte, die sich jeweils einem größeren Themengebiet widmen. Dazu gehört zunächst Abschnitt 2.1 mit einer Übersicht über die wichtigsten Aspekte der Rechnerarchitekturen für Mehrkernprozessoren. Darauf folgen in Abschnitt 2.2 einige Grundlagen zur parallelen Programmierung und in Abschnitt 2.3 eine Einführung in Compiler-Werkzeuge und die im Rahmen dieser Arbeit verwendeten Zwischendarstellungen für den Programmcode. Abschnitt 2.4 gibt schließlich einen kurzen Überblick über die Methoden der WCET-Analyse und die Ausführung von Echtzeit-Software auf zeitkritischen Systemen.

2.1. Rechnerarchitekturen für Mehrkernprozessoren

Die Architektur eines Rechnersystems beschreibt dessen grundsätzlichen Aufbau, einschließlich des Befehlssatzes, der funktionalen Organisation, des Logikentwurfs und der Implementierung [42]. Aus Sicht der Software ist zunächst vor allem die Architektur des Befehlssatzes (englisch *Instruction Set Architecture*, kurz ISA) von Bedeutung, während andere Aspekte (wie z. B. Cache-Speicher) für den Programmierer oft nicht direkt sichtbar sind. Sobald jedoch das Zeitverhalten der Software eine Rolle spielt, müssen zahlreiche weitere Details der Rechnerarchitektur berücksichtigt werden. Dies gilt insbesondere für Mehrkernprozessoren, deren Architektur naturgemäß komplexer ist als bei Systemen mit nur einem Prozessorkern. Im Bereich eingebetteter Systeme werden solche Rechnerarchitekturen häufig als sogenannte *Systems on Chip* (englisch für *Ein-Chip-System*, kurz SoC) realisiert, die sich durch die Integration aller relevanten Komponenten auf einem einzelnen Chip auszeichnen [80]. Enthält ein SoC mehrere Prozessorkerne,

so spricht man auch von einem *Multi-processor System on Chip* (englisch für *Ein-Chip-System mit mehreren Prozessoren*, kurz MPSoC).

Sowohl für die Rechenleistung als auch die parallelen Programmierparadigmen auf Mehrkernprozessoren spielt der Aufbau der Speicherhierarchie eine entscheidende Rolle. Die Realisierung schneller Speicher, z.B. auf Basis von statischem RAM (statischer Direktzugriffsspeicher, englisch *Static Random-Access Memory*, kurz SRAM), ist typischerweise teurer und benötigt mehr Chip-Fläche als langsamere Speicher in Form von dynamischem RAM (dynamischer Direktzugriffsspeicher, englisch *Dynamic Random-Access Memory*, kurz DRAM). So besteht eine DRAM Zelle aus nur einem Transistor und einem Kondensator, während für eine SRAM Zelle typischerweise sechs Transistoren benötigt werden. Um in diesem Zielkonflikt einen Kompromiss herzustellen, besitzen moderne Rechnerarchitekturen meist eine komplexe Speicherhierarchie bestehend aus schnellen aber vergleichsweise kleinen Zwischenspeichern und großen aber langsameren Hauptspeichern. Die klassische Hierarchie für Einzelkernprozessoren beinhaltet einen großen Hauptspeicher und mehrere Stufen aus schnelleren Cache-Speichern. Letztere puffern die kürzlich vom Prozessor verwendeten Daten aus dem Hauptspeicher, um sie bei potentiellen späteren Zugriffen mit geringerer Latenz zur Verfügung stellen zu können. Die Caches sind für die Software transparent, da sämtliche Verwaltungsaufgaben, einschließlich der Entscheidung welche Daten im Cache verbleiben und welche in den Hauptspeicher zurückgeschrieben werden, durch die Hardware übernommen werden. Um selten verwendete Daten im Cache durch andere ersetzen zu können, gibt es eine Reihe gängiger Cache-Verdrängungsstrategien. Eine häufig verwendete Strategie besteht z. B. darin, stets die am längsten nicht genutzten Daten im Cache zu verdrängen. Diese Strategie wird im englischen auch als *„least recently used“* (LRU) bezeichnet.

Im Hinblick auf Mehrkernprozessoren gibt es grundsätzlich zwei unterschiedliche Ansätze für den Aufbau der Speicherhierarchie (siehe z. B. [42]). Zum einen gibt es das Modell des *gemeinsamen Hauptspeichers* (englisch *Shared Memory*, kurz SHM), in dem alle Kerne gleichberechtigten Zugriff auf einen gemeinsamen Speicher haben. Zum anderen nutzen insbesondere größere Mehrkernprozessoren auch *verteilte Speicher* (englisch *Distributed Memory*), auf die nicht alle Kerne gleichermaßen Zugriff haben. In solchen Architekturen kann der Zugriff auf eine Speicherkomponente z. B. auf bestimmte Kerne beschränkt sein und/oder die Zugriffszeiten können sich je nach Prozessorkern signifikant unterscheiden.

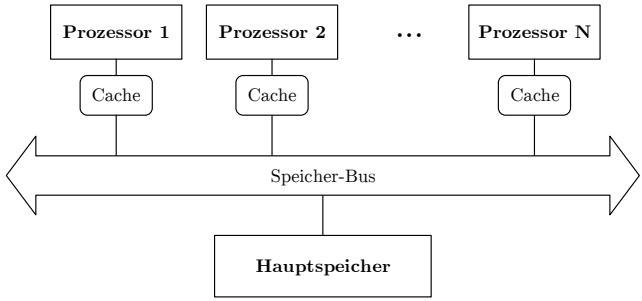


Abbildung 2.1.: Einfache Mehrkernprozessorarchitektur mit gemeinsam genutztem Hauptspeicher (vgl. [42])

2.1.1. Architekturen mit gemeinsam genutzten Speichern

Abbildung 2.1 zeigt den grundlegenden Aufbau einer einfachen Mehrkernprozessorarchitektur mit gemeinsamem Hauptspeicher. Die Adressierung des Hauptspeichers ist oft symmetrisch ausgelegt, sodass dieselbe Adresse auf jedem der Kerne auf dieselbe Stelle im Speicher verweist. In diesem Fall spricht man auch von einem *symmetrischen* Mehrkernprozessor (englisch *Symmetric Multiprocessor*, kurz SMP) [42]. Das SMP-Prinzip erleichtert die Programmierung mitunter erheblich, da es z.B. einem Betriebssystem erlaubt, Softwareprozesse dynamisch zur Laufzeit auf andere Kerne zu verschieben, ohne besondere Rücksicht auf die Speicherhierarchie nehmen zu müssen.

Ausgehend von den Kernen beginnt die Speicherhierarchie typischerweise mit den Caches der ersten Stufe (auch als L1-Caches bezeichnet), die jeweils einem Prozessorkern exklusiv zugeordnet sind. Solche privaten Caches können sehr performant an den Prozessorkern angebunden werden, haben jedoch den Nachteil, dass Zusatzaufwand zur Erhaltung der sogenannten *Cache-Kohärenz* erforderlich ist. Ohne geeignete Maßnahmen zur Erhaltung der Kohärenz kann z.B. Prozessorkern 1 aus Abbildung 2.1 die Daten im Hauptspeicher verändern, während sie gleichzeitig noch im privaten Cache von Prozessorkern 2 vorgehalten werden. Kern 2 würde die vorgenommenen Änderungen in diesem Fall nicht bemerken, da der Cache noch eine veraltete Version der Daten enthält. Das Problem lässt sich hardwareseitig durch Kohärenzprotokolle lösen, die sicherstellen, dass geänderte Daten auch in den lokalen Caches anderer Kerne aktualisiert werden. Bei einer moderaten Anzahl von Kernen lässt sich ein Kohärenzprotokoll noch mit überschaubarem Aufwand implementieren. Die meisten Verfahren zum

Erhalt der Cache-Kohärenz skalieren jedoch nur schlecht mit der Anzahl der Prozessorkerne. Eine Alternative stellt Software-gesteuerte Kohärenz dar, bei der das Verwerfen (englisch *invalidation*) bzw. das Zurückschreiben (englisch *write-back*) von Cache-Inhalten manuell durch die Software ausgelöst werden kann. Es liegt dann in der Verantwortung der Software, die Caches so anzusteuern, dass beim Datenaustausch keine Probleme mit der Cache-Kohärenz auftreten.

2.1.2. Architekturen mit verteilten Speichern

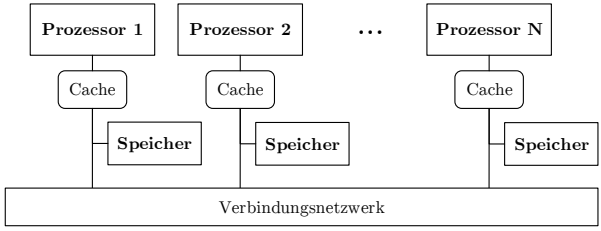


Abbildung 2.2.: Einfache Mehrkernprozessorarchitektur mit verteiltem Speicher (vgl. [42])

Architekturen mit verteilten Speichern werden häufig bei Systemen mit einer großen Zahl von Rechenkernen eingesetzt, um den Engpass eines zentralen Hauptspeichers und die eingeschränkte Skalierbarkeit von Cache-Kohärenzprotokollen zu umgehen. Ein Beispiel für eine Rechnerarchitektur mit verteiltem Speicher ist in Abbildung 2.2 dargestellt. Solche Architekturen enthalten mehrere, im System verteilte Speicher, die lokal an die Prozessorkerne angebunden sind. Speicherzugriffe auf die lokalen Speicher müssen deshalb nicht über das gemeinsame Verbindungsnetzwerk abgewickelt werden, sodass Engpässe vermieden und die Latenzen der Speicheranbindungen verbessert werden können. Der Datenaustausch zwischen den Kernen ist in diesem Aufbau jedoch nicht mehr implizit durch den gemeinsam genutzten Speicher gegeben. Datenübertragungen zwischen den einzelnen verteilten Speichern müssen daher explizit erfolgen, z. B. mittels Nachrichten-basierter Kommunikation. Um diesen Datenaustausch effizienter zu gestalten, verfügen viele Architekturen über sogenannte *Direct Memory Access* (DMA) Einheiten, die die Nutzdaten in andere Speichersegmente kopieren, ohne dabei einen der Prozessorkerne in Anspruch nehmen zu müssen.

Neben der Reinform einer Architektur mit verteiltem Speicher gibt es auch verschiedene Mischformen, in denen der Speicher zwar verteilt ist, aber dennoch (zumindest teilweise) von den Rechenkernen gemeinsam genutzt werden kann. Zu diesen Mischformen gehört das Konzept des verteilten gemeinsamen Speichers (englisch *Distributed Shared Memory*), bei dem die Speicher zwar im System verteilt sind, die Adressierung jedoch für jeden Kern identisch ist. Dadurch wird die einfachere Programmierbarkeit von Architekturen mit gemeinsamen Speichern mit der Effizienz und besseren Skalierbarkeit verteilter Speicher kombiniert. Das Problem der Cache-Kohärenz bleibt in solchen Architekturen jedoch grundsätzlich bestehen. Auch wenn sich der Adressraum für die Software einheitlich darstellt, muss aus Effizienzgründen außerdem beachtet werden, dass die Zugriffszeiten abhängig von dem jeweiligen Speichersegment stark variieren können. Man spricht aus diesem Grund auch von sogenannten *Non-Uniform Memory Access* (NUMA) Architekturen.

2.1.3. Chip-interne Kommunikation in Mehrkernprozessoren

Mit einer steigenden Zahl von Prozessorkernen werden auch aufwändigere Kommunikationsnetzwerke zum Datenaustausch zwischen den Prozessorkernen, Speichern und der Peripherie benötigt (siehe [83]). Eine sehr einfache Form der Kommunikationsverbindung stellt der zentrale Speicher-Bus aus Abbildung 2.1 dar. Alle Busteilnehmer kommunizieren dort über einen Satz gemeinsamer Übertragungsleitungen, wobei zu jedem Zeitpunkt nur ein Teilnehmer Daten senden darf. Aus naheliegenden Gründen muss mit steigender Zahl aktiver Busteilnehmer im Mittel länger auf das freiwerden des Busses gewartet werden. Dies begrenzt die Anzahl an Prozessorkernen, die effizient an einen gemeinsam genutzten Bus angebunden werden können. Für große Mehrkernprozessoren, oft auch als *Vielkernprozessoren* (englisch *Many-Core Processor*) bezeichnet, wird daher auf Alternativen wie Bus-Hierarchien oder Chip-interne Netzwerke (englisch *Network-on-Chip*, kurz NoC) zurückgegriffen.

In den Abbildungen 2.3 und 2.4 sind einfache Beispiele für Bus-Hierarchien bzw. Networks-on-Chip dargestellt (vgl. [83; 63]). In Bus-Hierarchien müssen nicht mehr alle Zugriffe über einen globalen Bus abgewickelt werden, sondern sie durchlaufen nur noch die Bus-Systeme der betroffenen Hierarchieebenen. Durch den hierarchischen Aufbau stellt die Topologie einen Baum dar, weshalb es für jedes Paar aus miteinander kommunizierenden Komponenten genau einen festgelegten Kommunikationspfad durch die Hierarchie gibt. Im Gegensatz dazu können die Daten ihr Ziel in einem NoC grundsätzlich über unterschiedliche Pfade und Netzwerk-Komponenten erreichen. Das gilt auch auf das NoC aus Abbildung 2.4, das die am häufigsten eingesetzte Topologie eines zweidimensionalen Gitters

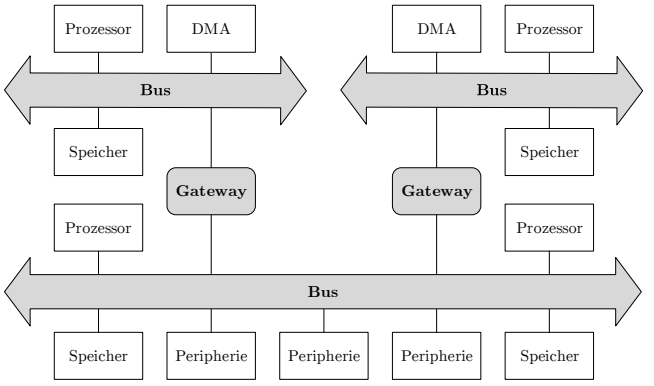


Abbildung 2.3.: Beispiel für eine Bus-Hierarchie (vgl. [83])

aufweist. Darin sind jeweils einer oder mehrere Prozessorkerne und deren lokale Speicher zu einer sogenannten *Kachel* zusammengefasst, die wiederum über einen Netzwerk Adapter (NA) an den lokalen NoC-Router (R in der Abbildung) angebunden ist. Die Router haben in dieser Topologie bis zu fünf vollduplexe Datenverbindungen, über welche sie mit bis zu vier Nachbar-Routern sowie einer lokalen Kachel verbunden sind. Größere gemeinsam nutzbare Speicher oder Peripherie können z. B. in Form von spezialisierten Kacheln (z. B. Kachel 1,1 in der Abbildung) in das NoC eingebunden werden.

Verglichen mit Bus-Hierarchien lässt sich die Kommunikationslast in NoCs grundsätzlich besser auf die verschiedenen Verbindungsleitungen verteilen. Mit den Routern müssen der Architektur jedoch auch vergleichsweise aufwändige Hardwareeinheiten hinzugefügt werden. Zwei wesentliche Aufgaben der Router sind in diesem Zusammenhang die *Routenplanung* und die *Datenvermittlung*:

Routenplanung: Um die Komplexität der Router überschaubar zu halten, werden meist einfache Strategien wie das sogenannte *XY-Routing* für die Routenplanung verwendet. Beim XY-Routing werden die Kacheln und die zugehörigen Router mit zweidimensionalen Koordinaten (x, y) versehen (vgl. Nummerierung der Kacheln in Abbildung 2.4). Daten mit einer Ursprungskachel (x_u, y_u) und einer Zielkachel (x_z, y_z) werden dann zunächst in X-Richtung bis zu den Router-Koordinaten (x_z, y_u) übertragen, bevor sie in Y-Richtung bis zur Zielkachel gelangen.

Datenvermittlung: Um den Datenfluss über die Router zu organisieren, gibt es für NoCs zwei gängige Klassen von Vermittlungsstrategien (englisch

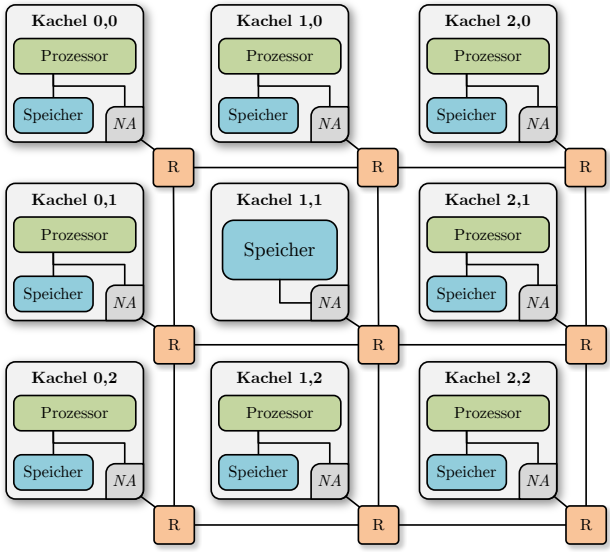


Abbildung 2.4.: Beispiel für eine NoC-Architektur (vgl. [63])

Switching Strategies) [83]. Das sind zum einen die *Leitungsvermittlung* (englisch *Circuit Switching*) und zum anderen die *Paketvermittlung* (englisch *Packet Switching*). Bei ersterer wird vor der eigentlichen Datenübertragung im Netzwerk ein Verbindungspfad, auch *Circuit* genannt, von der Quelle zum Ziel eingerichtet. Entlang dieses Verbindungspfads können dann so lange Daten übertragen werden, bis er wieder freigegeben wird. Bei der Paketvermittlung werden die Daten dagegen in Pakete unterteilt, die dann unabhängig voneinander durch das NoC geleitet werden. Eine Mischform stellt die *virtuelle Leitungsvermittlung* (englisch *Virtual Circuit Switching*) dar, bei der virtuelle Circuits in einem Paketvermittlungsnetzwerk angelegt werden können. Ein genereller Vorteil von Circuits ist, dass nach dem Verbindungsaufbau alle Daten über den gleichen Pfad im NoC übertragen werden. Die Router können Ressourcen, wie z. B. Pufferspeicher, beim Aufbau des Circuits reservieren, sodass sie während der Datenübertragung garantiert zur Verfügung stehen. Bei der Paketvermittlung müssen die Ressourcen dagegen für jedes Paket einzeln alloziert werden, was zu Schwankungen in der Übertragungsdauer und abweichenden Routen bei Paketen mit denselben Quellen und Zielen führen kann. Dieser Nachteil

steht einer größeren Flexibilität und dem Wegfall des Zusatzaufwands für das Anlegen und Freigeben von Circuits gegenüber.

2.1.3.1. Arbitrierung

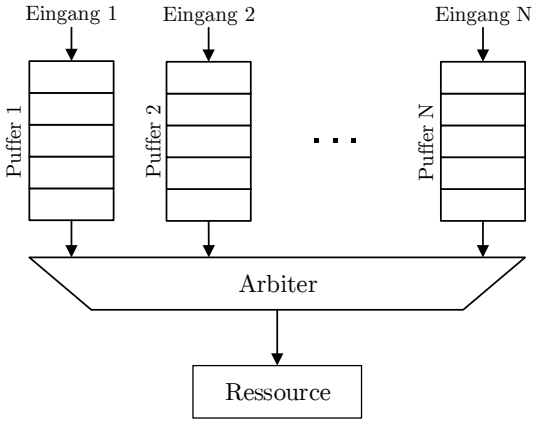


Abbildung 2.5.: Typische Hardware-Struktur zur Arbitrierung einer gemeinsam genutzten Ressource

Bus-Hierarchien und NoCs können die Kommunikationslast zwar bis zu einem gewissen Grad auf verschiedene Komponenten verteilen, es besteht jedoch weiterhin die Möglichkeit, dass unabhängige Kommunikationspfade über dieselbe Verbindung verlaufen. Bei parallelen Zugriffen muss auch hier die zeitliche Reihenfolge für den Zugriff auf die physikalischen Übertragungsleitungen geregelt werden. Dieser Vorgang wird auch als *Arbitrierung* bezeichnet. Die Arbitrierung spielt in Mehrkernprozessoren eine entscheidende Rolle für die Analyse des Zeitverhaltens in Verbindung mit dem eingangs beschriebene Problem der zeitlichen Interferenz zwischen den Prozessorkernen [53]. Abbildung 2.5 zeigt die typische Struktur zur Arbitrierung von Zugriffen auf eine gemeinsam genutzte Ressource. Die eingehenden Daten werden dort in Einheiten mit einer festen Größe unterteilt und nach dem „First In – First Out“-Prinzip (FIFO) in einem Puffer zwischengespeichert. Die FIFO-Puffer fungieren als Warteschlangen, aus denen die zuerst hinzugefügten Elemente auch als erste wieder entnommen werden. Der nachfolgende Arbiter entscheidet dann, wann die Elemente welcher Warteschlange an die gemeinsam genutzte Ressource weitergegeben werden. Im

einfachsten Fall ist die Länge der Eingangspuffer gleich eins, sodass sie einem einfachen Register entsprechen.

Der Datendurchsatz und die Latenz für Kommunikation zwischen den Kernen bei mehreren konkurrierenden Übertragungen hängt entscheidend von dem jeweils realisierten Arbitrierungsschema ab. Zwei für Echtzeitsysteme geeignete Arbitrierungsverfahren sind das *Zeitmultiplexverfahren* (englisch *Time Division Multiple Access*, kurz TDMA) und das *Rundlaufverfahren* (englisch *Round Robin*, kurz RR). Beide Schemata haben die für Echtzeitsysteme wichtige Eigenschaft, dass eine Datenübertragung nicht durch konkurrierenden Übertragungen völlig blockiert werden kann. Ohne diese Eigenschaft könnte es im Extremfall zu einer unendlichen WCET kommen.

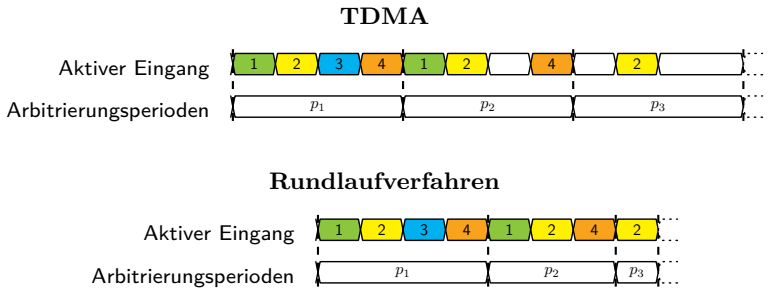


Abbildung 2.6.: Beispielhafter Zeitablauf am Ausgang des Arbiters bei vier Eingängen mit TDMA bzw. Round Robin Arbitrierung

Sowohl TDMA als auch RR verwenden feste Zeitschlitze, innerhalb derer jeweils eine Dateneinheit aus einer der Warteschlangen an die gemeinsam genutzte Ressource (z. B. eine NoC-Verbindungsleitung) weitergegeben wird. Abbildung 2.6 zeigt jeweils einen möglichen Zeitablauf für den Zugriff auf eine Ressource mit vier Arbiter-Eingängen. Die Eingänge erhalten innerhalb einer Arbitrierungsperiode p reihum Zugriff auf die Ressource, sodass sich die zur Verfügung stehende Bandbreite gleichmäßig auf die einzelnen Eingänge verteilt. Bei vollständiger Auslastung aller Eingänge unterscheiden sich RR und TDMA zunächst nicht (z. B. in Periode p_1 aus der Abbildung). Einen Unterschied gibt es erst dann, wenn einer oder mehrere Eingangspuffer leer sind. Im Fall von RR wird der Zeitschlitz der aktuell inaktiven Eingänge übersprungen, während er bei TDMA ungenutzt bleibt. Die Länge der Arbitrierungsperioden bleibt dadurch bei TDMA konstant, während sie sich bei RR dynamisch verkürzen kann. In dem Beispiel aus Abbildung 2.6 ist dies für die Perioden p_2 und p_3 der Fall, wo für einige

Eingänge keine Daten mehr im Puffer sind. Aufgrund der festen Arbitrierungsperioden ist TDMA deterministisch und bietet einen konstanten Datendurchsatz für jeden Eingang. Es können folglich keine zeitlichen Interferenzen zwischen den Eingangsdatenströmen auftreten. Das RR-Schema stellt den Eingängen dagegen, je nach Gesamtauslastung, einen variablen Durchsatz zur Verfügung, sodass die Bandbreite der Ressource i. A. effizienter genutzt werden kann. Im ungünstigsten Fall garantiert RR jedoch nach wie vor einen Mindestdurchsatz, der dem von TDMA entspricht.

Sowohl bei TDMA als auch bei RR handelt es sich um *faire* Arbitrierungsverfahren. Im Kontext dieser Arbeit wird dabei die folgende Definition für faire Arbitrierung zugrunde gelegt:

Definition 2.1 (Faire Arbitrierung)**.** *Ein Arbitrierungsverfahren heißt fair, wenn die verfügbare Bandbreite gleichmäßig auf alle nicht-leeren Eingangspuffer eines Arbiters (gemäß Abbildung 2.5) verteilt wird.*

Diese Definition impliziert, dass eine Dateneinheit an erster Position in der Warteschlange eines Eingangspuffers auf maximal eine Dateneinheit jedes anderen Eingangspuffers warten muss, bevor sie an die Ressource weitergeleitet wird. Bei fairer Arbitrierung kann die Weitergabe einer Dateneinheit aus Eingang 1 dementsprechend nicht durch zwei aufeinanderfolgende Dateneinheiten aus Puffer 2 verzögert werden. Andernfalls wäre die Bandbreite für Eingang 2 zumindest kurzzeitig höher als die von Eingang 1.

Zur Bestimmung einer WCET müssen der Durchsatz DS sowie die maximale Latenz L für alle relevanten Kommunikationsverbindungen und Arbitrierungseinheiten bekannt sein. Für diese beiden Größen sollen im Rahmen dieser Arbeit die folgenden Definitionen zugrunde gelegt werden:

Definition 2.2 (Durchsatz)**.** *Der Durchsatz DS einer Kommunikationsverbindung oder Hardware-Komponente bezeichnet die Anzahl an Dateneinheiten, die pro Zeiteinheit übertragen werden können.*

Definition 2.3 (Latenz)**.** *Die Latenz L einer Kommunikationsverbindung oder Hardware-Komponente bezeichnet die Zeitspanne, die ein Datum zum Durchlaufen der Komponente von einem definierten Eingang bis zu einem definierten Ausgang benötigt.*

Ausgehend von dem Basisdurchsatz DS^{res} der Ressource kann der Durchsatz DS^{TDMA} bzw. $DS^{RR}(p_i)$ pro Eingang des Arbiters für das Zeitmultiplexverfahren bzw. das Rundlaufverfahren bestimmt werden. Mit Hilfe der Anzahl der in einer

Arbitrierungsperiode p_i verwendeten Eingänge $n^{aktiv}(p_i)$ sowie der Gesamtzahl der Eingänge n^{gesamt} ergeben sich dafür die folgenden Zusammenhänge:

$$DS^{TDMA} = \frac{DS^{res}}{n^{gesamt}}, \tag{2.1}$$

$$DS^{RR}(p_i) = \frac{DS^{res}}{n^{aktiv}(p_i)}. \tag{2.2}$$

Dies folgt unmittelbar daraus, dass die verfügbare Bandbreite während einer Arbitrierungsperiode gleichmäßig verteilt wird. Zur Berechnung der Latenz wird im Folgenden angenommen, dass eine Dateneinheit vor Beginn des zugehörigen Zeitschlitzes in dem Eingangspuffer angekommen sein muss, um noch in derselben Arbitrierungsperiode berücksichtigt werden zu können. Kommt die Dateneinheit gleichzeitig mit dem Beginn des Zeitschlitzes an, so kann sie den Arbiter erst in der darauffolgenden Periode passieren. Die maximale Latenz kann dann durch die Dauer einer Arbitrierungsperiode nach oben abgeschätzt werden. Da die Perioden bei RR veränderlich sind, muss hier das Maximum der Dauer der Ankunftsperiode p_i und der Dauer der darauffolgenden Übertragungsperiode p_{i+1} verwendet werden. Die maximale Latenz L^{TDMA} bzw. L^{RR} für TDMA bzw. das Rundlaufverfahren ergibt sich dann mit der Dauer eines Zeitschlitzes T^{ZS} zu:

$$L^{TDMA} = T^{ZS} \cdot n^{gesamt}, \tag{2.3}$$

$$L^{RR}(p_i) = T^{ZS} \cdot \max\left(n^{aktiv}(p_i), n^{aktiv}(p_{i+1})\right). \tag{2.4}$$

Für Kommunikationspfade in NoCs oder Bus-Hierarchien, die über mehrere Komponenten hinweg verlaufen, sind sichere Grenzen für den Gesamtdurchsatz bzw. die Gesamtlatenz durch die Summe aller Latenzen und den minimalen Datendurchsatz aller Komponenten des Pfades gegeben.

Die mit den Gleichungen (2.3) und (2.4) berechneten Latenzen gelten für den Fall ohne Rückstau der Daten in den Eingangspuffern des Arbiters. Ein solcher Rückstau entsteht dann, wenn die Rate der an einem Eingang eintreffenden Daten höher ist als der aktuell vom Arbiter bereitgestellte Durchsatz. In diesem Fall füllt sich die Warteschlange im Eingangspuffer kontinuierlich auf, sodass die neu eingereihten Datenelemente eine deutlich höhere Wartezeit bzw. Gesamtlatenz erfahren.

2.2. Parallele Programmierung

Mehrkernprozessoren können nur dann effizient genutzt werden, wenn in der darauf ausgeführten Software mehrere Ausführungsstränge parallel abgearbeitet werden können. Die Hauptmotivation der dabei verwendeten parallelen Programmierparadigmen besteht deshalb darin, die Berechnungen durch möglichst effiziente Verteilung der Rechenlast auf mehrere Prozessorkerne zu beschleunigen. Ein in diesem Zusammenhang wichtiges Kriterium ist der sogenannte *Speedup* SP (englisch für *Beschleunigung*), der die Ausführungszeit T^{par} des parallelen Programms in ein Verhältnis zur Laufzeit T^{seq} eines äquivalenten sequentiellen Programms setzt:

$$SP := \frac{T^{seq}}{T^{par}} . \tag{2.5}$$

Bei der Parallelisierung, d. h. der Aufteilung einer Berechnung in parallele Ausführungsstränge, stellt der Speedup meist das primäre Optimierungskriterium dar (gelegentlich ergänzt durch Kriterien wie den zu erwartenden Energieverbrauch).

Die meisten Algorithmen bzw. Berechnungsaufgaben lassen sich nicht in vollständig in unabhängigen Ausführungsstränge zerlegen, da die einzelnen Teilberechnungen auf die Ergebnisse anderer Teilberechnungen zurückgreifen müssen. In mehr oder weniger langen Zeitabständen muss deshalb eine Datenübertragung zwischen den einzelnen Ausführungssträngen erfolgen, um solche Zwischenergebnisse austauschen zu können. Da die Ausführungszeit einer Berechnung zur Laufzeit mitunter stark variieren kann (vgl. Abbildung 1.1), liegen die Zwischenergebnisse jedoch unter Umständen noch nicht vor, wenn sie von einem der anderen Ausführungsstränge benötigt werden. In diesem Fall muss der betroffene Ausführungsstrang auf die Bereitstellung der Daten durch die anderen Stränge warten und bleibt so lange untätig. Im Vergleich zu sequentiellen Programmen stellt dies einen zusätzlichen Kostenfaktor mit unmittelbar senkender Wirkung auf den Speedup dar. Ein wesentliches Ziel der Parallelisierung besteht deshalb i. A. darin, möglichst unabhängige Ausführungsstränge zu erzeugen und den Datenaustausch so zu minimieren.

Ausgehend von dieser Grundproblematik geben die folgenden Unterabschnitte einen Überblick über einige für diese Arbeit relevante Konzepte aus dem Bereich der Parallelisierung und der parallelen Programmierung.

2.2.1. Tasks, Prozesse und Schedules

Ein Ansatz zur Parallelisierung von Berechnungen besteht darin, sie in sogenannte *Tasks* (englisch für *Aufgaben*) zu unterteilen. Zwischen zwei Tasks können eine oder mehrere *Abhängigkeiten* bestehen, z. B. wenn der abhängige Task auf Zwischenergebnisse des anderen Tasks zurückgreifen muss. Existiert eine solche Abhängigkeit, so darf der abhängige Task erst nach der Beendigung des unabhängigen Tasks (wenn auch die Zwischenergebnisse vorliegen) gestartet werden. Die Abhängigkeiten definieren damit eine Reihe von Vorgaben hinsichtlich der Reihenfolge, in der die Tasks auszuführen sind. Sie werden deshalb auch als *Präzedenzbedingungen* (englisch *Precedence Constraints*) bezeichnet. Die Tasks und ihre Abhängigkeiten konstituieren zusammen den sogenannten Task-Graphen (vgl. [80]):

Definition 2.4 (Task-Graph)**.** *Ein Task-Graph $TG = (V, E, w)$ besteht aus den Task-Knoten $TN_i \in V$ und den Abhängigkeiten $d_{i,j} = (TN_i, TN_j) \in E$. Die gerichteten Kanten $d_{i,j}$ verlaufen dabei stets von dem unabhängigen Task-Knoten TN_i zu dem abhängigen Knoten TN_j. Das Gewicht $w(TN_i)$ ist ein Maß für die Ausführungszeit des Tasks TN_i.*

Ein beispielhafter Task-Graph mit sechs Tasks ist in Abbildung 2.7 dargestellt. Nach der Zerlegung einer Berechnung in Tasks kann der dabei entstandene Task-Graph in nebenläufige Ausführungsstränge aufgeteilt und somit parallelisiert werden. Die Ausführungsstränge werden im Folgenden analog zu [64] als *Prozesse* π_k bezeichnet. Zur effizienten parallelen Ausführung der Tasks TN_i, müssen sie jeweils einem der Prozesse π_k so zugewiesen werden, dass sich insgesamt ein möglichst hoher Speedup ergibt. Hierbei ist nicht nur Zuordnung der Tasks zu den Prozessen (englisch *Mapping*), sondern auch die zeitliche Reihenfolge der Tasks innerhalb eines Prozesses von Bedeutung. Die Zuweisungen und Reihenfolgen der Tasks bilden zusammen einen sogenannten *Schedule* (englisch für *Zeitplan*, vgl. [80]):

Definition 2.5 (Schedule)**.** *Ein Schedule für den Task-Graphen $TG = (V, E, w)$ und eine endliche Menge von Prozessen $\mathcal{P} = \{\pi_1, \pi_2, \dots\}$ ist definiert durch die Funktionen $\sigma : V \mapsto \mathbb{N}_0$ und $\mu : V \mapsto \mathcal{P}$. Ein Schedule ist gültig, wenn*

- $d_{i,j} \in E \implies \sigma(TN_i) + w(TN_i) \leq \sigma(TN_j)$,
- $\mu(TN_i) = \mu(TN_j) \implies \begin{cases} & \sigma(TN_i) + w(TN_i) \leq \sigma(TN_j) \\ \textit{oder} & \sigma(TN_j) + w(TN_j) \leq \sigma(TN_i) \end{cases}$

für alle Abhängigkeiten und Task-Paare gilt.

Die Funktion σ repräsentiert dabei ein Maß für die Startzeit der Tasks (und damit deren Reihenfolge), während die Funktion μ die Zuordnung von Tasks zu den Prozessen festlegt. Die beiden Nebenbedingungen sorgen dafür, dass abhängige Tasks stets in der korrekten Reihenfolge ausgeführt werden, bzw. dass sich Tasks innerhalb desselben Prozesses zeitlich nicht überlappen. Abbildung 2.8 zeigt eine graphische Darstellung für einen möglichen Schedule zu dem Task-Graphen aus Abbildung 2.7 (Abhängigkeiten innerhalb eines Prozesses sind nicht dargestellt).

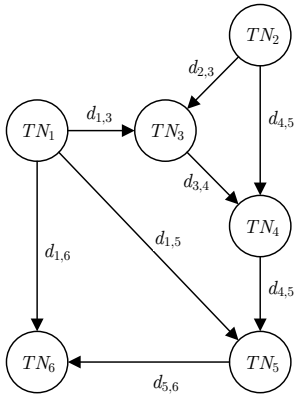


Abbildung 2.7.: Beispiel für einen Task-Graphen

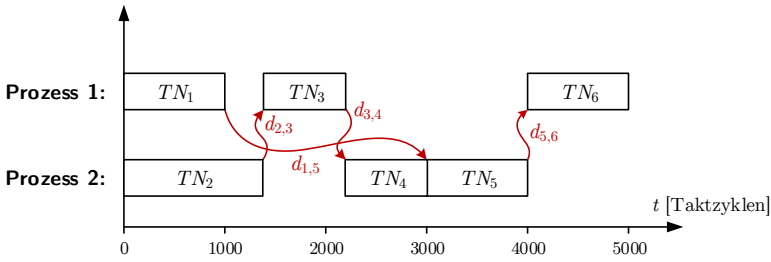


Abbildung 2.8.: Beispielhafter Schedule für den Task-Graphen aus Abbildung 2.7 über die Zeit t mit den prozessübergreifenden Abhängigkeiten

Das Optimierungsproblem, einen Schedule mit maximalem Speedup für den Task-Graphen $TG = (V, E, w)$ unter Verwendung von $|\mathcal{P}|$ Prozessen zu generieren, ist im Allgemeinen NP-vollständig [80]. Es existieren jedoch zahlreiche heuristische Verfahren zur approximativen Lösung von Scheduling-Problemen unterschiedlichster Ausprägung (siehe z.B. [9]). Auf eine nähere Betrachtung soll an dieser Stelle verzichtet werden, da klassische Scheduling-Algorithmen im Rahmen der vorliegenden Arbeit weniger von Bedeutung sind.

2.2.2. Klassifizierung von Abhängigkeiten

Mit Hilfe von Task-Graphen können die verschiedenen Arten von Abhängigkeiten zwischen den Tasks abstrakt durch Kanten dargestellt werden, um z. B. Scheduling-Probleme formulieren und lösen zu können. Entscheidend ist dabei lediglich die gemeinsame Eigenschaft aller Abhängigkeiten, dass sie eine bestimmte Reihenfolge für die betroffenen Tasks vorschreiben. Bei detaillierterer Betrachtung kann jedoch zwischen verschiedenen Klassen von Abhängigkeiten unterschieden werden (siehe z. B. [84; 80]):

Echte Datenabhängigkeiten (englisch *True Dependences*) entstehen, wenn ein Task Zwischenergebnisse erzeugt, die von einem anderen Task benötigt werden. Man spricht auch von Erzeuger/Konsumenten oder *Read-after-Write*-Abhängigkeiten (englisch für Lesen-nach-Schreiben, kurz RAW).

Ausgabeabhängigkeiten (englisch *Output Dependences*) bestehen dann, wenn zwei Tasks nacheinander Schreibzugriffe auf dieselbe Speicherstelle durchführen. In diesem Fall muss die Reihenfolge der Tasks beibehalten werden, damit der Speicher nach der Ausführung beider Tasks den korrekten Inhalt aufweist. Ausgabeabhängigkeiten werden auch als *Write-after-Write*-Abhängigkeiten (englisch für Schreiben-nach-Schreiben, kurz WAW) bezeichnet.

Gegenabhängigkeiten (englisch *Anti-Dependences*) entstehen, wenn ein Task eine Speicherstelle ausliest, bevor sie von einem anderen Task überschrieben wird. Die Reihenfolge der Tasks muss dann beibehalten werden, um zu verhindern, dass die Daten überschrieben werden bevor sie ausgelesen werden konnten. Gegenabhängigkeiten werden auch *Write-after-Read*-Abhängigkeiten (englisch für Schreiben-nach-Lesen, kurz WAR) genannt.

Kontrollflussabhängigkeiten liegen vor, wenn eine Programmverzweigung in einem Task von den Ergebnissen eines anderen Tasks abhängt, oder, wenn ein Task abhängig von Ergebnissen anderer Tasks überhaupt nicht ausgeführt wird.

Bei der Erzeugung eines parallelen Schedules können zwei von einer Abhängigkeit betroffene Tasks entweder demselben Prozess oder zwei unterschiedlichen Prozessen zugewiesen werden. In letzterem Fall verläuft die Abhängigkeiten über Prozess-Grenzen hinweg, sodass zur Laufzeit durch Kommunikations- und Synchronisationsmechanismen sichergestellt werden muss, dass die Tasks entsprechend der Präzedenzbedingung in der vorgeschriebenen Reihenfolge abgearbeitet werden. So muss z. B. in dem Schedule aus Abbildung 2.8 am Ende von TN_3 durch Synchronisation sichergestellt werden, dass TN_4 in Prozess 2 nicht schon vorher gestartet wird. Andernfalls würde dies zu einer Verletzung der Abhängigkeit $d_{3,4}$ führen.

2.2.3. Kommunikation und Synchronisation

In der parallelen Programmierung gibt es eine Reihe gängiger Synchronisationsprimitive, darunter z. B. die Semaphoren- oder die Barrieren-Synchronisation. Für das Forcieren der Reihenfolge zweier abhängiger Tasks in unterschiedlichen Prozessen bietet sich besonders das *Signal/Wait*-Synchronisationsmodell an. In diesem Modell sendet ein Prozess π_i mit Hilfe einer *Signal*-Operation ein Signal an einen Empfänger-Prozess π_j. Parallel dazu startet der Empfänger π_j eine zugehörige *Wait*-Operation, die den Prozess solange blockiert, bis das Signal erfolgreich empfangen wurde. Dadurch ist nach dem Ende der *Wait*-Operation in π_j sichergestellt, dass π_i alle Berechnungen, die der zugehörigen *Signal*-Operation vorangestellt sind, abgeschlossen hat. Für eine gegebene Abhängigkeit $d_{i,j}$ kann also durch das Einfügen eines *Signal/Wait*-Paares eine definierte Reihenfolge zwischen den Tasks in π_i vor der *Signal*-Operation und den Tasks in π_j nach der *Wait*-Operation erzwungen werden.

Für Ausgabe- und Gegenabhängigkeiten genügt das beschriebene Vorgehen, durch *Signal/Wait*-Operationen eine definierte Reihenfolge für die betroffenen Tasks zu erzwingen. Bei echten Datenabhängigkeiten sowie Kontrollflussabhängigkeiten müssen jedoch zusätzliche Informationen zwischen den Prozessen ausgetauscht werden, um die erzeugten Nutzdaten bzw. Informationen über den gewählten Programmzweig zu übermitteln. Die Kommunikation solcher Informationen zwischen den Prozessen erfolgt typischerweise entweder über den Austausch von Nachrichten (*Nachrichten-basierte Kommunikation*) oder über gemeinsam genutzte Speicherbereiche (*Shared Memory*). Die Wahl zwischen diesen beiden Methoden hängt meist stark von der später eingesetzten Hardware-Plattform ab. Kommunikation über gemeinsam genutzten Speicher setzt z. B. geeignete Speicherkomponenten in der Hardware voraus, die in Architekturen mit verteiltem Speicher jedoch nicht notwendigerweise vorhanden sind (vgl. Abschnitt 2.1).

2.2.3.1. Nachrichten-basierte Kommunikation

Bei Nachrichten-basierter Kommunikation tauschen die Prozesse ihre Daten in Form der namensgebenden Nachrichten aus. Die einfachste Form ist eine Punkt-zu-Punkt-Übertragung von einem sendenden Prozess π_i zu einem empfangenden Prozess π_j. Es handelt sich dabei um eine zweiteilige Operation, in der der Sender-Prozess π_i die Daten in einer Sendeoperation aus seinem Speicher in eine Nachricht überträgt, die dann vom Empfänger π_j in einer Empfangsoperation ausgelesen und in dessen lokalen Speicher geschrieben wird.

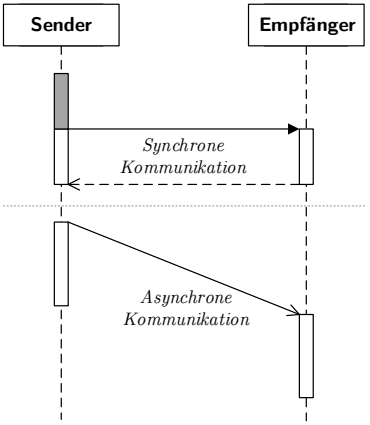


Abbildung 2.9.: Sequenzdiagramm mit synchroner und asynchroner Nachrichten-basierter Kommunikation

Für die Realisierung von Anwendungen mit Nachrichten-basierter Kommunikation gibt es standardisierte Schnittstellen wie z. B. das *Message Passing Interface* (englisch für *Nachrichtenübertragungsschnittstelle*, kurz MPI) [101]. Der MPI Standard unterscheidet grundsätzlich zwischen *synchroner* und *asynchroner* Kommunikation. Abbildung 2.9 zeigt beide Varianten in einem Sequenzdiagramm nach dem Standard der *Unified Modeling Language* (englisch für *vereinheitlichte Modellierungssprache*, kurz UML) [96]. Bei synchroner Kommunikation kann eine Sendeoperation erst nach Ausführung der zugehörigen Empfangsoperation abgeschlossen werden. Auf Senderseite gibt es dadurch u. U. eine zusätzliche Wartezeit, wenn der Empfänger die Empfangsoperation zum Sendezeitpunkt noch

nicht erreicht hat. Bei asynchroner Kommunikation kann die Sendeoperation dagegen unabhängig von der Empfangsoperation beendet werden. Die Empfangsoperation kann daher zeitlich auch nach der Beendigung der Sendeoperation stattfinden.

Die Grundvoraussetzung für asynchrone Kommunikation ist, dass die Nachricht für den Zeitraum zwischen der Sende- und Empfangsoperation entweder auf Senderseite, Empfängerseite oder dem Übertragungsweg vollständig zwischengespeichert werden kann. Bei synchroner Kommunikation entfällt diese Voraussetzung, da die Nachricht von der Sendeoperation in kleineren Einheiten übertragen und in der Empfangsoperation direkt im Speicher des Empfängers wieder zusammengesetzt werden kann. Eine Mischform aus synchroner und asynchroner Kommunikation ergibt sich, wenn für die Zwischenspeicherung nur eine begrenzte Menge an Speicher zur Verfügung steht. Die Sendeoperation ist dann asynchron solange genügend Zwischenspeicher für die aktuell zu sendende Nachricht vorhanden ist. Bei vollen Zwischenspeichern muss der Sender hingegen auf Empfangsoperationen warten, bis letztere durch das Entnehmen vorangegangener Nachrichten (und/oder bereits gesendete Teile der aktuellen Nachricht) aus dem Speicher ausreichend freie Kapazität schafft.

Sowohl bei synchroner als auch bei asynchroner Kommunikation kann die Empfangsoperation den aufrufenden Prozess so lange blockieren bis die Nachricht vollständig empfangen wurde. Zumindest in Richtung des Empfängers fungiert Nachrichten-basierte Kommunikation also gleichzeitig auch als Synchronisation. Die Synchronisationswirkung der Sende-/Empfangsoperationen entspricht dabei dem zuvor beschriebenen *Signal/Wait*-Modell. Die Operationen können folglich als Erweiterung von *Signal/Wait* mit zusätzlichem Austausch von Nutzdaten interpretiert werden.

2.2.3.2. Kommunikation über gemeinsam genutzten Speicher

Der Vorteil von Kommunikation über gemeinsam genutzten Speicher ist, dass kein kostspieliger Zusatzaufwand zur Kommunikation der Daten in Form von Nachrichten erforderlich ist. Stattdessen können die betroffenen Prozesse ohne vorherige Datenübertragung direkt auf eine gemeinsame Instanz der Daten im gemeinsamen Speicher zugreifen. Insbesondere bei größeren Datenmengen kann dies einen deutlichen Effizienzgewinn mit sich bringen. Zugriffe auf gemeinsam genutzte Daten erfordern aber dennoch eine geeignete Synchronisation zwischen den Prozessen, um sogenannte *Race Conditions* zu verhindern. Unter Race Conditions versteht man Konstellationen in parallelen Programmen, in denen Zugriffe auf gemeinsam genutzte Ressourcen (meist gemeinsame Speicher) bei mehrfachem Ausführen desselben Programms zu unterschiedlichen Ergebnissen

führen können (vgl. [80]). Ein solches nichtdeterministisches Verhalten entsteht z. B., wenn zwei Prozesse Schreibzugriffe auf denselben Speicherbereich durchführen, ohne dass die Reihenfolge der Zugriffe durch Synchronisation festgelegt wird (d.h. die vorliegende Ausgabeabhängigkeit wird ignoriert). In solchen Fällen hängt die Reihenfolge der Zugriffe und damit der Wert, der anschließend an der Speicherstelle abgelegt ist, von den Laufzeiten der Berechnungen in beiden Prozessen ab. Diese Laufzeiten können, wie eingangs beschrieben (vgl. Kapitel 1 und Abbildung 1.1), aufgrund von unterschiedlichen Verzweigungen, Cache-Effekten oder Zugriffskonflikten i. A. variieren.

Zur vollständigen Vermeidung von nichtdeterministischem Verhalten bei gemeinsam genutztem Speicher müssen daher sämtliche Ausgabe- und Gegenabhängigkeiten sowie die echten Datenabhängigkeiten zwischen den Prozessen zur Laufzeit durch Synchronisation forciert werden. Bei Nachrichten-basierter Kommunikation müssen hingegen nur echte Datenabhängigkeiten berücksichtigt werden, da die Daten hier mit dem Kommunikationsvorgang de facto in den Speicher des Empfängers kopiert werden. Es existiert dann keine gemeinsam genutzte Instanz der Daten, sodass Ausgabe- und Gegenabhängigkeiten keine Rolle spielen. Insgesamt hat Kommunikation über gemeinsame Speicher damit zwar den Vorteil, dass die aufwändige Datenübertragung eingespart wird. Im Gegenzug wird jedoch zusätzliche Synchronisation für Ausgabe- und Gegenabhängigkeiten benötigt.

2.2.4. Deadlocks

Ein weiteres Problem, das bei der parallelen Programmierung auftreten kann, sind die sogenannten *Deadlocks* (englisch für *Verklemmungen*). Dabei handelt es sich um eine Situation, in der sich mehrere Prozesse gegenseitig blockieren, sodass das Programm nicht mehr voranschreiten kann. Im Falle der zuvor diskutierten *Signal/Wait*-Synchronisation können Deadlocks durch zyklische Ketten aus *Signal*- und *Wait*-Operationen (oder analog Sende-/Empfangsoperationen) hervorgerufen werden.

Insbesondere in Anwesenheit von harten Echtzeitanforderungen müssen Deadlock-Situationen zur Entwurfszeit nachweisbar ausgeschlossen werden, da das Programm sonst im ungünstigsten Fall niemals endet und damit eine unendliche WCET hätte. Es lässt sich jedoch zeigen, dass *serialisierbare* parallele Programme garantiert frei von Deadlocks sind [97]. Ein paralleles Programm ist genau dann serialisierbar, wenn die Prozesse sich zu einem einzigen Prozess zusammenfassen lassen, ohne dabei eine *Wait*-Operation zeitlich vor der zugehörigen *Signal*-Operation einordnen zu müssen (analoges gilt für Sende-/Empfangsoperationen).

Die Serialisierung kann damit als eine inverse Transformation zur Parallelisierung eines Programms interpretiert werden.

2.2.5. Parallele Berechnungsmodelle

Ein *paralleles Berechnungsmodell* (englisch *Model-of-Computation*, kurz MoC) definiert wie Berechnungen, Algorithmen und Programme unter Nutzung von Nebenläufigkeit ausgeführt werden. Im Folgenden werden zwei der gängigsten Berechnungsmodelle für parallele Programme eingeführt.

2.2.5.1. Prozessnetzwerke

Das bisher diskutierte Modell mit den Prozessen als parallel ablaufende Einheiten orientiert sich am Berechnungsmodell der *Prozessnetzwerke* (englisch *Process Networks*, kurz PN). Ein wichtiger Vertreter dieser Modelle sind die sogenannten *Kahn Process Networks* (KPN) [51]. Ein KPN zeichnet sich dadurch aus, dass die Prozesse deterministisch sind und über FIFO-Kanäle von unendlicher Kapazität miteinander kommunizieren. Ein FIFO-Kanal bildet dabei eine Punkt-zu-Punkt-Verbindung zwischen zwei Prozessen π_i und π_j, in den π_i Daten in Form sogenannter *Tokens* eingefügt, sodass sie später von π_j konsumiert werden können. Aufgrund der unbegrenzten Kapazität der FIFO-Kanäle ist diese Form der Nachrichten-basierten Kommunikation (mit den Tokens als Nachrichten) vollständig asynchron: Während das Empfangen bzw. Auslesen von Daten aus dem Kanal so lange blockiert, bis ein Token vorliegt, steht beim Senden von Daten stets genug Speicher zur Verfügung um die Tokens zu übertragen und direkt mit der Ausführung des Prozesses fortzufahren.

Da in realen Systemen nur endlicher Speicher zur Verfügung steht, sind in der Praxis vor allem Abwandlungen des KPN mit endlicher Kanalkapazität von Bedeutung. In einem solchen Fall kann das Senden von Daten einen Prozess ebenfalls blockieren, wenn der zugehörige Kanal nicht mehr ausreichend freie Kapazität aufweist. Es kommt dann zu der in Abschnitt 2.2.3.1 beschriebenen Mischform aus synchroner und asynchroner Kommunikation.

2.2.5.2. Aktor-Modelle

Aktor-basierte Berechnungsmodelle unterteilen das Programm in sogenannte *Aktoren* (englisch *Actors*). Jeder Aktor stellt ein autonomes Objekt dar, das über Nachrichten mit anderen Aktoren kommunizieren kann. Aktoren verfügen über einen eigenen Nachrichtenpuffer, die sogenannte *Mailbox*, und arbeiten die darin

enthaltenen Nachrichten unabhängig von anderen Aktoren ab. Bei einer leeren Mailbox bleibt der Aktor solange untätig bis eine neue Nachricht eintrifft. Jeder Aktor hat einen separaten eigenen Zustand, sodass er ohne weitere Konflikte oder Abhängigkeiten parallel zu den anderen Aktoren ausgeführt werden kann. Bei der Bearbeitung eingehender Nachrichten durch einen Aktor sind der Versand neuer Nachrichten, das Erzeugen neuer Aktoren und/oder die Aktualisierung des lokalen Zustands als mögliche Reaktionen zugelassen. Ein wesentliches Merkmal von Aktoren ist, dass die Bearbeitung einer eingehenden Nachricht *atomar* sein muss, sodass sie nicht durch weitere Kommunikation/Synchronisation mit anderen Aktoren unterbrochen werden kann (siehe [80]). Die Nachrichten-basierte Kommunikation über die Mailboxen muss dazu vollständig *asynchron* ausgelegt werden.

Die Unterteilung der Berechnungen in Aktoren erfolgt typischerweise auf Basis der logischen Struktur des Programms. Zur parallelen Ausführung wird meist ein *Laufzeit-Scheduler* verwendet, der auf Basis der aktuell aktiven Aktoren (d.h. der Aktoren mit unbearbeiteten Nachrichten in der Mailbox) zur Laufzeit entscheidet, wann welcher Aktor auf welchem Prozessorkern ausgeführt wird.

2.3. Compiler-Werkzeuge und Zwischendarstellungen

Ein *Compiler* ist ein Programm, das als Eingabe einen Programmcode in einer bestimmten Programmiersprache erwartet und daraus einen anderen Programmcode mit derselben Semantik erzeugt (vgl. [39]). Viele Compiler übersetzen Programmcode einer höheren Programmiersprache, auch *Quellcode* oder *Quelltext* genannt, in Maschinencode, der für Prozessoren mit einer bestimmten Befehlssatz-Architektur (ISA) geeignet ist. Es gibt jedoch auch sogenannte *Quellcode-zu-Quellcode-Compiler* (englisch *Source-to-Source Compiler*, kurz S2S), die den ursprünglichen Quellcode in einen transformierten Quellcode in derselben oder einer anderen Programmiersprache übersetzten. Während des Transformationsprozesses vom ursprünglichen Quellcode hin zum Ziel-Code durchläuft das Programm in beiden Compiler-Typen meist eine oder mehrere *Zwischendarstellungen* (englisch *Intermediate Representations*, kurz IRs).

2.3.1. Abstrakter Syntaxbaum

Nach dem Einlesen des Quelltextes ist der *abstrakte Syntaxbaum* (english *Abstract Syntax Tree*, kurz AST) typischerweise die erste Zwischendarstellung, die von einem Compiler erzeugt wird. Der Teil des Compilers, der für die Erzeugung des AST aus dem Quellcode zuständig ist, wird auch als *Front-End* (englisch für

vorderes Ende) bezeichnet. Der AST ist eine Baumstruktur, die im Speicher des Compilers aufgebaut wird, um die Syntax des Quellprogramms in strukturierter Form abzubilden. Die Knoten des AST repräsentieren jeweils ein syntaktisches Konstrukt der Quellsprache. Die Knoten für zusammengesetzte Konstrukte haben wiederum untergeordnete Knoten, die die einzelnen Teil-Konstrukte abbilden. Für den einfachen arithmetischen Ausdruck `d = a * (b + c)` könnte der abstrakte Syntaxbaum z. B. die Struktur aus Abbildung 2.10 haben. Zur besseren Handhabung werden die Knoten des AST meist mit einer Reihe von Attributen versehen, die über die reine Syntax hinausgehen können. Dabei kann es sich z. B. um den Datentyp eines Ausdrucks, um hilfreiche Informationen für die Programmoptimierung oder sonstige relevante Attribute zur Semantik des Knotens handeln (vgl. [39]).

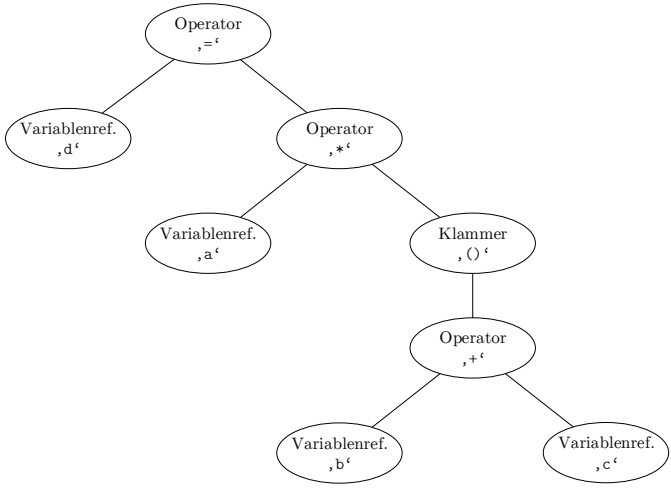


Abbildung 2.10.: Beispielhafter AST für den Ausdruck „`d = a * (b + c)`“

Der AST kann als Zwischendarstellung verschiedenen Transformationen unterzogen werden und/oder später in andere Zwischendarstellungen überführt werden. In Quellcode-zu-Quellcode-Compilern bildet ein (transformierter) AST meist auch die Grundlage zur Erzeugung des Ausgabequellcodes.

2.3.2. Kontrollflussgraph

Aus dem AST lässt sich durch einen einfachen Algorithmus (siehe [39]) der sogenannte *Kontrollflussgraph* (englisch *Control-Flow Graph*, kurz CFG) als eine weitere Zwischendarstellung ableiten. Der Kontrollflussgraph bildet die Verzweigungen im Programmcode ab und beschreibt damit alle grundsätzlich möglichen Programmabläufe (auch *Kontrollflüsse* genannt). Die Knoten des Kontrollflussgraphen werden durch die sogenannten *Basisblöcke* (englisch *Basic Blocks*) gebildet. Ein Basisblock ist definiert als eine maximal lange zusammenhängende Sequenz von Anweisungen im Programm, innerhalb derer es keine Verzweigung des Kontrollflusses gibt. Der Kontrollflussgraph lässt sich damit in Anlehnung an [97] wie folgt definieren:

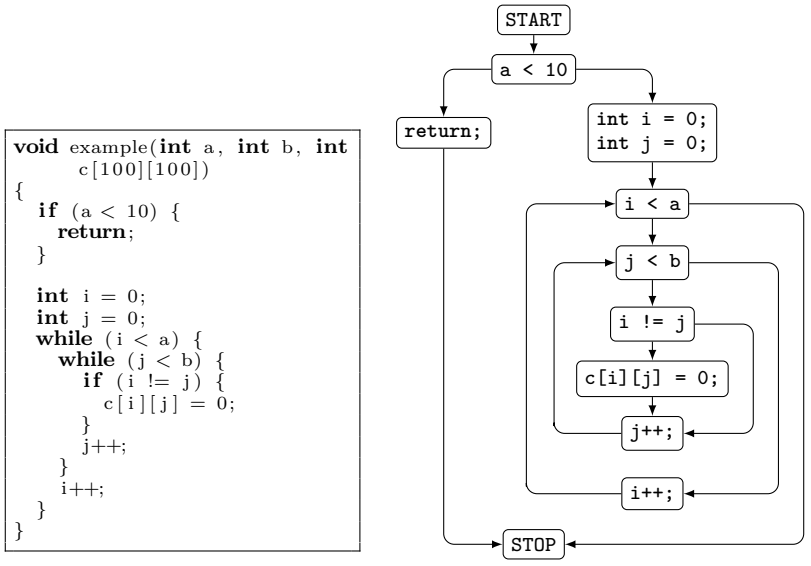
Definition 2.6 (Kontrollflussgraph)**.** *Ein Kontrollflussgraph CFG ist ein gerichteter Graph* (V, E^{cf}, TYP) *bestehend aus*

1. *einer Menge von Knoten* $bb_i \in V$*,*
2. *den Kontrollflusskanten* $cf_{i,j} = (bb_i, bb_j) \in E^{cf}$ *und*
3. *einem Knotentyp* $TYP(bb_i) \in \{BASISBLOCK, START, STOP\}$*.*

Eine Kontrollflusskante $cf_{i,j} \in E$ *existiert genau dann, wenn der Kontrollfluss des Programms am Ende von* bb_i *direkt zu* bb_j *springen kann. Jeder Kontrollflussgraph enthält genau einen Startknoten mit* $TYP(bb_i) = START$*, genau einen Endknoten mit* $TYP(bb_i) = STOP$ *und beliebig viele Basisblöcke mit* $TYP(bb_i) = BASISBLOCK$*.*

In der vorgestellten Form ist der CFG geeignet, um sequentielle Programme zu beschreiben, deren Kontrollfluss nur Sprünge zu statisch vorhersagbaren Sprungzielen enthält. Abbildung 2.11 zeigt ein Beispiel für einen Kontrollflussgraphen mit dem zugehörigen Quellcode einer Funktion in der Programmiersprache C. In prozeduralen Programmiersprachen wie C kann ein eigener CFG für jede Funktion definiert werden. Bei einer Ausführung des Programms durchläuft der Kontrollfluss dann eine Sequenz von Basisblöcken entlang eines zur Laufzeit festgelegten Pfades vom Start- zum Endknoten. Im Normalfall wird davon ausgegangen, dass mindestens ein solcher Pfad existiert. Ausnahmen sind z. B. Programme, in denen eine Berechnung über die gesamte Betriebszeit des Systems hinweg periodisch in einer Endlosschleife wiederholt werden soll.

Ein Kontrollflussgraph, in dem alle Schleifen nur einen einzigen Eintrittspunkt (d.h. nur einen Basisblock, über den die Schleife erreicht werden kann) haben, wird auch als *reduzierbarer* Kontrollflussgraph (englisch *Reducible Control-Flow*



(a) Ein Beispielquellcode in der Programmiersprache C

(b) Kontrollflussgraph für den Beispielquellcode

Abbildung 2.11.: Ein Beispielquellcode mit zugehörigem CFG

Graph) bezeichnet [97]. In einem solchen CFG kann zwischen *vorwärtsgerichteten* und *rückwärtsgerichteten* Kontrollflusskanten unterschieden werden. Letztere sind diejenigen Kanten, die von einem Basisblock innerhalb der Schleife zurück zum Eintrittspunkt führen. Der Kontrollflussgraph in Abbildung 2.11b zählt zu der Klasse der reduzierbaren Graphen, wobei die Knoten „`i < a`“ und „`j < b`“ jeweils den Eintrittspunkt der zugehörigen Schleife bilden. Durch das Entfernen aller Rückwärts-Kanten kann ein reduzierbarer CFG in einen azyklischen gerichteten Graphen überführt werden. In strukturierten Programmiersprachen wie C entstehen Schleifen meist ausschließlich durch Kontrollflusskonstrukte wie `for`- oder `while`-Blöcke. Diese Blöcke sind so konstruiert, dass im Normalfall (ohne zusätzliche Sprunganweisungen o. ä.) ein reduzierbar Kontrollfluss entsteht.

2.3.3. Parallele Programmgraphen

Das Konzept des Kontrollflussgraphen lässt sich gemäß [97] zu einem sogenannten *parallelen Programmgraphen* (englisch *Parallel Program Graph*, kurz PPG) erweitern, um auch parallele Programme beschreiben zu können.

Definition 2.7 (Paralleler Programmgraph)**.** *Ein Paralleler Programmgraph* $PPG = (V, E^{cf}, E^{sync}, TYP)$ *ist ein gerichteter Graph bestehend aus*

1. *einer Menge von Knoten* $bb_i \in V$*,*
2. *den Kontrollflusskanten* $cf_{i,j} = (bb_i, bb_j) \in E^{cf}$*,*
3. *einer Menge von Synchronisationskanten* $sy_{i,j} \in E^{sync}$ *mit einer zugehörigen Synchronisationsbedingung* $f_{i,j}$ *und*
4. *einem Knotentyp* $TYP(bb_i) \in \{BASISBLOCK, START, GABELUNG\}$*.*

Die Definitionen der Kontrollflusskanten sowie der Knotentypen BASISBLOCK und START entsprechen denen des CFG (siehe Definition 2.6). Eine Synchronisationskante $sy_{i,j} \in E^{sync}$ *existiert genau dann, wenn die Ausführungsreihenfolge* bb_i, bb_j *bei erfüllter Synchronisationsbedingung* $f_{i,j}$ *durch Synchronisation erzwungen wird. Der Knotentyp GABELUNG erzeugt einen Knoten, der den eingehenden Kontrollfluss in mehrere parallele Ausführungsstränge aufspaltet.*

Ein PPG hat damit weiterhin einen definierten Startknoten, im Gegensatz zum CFG jedoch keinen ausgezeichneten Endknoten. Potentiell nebenläufige Kontrollflusspfade müssen dementsprechend nicht notwendigerweise in demselben Knoten enden. Die Eigenschaft der Reduzierbarkeit für PPGs ist analog zu der entsprechenden Eigenschaft des Kontrollflussgraphen definiert [97].

2.3.4. Static Single Assignment Form

Für die Analyse und Optimierung des Datenflusses in einem Programm eignet sich die sogenannte *Static Single Assignment* (englisch für *Statische Einzelzuweisung*, kurz SSA) Form als Zwischendarstellung. Die Grundidee der SSA-Form besteht darin, dass jede Variable des Programms in dieser Darstellung nur genau einmal beschrieben werden darf (vgl. [39]). Ein Eingabeprogramm kann in seine SSA-Form überführt werden, indem für jede Zuweisung einer Variablen eine neue Kopie mit einer eindeutigen Versionsnummer generiert wird.

Ein Beispiel für ein C-Programm mit der zugehörigen SSA-Form ist in Abbildung 2.12 dargestellt. Da die Variable `a` im Originalprogramm mehrfach

```
int a = 0;
a++;

if (condition()) {
  a = a + 10;
} else {
  a = a + 5;
}

printf("%i\n", a);
```

(a) Beispielquellcode

```
int a0, a1, a2, a3, a4;
a0 = 0;
a1 = a0 + 1;
if (condition()) {
  a2 = a1 + 10;
} else {
  a3 = a1 + 5;
}
a4 = Φ(a2, a3);
printf("%i\n", a4);
```

(b) SSA-Form zu dem Beispielcode

Abbildung 2.12.: Ein Beispielprogramm in der Programmiersprache C mit der zugehörigen SSA-Form

überschrieben wird, enthält die SSA-Form mehrere durchnummerierte Kopien `a0` bis `a4`. Die Zuweisung eines Wertes zu einer neuen Variablenversion wird auch als *Definition* (englisch *Definition*) und das spätere Auslesen als *Verwendung* (englisch *Use*) der Variablen bezeichnet. Für jede Verwendung gibt es in der SSA-Form dann genau eine zugehörige Definition, die unmittelbar aus der Versionsnummer der verwendeten Variablen hervorgeht.

An Stellen mit mehreren zusammenlaufenden Kontrollflusskanten kann jedoch ein Sonderfall auftreten, wenn auf den zusammengeführten Kontrollflusspfaden unterschiedliche Definitionen der ursprünglichen Variablen vorhanden sind. Dies ist z. B. in der `if`-Verzweigung aus Abbildung 2.12 der Fall. Nach dem Zusammenführen der Kontrollflüsse gibt es dort keine eindeutige Definition mehr, da die korrekterweise zu verwendende Definition (in der Abbildung entweder `a2` oder `a3`) von dem zuvor gewählten Kontrollflusspfad abhängt. Die SSA-Form löst dieses Problem durch die Definition einer zusätzlichen Variablenversion unmittelbar nach dem Zusammenlaufen von Kontrollflusskanten mit konkurrierenden Definitionen. Zur Auswahl der korrekten vorherigen Definition wird die Funktion Φ eingeführt, die abhängig von der gewählten Kontrollflusskante den dazu passenden Variablenwert (in dem Beispiel `a2` oder `a3`) aus ihrer Argumenten-Liste zurückgibt.

Neben der direkten Zuordnung jeder Verwendung zu genau einer Definition hat die SSA-Form den Vorteil, dass sämtliche Ausgabe- und Gegenabhängigkeiten eliminiert werden. Dies ergibt sich unmittelbar daraus, dass jede Variable nur einmal geschrieben und danach nur noch gelesen werden kann. Ein Programm in SSA-Form enthält also nur noch Kontrollfluss- und echte Datenabhängigkeiten.

Die Änderungen durch die Erzeugung der SSA-Form werden meist vor der Generierung des Ausgabecodes (zumindest teilweise) wieder rückgängig gemacht. Der

Grund hierfür ist, dass die Variablenduplikate zusätzlichen Speicher benötigen, und dass eine tatsächliche Realisierung der Φ-Funktionen zur Laufzeit einen signifikanten Mehraufwand nach sich ziehen kann.

2.3.5. Hierarchische Task-Graphen

Die in Definition 2.4 eingeführten Task-Graphen können auch zum Aufbau einer entsprechenden Zwischendarstellung herangezogen werden. Dabei tritt jedoch das Problem auf, dass reale Programme meist diverse Schleifen enthalten. Bei einer ungünstigen Wahl der Tasks kann dann u. U. kein azyklischer Task-Graph mehr erzeugt werden, da eine Schleifeniteration im Allgemeinen auch von den Ergebnissen der vorherigen Iterationen abhängt. Die Berechnungen innerhalb der Schleife haben dann Abhängigkeiten zu sich selbst (sogenannte *Loop-Carried Dependences*) und es entsteht ein Zyklus. In diesem Fall ist es nicht mehr möglich, einen gültigen Schedule gemäß Definition 2.5 zu erzeugen, da die Nebenbedingung $d_{i,j} \in E \implies \sigma(TN_i) + w(TN_i) \leq \sigma(TN_j)$ innerhalb von Zyklen i. A. nicht für alle Abhängigkeiten eingehalten werden kann (zumindest bei nichttrivialen Task-Knoten mit $w(TN_i) > 0$). Für den einfachsten Fall einer gegenseitigen Abhängigkeit zwischen TN_i und TN_j mit $d_{i,j} \in E$ und $d_{j,i} \in E$ wird dies beim Einsetzen in die entsprechenden Nebenbedingungen sofort ersichtlich.

Das Problem der Zyklen kann durch die Erweiterung des Task-Graphen zu einem sogenannten *Hierarchischen Task-Graphen* (englisch *Hierarchical Task Graph*, kurz HTG) gelöst werden (siehe [35]). Der HTG kapselt dazu alle Schleifen bzw. alle stark zusammenhängenden Komponenten im Kontrollflussgraphen rekursiv in separaten Hierarchieebenen. Auf der obersten Hierarchieebene wird dabei zunächst ein azyklischer Task-Graph erzeugt, indem jede stark zusammenhängende Komponente des (reduzierbaren) CFG jeweils zu einem einzelnen hierarchischen Task-Knoten zusammengefasst wird. Nach dem Entfernen der Rückwärts-Kanten innerhalb der so erzeugten hierarchischen Knoten, wird darin wiederum nach untergeordneten stark zusammenhängenden Komponenten gesucht. Durch erneutes Gruppieren dieser Komponenten in untergeordnete hierarchische Knoten lässt sich das Konstruktionsverfahren so lange rekursiv fortsetzen, bis der CFG vollständig in eine Hierarchie aus azyklischen Teilgraphen zerlegt wurde. Aus diesen Teilgraphen lassen sich geeignete azyklische Task-Graphen erzeugen, die in ihrer Gesamtheit dann den HTG bilden. Die Task-Graphen der einzelnen Hierarchieebenen des HTG können nicht nur Abhängigkeiten zwischen ihren eigenen Knoten haben, sondern auch von der nächsthöheren Hierarchieebene abhängen. Jeder azyklische Task-Graph erhält deshalb jeweils einen Eingabe-

und einen Ausgabeknoten, über den alle eingehenden bzw. ausgehenden Abhängigkeiten der übrigen Knoten zu den darüber liegenden Hierarchieebenen verlaufen.

Ein Beispiel für den Hierarchischen Task-Graphen eines einfachen Programms ist in Abbildung 2.13 dargestellt (vgl. [RKB+19]). Die Schleifen des Beispielprogramms bilden jeweils einen hierarchischen Knoten, der wiederum einen azyklischen Teilgraphen zur Darstellung einer einzelnen Iteration enthält. Die *Loop-Carried Dependence* für die Variable `sum` verläuft dabei über die Eingabe- und Ausgabeknoten, sodass der Abhängigkeitszyklus aufgebrochen wird. Mit seiner hierarchischen Struktur exponiert der HTG dann die gesamte Parallelität des Programms auf allen Granularitäts-Stufen, bis hinunter zu den einzelnen Basisblöcken.

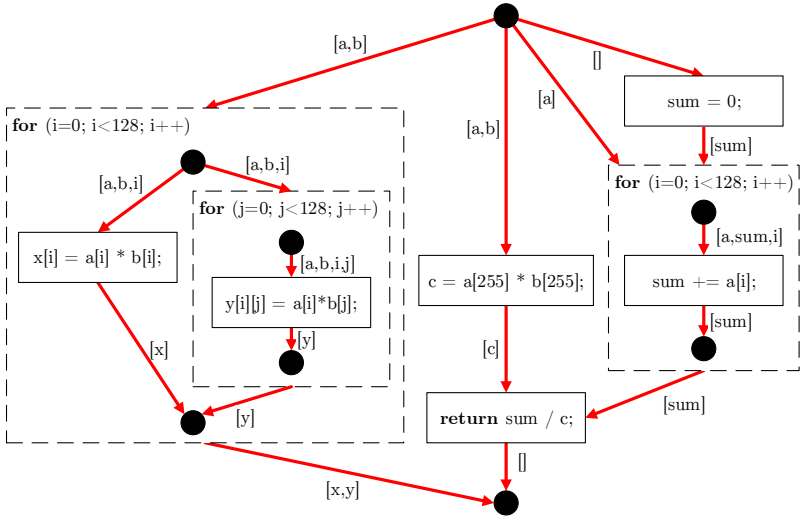
Mit Hilfe des HTG lässt sich für jede Hierarchieebene ein individueller Task-Graph-Schedule gemäß Definition 2.5 bestimmen. Die Scheduling-Probleme für die einzelnen Hierarchieebenen sind jedoch gekoppelt, sodass sie i. A. nicht isoliert betrachtet werden können. Ein möglicher Ansatz zur Generierung eines Schedules für den HTG als Ganzes besteht z. B. darin, entweder von den unteren Hierarchieebenen nach oben (englisch *Bottom-up*) oder von den oberen nach unten (englisch *Top-down*) sukzessive optimierte Schedules zu erzeugen (siehe z. B. [ATK+18]). Hierbei müssen in den späteren Scheduling-Problemen jeweils die Teilergebnisse aus den zuvor erzeugten Schedules berücksichtigt werden.

2.3.6. Compiler-Optimierung

Zu den wesentlichen Zielen der meisten Compiler zählt die Optimierung des erzeugten Ziel-Codes im Hinblick auf Kriterien wie Performanz, Code-Größe oder Speicherverbrauch. Ein häufiges Problem bei der Compiler-Optimierung besteht darin, dass der zu generierende Ziel-Code zum Zeitpunkt der Optimierung naturgemäß noch nicht vorhanden ist, weshalb die genannten Kriterien *a priori* nicht genau bestimmt werden können. Ein Optimierungsschritt muss daher auf vereinfachte Modelle (z. B. die bereits diskutierten Task-Graphen) und Schätzungen der Optimierungskriterien zurückgreifen, um eine geeignete Zielfunktion formulieren zu können. Hinzu kommt, dass Compiler meist aus mehreren Schritten zusammengesetzt sind, sodass die Genauigkeit der genannten Modelle und Schätzungen ggf. durch nachfolgende Compiler-Optimierungen beeinträchtigt werden. In der Folge hängt die Qualität des Gesamtergebnisses i. A. stark von der gewählten Reihenfolge der Optimierungsschritte ab. Die Problematik fehlender Informationen über die Ergebnisse nachfolgender Schritte wird deshalb auch als *Phase-Ordering Problem* (englisch für *Problem der Phasenreihenfolge*) bezeichnet (siehe z. B. [3]).

```
int htg(
    int a[256], int b[256],
    int x[128], int y[128][128])
{
  int c, i, j, sum;
  c = a[255] * b[255];
  for (i=0;i<128;i++) {
    x[i] = a[i] * b[i];
    for (j = 0; j < 128; j++)
      y[i][j] = a[i]*b[j];

  }
  sum = 0;
  for (i=1;i<128;i++)
    sum += a[i];
  return sum / c;
}
```

(a) Beispielquellcode in der Programmiersprache C



(b) HTG des Beispielcodes. Abhängigkeitskanten sind mit den zugehörigen Variablennamen beschriftet und Eingabe-/Ausgabeknoten werden als Kreise dargestellt.

Abbildung 2.13.: Ein Beispielprogramm mit zugehöriger HTG-Repräsentation

Die für Compiler relevanten Optimierungsprobleme zählen in der Regel zu der Klasse der kombinatorischen Optimierungsprobleme (siehe z. B. [61]) und beziehen sich oft auf gerichtete oder ungerichtete Graphen. Für diese Arbeit sind insbesondere die *ganzzahlige lineare Optimierung* (englisch *Integer Linear Programming*, kurz ILP) und das *Max-Flow-Min-Cut Problem* von Bedeutung.

2.3.6.1. Ganzzahlige lineare Optimierung

Die ganzzahlige lineare Optimierung sucht nach einem Satz ganzzahliger Werte, die eine lineare Zielfunktion maximieren und dabei eine Reihe von linearen Ungleichungen als Nebenbedingung einhalten.

Definition 2.8 (Ganzzahliges lineares Optimierungsproblem)**.** *Ein Optimierungsproblem*

$$\max_{x \in \mathbb{Z}^n} \{c^\intercal x : \ Ax \leq b\}$$

für eine Matrix $A \in \mathbb{Z}^{m \times n}$ und zwei Vektoren $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$ heißt ganzzahliges lineares Optimierungsproblem.

Der Vorteil der Formulierung eines Problems als ganzzahliges lineares Optimierungsproblem besteht darin, dass es Verfahren und Software-Werkzeuge gibt, die eine beweisbar optimale Lösung erzeugen können. Bisher ist jedoch kein Verfahren zur optimalen Lösung in Polynomialzeit bekannt, weshalb i. A. nur kleinere Instanzen in annehmbarer Zeit exakt gelöst werden können [61]. Es existieren allerdings auch heuristische Lösungsverfahren, die deutlich effizienter sind, aber keine optimale Lösung garantieren können. Wenn die Forderung nach ganzzahligen Werten für alle Variablen in x entfällt, so spricht man von einem linearen Optimierungsproblem. Für diese Problemklasse kann, anders als bei ILP-Problemen, eine optimale Lösung in Polynomialzeit gefunden werden. Eine Mischform ist die *gemischt-ganzzahlige lineare Optimierung* (englisch *Mixed-Integer linear Programming*, kurz MIP), in der nur für einen Teil der Variablen Ganzzahligkeit gefordert wird. Für weitere Details und mögliche Lösungsalgorithmen zu ILP und MIP Problemen sei an dieser Stelle auf die Literatur (z. B. [61]) verwiesen.

2.3.6.2. Max-Flow-Min-Cut Problem

Das Max-Flow-Min-Cut Problem bezieht sich auf einen Graphen $G = (V, E, w)$ mit positiven Gewichten $w : E \mapsto \mathbb{R}_+$ und zwei ausgewählten Knoten $s \in V$ und $t \in V$. Das Ziel besteht darin, einen minimalen Kantenschnitt $\delta(X) \subseteq E$ mit

$X \subset V$ zu finden, der den Graphen vollständig in zwei Teile X und $V \setminus X$ mit $s \in X$ und $t \notin X$ unterteilt. Ein solcher Schnitt wird auch *s-t-Schnitt* genannt, da jeder Pfad von s nach t über mindestens eine der Kanten in $\delta(X)$ verläuft. Die Kostenfunktion ist dabei definiert durch die Summe der Kantengewichte in $\delta(X)$, sodass sich das Optimierungsproblem

$$\min_{\delta(X) \subseteq E} \left\{ \sum_{e \in \delta(X)} w(e) \right\} \tag{2.6}$$

ergibt.

Es kann gezeigt werden, dass das Gewicht des minimalen s-t-Schnitts in G stets dem maximal möglichen Fluss von s nach t entspricht [61]. Die beiden zugehörigen Optimierungsprobleme sind damit äquivalent und können dementsprechend mit denselben Algorithmen gelöst werden. Daraus resultiert auch die Bezeichnung als *Max-Flow-Min-Cut Problem* (englisch für *Maximaler-Fluss-Minimaler-Schnitt*). Das Optimierungsproblem kann mit Algorithmen wie dem *Edmonds-Karp Algorithmus* (siehe [61]) in Polynomialzeit exakt gelöst werden.

2.4. Echtzeit-Software

2.4.1. Statische WCET-Analyse

Wie in Kapitel 1 beschrieben, erfordert die Verifikation harter Echtzeitanforderungen eine Abschätzung der WCET zur Entwurfszeit. Entsprechende Werkzeuge zur statischen WCET-Analyse verwenden den kompilierten Maschinencode des Programms und ermitteln daraus einen sicheren oberen Grenzwert für die WCET. Das Zeitverhalten der Hardware wird dabei durch ein geeignetes Modell beschrieben. Das Problem der WCET-Analyse lässt sich gemäß Wilhelm et al. [116] in die folgenden Teilprobleme unterteilen:

Datenwert-Analyse: Ziel dieser Analyse ist es, die möglichen Wertebereiche von Prozessorregistern und lokalen Variablen zur Entwurfszeit einzugrenzen. Mit den Ergebnissen können später die Adressen von Speicherzugriffen oder Schleifengrenzen ermittelt werden.

Kontrollfluss-Analyse: Diese Analyse extrahiert Informationen über Kontrollflusspfade, die von dem Programm potentiell durchlaufen werden können. Das Ziel besteht darin, die Menge der möglichen Pfade unter Berücksichtigung von Schleifengrenzen und Registerwertebereichen möglichst weit einzugrenzen.

Analyse des Prozessorverhaltens: Dieser Teilschritt dient der Bestimmung von Ausführungszeiten für die einzelnen Instruktionen unter Berücksichtigung der Prozessorarchitektur. Auf komplexeren Architekturen hängen diese Zeiten vom Ausführungskontext ab, da z. B. Caches zu Variationen bei den Speicherzugriffszeiten führen können. Dabei müssen insbesondere die sogenannten *Laufzeitanomalien* (englisch *Timing Anomalies*), bei denen eine lokale Verkürzung der Ausführungszeit insgesamt zu einer längeren globalen Laufzeit führt, berücksichtigt werden.

Grenzwertberechnung: In diesem Schritt wird der eigentliche obere Grenzwert der WCET als Endergebnis der WCET-Analyse berechnet. Die Teilergebnisse der vorherigen Analysen dienen hierbei als Grundlage.

Zur Durchführung von WCET-Analysen existieren verschiedene kommerzielle Software-Werkzeuge und Forschungsprototypen. Hierzu zählen z. B. die kommerziellen Werkzeuge aiT [29] und Bound-T[1] sowie akademische Software wie OTAWA [6] oder Chronos [70]. Diese Werkzeuge enthalten teils umfassende Analysen zur Bestimmung von Schleifengrenzen, Speicheradressen und Wertebereichen. Abhängig vom Programmaufbau und dem verwendeten Compiler können in vielen Fällen dennoch nicht alle nötigen Informationen automatisch bestimmt werden. Aus diesem Grund bieten Werkzeuge wie aiT dem Benutzer die Möglichkeit, umfassende Annotationen mit ergänzenden Informationen in die Analyse mit einfließen zu lassen.

Herkömmliche Werkzeuge zur WCET-Analyse gehen i. A. von einem einzelnen Prozessor aus, dessen Speicherzugriffszeiten nicht durch zusätzliche Hardware-Einheiten oder Prozessorkerne beeinflusst werden. Um eine gültige WCET auf Mehrkernprozessoren zu erhalten, müssen entweder spezialisierte Rechnerarchitekturen mit geringer zeitlicher Interferenz zwischen den Kernen oder neuartige Analysewerkzeuge mit Interferenz-Abschätzung eingesetzt werden.

2.4.2. Echtzeitbetriebssysteme

Die statische WCET-Analyse betrachtet üblicherweise ein einzelnes Echtzeitprogramm oder eine spezifische Software-Routine in Isolation. Oft ist es jedoch wünschenswert, mehrere Echtzeitprogramme mit bekannten WCET-Grenzen auf demselben Rechnersystem auszuführen, um z. B. die Kosten für zusätzliche Hardware einzusparen. Zu diesem Zweck werden *Echtzeitbetriebssysteme* (englisch *Real-Time Operating System*, kurz RTOS) eingesetzt, die die verfügbare Rechenzeit zur Laufzeit auf die einzelnen (Teil-)Programme (oft auch als Prozesse oder Tasks bezeichnet) aufteilen. Im Gegensatz zu herkömmlichen Betriebssystemen

[1] `http://www.bound-t.com/`

muss ein RTOS dabei sicherstellen, dass die einzelnen Betriebssystemprozesse genügend Rechenzeit erhalten, um die festgelegten Zeitlimits einhalten zu können. Grundlage hierfür bildet die WCET der relevanten Programmteile.

Echtzeitbetriebssysteme unterstützen meist eine Priorisierung der Prozesse, sodass weniger kritischen (Teil-)Programmen eine niedrigere Priorität eingeräumt werden kann als zeitkritischen Prozessen mit harten Echtzeitanforderungen. Werden Betriebssystemprozesse mit unterschiedlicher Kritikalität auf demselben Prozessorsystem ausgeführt, so spricht man auch von einem *gemischt-kritischen System* (englisch *Mixed-Criticality System*) [11; 43]. Solche Systeme haben den Vorteil, dass die verbleibende Rechenzeit nach Beendigung aller kritischen Berechnungen für nicht-kritische Aufgaben verwendet werden kann. Da die Ausführungszeiten zeitkritischer Berechnungen im Mittel meist deutlich unterhalb der WCET liegen, kann die durchschnittliche Auslastung der Prozessoren dadurch i. A. deutlich verbessert werden.

Für eine konkrete Konfiguration von RTOS-Prozessen/Tasks in einem Echtzeitbetriebssystem muss zur Entwurfszeit nachgewiesen werden, dass das RTOS zur Laufzeit alle geforderten Zeitlimits einhalten kann. Hierzu kann eine sogenannte *Schedulability-Analyse* für die vorgesehene Menge an RTOS-Prozessen durchgeführt werden [19]. Diese ermittelt auf Basis der WCET aller relevanten Berechnungen und Abhängigkeiten, ob der Laufzeit-Scheduler des Betriebssystems im ungünstigsten Ausführungsfall alle Zeitlimits einhalten kann. Im Gegensatz zu einem statischen Schedule, der zur Entwurfszeit festgelegt wird, muss hierbei berücksichtigt werden, dass das RTOS ggf. die Startzeit σ und/oder den verwendeten Prozessorkern μ für bestimmte Tasks/Prozesse erst zur Laufzeit festlegt (vgl. σ und μ in Definition 2.5).

Im Rahmen dieser Arbeit wird überwiegend die Parallelisierung von zusammengehörigen zeitkritischen Berechnungen unter Verwendung von statischen Schedules untersucht. Echtzeitbetriebssysteme und Laufzeit-Scheduling spielen daher im Folgenden nur eine untergeordnete Rolle. Nichtsdestotrotz können parallelisierte Programme mit statischen Schedules in Form von monolithischen Tasks/Prozessen in ein Gesamtsystem auf RTOS-Basis eingebunden werden.

Kapitel 3.

Stand der Technik

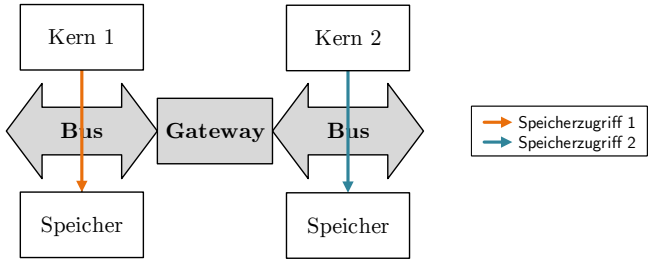
Dieses Kapitel fasst den Stand der Technik hinsichtlich der Anwendbarkeit von Mehrkernprozessoren in Echtzeitsystemen und der zugehörigen Entwurfswerkzeuge für parallele Software zusammen. Letzteres schließt insbesondere WCET-optimierte Compiler und Werkzeugketten zur automatischen Parallelisierung von Echtzeitanwendungen mit ein. Die ersten drei Abschnitte dieses Kapitels befassen sich jeweils mit einem dieser Aspekte, bevor Abschnitt 3.4 die neuen Beiträge dieser Arbeit zum Stand der Technik darlegt und einordnet.

3.1. Mehrkernprozessoren in Echtzeitsystemen

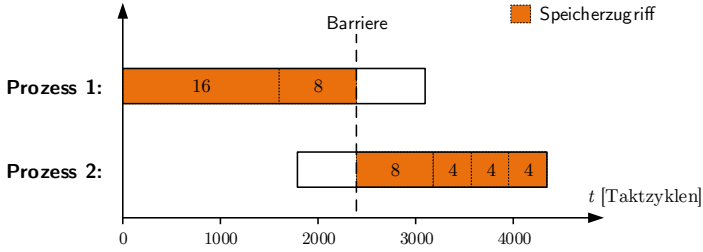
Vor allem aufgrund des in Kapitel 1 beschriebenen Problems der zeitlichen Interferenz zwischen den Prozessorkernen ist die Verwendung von Mehrkernprozessoren in Echtzeitsystemen noch mit einigen ungelösten Problemstellungen verbunden. Gegenstand aktueller Forschung sind dabei insbesondere die Methoden zur statischen WCET-Analyse für Mehrkernprozessoren und die damit verbundenen Anforderungen und Einschränkungen für die verwendeten Programmiermodelle und Hardware-Plattformen. Grundsätzlich zeichnen sich in bisherigen Arbeiten drei Herangehensweisen an das Problem der Interferenz ab (siehe z. B. [72]):

1. Die Verwendung von Hardware-Architekturen, die Interferenzeffekte durch *zeitliche Isolation* (englisch *temporal Isolation*) weitgehend vermeiden.
2. Die Gewährleistung von *Interferenz-freien parallelen Schedules* (vgl. Abschnitt 2.2) um eine zeitliche Isolation auf Software-Ebene herzustellen.
3. Die Erweiterung der statischen WCET-Analyse durch eine *Interferenz-Analyse*, die die Kosten von Interferenzeffekten nach oben abschätzt.

Die ersten beiden Herangehensweisen vermeiden Interferenzeffekte durch zeitliche Isolation vollständig. Abbildung 3.1 zeigt zwei mögliche Methoden, um dies zu erreichen. Bei zeitlicher Isolation in Hardware kann zur Vermeidung von



(a) Zeitliche Isolation in Hardware durch räumliche Trennung



(b) Zeitliche Isolation in Software durch zeitliche Trennung

Abbildung 3.1.: Zwei Möglichkeiten zur Realisierung von zeitlicher Isolation in Hardware bzw. Software

Interferenz entweder eine *räumliche* oder eine *zeitliche* Trennung konkurrierender Speicherzugriffe erfolgen. Die erste Option ist in Abbildung 3.1a dargestellt und stattet jeden Prozessorkern mit räumlich getrennten Hardware-Einheiten, wie z. B. eigenen Speicherkomponenten und Bussen, aus. Auf diese Weise können sich die Zugriffe unterschiedlicher Kerne gegenseitig nicht mehr beeinflussen. Eine *zeitliche* Trennung in Hardware kann dagegen durch TDMA-Arbitrierung erreicht werden, bei der die statische Zuordnung der Zeitschlitze eine konstante Zugriffszeit auf gemeinsam genutzte Hardware-Ressourcen sicherstellt. In beiden Ausprägungen besteht der Vorteil der zeitlichen Isolation in Hardware darin, dass existierende kommerziell erhältliche Werkzeuge zur WCET-Analyse im Idealfall unverändert eingesetzt werden können. Im Fall von TDMA-Arbitrierung geht dies jedoch auf Kosten der Performanz, da eine Hardware-Ressource bei

ungenutzten TDMA-Zeitschlitzen nicht mehr vollständig ausgelastet werden kann (vgl. Abbildung 2.5). Für die räumliche Trennung sind dagegen zusätzliche Hardware-Ressourcen nötig, die die Chip-Fläche und damit die Kosten einer solchen Plattform in die Höhe treiben. Verschiedene Ansätze zur Vermeidung von Interferenz auf Hardware-Ebene wurde in den vergangenen Jahren in Arbeiten wie [100; 88; 37; 90] verfolgt. Dabei wird meist auf durchgängige TDMA-Arbitrierung in Networks-on-Chip, Systembussen oder Speicherressourcen gesetzt. Aufgrund der Verwendung von TDMA sinkt jedoch sowohl die maximale Bandbreite pro Prozessorkern als auch die mittlere Ressourcen-Auslastung, weshalb derartige Architekturkonzepte in kommerziell erhältlichen Mehrkernprozessoren kaum zu finden sind. Außerdem ist die Skalierbarkeit erheblich eingeschränkt, da mit der Anzahl der Kerne meist auch die Anzahl der TDMA-Zeitschlitze erhöht werden muss.

Bei der zweiten Herangehensweise wird die zeitliche Isolation in Software hergestellt, sodass die Hardware weniger restriktiv aufgebaut werden kann. Hierbei werden Zugriffe auf gemeinsam genutzte Hardware-Ressourcen durch softwareseitige Synchronisation zeitlich voneinander getrennt, sodass zu jedem Zeitpunkt nur ein Prozessorkern auf die Ressource zugreifen kann. Ein entsprechendes Beispiel ist in Abbildung 3.1b dargestellt, wo durch eine Barrieren-Synchronisation sichergestellt wird, dass die Zugriffe in beiden Prozessen nicht zur gleichen Zeit auftreten können. Der Nachteil dieser Methode besteht darin, dass nur Programmteile, die ausschließlich mit Daten in privaten Speichern arbeiten, parallel ausgeführt werden dürfen. Bei Anwendungen mit hohem Speicherbedarf sind solche lokalen Speicher jedoch oft zu klein, sodass ein Großteil des Programms nicht auf regelmäßige Zugriffe auf einen gemeinsam genutzten Hauptspeicher verzichten kann. Mit Interferenz-freien Schedules lässt sich ein solches Programm dann nur sehr eingeschränkt parallelisieren. Der Ansatz eignet sich deshalb in erster Linie für Algorithmen mit hoher Datenlokalität, bietet bei weniger geeigneten Programmen jedoch nur eine begrenzte Skalierbarkeit. Zu den existierenden Arbeiten, die Interferenz-freie Schedules nutzen, zählen [87; 7; 73; 32]. Perret et al. formulieren das Problem der Zuweisung von Tasks in [87] mit einem Satz von Nebenbedingungen, die die zeitliche Isolation sicherstellen sollen. Die Arbeit fokussiert sich auf die kommerzielle Kalray MPPA Plattform, die mit 256 Kernen zu den Vielkernprozessoren zählt. Ebenfalls für die Kalray MPPA Plattform können mit dem Ansatz von Becker et al. [7] Interferenz-freie parallele Schedules generiert werden. Berechnungen werden dabei in eine Lese-, Ausführungs- und Schreibphase unterteilt, wobei nur erstere und letztere auf gemeinsam genutzte Speicher zugreifen dürfen. Durch zeitliche Isolation der Lese- und Schreibphasen lässt sich dann ein Interferenz-freier Schedule erzeugen. Ein ähnlicher Ansatz wird in [73] verfolgt, wobei eine vergleichbare Unterteilung in die genannten drei Phasen verwendet wird. Freitag et al. [32] gehen einen etwas anderen Weg

und schlagen eine sogenannte *virtuelle* zeitliche Isolation für gemischt-kritische Systeme vor. Dabei dürfen nicht-kritische Tasks bis zu einem gewissen Grad mit kritischen Tasks interferieren. Läuft ein kritischer Task zur Laufzeit Gefahr seine Zeitlimits zu verletzen, so werden alle nicht-kritischen Tasks vom Laufzeitsystem gedrosselt, um Interferenzen zu verringern und den kritischen Task dadurch zu beschleunigen. Die parallele Ausführung mehrerer zeitkritischer Berechnungen wird dadurch jedoch nicht ermöglicht.

Die dritte Herangehensweise der Interferenz-Analyse ist nicht von den Nachteilen der zeitlichen Isolation betroffen und erlaubt eine höhere Parallelität der Software ohne dabei eine spezialisierte Zielplattform vorauszusetzen. Durch die Interferenz-Analyse kann bei der Entwicklung paralleler Programme explizit zwischen Interferenz-Kosten und Parallelität abgewogen werden, um einen möglichst guten Kompromiss zu erzielen. Die präzise Eingrenzung von Interferenzeffekten bei überschaubarer pessimistischer Überschätzung erfordert jedoch eine detaillierte Kenntnis der Programmstruktur und/oder geeignete einschränkende Annahmen. Hierzu wird typischerweise ein entsprechend ausgelegtes paralleles Programmiermodell vorausgesetzt, das von Software-Entwicklern eingehalten werden muss, um eine korrekte WCET-Abschätzung zu gewährleisten. Die Herangehensweise der Interferenzabschätzung erfordert damit zum einen aufwändigere WCET-Analysewerkzeuge und erschwert zum anderen die Entwicklung paralleler Software durch die zusätzlichen Einschränkungen. Diese Nachteile können jedoch durch die in den nachfolgenden Abschnitten 3.2 und 3.3 diskutierte Integration von Compiler-Werkzeugen und WCET-Analysen stark abgemildert werden (siehe auch [RKB+19]). Das trifft insbesondere dann zu, wenn der Compiler auch die Parallelisierung des Eingabeprogramms automatisiert übernehmen kann.

Insgesamt verspricht die Interferenz-Analyse im Vergleich zur zeitlichen Isolation einen höheren Speedup bei der Parallelisierung, da mehr Parallelität bei Zugriffen auf gemeinsam genutzte Ressourcen möglich ist. Zugleich ist die Lösung jedoch auch mit einer erhöhten Komplexität bei der Verifikation der Echtzeitanforderungen verbunden, da eine isolierte WCET-Analyse für einzelne Prozessorkerne nicht mehr ausreicht, um sichere WCET-Grenzen zu bestimmen. Ansätze zur Lösung der damit verbundenen Herausforderungen sind nach wie vor Gegenstand aktueller Forschung. Da auch in dieser Arbeit auf die Forderung nach zeitlicher Isolation verzichtet wird, gibt der nachfolgende Unterabschnitt einen detaillierteren Überblick über existierende Ansätze zur statischen Analyse von Interferenzeffekten.

3.1.1. Ansätze zur Interferenz-Analyse

Die existierenden Arbeiten [93; 55; 56; 78; 2; 94; 24] befassen sich mit der Modellierung und Eingrenzung von Interferenzeffekten in paralleler Echtzeit-Software. Die Interferenzanalysen dienen in [93; 55; 56; 78; 2] der Ermittlung von WCET-Grenzen bzw. Software-Reaktionszeiten, während sie in [94; 24] zur Bestimmung paralleler Schedules verwendet werden.

Die Arbeiten [78; 55; 56; 94] beschränken das parallele Programmiermodell auf *zeitgesteuerte Schedules* (englisch *Time-triggered Schedule*). Dabei werden feste Zeitintervalle definiert, innerhalb derer jeder Prozessorkern maximal einen Task bearbeitet. Die Startzeiten der Zeitintervalle werden zwischen den Kernen synchronisiert, sodass jeder Kern zur gleichen Zeit mit dem Task des jeweiligen Intervalls beginnt. Ist die Ausführung eines Tasks bereits vor Ende des Zeitintervalls abgeschlossen, so bleibt der Prozessorkern für den Rest der Zeit untätig [78]. Die Belegung und Länge der Zeitintervalle werden typischerweise zur Entwurfszeit anhand der WCET der einzelnen Tasks festgelegt. Ein Vorteil zeitgesteuerter Schedules ist, dass Interferenz nur zwischen Tasks innerhalb desselben Zeitintervalls auftreten kann. Auf diese Weise lässt sich die Anzahl gleichzeitiger Zugriffe auf gemeinsam genutzte Ressourcen sicher eingrenzen. Für gemischt-kritische Systeme sind solche Schedules jedoch eher ungeeignet, da die Rechenzeit innerhalb der Intervalle nach Beendigung der zeitkritischen Tasks nicht mehr für andere Berechnungen genutzt werden kann. Im durchschnittlichen Fall ist deshalb mit erheblichen Leerlaufzeiten zu rechnen.

Die Ansätze [2; 93] definieren ebenfalls feste Zeitfenster, innerhalb derer ein Task ausgeführt werden kann, um Interferenzen eingrenzen zu können. Sie setzen im Gegensatz zu zeitgesteuerten Schedules jedoch keine global synchronisierten Zeitintervalle voraus, sondern erlauben beliebige Startzeiten für die einzelnen Tasks. Die Startzeit eines Tasks zusammen mit seiner WCET definiert dann das Zeitfenster, innerhalb dessen der Task ausgeführt werden kann. In [93] werden die Startzeiten anhand eines Datenflussgraphen bestimmt, wobei Datenabhängigkeiten jeweils dazu führen, dass der abhängige Task eine Startzeit größer gleich der maximalen Endzeit des unabhängigen Tasks erhält. Ähnlich wie bei zeitgesteuerten Schedules, bleibt die Rechenzeit bei Beendigung eines Tasks vor dem Ablauf seiner WCET i. A. ungenutzt.

Im Gegensatz zu den übrigen Arbeiten setzt [24] als einziger Ansatz nicht auf feste Zeitfenster zur Eingrenzung von Interferenzen. Stattdessen wird die Synchronisationsstruktur des parallelen Programms so eingeschränkt, dass Tasks nur dann parallel ausgeführt werden können, wenn sie sich im Schedule überlappen (siehe auch Abschnitt 3.3.2.1). Dadurch lassen sich Interferenzen ähnlich effizient eingrenzen wie bei der Verwendung von Zeitfenstern, allerdings sind die Start-

und Endzeiten der Tasks flexibel, da die Synchronisation nicht auf Zeit-Basis erfolgt. Vor dem Start eines Tasks muss also nicht mehr zwingend die gesamte WCET der vorherigen Tasks abgewartet werden. Stattdessen kann die Ausführung des Tasks unmittelbar nach der entsprechenden Synchronisationsoperation gestartet werden, sodass die Gesamtlaufzeit des Programms im durchschnittlichen Fall weniger stark beeinträchtigt wird. Ein Nachteil des Ansatzes besteht darin, dass die Parallelisierung von Schleifen nicht unterstützt wird. Für die Eingrenzung von Interferenzen muss außerdem ein erheblicher größerer Synchronisationsaufwand betrieben werden als es zur alleinigen Einhaltung von Abhängigkeiten nötig wäre. Letzteres kann sowohl die WCET, aber insbesondere auch die Ausführungszeit im durchschnittlichen Fall negativ beeinflussen.

Die genannten Arbeiten gehen von einer weitgehend festgelegten Architektur der Zielplattform aus. Die Ansätze [55; 56; 2] verwenden eine einfache Mehrkernarchitektur bestehend aus mehreren Kernen mit lokalen Zwischenspeichern, einem gemeinsam genutzten Speicherbus, sowie einem daran angebundenen Hauptspeicher. Die in [2] vorgestellte Implementierung bietet zwar Möglichkeiten zur Erweiterung auf komplexere Speicherhierarchien, was von den Autoren jedoch nicht näher ausgeführt wird. Die Ansätze aus [94; 93] nutzen die bereits erwähnte kommerziell erhältliche Kalray MPPA Plattform. Die Autoren von [78] setzten auf den ebenfalls kommerziell erhältlichen Freescale P4080 Prozessor.

Insgesamt setzen, abgesehen von [24], alle genannten Arbeiten auf feste Start- und Endzeiten für die Tasks, um die Anzahl interferierender Speicherzugriffe eingrenzen zu können. Die dabei verwendete Synchronisation auf Basis der Ausführungszeit eignet sich jedoch nur für reine Echtzeitsysteme, während gemischt-kritische Systeme nicht sinnvoll ausgelastet werden können. In [24] ist dieses Problem zwar nicht gegeben, allerdings wird ein vergleichsweise restriktives Programmiermodell mit hohem Synchronisationsaufwand vorausgesetzt. Darüber hinaus sind die existierenden Arbeiten auf einzelne Hardware-Architekturen festgelegt und beinhalten keine Unterstützung für allgemeine Beschreibungsformate zur Definition der Zielarchitektur.

3.1.2. Modellierung von Mehrkernprozessoren

Die Modellierung von Mehrkernprozessorsystemen im Allgemeinen ist ein weites Feld mit zahlreichen existierenden Forschungsarbeiten und kommerziellen Werkzeugen. Im Rahmen dieser Arbeit sind vor allem Ansätze zur Beschreibung der Plattform mittels Architekturbeschreibungssprachen (ADLs) sowie mathematische Modelle zur Beschreibung der Echtzeit-Performanz in Chip-internen Netzwerken von Bedeutung.

3.1.2.1. Architekturbeschreibungssprachen

Architekturbeschreibungssprachen haben zum Ziel, die Hardware-/Software-Architektur einer Zielplattform auf einer geeigneten Abstraktionsebene zu beschreiben. Zu den kommerziell eingesetzten Vertretern gehören z. B. die *Architecture Analysis & Design Language* (AADL) [28], die *Electronics Architecture and Software Technology - Architecture Description Language* (EAST-ADL) [22], das *Software-Hardware Interface for Multi-Many-Core* (SHIM) [111] sowie das UML-Profil *MARTE* (*Modeling and Analysis of Real-time and Embedded systems*)[108]. Für die verbreiteten ADLs gibt es oft ganze Ökosysteme aus Software-Werkzeugen, die das ADL-Format entweder als Eingabe nutzen, um z. B. Informationen über die Zielplattform berücksichtigen zu können, und/oder den Benutzer bei der Erstellung von Hardware-Modellen unterstützen. Im Bereich der Werkzeuge zur automatisierten Parallelisierung wurden ADL Beschreibungen z. B. von Stripf et al. in [105] eingesetzt, um mehrere Zielplattformen unterstützen zu können. Der Schwerpunkt liegt dabei auf der Generierung einer Simulationsumgebung für die beschriebene Hardware-Plattform, während Echtzeitanforderungen keine Rolle spielen.

Im Hinblick auf harte Echtzeitgarantien muss vor allem das Zeitverhalten im ungünstigsten Ausführungsfall modelliert werden. Die allgemeine Beschreibung der Performanz von Systemkomponenten und damit des Zeitverhaltens gehört typischerweise zu den Kernbestandteilen einer ADL. Hierbei variieren jedoch die Abstraktionsebenen, auf denen die entsprechenden Informationen modelliert werden. Meist sind die Modelle auf höheren Abstraktionsebenen angesiedelt, sodass sie für Aufgaben, wie z. B. die frühe Abschätzung von Latenzen oder die Schedulability-Analyse (vgl. Abschnitt 2.4.2), ausreichend präzise sind. Für das Ableiten von sicheren WCET-Grenzen, insbesondere im Zusammenhang mit der Analyse von Interferenzeffekten, fehlen jedoch meist die nötigen Hardware-Details. Die nachfolgende Übersicht beschränkt sich daher auf eine Auswahl existierender Arbeiten, die einem detaillierten Modell des Zeitverhaltens im ungünstigsten Ausführungsfall am nächsten kommen.

Zu den existierenden Arbeiten, die das Zeitverhalten mit Hilfe von ADLs beschreiben, gehören u. a. [36; 52; 75; 67; 85; 95; 112]. Die Autoren von [36] verwenden Modelle auf Basis der EAST-ADL zur Unterstützung einer Schedulability-Analyse. Ebenfalls unter Verwendung der EAST-ADL wird in [52] eine Methode zur simulations-basierten und formalen Verifikation von abstrakten Anwendungsmodellen vorgeschlagen. Es wird ein ereignisgesteuertes Anwendungsmodell vorausgesetzt, wobei die Ausführungszeiten der Softwarekomponenten vorgegeben werden müssen. Eine Analyse der Hardwareeigenschaften findet dabei nicht statt. Weitere Arbeiten aus dem automobilen Umfeld, das die hauptsächliche

Anwendungsdomäne der EAST-ADL darstellt, betrachten zudem verteilte Architekturen aus mehreren räumlich verteilten Steuergeräten. Hierzu zählt z. B. [75] mit einem Ansatz zur Analyse des Zeitverhaltens in verteilten Systemarchitekturen.

Auf Basis der AADL stellt [85] ein Software-Werkzeug vor, mit dem das Zeitverhalten von gemischt-kritischen Applikationen modelliert werden kann. Die Methode deckt jedoch nur Einzelkernprozessoren ab und berücksichtigt keine Details zum Zeitverhalten der Hardware. Die Arbeiten [67; 95] betrachten dagegen auch Mehrkernprozessoren und berücksichtigen, im Gegensatz zu den genannten Ansätzen auf Basis der EAST-ADL, auch den internen Aufbau der Hardware um Aussagen über das Zeitverhalten der Software abzuleiten. Hierbei handelt es sich jedoch um vergleichsweise abstrakte Modelle, die keine detaillierte Analyse von Arbitrierungsschemata oder Interferenzeffekten erlauben.

Auf Basis von UML-MARTE schlagen die Autoren von [112] eine Methode zur Performanz-Analyse für Echtzeitsysteme vor. Der Ansatz verwendet Profiling, eine Schedulability-Analyse und Simulationen des System-Modells um die Performanz im ungünstigsten Fall abzuschätzen. Da aus den Ergebnissen des Profiling bzw. der Simulation keine harten Garantien abgeleitet werden können, eignet sich der Ansatz jedoch vorwiegend für weiche Echtzeitsysteme.

Insgesamt gibt es zahlreiche Forschungsarbeiten und kommerzielle Software-Werkzeuge, die ADL Beschreibungen, von der Entwurfsraumexploration bis hin zur Verifikation, in verschiedenen Bereichen des Systementwurfs einsetzen. Das Abstraktionsniveau der vorgeschlagenen Modelle bildet die Hardware-Plattform jedoch nicht ausreichenden detailliert ab, um Interferenzeffekte oder die WCET auf Mehrkernprozessoren mit ausreichender Genauigkeit nach oben abschätzen zu können.

3.1.2.2. Mathematische Performanz-Modelle für Netzwerke

Mit Hilfe mathematischer Modelle lässt sich die Performanz bzw. das Zeitverhalten eines Netzwerks auf einer abstrakten Ebene beschreiben, ohne alle Details der Implementierung berücksichtigen zu müssen. Im Rahmen dieser Arbeit ist vor allem die formale Beschreibung von Chip-Internen Kommunikationsnetzwerken wie z. B. NoCs (vgl. Abschnitt 2.1.3) von Interesse. Im Gegensatz zu den in Abschnitt 3.1.1 vorgestellten Ansätzen zur Interferenz-Analyse betrachten solche NoC-Modelle hauptsächlich die Netzwerk-Hardware, lassen die Software-Architektur jedoch außen vor. Bei den mathematischen NoC-Modellen basieren die gängigen Ansätze typischerweise auf den Methoden der *Warteschlangen-Theorie* (englisch *Queueing Theory*), des *Netzwerk-Kalküls* (englisch *Network*

Modell	Modell des Datenaufkommens	Ableitbare Aussagen zur Performanz
Warteschlangen-Theorie	Wahrscheinlichkeitsverteilung für die Ankunft von Paketen	Durchschnittswerte für Latenz & Durchsatz
Netzwerk-Kalkül	Ankunftsfunktionen	Latenz & Puffernutzung im ungünstigsten Fall
Datenfluss-Analyse	Tokens, die von Aktoren generiert und konsumiert werden	Durchsatz und Latenz im ungünstigsten Fall

Tabelle 3.1.: Übersicht über formale Performanz-Modelle für Netzwerke (vgl. [58])

Calculus) oder der *Datenfluss-Analyse* (siehe [58]). Eine Übersicht über einige wesentliche Merkmale dieser Methoden ist in Tabelle 3.1 dargestellt.

Ansätze auf Basis der *Warteschlangen-Theorie* modellieren das Netzwerk durch einen gerichteten Graphen aus Warteschlangen und sogenannten Server-Knoten [58]. Dabei wird ein probabilistischer Ansatz verfolgt, der die mittlere Zeit zwischen der Ankunft zweier aufeinanderfolgender Elemente durch eine Wahrscheinlichkeitsverteilung beschreibt. Für die Entnahme von Elemente aus der Warteschlange durch einen Server-Knoten können verschiedene Service-Modelle zugrunde gelegt werden, wodurch sich z. B. das Rundlaufverfahren (vgl. Abschnitt 2.1.3.1) modellieren lässt. Die Klasse der Warteschlangen-Modelle eignet sich durch den probabilistischen Ansatz vorwiegend zur Bestimmung der Performanz im durchschnittlichen Fall. Harte Echtzeitgarantien lassen sich daraus jedoch nicht ableiten.

Für die Methode des *Netzwerk-Kalküls* existiert die Unterklasse des *Echtzeit-Kalküls* (englisch *Real-Time Calculus*), die speziell auf Echtzeit-Systeme abzielt (siehe [58]). Darin werden Netzwerk-Aktivitäten durch Ereignisse beschrieben, wobei die minimale und maximale Anzahl von Ereignissen pro Zeitintervall und Systemkomponente jeweils durch eine *Ankunftsfunktion* (englisch *Arrival Curve*) modelliert wird. Mit Hilfe der *min-plus Algebra* lässt sich aus allen Ankunftsfunktionen einer Komponente deren *Ausgabefunktion* ermitteln. Innerhalb eines Netzwerks kann diese Ausgabefunktion wiederum als Ankunftsfunktion für nachfolgende Komponenten dienen. Weiterhin können die Verzögerungen der Ereignisse beim Durchlaufen der Komponente sowie die Anzahl der in einer Komponente zwischengespeicherten Ereignisse, auch *Backlog* genannt, bestimmt werden. Letzteres dient als Maß für die Füllstände von Puffer-Ressourcen im

Netzwerk. Das Echtzeit-Kalkül ermöglicht harte Echtzeitgarantien, sofern korrekte Ausgabefunktionen für alle Datenquellen im Netzwerk vorhanden sind. Die Methode des Netzwerk-Kalküls kann dann, wie in [91], zur Bestimmung von Latenzen für die Datenübertragung in NoCs verwendet werden. Die Autoren von [91] setzen dazu jedoch spezielle Hardware-Einheiten zur Steuerung der Datenrate voraus, um die Einhaltung der modellierten Ankunftsfunktionen zur Laufzeit sicherzustellen.

Bei der Beschreibung mit den Methoden der *Datenfluss-Analyse* wird das Netzwerk durch einen Aktor-basierten Datenflussgraphen dargestellt (vgl. Abschnitt 2.2.5). In solchen Datenflussnetzwerken werden typischerweise die Latenz und der Durchsatz als wesentliche Kenngrößen betrachtet (siehe [58]). Um das Zeitverhalten zu modellieren, können die Aktoren mit einer Verzögerungszeit beaufschlagt werden, die ggf. auch Arbitrierungs-Verzögerungen beinhaltet (vgl. [58; 115]). Bei Verwendung entsprechender Verzögerungszeiten ermöglichen Datenflussmodelle auch Aussagen über das Zeitverhalten im ungünstigsten Fall und eignen sich damit auch für Echtzeit-Systeme.

Die genannten mathematischen Modelle für allgemeine Netzwerke erlauben es, beliebige Topologien von Kommunikationsverbindungen und NoCs in Mehrkernprozessorarchitekturen zu beschreiben. Mit Ausnahme der probabilistischen Ansätze auf Basis der Warteschlangen-Theorie lassen sich mit diesen Methoden, unter bestimmten Voraussetzungen, auch Echtzeitgarantien ableiten. Um Aussagen über die WCET eines parallelen Programms treffen zu können müssen diese Methoden jedoch noch mit einer Interferenz-Analyse auf Software-Ebene (vgl. Abschnitt 3.1.1) verknüpft werden. Dies kann nur dann zu sinnvollen Ergebnissen führen, wenn die Speicherzugriffsmuster der Software den Annahmen des Modells entsprechen, sodass sich z. B. eine konkrete Ankunftsfunktion ableiten lässt.

3.2. WCET-optimierte Compiler

Werkzeuge zur statischen WCET-Analyse nutzen teils aufwändige Analysen um Schleifengrenzen oder Speicheradressen aus dem kompilierten Programm zu extrahieren. Viele Compiler sind jedoch auf gute Performance im durchschnittlichen Fall optimiert und generieren daher oft Maschinencode, der nur eingeschränkt analysierbar ist. In der Folge können statische Analysen ggf. nur unvollständige Ergebnisse oder pessimistische Abschätzungen liefern, weshalb die fehlenden Informationen manuell vom Benutzer vorgegeben werden müssen. Dieses Problem lässt sich durch WCET-optimierte Compiler und eine enge Kopplung von Compiler und WCET-Analyse entschärfen. Auf diese Weise können die statische Analysierbarkeit des Maschinencodes verbessert und interne

Informationen des Compilers an die WCET-Analyse weitergegeben werden (siehe z. B. [116]).

Der umfangreichste existierende Ansatz in diese Richtung ist der von Falk et al. in [27] vorgeschlagene *WCET-aware C Compiler* (WCC). Die Generierung von Maschinencode aus C-Programmen ist in diesem Compiler vollständig auf das WCET-Kriterium optimiert. Eine enge Kopplung mit dem WCET-Analysewerkzeug aiT [29] ermöglicht dabei eine Rückkopplung von WCET-Informationen in den Optimierungsprozess. Der Compiler nutzt zwei Zwischendarstellungen, von denen sich die erste am Quellcode orientiert, während die zweite nah am Maschinencode angesiedelt ist. Informationen wie die Anzahl der Iterationen eines Basisblocks können so bereits auf Quelltext-Ebene vom Compiler gesammelt werden, bevor relevante Zusatzinformationen, wie z. B. die Datentypen, beim Übergang zum Maschinencode verloren gehen. Der WCC implementiert eine Reihe von Optimierungen zur Verbesserung der WCET, darunter die Duplikation und Positionierung von Prozeduren, eine WCET-optimierte Registerallokation sowie die automatische Allokation schneller Zwischenspeicher (englisch *Scratchpad Memories*, kurz SPM) für Daten und Instruktionen.

Die letztgenannte Nutzung schneller aber relative kleiner Zwischenspeicher zählt zu den am häufigsten untersuchten Compiler-Optimierungen im Zusammenhang mit Echtzeit-Software. Einer der Hauptgründe dafür ist, dass solche Speicher in Echtzeitsystemen eine gute Alternative zu Caches darstellen. Im Gegensatz zu Caches werden sie von der Software verwaltet, sodass ein Compiler zur Entwurfszeit eine geeignete Speicherallokation vorgeben kann. Die statische WCET-Analyse kann dadurch besser vorhersagen, ob ein Speicherzugriff auf den langsameren Hauptspeicher zurückgreifen muss oder nicht. Im Fall von Caches entscheidet sich Letzteres erst zur Laufzeit, wobei der zuvor durchlaufene Kontrollflusspfad eine maßgebliche Rolle spielt.

3.2.1. Speicherallokation

Unter *Speicherallokation* versteht man das Problem, die Datenobjekte eines Programms (z. B. Variablen oder Basisblöcke aus Maschinencode-Instruktionen) möglichst optimal auf die verfügbaren Speicher zu verteilen. In Echtzeit-Software wird die Speicherallokation in der Regel zur Entwurfszeit festgelegt, um die statische Vorhersagbarkeit im Hinblick auf die WCET-Analyse zu verbessern. In den existierenden Ansätzen zur Allokation von Scratchpad-Speichern wird meist von dem Spezialfall einer Speicherhierarchie mit einem schnellen Zwischenspeicher von überschaubarer Größe und einem langsameren Hauptspeicher ausgegangen. In größeren Mehrkernprozessoren mit verteilten Speichern sind jedoch auch

Ansatz	WCET-optimiert	Dynamisch	Code / Daten	Mehrkern	Mehrstufig
Avissar et al. [4]	Nein	Nein	Daten	Nein	Ja
Suhendra et al. [106]	Ja	Nein	Daten	Nein	Nein
Deverge et al. [23]	Ja	Ja	Daten	Nein	Nein
Falk et al. [27]	Ja	Nein	Beides	Nein	Nein
Wan et al. [114]	Ja	Ja	Daten	Nein	Nein
Kim et al. [59]	Ja	Ja	Code	Ja	Nein
Liu et al. [71]	Ja	Ja	Beides	Ja	Ja
Kafshdooz et al. [50]	Ja	Ja	Beides	Ja	Ja
Oehlert et al. [79]	Ja	Nein	Code	Ja	Nein

Tabelle 3.2.: Eine Auswahl relevanter Ansätze zur Speicherallokation

komplexere Speicherhierarchien möglich, sodass sich im Allgemeinen ein weiter gefasstes Speicherallokationsproblem ergibt.

Das Problem der Allokation von Scratchpad-Speichern für Echtzeit-Systeme wurde bereits in einer Reihe von Arbeiten, darunter [106; 23; 27; 114; 59; 71; 50; 79], untersucht. Darüber hinaus gibt es zahlreiche Arbeiten, die nicht auf Echtzeitsysteme spezialisiert sind, und deshalb an dieser Stelle nicht näher betrachtet werden. Davon ausgenommen ist der Ansatz von Avissar et al. [4], der im Gegensatz zu den genannten WCET-optimierten Ansätzen beliebige heterogene Speicherhierarchien unterstützt. Letzteres macht ihn für die vorliegende Arbeit interessant, da die angestrebte plattformunabhängige Lösung für ein breiteres Spektrum von Speicherarchitekturen geeignet sein muss.

Tabelle 3.2 vergleicht die genannten Ansätze anhand verschiedener Kriterien. Grundsätzlich kann bei der Speicherallokation zwischen statischer und dynamischer Allokation unterschieden werden. Statische Ansätze weisen einem Datenobjekt genau ein festes Speichersegment zu, in dem es über die gesamte Lebensdauer hinweg verbleibt. Bei der dynamischen Allokation kann ein Datenobjekt hingegen zur Laufzeit zwischen den Speichern verschoben werden. In Echtzeit-Software findet Letzteres meist nur an zur Entwurfszeit festgelegten Stellen im Programm statt, sodass auf einen Laufzeitmechanismus zur Speicherverwaltung verzichtet werden kann.

Als weiteres Kriterium unterscheidet Tabelle 3.2 zwischen der Allokation von Daten, Programmcode oder einer Kombination aus beidem. Hinzu kommt die Klassifizierung nach der Eignung des Ansatzes für Mehrkernprozessoren sowie dessen Unterstützung für mehrstufige Speicherhierarchien. Eine Speicherhierarchie zählt dabei als mehrstufig, wenn ein Prozessorkern Zugriff auf mehr als nur einen Scratchpad-Speicher und den Hauptspeicher hat.

Unter den Ansätzen aus Tabelle 3.2 verwenden die Arbeiten [4; 106; 23; 27; 59; 71; 50] ganzzahlige lineare Optimierung (ILP, siehe Abschnitt 2.3.6), um das Optimierungsproblem darzustellen und optimal zu lösen. Die Ansätze in [106; 114; 59; 50; 79] stellen außerdem verschiedene Heuristiken zur Lösung des Allokationsproblems vor.

Der Ansatz von Avissar et al. [4] nutzt eine ILP-Formulierung, um eine optimale statische Allokation für eine heterogene Speicherhierarchie zu berechnen. Dabei werden globale Variablen, Variablen auf dem Programm-Stack sowie zur Laufzeit allozierte Variablen berücksichtigt. Die Arbeit zielt zwar nicht auf Echtzeitprogramme ab, setzt jedoch als einziger Ansatz in Tabelle 3.2 keine feste Speicherarchitektur in der Hardware voraus.

Die in [106; 27; 79] vorgestellten Ansätze zur Scratchpad-Allokation verwenden die WCET des längsten Kontrollflusspfades als Zielfunktion für die Optimierung. Hierzu wird der Kontrollflussgraph des Programms durch Entfernen aller rückwärtsgerichteten Kanten in einen azyklischen Graphen umgewandelt und die Berechnung des längsten (kritischen) Pfades als ILP-Modell implementiert. Die WCET jedes Basisblocks wird dabei als Funktion der Speicherzuweisung aller darin verwendeten Variablen dargestellt. Die Lösung des ILP-Problems führt (idealerweise) zu einer Speicherzuweisung mit minimaler WCET auf dem kritischen Pfad. In [79] wird darüber hinaus eine Erweiterung präsentiert, die das Zeitverhalten von Bussen mit TDMA-Arbitrierung berücksichtigt.

Der Ansatz zur dynamischen Allokation von Deverge et al. [23] verwendet eine Graphen-Darstellung des Programms und konstruiert ein ILP-Problem, das die Verschiebung beliebiger Variablen zwischen Scratchpad-Speicher und Hauptspeicher an jeder Kante des Graphen in Betracht zieht. Der kritische Pfad wird im Gegensatz zu [106] vor dem Erzeugen des ILP-Problems berechnet, sodass eventuelle Änderungen dieses Pfades in Folge der Scratchpad-Allokation zunächst nicht berücksichtigt werden. Die Autoren kompensieren dies durch iteratives Wiederholen der ILP-Optimierung bis der kritische Pfad sich nicht mehr ändert.

Die Autoren von [114] schlagen einen heuristischen Algorithmus zur dynamischen Scratchpad-Allokation vor, der nicht nur den kritischen Pfad, sondern die k längsten Kontrollflusspfade berücksichtigt. Hierdurch soll die wiederholte Neuberechnung des längsten Pfades im Gegensatz zu [23] vermieden werden.

Anders als die übrigen Ansätze untersuchen Kim et al. in [59] den Spezialfall eines Mehrkernprozessors, in dem der Hauptspeichen nicht direkt von den Prozessorkernen angesprochen werden kann. Instruktionen und Daten müssen erst durch eine DMA-Einheit in den Scratchpad-Speicher kopiert werden, bevor sie von den Prozessoren verwendet werden können. Die Autoren schlagen in diesem Zusammenhang ein ILP-basiertes sowie ein heuristisches Optimierungsverfahren

zur Allokation von Scratchpad-Speichern für Instruktionen vor. Der Ansatz zielt zwar auf Mehrkernprozessoren ab, geht jedoch von einer konstanten Laufzeit für DMA-Transfers aus, ohne die potentielle Verlangsamung durch Interferenz mit parallelen Datentransfers anderer Kerne zu berücksichtigen.

Ein WCET-optimierter Ansatz für Mehrkernprozessoren, der mehrstufige Speicherhierarchien unterstützt, wird von Liu et al. in [71] vorgeschlagen. Neben je einem privaten Scratchpad-Speicher für Daten und Instruktionen, können die Prozessorkerne auf einen weiteren, gemeinsam genutzten Scratchpad-Speicher zurückgreifen. Das vorgeschlagene dynamische Allokationsschema teilt den gemeinsamen Scratchpad-Speicher jedoch vor der Optimierung in Partitionen auf und optimiert die Allokation für jeden Prozessorkern in Isolation. Es findet also weder Datenaustausch über den gemeinsamen Scratchpad-Speicher statt, noch sind die Optimierungsprobleme der einzelnen Kerne gekoppelt, um eine Co-Optimierung zu ermöglichen.

Die von Kafshdooz et al. in [50] vorgestellte Arbeit nimmt ebenfalls eine mehrstufige Speicherhierarchie mit privaten und gemeinsam genutzten Scratchpad-Speichern an. Dabei ist die Scratchpad-Allokation in ein Verfahren zum Task-Scheduling integriert, um die Auswirkungen der Speicherallokation auf die WCET einzelner Tasks bei der Generierung eines statischen Schedules einbeziehen zu können. Im Gegensatz zu anderen Ansätzen werden die Kosten für das Verschieben von Daten zwischen den Speichern jedoch nicht berücksichtigt. Ebensowenig berücksichtigt das Optimierungsmodell zeitliche Interferenz auf gemeinsam genutzten Speichern.

Zusammengefasst setzen alle genannten Arbeiten mit Ausnahme von [4] eine festgelegte Speicherhierarchie voraus. Mehrkernprozessoren in Verbindung mit mehrstufigen Speicherhierarchien werden nur in [71] und [50] adressiert, wobei [71] das Optimierungsproblem entkoppelt und für jeden Kern separat betrachtet. In [50] wird der Speicherbedarf zwischen den Prozessorkernen gegeneinander abgewogen, das Optimierungsmodell ist jedoch vergleichsweise abstrakt und berücksichtigt weder zeitliche Interferenz noch die Kosten für das Einlesen und Zurückschreiben der Inhalte des Scratchpad-Speichers beim Übergang zwischen zwei Tasks. Es fehlt also ein WCET-optimierter Ansatz, der beliebige Speicherhierarchien unterstützt, zeitliche Interferenz berücksichtigt und den Speicherbedarf mehrerer Prozessorkerne mit feiner Granularität gegeneinander abwägt.

3.3. Automatische Software-Parallelisierung

Schon seit mehreren Jahrzehnten befasst sich die Forschung mit Ansätzen zur automatischen Software-Parallelisierung in verschiedenen Anwendungsfeldern. Das Ziel besteht i. A. darin, den Prozess der Erzeugung von parallelem Programmcode aus existierendem sequentiellem Programmcode weitgehend zu automatisieren. Seit dem verstärkten Aufkommen von Mehrkernprozessoren gewinnt diese Problemstellung zunehmend an Bedeutung, da der Entwicklungsaufwand für effiziente Software auf solchen Systemen durch die Entwurfsautomatisierung erheblich reduziert werden kann.

In parallelen Programmen außerhalb der Domäne der eingebetteten Systeme werden meist Laufzeitsysteme mit umfassender Hardware-Abstraktion verwendet. Hierbei kommen oft generische Programmierschnittstellen wie *OpenMP* [13] oder *POSIX Threads* (kurz *Pthreads*) [77] zum Einsatz, die jedoch nur bedingt für eingebettete Systeme und noch weniger für harte Echtzeitsysteme geeignet sind. In der Domäne der eingebetteten und Cyber-Physikalischen Systeme finden sich deshalb oft spezialisierte Laufzeitsysteme wie die *AUTomotive Open System ARchitecture* (kurz AUTOSAR, siehe z. B. [34]) im Automobilbereich. Häufig wird eingebettete Software aber auch ohne Betriebssystem direkt auf der Hardware ausgeführt (im Englischen auch als *bare-metal* bezeichnet). Diese Unterschiede spiegeln sich auch bei den Werkzeugen zur automatischen Parallelisierung wider, wo es eine Reihe spezialisierter Ansätze für eingebettete Software gibt. Die Lösungen außerhalb des eingebetteten Anwendungsbereichs sind im Rahmen dieser Arbeit weniger relevant, weshalb sie in der nachfolgenden Übersicht zum Stand der Technik nicht näher betrachtet werden.

Werkzeuge zur automatischen Parallelisierung im Bereich eingebetteter Systeme gliedern sich typischerweise in die folgenden drei Teilschritte:

1. Abhängigkeitsanalyse und Task-Extraktion.
2. Berechnung eines parallelen Schedules.
3. Generierung von parallelem Programmcode.

Je nach Ansatz können neben diesen Kernbestandteilen noch weitere Optimierungs- und Transformationsschritte ergänzt werden, um z. B. die Qualität der generierten Parallelisierung zu verbessern. Schritt 1 dient i. A. der Extraktion eines Task-Graphen TG (siehe Definition 2.4) aus dem sequentiellen Eingabeprogramm. Hierbei muss das Programm mit einer geeigneten Granularität in möglichst unabhängige Teile untergliedert werden, die dann die Tasks des Task-Graphen bilden. Eine Möglichkeit um dies zu bewerkstelligen ist der in Abschnitt 2.3.5 vorgestellte Hierarchische Task-Graph (HTG). Für einige Berechnungsmodelle, wie z. B. Datenflussprogramme oder Aktor-Modelle, kann

die Task-Extraktion weitgehend entfallen, da der Aufbau des Programms dort bereits eine Untergliederung in kleinere Einheiten mit bekannten Abhängigkeiten vorgibt.

Schritt 2 dient der Berechnung eines möglichst optimalen Schedules gemäß Definition 2.5 für den zuvor extrahierten Task-Graphen. Es handelt sich dabei oft um einen *statischen* Schedule, da er zur Entwurfszeit und nicht dynamisch zur Laufzeit festgelegt wird. Zur Verifikation von Echtzeitanforderungen und zur Abschätzung von Interferenzeffekten sind statische Schedules von Vorteil, da die Reihenfolge der Tasks und ihre Zuordnung zu den Prozessoren zur Entwurfszeit bekannt sind.

Der berechnete Schedule wird durch Schritt 3 im Programmcode umgesetzt, indem eine geeignete Parallelisierung des Eingabeprogramms erzeugt wird. Bestandteil dieser Programmtransformation können u. a. auch zusätzliche Optimierungsschritte sein, um z. B. feingranulare Verbesserungen vorzunehmen, die sich im Scheduling-Problem nicht oder nur eingeschränkt darstellen lassen. Hierzu zählen beispielsweise die Ausgestaltung und Platzierung von Kommunikation und Synchronisation oder die detaillierte Speicherverwaltung.

Im Folgenden werden relevante existierende Arbeiten und Werkzeuge zur automatischen Parallelisierung für eingebettete Systeme vorgestellt und anhand der drei genannten Hauptbestandteile klassifiziert. Hierbei wird zwischen Ansätzen mit und ohne Berücksichtigung harter Echtzeitanforderungen unterschieden. Letztere sind für harte Echtzeitanwendungen oft ungeeignet, da der generierte parallele Code den Anforderungen einer statischen WCET-Analyse nicht genügt. Eine besondere Rolle im Stand der Technik zu dieser Arbeit nimmt die eingangs erwähnte ARGO-Werkzeugkette ein, die eine wesentliche Grundlage der vorgestellten Beiträge bildet und deshalb ausführlicher vorgestellt wird.

3.3.1. Parallelisierung für eingebettete Systeme ohne Echtzeitanforderungen

Zu den neueren Arbeiten im Kontext der automatischen Parallelisierung von C-Programmen für eingebettete Systeme ohne harte Echtzeitanforderungen gehören [18; 105; 38; 17; 57; 69].

Kempf et al. fokussieren sich in [57] auf den Bereich der Industrieautomatisierung und untersuchen den theoretischen Speedup, der durch automatische Parallelisierung erzielt werden kann. Aufgrund des Aufbaus der Programme in dieser Domäne untersucht die Arbeit vor allem die parallele Ausführung verschiedener Funktionen und schlägt eine Methode zur Analyse der Datenabhängigkeiten über Funktionsgrenzen hinweg vor. Mit einer anschließenden Parallelitäts-Analyse

werden die theoretisch möglichen Speedups abgeschätzt, jedoch ohne eine reale Anwendung zu parallelisieren und das Ergebnis zu evaluieren. An dieser Stelle gehen andere Ansätze deutlich weiter und stellen vollständige Compiler-Werkzeuge zur Erzeugung von parallelen Programmen ohne die Beschränkung auf eine spezifische Anwendungsdomäne vor.

Cordes et al. schlagen in [18; 17] einen Ansatz zur automatischen Parallelisierung von C-Code vor, der den HTG (vgl. Abschnitt 2.3.5) zur Task-Extraktion verwendet. Die Berechnung paralleler Schedules basiert auf einem ILP-Optimierungsmodell, wobei dem Kostenmodell verschiedene Ausführungszeiten und Statistiken aus einer Simulation der Zielplattform zugrunde liegen. Zur Erzeugung des parallelen Codes wird das von Baert et al. in [5] vorgestellte Werkzeug „MPA/ATOMIUM“ verwendet. Letzteres erzeugt innerhalb des Eingabeprogramms eine Reihe von parallelen Programmabschnitten, deren genaue Ausgestaltung vom Scheduler durch eine Spezifikations-Datei vorgegeben wird. Der resultierende parallelisierte Kontrollfluss spaltet sich innerhalb der parallelen Programmabschnitte in mehrere Ausführungsstränge auf, während er ansonsten sequentiell abläuft. Bei Bedarf fügt das Werkzeug innerhalb der parallelen Abschnitte außerdem Kommunikation über FIFO-Kanäle (vgl. Abschnitt 2.2.5) ein. Die experimentellen Ergebnisse der Autoren zeigen insgesamt, dass die durchschnittliche Ausführungszeit verschiedener Algorithmen durch die HTG-basierte Parallelisierung deutlich verbessert werden kann.

Die Arbeiten von Stripf, Goulas et al. [105; 38] beschreiben die Werkzeugkette „ALMA“ zur Generierung paralleler C-Programme aus sequentiellen Eingabeprogrammen in der Programmiersprache *Scilab* [26]. Scilab ist auf den Bereich der numerischen Mathematik spezialisiert und eignet sich insbesondere für Berechnungen aus der linearen Algebra. Die Werkzeugkette nutzt eine eigens spezifizierte Architekturbeschreibungssprache (ADL), mit deren Hilfe alle für die Parallelisierung relevanten Eigenschaften der Zielplattform beschrieben werden können. Anders als [18; 17], setzt der Ansatz auf mehrere konsekutive Optimierungsschritte, die die ebenfalls ILP-basierte Optimierung des Schedules ergänzen. Dabei finden zunächst eine Reihe von Optimierungen des ursprünglichen Scilab Programms statt, bevor der Code in die Programmiersprache C übersetzt wird. Anschließend durchläuft die zugehörige Zwischendarstellung zwei aufeinanderfolgende Optimierungsphasen, zunächst zur feingranularen und dann zur grobgranularen Parallelisierung des Programms. Zu den feingranularen Optimierungen zählen u. a. die Transformation von Schleifen und die Befehlssatzabhängige Nutzung von Vektorinstruktionen. Die grobgranulare Optimierung beinhaltet dagegen die Erzeugung eines HTG sowie die Berechnung eines zugehörigen Schedules mittels ILP. Beide Optimierungsphasen können bei Bedarf auch iterativ wiederholt werden. Die Erzeugung von parallelisiertem C-Code erfolgt dann in zwei Schritten, wobei zunächst die Variablenzuordnung bestimmt

und Kommunikation eingefügt wird, bevor die abschließende Parallelisierung des Kontrollflusses erfolgt. Als Kommunikationsschnittstelle wird auf eine Teilmenge von MPI zurückgegriffen, während plattformspezifische Kommunikationsmechanismen als zukünftige Erweiterung in Betracht gezogen werden, um bessere Ergebnisse auf eingebetteten Prozessorarchitekturen zu erzielen.

Leupers et al. präsentieren in [69] die „MAPS“ Entwicklungsumgebung für eingebettete Anwendungen auf Mehrkernprozessoren mit integrierter Unterstützung für die automatische Parallelisierung von C-Programmen. Der Ansatz ist in drei Hauptbestandteile unterteilt, die im Wesentlichen den oben genannten Schritten der Task-Extraktion, der Berechnung eines Schedules sowie der Erzeugung des parallelisierten Codes entsprechen. Im Gegensatz zu HTG-basierten Arbeiten, verwendet MAPS eine Zwischendarstellung auf Basis von Prozessnetzwerken (vgl. Abschnitt 2.2.5). Das sequentielle Programm wird dabei zunächst durch eine Heuristik in möglichst unabhängige Prozesse im Sinne eines Kahn Process Networks (KPN) zerlegt. Für die erzeugten Prozesse wird anschließend ein Schedule berechnet, der festlegt, wann welcher Prozess auf welchem Prozessorkern ausgeführt wird. Hierbei werden auch Kontextwechsel durch ein Laufzeitsystem explizit zugelassen und mitberücksichtigt. In einem letzten Schritt erfolgt dann die Erzeugung von plattformspezifischem parallelem C-Code, der, wie die zuvor verwendete KPN-Zwischendarstellung, FIFO-basierte Kommunikation verwendet.

Den in diesem Unterabschnitt genannten Arbeiten ist gemein, dass sie nicht die WCET, sondern gemessene Laufzeiten aus Simulationen oder Profiling-Läufen als Grundlage für die Parallelisierung heranziehen. Für harte Echtzeitanwendungen, in denen vor allem der ungünstigste Ausführungsfall von Bedeutung ist, sind durch die so erzeugten Parallelisierungen deshalb kaum Verbesserungen zu erwarten. Außerdem weisen die Programmiermodelle der generierten parallelen Programme in [18; 105; 38; 17; 69] Eigenschaften auf, die eine akkurate WCET-Analyse zumindest bei Plattformen ohne zeitliche Isolation erschweren oder unmöglich machen. So erzeugen Cordes et al. [18; 17] z. B. zur Laufzeit neue Ausführungsstränge, die dann von einem Betriebssystem verwaltet werden müssen. Dies geht mit zusätzlicher Komplexität in der Software (durch Systemaufrufe und ggf. Speicher-Virtualisierung) einher, was bei der WCET-Analyse zu pessimistischen Ergebnissen führen kann. Hinzu kommt, dass entsprechende Betriebssysteme meist nur für Rechnerarchitekturen mit gemeinsamem Hauptspeicher verfügbar sind. Auch MAPS [69] setzt ein geeignetes Laufzeitsystem voraus, um die Kontextwechsel zu ermöglichen, was bei harten Echtzeitanwendungen dieselben Probleme nach sich ziehen kann. Die ALMA-Werkzeuge [105; 38] setzen dagegen auf die MPI-Schnittstelle, deren Implementierungen meist ebenso wenig auf harte Echtzeitsysteme ausgelegt sind.

3.3.2. Automatische Parallelisierung für Echtzeitsysteme

Aufgrund der spezifischen Anforderungen an harte Echtzeitsysteme, befassten sich einige Forschungsarbeiten der letzten Jahre mit spezialisierten Lösungen zur Generierung von parallelen Echtzeitprogrammen. Anders als bei den bisher diskutierten Ansätzen stellt dort die WCET auf einem Mehrkernprozessor das primäre Zielkriterium dar und das erzeugte parallele Programm muss statisch vorhersagbar sein. Unter den WCET-optimierten Ansätzen gibt es einige, die ein vereinfachtes Anwendungsmodell in Form von azyklischen Task-Graphen voraussetzen, während andere Werkzeuge die Parallelisierung allgemeiner Ausführungsmodelle unterstützen. In ersterem Fall vereinfacht sich das Problem der Parallelisierung erheblich, da lediglich ein Schedule für den azyklischen Graphen generiert werden muss. Durch die Forderung nach Azyklizität eignen sich diese Ansätze jedoch i. A. nicht zur Parallelisierung allgemeiner Programme, insbesondere, wenn durch Schleifen im Programm zyklische Datenabhängigkeiten entstehen.

3.3.2.1. Ansätze auf Basis von azyklischen Task-Graphen

Panić et al. schlagen in [82] einen Ansatz zur parallelen Ausführung von Anwendungen aus dem Automobilbereich vor. Der Ansatz schränkt Eingabeprogramme auf das AUTOSAR-Programmiermodell ein, welches eine Anwendung in AUTOSAR-Tasks unterteilt, die sich wiederum aus sogenannten *Runnables* zusammensetzen. Zur Parallelisierung werden die Datenabhängigkeiten zwischen Runnables analysiert, um einen azyklischen Task-Graphen (vgl. Definition 2.4) zu extrahieren. Für diesen Graphen wird eine Zuweisung der Runnables zu den Prozessorkernen berechnet, wodurch die parallele Ausführung mehrerer Runnables ermöglicht wird. Die WCET eines Runnables dient dabei als Maß für die Ausführungszeit. Kehr et al. erweitern diesen Ansatz in [54], indem sie mehrere AUTOSAR-Tasks zu sogenannten Supertasks zusammenfassen. Durch die größere Zahl von Runnables innerhalb eines solchen Supertasks kann zusätzliche Parallelität exponiert und genutzt werden, sodass sich Mehrkernprozessoren effizienter auslasten lassen. Die Umsetzung der Parallelisierung erfolgt mit Hilfe des AUTOSAR-Laufzeitsystems, das dazu entsprechend konfiguriert und erweitert wird. Durch die Beschränkung auf die Struktur der jeweiligen AUTOSAR-Anwendung lässt sich jedoch nur Parallelität nutzen, die durch die vorgegebene Aufteilung in Runnables bereits exponiert ist. Komplexe Algorithmen, die nicht auf mehrere Runnables verteilt wurden, können somit nur sequentiell ausgeführt werden. Des Weiteren setzt der Ansatz Mehrkernprozessoren mit zeitlicher Isolation (vgl. Abschnitt 3.1) voraus, da Interferenzeffekte nicht berücksichtigt werden.

Matějka et al. stellen in [73] ein Werkzeug vor, das C/C++-Code in einen azyklischen Task-Graphen umwandelt und einen zugehörigen parallelen Schedule bestimmt. Zur Vermeidung von Interferenzen werden Interferenz-freie Schedules mit einer Unterteilung der Tasks in eine Lese-, Ausführungs- und Schreibphase vorausgesetzt (vgl. Abschnitt 3.1). Der Scheduler muss dann sicherstellen, dass die Lese- und Schreibphasen niemals parallel zueinander ausgeführt werden können. In der Folge tragen nur die Ausführungsphasen der Tasks zum Gewinn der Parallelisierung bei, während die übrigen Phasen sequentiell abgearbeitet werden müssen. Der Ansatz ist außerdem darauf angewiesen, dass alle benötigten Daten während der Ausführungsphase im Cache vorgehalten werden können, weshalb der Speicherbedarf einzelner Tasks hinreichend niedrig sein muss. Hinzu kommt, dass die Autoren für die Bewertung ihres Ansatzes keine statische Analyse, sondern lediglich stichprobenartige Messungen heranziehen.

Pagetti et al. präsentieren in [81] einen Prozess zur Generierung paralleler Echtzeitanwendungen, der weitgehend auf existierende Entwicklungs- und Analysewerkzeuge zurückgreift. Die Eingabeprogramme werden auf Basis der synchronen Datenfluss-orientierten Programmiersprache Lustre [89] implementiert. Die einzelnen Programmbestandteile werden daraufhin in C-Code umgewandelt und mit dem *WCET-aware C Compiler* (WCC) [27] kompiliert. Die Parallelisierung wird durch die Berechnung eines statischen Schedules für die einzelnen Programmbestandteile erreicht. Die WCET-Analysierbarkeit wird im Wesentlichen dadurch sichergestellt, dass die generierten Schedules Interferenzeffekte vollständig vermeiden. Als Zielplattform kommt eine Mehrkernarchitektur mit TDMA-Speicherbus und lokalen Scratchpad-Speichern für jeden Kern zum Einsatz. Die Arbeit beinhaltet keine WCET-Ergebnisse, sodass die Effizienz der vorgestellten Methode nicht näher bewertet werden kann. Durch den TDMA-Bus in der Zielplattform entsteht jedoch ein zentraler Engpass, der bei Interferenzfreien Schedules für die zeitliche Isolation nicht zwingend erforderlich wäre. Es ist zu erwarten, dass dies mit entsprechenden Leistungseinbußen einhergeht.

Didier et al. präsentieren in [24] einen Ansatz zur Parallelisierung von synchronen Datenflussprogrammen, die auf einer an Lustre angelehnten Programmiersprache basieren. Im Gegensatz zu dem ebenfalls auf Lustre basierten Ansatz [81] werden dabei auch Interferenzeffekte berücksichtigt. Die Eingabeprogramme bestehen i. A. aus einer Hauptschleife, die in jeder Iteration eine Struktur aus Datenflussknoten ausführt. Jeder der Knoten wird in eine Funktion der Programmiersprache C übersetzt, kompiliert und anschließend statisch analysiert, um seine WCET in Isolation (d.h. ohne Interferenz- und Synchronisationskosten) zu bestimmen. Die Knoten des Datenflussprogramms bilden mit ihren Abhängigkeiten einen (per Definition des Programmiermodells) azyklischen Graphen, für den mittels Listen-basierter Scheduling-Verfahren ein Schedule berechnet wird. Um die Interferenz-Kosten abschätzen zu können, werden bei der Umsetzung

des Schedules zusätzliche Einschränkungen gefordert: Sofern sich die WCET-Zeitspannen zweier Datenflussknoten im Schedule nicht überlappen, so muss ihre zeitliche Reihenfolge zur Laufzeit durch Synchronisation erzwungen werden. Die Synchronisation sorgt dann dafür, dass es auch bei Laufzeiten deutlich unterhalb der WCET keine relevanten Abweichungen von dem statischen Schedule gibt. Das Ziel dieser Maßnahme besteht darin, die potentiell parallelen Berechnungen so weit einzuschränken, dass Interferenzen direkt am Schedule abgelesen werden können. Im Vergleich zu zeitgesteuerten Schedules (vgl. Abschnitt 3.1.1) reduziert dies die durchschnittliche Leerlaufzeit der Prozessoren, während die Menge der potentiell gleichzeitig ablaufenden Tasks/Berechnungen weiterhin durch den Schedule vorgegeben wird. Letzteres dient den Autoren schließlich auch als Grundlage für die Bestimmung der Interferenz-Kosten und damit der WCET des generierten Schedules.

3.3.2.2. Ansätze für allgemeinen C-Code

Das allgemeinere Problem der Parallelisierung von Programmen mit zyklischen Datenabhängigkeiten wird von den Arbeiten [103; 33; 113; 48] sowie der bereits erwähnten ARGO-Werkzeugkette [RKB+19; DPA+17] adressiert. Letztere wird aufgrund ihrer Bedeutung für die vorliegende Arbeit in dem gesonderten Abschnitt 3.3.3 im Detail vorgestellt.

Ungerer, Jahr, Frieb et al. präsentieren in [33; 113; 48] einen Werkzeug-gestützten Prozess, um Software-Entwickler bei der Parallelisierung harter Echtzeitprogramme für Mehrkernprozessoren zu unterstützen. Verglichen mit den in Abschnitt 3.3.1 genannten Arbeiten handelt es sich dabei um einen weniger stark automatisierten Ansatz. Der erste Schritt der Methodik besteht darin, das Eingabeprogramm durch ein Modell in Form eines sogenannten „Activity and Pattern Diagram“ (englisch für „Aktivitäts- und Entwurfsmuster-Diagramm“) darzustellen. Bei diesem Diagrammtyp handelt es sich um eine Abwandlung der Aktivitätsdiagramme aus dem UML2 Standard (siehe [96]). In einem darauffolgenden Schritt wird dieses Diagramm verwendet, um Parallelität innerhalb des Eingabeprogramms zu identifizieren. Diese muss dann im Aktivitätsdiagramm durch das Ersetzen der entsprechenden Aktivitäten durch *parallele Entwurfsmuster* (englisch *parallel design patterns*) aus einem Musterkatalog exponiert werden. Ein anschließender Optimierungsschritt weist die Aktivitäten jeweils einem Prozessorkern zu und fasst sie entsprechend zusammen. In einem letzten Schritt wird die Parallelisierung des Activity and Pattern Diagram manuell auf das sequentielle Eingabeprogramm übertragen. Der Software-Entwickler wird dabei durch die Bereitstellung von „Algorithmen-Skeletten“ (englisch „algorithmic skeletons“) unterstützt, die für die jeweils identifizierten Entwurfsmuster in einem Katalog hinterlegt sind. Das resultierende parallele Programm ist für die

statische WCET-Analyse grundsätzlich geeignet, die in [33] präsentierten Ergebnisse weisen jedoch auf eine deutliche Überschätzung der tatsächlichen WCET hin. Der Grund hierfür sind pessimistischen Annahmen über Interferenzeffekte, da der Ansatz keine detaillierte Interferenz-Analyse beinhaltet.

Stegmeier et al. wenden in [103] die Entwurfsmuster-basierten Parallelisierung von Jahr et al. [48] auf AUTOSAR-Anwendungen an. Hierzu werden ausgewählte AUTOSAR-Runnables mit dem Ansatz aus [48] parallelisiert und die Ergebnisse anhand des WCET-Speedups bewertet. Die WCET-Grenzen werden dabei jedoch nicht durch statische Analysen gewonnen, sondern mit einem auf Zeitmessungen basierten Werkzeug approximiert. Die resultierenden Echtzeitgarantien sind dementsprechend weniger sicher.

3.3.2.3. Diskussion

In den Ansätzen zur automatischen Parallelisierung ohne Echtzeitanforderungen aus Abschnitt 3.3.1 konnten Konzepte wie der HTG erfolgreich eingesetzt werden, um allgemeine C-Programme zu parallelisieren. Allerdings lassen sich die dabei verwendeten Optimierungsverfahren und Programmiermodelle nicht direkt auf harte Echtzeitsysteme übertragen, da insbesondere das Problem der Interferenz sowie die statische Vorhersagbarkeit der generierten Programme dort nicht berücksichtigt werden. Die Arbeiten zur Parallelisierung harter Echtzeitanwendungen aus Abschnitt 3.3.2 beschränken sich dagegen zum Großteil auf vereinfachte Anwendungsmodelle. Lediglich die Arbeiten von Ungerer, Jahr, Frieb et al. [33; 113; 48] unterstützen grundsätzlich die Parallelisierung allgemeiner Algorithmen und Programme. Die übrigen Ansätze aus Abschnitt 3.3.2.1 vereinfachen das Problem der Parallelisierung, indem sie azyklische Datenabhängigkeiten voraussetzen, um das Programm durch einen Task-Graphen darstellen zu können. Sie sind damit jedoch nicht mehr für die Parallelisierung iterativer Algorithmen mit zyklischen Abhängigkeiten oder allgemeiner Kontrollfluss-Strukturen geeignet. Abgesehen von der Arbeit von Matejka et al. [73], die einen Compiler zur Generierung des azyklischen Task-Graphen verwendet, muss außerdem bereits das Eingabeprogramm in parallelisierbare Datenflussknoten bzw. AUTOSAR-Runnables unterteilt sein. Falls keine ausreichende Zahl von Datenflussknoten/Runnables vorhanden ist, so müssen komplexere Datenflussknoten/Runnables manuell oder wie in [103] semi-automatisiert mit feinerer Granularität parallelisiert werden.

Die Ansätze in [33; 113; 48; 103] unterstützen auch zyklische Abhängigkeiten, erfordern jedoch einen hohen Anteil an manueller Entwicklungsarbeit, insbesondere bei der Identifizierung von Entwurfsmustern und der Übersetzung des originalen Programms in eine parallele Entsprechung. Anders als die hoch-automatisierten

Werkzeuge [17; 69; 38] zur Parallelisierung ohne Echtzeitanforderungen gibt es unter den genannten Arbeiten für harte Echtzeit-Software folglich keinen Ansatz, der feine Granularität und die Unterstützung zyklischer Abhängigkeiten mit einem hohen Grad an Automatisierung verbindet. Die im Folgenden vorgestellte ARGO-Werkzeugkette zielt auf ebendiese Lücke im Stand der Technik ab. Sie verbindet außerdem die automatische Parallelisierung mit einer integrierten WCET-Analyse für Mehrkernprozessoren, um Informationen aus der Code-Generierung während der statischen Analyse nutzen zu können.

Während u. a. Stripf et al. [105] einen mehrstufigen Optimierungsprozess für Programme ohne Echtzeitgarantien vorschlagen, fokussieren sich die Ansätze für harte Echtzeitsysteme vor allem auf die Bestimmung eines parallelen Schedules. Feingranulare Optimierungsschritte auf Code-Ebene, z. B. während der Generierung des parallelen Codes, finden nicht statt oder bleiben dem Software-Entwickler überlassen. Die konkrete Ausgestaltung von Kommunikation wird dabei entweder durch das Datenfluss- bzw. AUTOSAR-Programmiermodell und/oder die Entwurfsmuster bzw. Algorithmen-Skelette vorgegeben. Auch in dieser Hinsicht bedarf es also einer umfassenderen und auf Echtzeitanwendungen spezialisierten Lösung wie der ARGO-Werkzeugkette.

3.3.3. Die ARGO-Werkzeugkette

Die in dieser Arbeit vorgestellten Beiträge sind wesentliche Bestandteile der ARGO-Werkzeugkette, die den vorherigen Stand der Technik aus Abschnitt 3.3.2 um einen Quellcode-zu-Quellcode-Compiler (S2S) zur hoch-automatisierten Parallelisierung von allgemeinen Echtzeit-Programmen einschließlich iterativer Algorithmen erweitert (vgl. [RKB+19; DPA+17]). Der grundsätzliche Aufbau der Werkzeugkette ist in Abbildung 3.2 dargestellt. Die Beiträge dieser Arbeit sind hervorgehoben und in Form der *Architekturbeschreibungssprache* (ADL), der Komponenten für *Daten-Management & Synchronisation*, der Zwischendarstellung für das parallele Programm (*Parallele Programm IR*, kurz PPIR) und der *Quellcode-Erzeugung* in die Werkzeugkette integriert. Während die genannten Komponenten in den Kapiteln 4 bis 7 im Detail behandelt werden, folgt an dieser Stelle ein Überblick über die restlichen Bestandteile und die Werkzeugkette als Ganzes.

Die Eingabeprogramme können entweder als sequentieller Code in der Programmiersprache C (vgl. [46]) oder als Modell bzw. Skript in der Softwareumgebung *Scilab/Xcos* (siehe [26]) entwickelt werden. Scilab ist dabei insbesondere für Anwendungen aus der numerischen Mathematik und Xcos für Datenfluss-orientierte grafische Modelle geeignet. Alle Scilab- und Xcos-Bestandteile der Eingabe-Applikation werden im *Scilab/Xcos Front-End* als erstem Schritt der

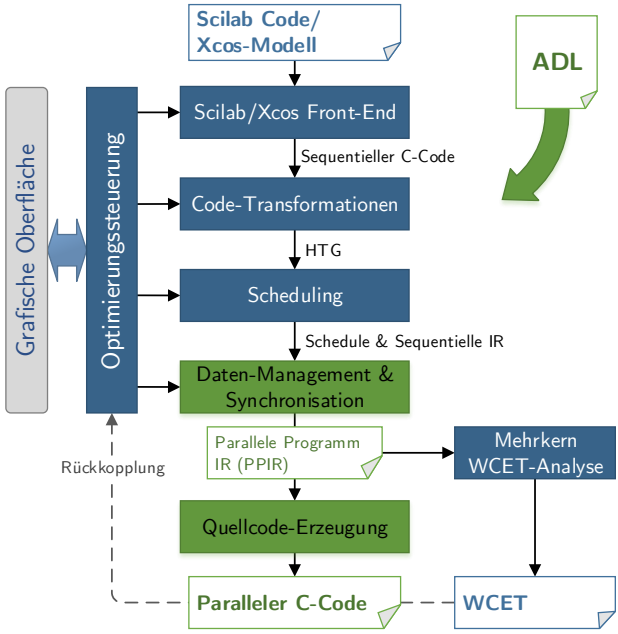


Abbildung 3.2.: Aufbau der ARGO-Werkzeugkette. Die Beschriftung der Pfeile zwischen den Einzelschritten benennt die übergebene Zwischendarstellung.

ARGO-Werkzeugkette zunächst in sequentiellen C-Code umgewandelt. Das so entstandene sequentielle C-Programm dient dann als Ausgangspunkt für die Compiler-Werkzeuge zur automatischen Parallelisierung.

Die Komponenten zur automatischen Parallelisierung basieren auf der Quellcode-zu-Quellcode-Compiler-Infrastruktur „GeCoS“ (siehe [30]), die wiederum auf dem „Eclipse Modelling Framework“ (siehe z. B. [104]) aufbaut. Die GeCoS Infrastruktur ist flexibel erweiterbar und enthält bereits eine Reihe gängiger Zwischendarstellungen für die Programmiersprache C, einschließlich eines abstrakten Syntaxbaums (AST), des Kontrollflussgraphen (CFG), der SSA-Form und des HTG. Sämtliche Komponenten der Werkzeugkette können über eine

Beschreibung der Zielplattform mit der in Kapitel 5 vorgestellten Architekturbeschreibungssprache auf Hardware-spezifische Informationen zurückgreifen.

3.3.3.1. Code-Transformationen

Nach dem Einlesen des C-Quellcodes in eine AST-Zwischendarstellung folgt ein Schritt zur Anwendung verschiedener Code-Transformationen. Dieser Schritt ergänzt die erwähnten drei Hauptschritte der automatischen Parallelisierung mit dem Ziel, die Struktur des Programms auf eine bessere Parallelisierbarkeit und Datenlokalität zu optimieren. Letztere ist im Hinblick auf das spätere parallele Programm vorteilhaft, da der voraussichtliche Kommunikationsbedarf sowie die Anfälligkeit für zeitliche Interferenzen dadurch tendenziell verringert wird. Die Code-Transformationen zielen insbesondere auf Datenstrukturen mit mehrdimensionalen Feldern (englisch *Arrays*) und die damit verbundenen verschachtelten Schleifenstrukturen ab. Mit Hilfe sogenannter Polyedermodelle als Zwischendarstellung (siehe z. B. [14; 8]) lassen sich z. B. Transformationen zur Reduktion von Array-Größen realisieren, die bereits in sequentiellen Programmen zu einer Verbesserung der WCET beitragen können [68]. Durch geschicktes Aufteilen von verschachtelten Schleifen und der darin verwendeten Array-Variablen lässt sich zudem nicht nur die Datenlokalität, sondern auch die Parallelisierbarkeit verbessern, da die einzelnen Teilschleifen später unabhängig voneinander nebenläufig ausgeführt werden können.

3.3.3.2. HTG-Erzeugung und Scheduling

Aus der sequentiellen Zwischendarstellung des C-Programms, die durch den Transformationsschritt optimiert wurde, wird mit Hilfe des HTG anschließend eine Menge von hierarchischen Tasks extrahiert. Hierzu wird das Programm zunächst in seine SSA-Form umgewandelt, um eine explizite Darstellung der Datenabhängigkeiten des Programms zu erhalten. Anschließend erfolgt die Aufteilung in Task-Knoten, wobei stark zusammenhängende Komponenten jeweils separate Hierarchieebenen bilden (vgl. Definition des HTG in Abschnitt 2.3.5). Die Abhängigkeiten zwischen den Task-Knoten gehen dann direkt aus den SSA-Definitionen und deren Verwendungen hervor, wobei in der SSA-Form zunächst nur die echten Datenabhängigkeiten eine Rolle spielen (vgl. Abschnitt 2.3.4).

Zur Berechnung eines Schedules ist außerdem ein Gewicht $w(TN_i)$ für die Task-Knoten TN_i erforderlich, um die Ausführungszeiten der Tasks zu quantifizieren (vgl. Definition 2.5). Da die ARGO-Werkzeuge in erster Linie auf eine Verbesserung der WCET abzielen, werden die Gewichte durch eine statische WCET-Analyse ermittelt. Hierzu wird aus der Zwischendarstellung C-Code mit

Assembler-Sprungmarken (englisch *Labels*) generiert, kompiliert und mit dem WCET-Analysewerkzeug aiT (siehe [29]) analysiert. Die Sprungmarken im Code erlauben es dabei, die WCET-Beiträge der Task-Knoten während der Analyse einzeln zu bestimmen, um sie später als Gewichte auf die HTG-Repräsentation übertragen zu können. Um den Effekt unterschiedlicher Speicherzugriffszeiten (z. B. in Systemen mit verteiltem Speicher) abschätzen zu können, wird die WCET-Analyse zweimal wiederholt, sodass je ein HTG-Gewicht für Zugriffe auf den schnellsten bzw. den langsamsten Speicher der Hardware-Plattform zur Verfügung steht.

Im Zusammenhang mit dem in Abschnitt 2.3.6 erwähnten *Phase-Ordering Problem* stellen die so ermittelten Gewichte $w(TN_i)$ des HTG lediglich eine *a priori* Schätzung aus dem sequentiellen Programm dar. Die nachfolgenden Schritte zu *Daten-Management & Synchronisation* sowie zur *Mehrkern WCET-Analyse* enthalten ihrerseits komplexe Optimierungs- und Analyseverfahren, deren Ergebnisse zum Zeitpunkt des Scheduling noch nicht vollständig vorhergesagt werden können. Die tatsächlichen WCET-Werte sind daher erst nach Beendigung der vollständigen Werkzeugkette bekannt und stehen beim Scheduling noch nicht zur Verfügung.

Die Scheduling-Komponente der ARGO-Werkzeugkette stellt eine Reihe verschiedener Scheduling-Algorithmen für den erzeugten HTG bereit, die sich auf den unterschiedlichen Hierarchieebenen beliebig kombinieren lassen (siehe [RKB+19; ATK+18]). Hierzu zählen Verfahren auf Basis von ILP sowie verschiedene Heuristiken wie z. B. das *Heterogeneous Earliest Finish Time* Verfahren mit *Look Ahead* (HEFT-LA).

Alle umgesetzten Scheduling-Algorithmen verfügen über eine einheitliche Schnittstelle und geben am Ende, in Übereinstimmung mit Definition 2.5, eine Startzeit $\sigma(TN_i)$ und eine Kern-Zuweisung $\mu(TN_i)$ für jeden Task-Knoten TN_i zurück. Mit diesen Informationen kann das in Kapitel 4 vorgestellte Compiler-Werkzeug automatisiert eine vorhersagbare PPIR erzeugen, die auf dem in dieser Arbeit definierten Programmiermodell aufbaut (siehe Abschnitt 4.1 und [RMB+18]). Die PPIR kann durch einen parallelen Programmgraphen (PPG) gemäß Definition 2.7 beschrieben werden.

3.3.3.3. ARGO WCET-Analyse für Mehrkernprozessoren

Die WCET-Analyse für Mehrkernprozessoren basiert auf einer Darstellung der PPIR als PPG. Für den PPG werden einige Einschränkungen vorausgesetzt, die in Abschnitt 4.1 genauer betrachtet werden. Um Interferenzeffekte zur Entwurfszeit eingrenzen zu können, muss statisch vorhergesagt werden, welche Teile des PPG potentiell parallel zueinander ausgeführt werden können. Nur die

Zugriffe solcher paralleler Programmteile auf eine gemeinsam genutzte Ressource können zur Laufzeit Interferenzen verursachen.

Da die Nachteile zeitgesteuerter Schedules vermieden werden sollen, können die Startzeiten einzelner Tasks/Programmteile bei jeder Ausführung des Programms variieren. Es ist deshalb nicht mehr möglich, die zeitliche Überlappung der Tasks in einem statischen Schedule als alleiniges Kriterium für eine potentiell gleichzeitige Ausführung heranzuziehen. Aus diesem Grund verwendet ARGO – im Gegensatz zu den in Abschnitt 3.1.1 diskutierten Ansätzen – die Synchronisationsstruktur des parallelen Programms zur Eingrenzung von Interferenz. Hierzu muss der PPG einer sogenannten „May-Happen-in-Parallel"-Analyse (kurz MHP-Analyse) unterzogen werden. Die MHP-Analyse betrachtet Programmteile genau dann als potentiell parallel, wenn sie (I) unterschiedlichen Prozessorkernen zugewiesen sind und (II) die Synchronisationsstruktur eine parallele Ausführung nicht unterbindet (siehe z. B. [92]).

Zur effizienten Durchführung der MHP-Analyse wird der PPG zunächst zum sogenannten azyklischen PPIR Graphen *aPPIR* vereinfacht. Dabei werden Basisblöcke, die nicht für die Synchronisationsstruktur relevant sind, zu größeren Code-Segmenten $cs_i \in \mathcal{CS}$ zusammengefasst [RB20a]. Zudem werden alle Schleifen aufgelöst, indem sie zweifach ausgerollt (d. h. die Basisblöcke der Schleife im CFG werden dupliziert und zusammen mit den Originalen in eine äquivalente Schleife mit halber Iterationszahl überführt) werden, während alle rückwärtsgerichteten Kanten entfallen. Hierzu muss für jeden parallelen Prozess ein reduzierbarer Kontrollflussgraph vorausgesetzt werden (vgl. Abschnitt 2.3.2). Die vollständige Definition der azyklischen PPIR gestaltet sich wie folgt:

Definition 3.1 (Azyklischer PPIR Graph)**.** *Sei* $aPPG = (V, E^{cf}, E^{sync}, TYP)$ *ein paralleler Programmgraph mit zweifach ausgerollten Schleifen und entfernten Rückwärts-Kanten. Ferner sei* $\mathcal{CS}$ *eine Menge von disjunkten Code-Segmenten* $cs_i \subseteq V$*. Dann heißt der Graph* $aPPIR = (\mathcal{CS}, E^{cfp}, E^{sync})$ *azyklischer PPIR Graph von aPPG, wenn für die Menge* E^{cfp} *aller vorwärtsgerichteten Kontrollflusskanten* $cfp_{i,j} := (cs_i, cs_j)$ *gilt:*

$$cfp_{i,j} \in E^{cfp} \iff \exists (bb_k, bb_l) \in E^{cf} \text{ mit } bb_k \in cs_i \text{ und } bb_l \in cs_j \ .$$

Die Menge $\mathcal{CS}$ *ist dabei so zu wählen, dass die folgenden Nebenbedingungen gelten:*

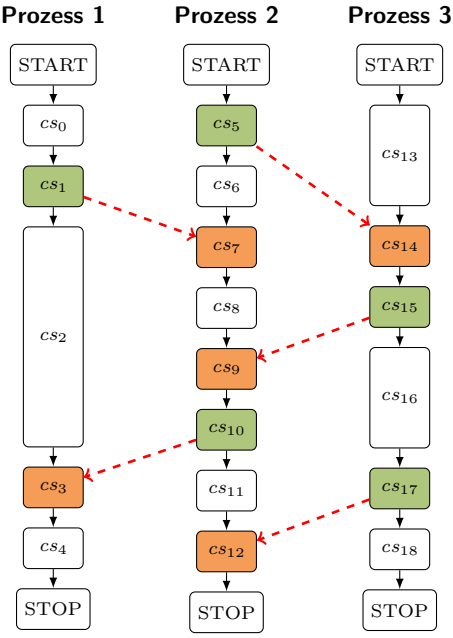
1. *Für jedes Code-Segment* cs_i *existiert maximal ein Eintrittsknoten* $bb^- \in cs_i$*, sodass* $\left\{(bb_k, bb_l) \in E^{cf} : bb_k \notin cs_i \wedge bb_l \in cs_i \wedge bb_l \neq bb^-\right\} = \emptyset$ *gilt.*

2. *Für jedes Code-Segment cs_i existiert maximal ein Austrittsknoten $bb^+ \in cs_i$, sodass $\left\{(bb_k, bb_l) \in E^{cf} : bb_k \in cs_i \wedge bb_l \notin cs_i \wedge bb_k \neq bb^+\right\} = \emptyset$ gilt.*
3. *Für Synchronisationskanten muss $sy_{i,j} := (cs_i, cs_j) \in E^{sync} \implies |cs_i| = |cs_j| = 1$ gelten.*

Bei der Konstruktion des azyklischen PPIR Graphen stellt die konkrete Wahl der Code-Segmente cs_i als Gruppe von Blöcken des PPG einen offenen Freiheitsgrad dar. Nur die Segmente, die einen Synchronisationsblock enthalten, sind durch die dritte Nebenbedingung auf eine Menge aus genau einem Block festgelegt. Um die Effizienz der MHP-Analyse zu verbessern, empfiehlt es sich, die übrigen Code-Segmente so groß wie möglich zu wählen. Aus den ersten beiden Nebenbedingungen folgt dabei, dass jedes Code-Segment eine sogenannte „Single-Entry Single-Exit"-Region (englisch für „einzelner Eingang/einzelner Ausgang", kurz SESE) mit höchstens einem Eingangs- und einem Ausgangsknoten bilden muss.

Unter Verwendung eines geeigneten Graphen *aPPIR* liefert eine MHP-Analyse für jedes Code-Segment $cs_i \in \mathcal{CS}$ eine Menge von möglicherweise parallel ausgeführten Code-Segmenten $MHP(cs_i)$ (vgl. [RKB+19]). Durch das Ausrollen der Schleifen lässt sich auch der Fall abbilden, dass mehrfach über denselben Basisblock iteriert wird, wodurch er ggf. mehr als einmal parallel zu dem betrachteten Code-Segment cs_i ausgeführt werden kann. Letzteres ist immer dann gegeben, wenn eine Schleife in einem anderen Prozess während der Ausführung von cs_i potentiell mehrere Iterationen durchläuft.

Abbildung 3.3 veranschaulicht die Ergebnisse der MHP-Analyse anhand eines Beispiels für einen azyklischen PPIR Graphen. Durchgezogene Linien repräsentieren dabei die Kontrollflusskanten der einzelnen Prozesse, und unterbrochene Linien die Synchronisationskanten. Code-Segmente mit Synchronisationsoperationen sind farblich hervorgehoben. Die Ergebnisse der MHP-Analyse für den Beispielgraphen sind in Abbildung 3.3b für alle Segmente ohne Synchronisation aufgelistet. Grundsätzlich können zwei Code-Segmente cs_i und cs_j niemals parallel zueinander ausgeführt werden, wenn zwischen den Segmenten in der *aPPIR* ein Pfad aus Kontrollfluss- und Synchronisationskanten existiert, der eine definierte Reihenfolge für die Ausführung erzwingt. Ein Sonderfall besteht bei Synchronisationsknoten, die aufgrund von Programmverzweigungen nicht notwendigerweise vom Kontrollfluss erreicht werden. Hier ist die Existenz eines Pfades zwischen cs_i und cs_j eine notwendige, aber keine hinreichende Bedingung dafür, dass die Knoten nicht parallel ausgeführt werden können. Es muss also zusätzlich geprüft werden, ob alle alternativen Kontrollflusszweige einen solchen Pfad enthalten.



(a) Azyklischer PPIR Graph für 3 Prozesse. Synchronisationskanten sind durch unterbrochene Linien dargestellt.

Code-Segment	Potentiell parallele Segmente $MHP(cs_i)$
cs_0	$cs_5, cs_6, cs_{13}, cs_{14}, cs_{15}, cs_{16}, cs_{17}, cs_{18}$
cs_2	$cs_5, cs_6, cs_8, cs_9, cs_{10}, cs_{11}, cs_{12}, cs_{13}, cs_{14}, cs_{15}, cs_{16}, cs_{17}, cs_{18}$
cs_4	$cs_{11}, cs_{12}, cs_{16}, cs_{17}, cs_{18}$
cs_6	$cs_0, cs_1, cs_2, cs_{13}, cs_{14}, cs_{15}, cs_{16}, cs_{17}, cs_{18}$
cs_8	$cs_2, cs_{13}, cs_{14}, cs_{15}, cs_{16}, cs_{17}, cs_{18}$
cs_{11}	$cs_2, cs_3, cs_4, cs_{13}, cs_{14}, cs_{15}, cs_{16}, cs_{17}, cs_{18}$
cs_{13}	$cs_0, cs_1, cs_2, cs_5, cs_6, cs_7, cs_8$
cs_{16}	$cs_0, cs_1, cs_2, cs_3, cs_4, cs_6, cs_7, cs_8, cs_9, cs_{10}, cs_{11}$
cs_{16}	$cs_0, cs_1, cs_2, cs_3, cs_4, cs_6, cs_7, cs_8, cs_9, cs_{10}, cs_{11}, cs_{12}$

(b) MHP-Ergebnisse für die Code-Segmente ohne Synchronisation

Abbildung 3.3.: Beispiel für einen azyklischen PPIR Graphen mit den zugehörigen Ergebnissen der MHP-Analyse (vgl. [RB20a])

Für Code-Segmente cs^{sync}, die eine Synchronisationsoperation enthalten, kann $MHP(cs^{sync})$ grundsätzlich analog zu den übrigen, in Abbildung 3.3b aufgeführten Segmenten bestimmt werden. Der einzige Unterschied besteht darin, dass zwei durch die Synchronisationskante $(cs_i, cs_j) \in E^{sync}$ verbundene Knoten (z. B. Sende- und Empfangsoperationen) trotz der Kante parallel zueinander ablaufen können. Es gilt also $cs_i \in MHP(cs_j)$ und $cs_j \in MHP(cs_i)$. Ob diese Annahme erforderlich ist oder nicht, hängt von der konkreten Implementierung der Sende- und Empfangsoperationen ab.

Auf Basis der aPPIR, der MHP-Ergebnisse und einer kompilierten Binärdatei des zugehörigen C-Programms läuft die Mehrkern-WCET-Analyse in den folgenden fünf Schritten ab [RKB+19]:

Bestimmung der WCET auf Kern-Ebene: Mit Hilfe des kommerziellen Analysewerkzeugs aiT [29] wird der WCET-Beitrag $WCET^{-}(cs_i)$ jedes Code-Segments cs_i in Isolation (d. h. ohne Interferenz-Kosten) bestimmt.

Bestimmung der Zugriffszahl: Ebenfalls auf Basis von aiT wird daraufhin für jedes Speichersegment m_l der Zielplattform die maximal mögliche Anzahl $N^{mem}(cs_i, m_l)$ der Speicherzugriffe innerhalb des Code-Segments cs_i ermittelt.

Interferenz-Analyse: Anhand der MHP-Ergebnisse $MHP(cs_i)$ und der Anzahl der Speicherzugriffe N^{mem} aller beteiligten Code-Segmente werden die möglichen Zugriffskonflikte für alle gemeinsam genutzten Komponenten der Zielplattform bestimmt. Durch Analyse der jeweiligen Arbitrierungsszenarien kann daraus für jedes Code-Segment cs_i die zusätzliche Verzögerungszeit $IF(cs_i)$ durch Interferenzeffekte ermittelt werden. Das zugrundeliegende Berechnungsverfahren basiert auf dem in Kapitel 5 vorgestellten Plattformmodell, weshalb die weiteren Details dort diskutiert werden.

Analyse der Synchronisationszeiten: In diesem Schritt werden die Startzeiten $\sigma(cs_i)$ und Endzeiten $\varepsilon(cs_i)$ jedes Code-Segments im ungünstigsten Ausführungsfall bestimmt, wobei auch Synchronisationskanten berücksichtigt werden. Die Endzeiten sind dabei jeweils durch $\varepsilon(cs_i) = \sigma(cs_i) + WCET^{-}(cs_i) + IF(cs_i)$ gegeben. Die ersten Code-Segmente jedes Prozesses erhalten außerdem eine Startzeit von $\sigma = 0$. Die übrigen Segmente können erst gestartet werden, sobald sie (I) vom lokalen Kontrollfluss erreicht wurden und (II) die übrigen Prozesse die Ursprungsknoten aller eingehenden Synchronisationskanten durchlaufen haben. Für den Graphen in Abbildung 3.3a kann cs_9 z. B. erst dann starten, wenn cs_8 in Prozess 1 und cs_{15} in Prozess 2 vollständig durchlaufen wurden. Im Allgemeinen gilt daher

$$\sigma(cs_i) = \max_{cs_j \in adj^{-}(cs_i)} \Big(\varepsilon(cs_j) \Big), \tag{3.1}$$

wobei $adj^{-}(cs_i)$ die Menge aller Vorgängerknoten cs_j mit einer Kante $(cs_j, cs_i) \in E^{cfp} \cup E^{sync}$ repräsentiert. Die Synchronisationszeiten $SYNC(cs_i)$, die durch das Warten auf Synchronisation zusätzlich anfallen, lassen sich dann aus der Differenz zwischen der Startzeit $\sigma(cs_i)$ und dem Maximum der Endzeiten aller Vorgänger in demselben Prozess bestimmen.

Bestimmung der WCET auf System-Ebene: Unter Berücksichtigung der Zusatzkosten für Interferenz und Synchronisation ergibt sich die WCET des gesamten Programms $WCET^{+}$ aus der WCET desjenigen parallelen Prozesses mit der längsten individuellen WCET. Letztere werden erneut mit aiT ermittelt, indem die Zusatzkosten $IF(cs_i) + SYNC(cs_i)$ für jedes Code-Segment an den entsprechenden Stellen im Maschinencode beaufschlagt werden. Dies lässt sich durch das Einbinden automatisch generierter Annotationen in die WCET-Analyse bewerkstelligen.

3.3.3.4. Iterative Optimierungssteuerung

Die iterative Optimierungssteuerung dient zum einen dazu, dass der Endnutzer mit Hilfe einer grafischen Benutzeroberfläche interaktiv in den Parallelisierungsprozess eingreifen kann. Zum anderen können die Ergebnisse eines erfolgreichen Durchlaufs der Werkzeugkette über die in Abbildung 3.2 angedeutete Möglichkeit zur Rückkopplung bei einem erneuten Durchlauf berücksichtigt werden. Hierdurch lässt sich das *Phase-Ordering Problem* (siehe Abschnitt 2.3.6) teilweise entschärfen, indem die *a posteriori* Resultate der Mehrkern-WCET-Analyse aus dem vorherigen Durchlauf z. B. bei der Erzeugung eines neuen Schedules berücksichtigt werden.

3.3.3.5. Einschränkungen und Annahmen

Um die Analysierbarkeit der generierten parallelen Programme sicherzustellen, beschränkt die Werkzeugkette die Eingabeprogramme auf eine Teilmenge des C-Standards nach ISO/IEC 9899:1999 [46]. Neben der Abwesenheit von undefiniertem Verhalten (z. B. durch Überschreitung der Grenzen von Arrays) werden für Eingabeprogramme daher die folgenden Einschränkungen gefordert:

1. Das sequentielle Eingabeprogramm muss für die statische WCET-Analyse geeignet sein.
2. Keine dynamische Speicherallokation unter Verwendung der Funktionen `malloc()` und `free()` aus der C-Standartbibliothek.
3. Keine Zeigerarithmetik abseits von Zugriffen auf mehrdimensionale Arrays.

4. Seiteneffekte von Funktionen, deren Quellcode bei der Parallelisierung nicht mit einbezogen wird, müssen manuell spezifiziert werden.
5. Die Länge von Arrays muss zu jeder Zeit bekannt sein, insbesondere auch bei der Übergabe als Funktionsargument.
6. Die maximale Anzahl der Iterationen muss für jede Schleife zur Entwurfszeit bekannt sein.
7. Keine rekursiven Funktionsaufrufe.
8. Keine Verwendung von Funktionszeigern.
9. Keine Verwendung der Sprunganweisung `goto`.

Die Einschränkungen 1, 2 und 6 sind Voraussetzung für die WCET-Analyse, weshalb sie in der Praxis auch bei rein sequentiellen Echtzeitprogrammen erforderlich sind. Die Einschränkungen 3 und 4 dienen der Analysierbarkeit von Datenabhängigkeiten bei der Erzeugung der SSA-Form, während Forderung 5 sicherstellt, dass das Datenvolumen für eingefügte Kommunikationsoperationen zur Entwurfszeit bekannt ist. Außerdem sorgen die Einschränkungen 7 bis 9 für einen statisch vorhersagbaren und reduzierbaren Kontrollflussgraphen. Dies ist insbesondere für den Aufbau des HTG von Bedeutung, da sonst u. U. keine eindeutige Hierarchie erzeugt werden kann (z. B., wenn mittels `goto` zwischen zwei Schleifen hin und her gesprungen wird).

Ein großer Teil dieser Einschränkungen gehört auch schon auf Einzelkernprozessoren zur gängigen Praxis bei der Entwicklung harter Echtzeitanwendungen. Wird der C-Code durch die ARGO-Werkzeuge aus abstrakteren Scilab/Xcos Programmen erzeugt, so werden diese Einschränkungen automatisch berücksichtigt. Im Vergleich zu den in Abschnitt 3.3.2 genannten Arbeiten auf Basis azyklischer Graphen, kann trotz dieser Einschränkungen ein erheblich breiteres Spektrum an Programmen und Algorithmen parallelisiert werden.

3.4. Zusammenfassung und Abgrenzung

An der Vielzahl von Forschungsarbeiten zu spezialisierten Entwicklungswerkzeugen für harte Echtzeit-Software zeigt sich, dass sich allgemeine Ansätze für unkritische Software oftmals nicht oder nur ineffizient auf harte Echtzeit-Software übertragen lassen. Aus diesem Grund wurden u. a. in den Teilbereichen der C-Compiler (z. B. mit dem WCC [27]), der Speicherallokation sowie der automatischen und semi-automatischen Software-Parallelisierung eine Reihe spezialisierter Ansätze für Echtzeit-Software entwickelt und erfolgreich evaluiert. Besonders für Mehrkernprozessoren gibt es jedoch nur wenige Arbeiten mit dem Ziel einer umfassenden Werkzeugunterstützung für die Entwicklung,

Optimierung und Parallelisierung von harter Echtzeit-Software. Im Bereich der Speicherallokation existieren zwar einige erfolgversprechende Ansätze zur Nutzung von Scratchpad-Speichern in Echtzeitsystemen, das Problem der Interferenz in Mehrkernprozessoren wurde dabei jedoch nicht berücksichtigt. Bei der Parallelisierung von Echtzeit-Software demonstrieren erste Ansätze, dass die WCET trotz der Interferenz-Problematik von der Nutzung mehrerer Prozessorkerne profitieren kann. Bisher ist die Parallelisierung jedoch entweder mit einem hohen Anteil an manueller Entwicklungsarbeit (im Fall der Ansätze mit Entwurfsmustern) oder einem stark spezialisierten bzw. eingeschränkten Programmiermodell für die sequentiellen Eingabeprogramme verbunden.

Auch bei der WCET-Analysierbarkeit für parallele Echtzeitprogramme auf Mehrkernprozessoren gibt einige Fortschritte in der Forschung. Zur Vermeidung bzw. Eingrenzung von Interferenzeffekten werden bisher jedoch meist Plattformen mit *zeitlicher Isolation* oder spezialisierte Scheduling-Techniken wie z. B. *zeitgesteuerte Schedules* vorausgesetzt. Beides ist, insbesondere im Hinblick auf gemischt-kritische Systeme, mit erheblichen Nachteilen für die Rechenleistung im durchschnittlichen Fall verbunden. Die Eingrenzung von Interferenz auf Basis der Synchronisationsstruktur des parallelen Programms wurde bisher hingegen kaum untersucht. Die Arbeit von Didier et al. [24] zur Parallelisierung von Datenflussprogrammen geht zwar in diese Richtung, die Synchronisationsstruktur wird hier jedoch an den Schedule angepasst, wodurch unnötig hohe Synchronisationskosten im parallelen Programm entstehen können.

Mit der in Abschnitt 3.3.3 beschriebenen ARGO-Werkzeugkette wurde erstmals der Versuch unternommen, automatische Parallelisierung für ein breites Spektrum harter Echtzeit-Software zu ermöglichen und eine entsprechend ausgelegte WCET-Analyse für Mehrkernprozessoren zu integrieren. Im Vergleich zu vorherigen Arbeiten für Echtzeitsysteme wird ein allgemeines Programmiermodell und ein hoher Automatisierungsgrad unterstützt, sodass auch die effiziente Migration von existierendem C-Code möglich wird. In einem solchen Ansatz gestalten sich alle Schritte der Parallelisierung – angefangen von der Task-Extraktion, über die Berechnung des Schedules und die Generierung des parallelen Programms bis hin zur WCET-Analyse – deutlich komplexer als z. B. in Ansätzen für azyklische Datenflussmodelle. Einer der Hauptgründe hierfür ist, dass durch die Parallelisierung von Schleifen und Kontrollflussstrukturen auch Kommunikation und Synchronisation innerhalb verschachtelter Schleifen stattfinden kann. Die Programme lassen sich dann nicht mehr durch azyklische Graphen darstellen und die Wahl der Synchronisationszeitpunkte spielt aufgrund der iterativen Wiederholung eine größere Rolle. Die Optimierung dieser Zeitpunkte wird in existierenden Arbeiten typischerweise als Teil der Berechnung des Schedules betrachtet. Dabei werden die Kommunikationskosten jedoch oft nur als zusätzlicher Beitrag zur Laufzeit/WCET eines Tasks modelliert (siehe z. B. [45;

94]). Eine Optimierung zur Neupositionierung von Kommunikation über die Task-Grenzen des HTG hinweg wurde in bisherigen Arbeiten dagegen noch nicht untersucht. Dieses bisher ungenutzte Optimierungspotential eröffnet eine zusätzliche Möglichkeit, die WCET einer Parallelisierung durch neue feingranulare Optimierungsschritte im Anschluss an die Scheduling-Phase weiter zu verbessern. Ein solcher mehrstufiger Ansatz hat außerdem den Vorteil, dass die ohnehin schon hohe Komplexität des Scheduling-Problems nicht noch weiter gesteigert wird.

Ein weiterer Aspekt, der in bisherigen Arbeiten kaum betrachtet wurde, ist die Plattformunabhängigkeit von Entwicklungswerkzeugen für parallele Echtzeit-Software. Mit Ausnahme der ARGO-Werkzeuge sind die existierenden Ansätze auf eine spezifische Hardware-Architektur ausgelegt oder nutzen vereinfachte Hardware-Modelle, die Interferenzeffekte nicht abbilden können. Plattformmodelle auf Basis von Architekturbeschreibungssprachen (ADLs) werden zwar in einigen Entwicklungswerkzeugen genutzt, keines der gängigen Formate bietet jedoch ausreichende Details, um eine plattformunabhängige Interferenz-Analyse zu ermöglichen. Im Anbetracht der zunehmenden Heterogenität in modernen Rechnerarchitekturen gewinnt diese Plattformunabhängigkeit in Entwicklungswerkzeugen zunehmend an Bedeutung, da die Software dadurch ohne großen Aufwand an eine andere Zielarchitektur angepasst werden kann. Das gilt insbesondere für die WCET-orientierte Optimierung der Speicherallokation, die in hohem Maße von der Zielplattform und den darin verfügbaren Speichersegmenten abhängt. Der Stand der Technik zur Speicherallokation in Echtzeit-Systemen bietet hierzu weder einen existierenden Ansatz, der Interferenzen berücksichtigt, noch ein plattformunabhängiges Allokationsschema für verteilte und/oder heterogene Speicherhierarchien.

Vor diesem Hintergrund, beinhaltet die vorliegende Arbeit neuartige Beiträge zum Stand der Technik, die die Generierung von WCET-optimiertem parallelem C-Code für ein allgemeines Programmiermodell mit Schleifen und Kontrollflussstrukturen bei gleichzeitiger Plattformunabhängigkeit ermöglichen (vgl. Zielsetzung der Arbeit in Abschnitt 1.2). Im Einzelnen wird der Stand der Technik durch diese Arbeit wie folgt erweitert:

1. Eine neue *Methode zur Generierung von parallelem C-Code für harte Echtzeitsysteme*, die Deadlocks per Konstruktion ausschließt und im Gegensatz zu vorherigen Ansätzen für harte Echtzeit-Software auch Kontrollflussstrukturen automatisiert parallelisieren kann.
2. Ein *paralleles Programmiermodell*, dass eine präzise Interferenz-Analyse erlaubt und Synchronisation in Schleifen/Kontrollflussstrukturen unterstützt. Im Gegensatz zum Stand der Technik werden keine zeitgesteuerten

Schedules vorausgesetzt, sodass es auch in gemischt-kritischen Systemen effizient eingesetzt werden kann.

3. Ein generisches *Hardware-Modell und ADL-Format zur detaillierten Beschreibung des Zeitverhaltens* einer Mehrkern-Rechnerarchitektur. Im Gegensatz zum Stand der Technik ist das Modell hinreichend detailliert, um die Verzögerung von Speicherzugriffen durch Interferenzeffekte in komplexen Kommunikationsverbindungen wie Networks-on-Chip (NoCs) zur Entwurfszeit berechnen zu können. Anders als bei mathematischen Netzwerk-Modellen, wie z. B. dem Netzwerk-Kalkül, werden dabei keine speziellen Zugriffsmuster oder Hardware-Einheiten zur Steuerung der Datenrate vorausgesetzt.
4. Eine neuartige *Methode zur feingranularen Optimierung von Kommunikation und Synchronisation* in parallelisierten Echtzeitprogrammen. Im Vergleich zu existierenden Arbeiten zur automatischen Parallelisierung von Echtzeit-Software findet die Optimierung nicht während der Berechnung des Schedules statt, sondern arbeitet mit feinerer Granularität auf dem Kontrollflussgraphen. Sie eignet sich damit als Ergänzung zur klassischen Optimierung des Schedules.
5. Ein *Optimierungsschema zur Speicherallokation in Echtzeitsystemen mit verteiltem Speicher*, das im Gegensatz zu existierenden Ansätzen Interferenzeffekte berücksichtigt und plattformunabhängig ist.

Kapitel 4.

Entwurfsautomatisierung für parallele Echtzeitprogramme

Werkzeuge zur Entwurfsautomatisierung spielen in der Entwicklung digitaler Systeme bereits an vielen Stellen eine entscheidende Rolle bei der Handhabung der Entwurfs-Komplexität. Auch für die parallele Programmierung echtzeitfähiger Mehrkernprozessoren ist die Entwurfsautomatisierung ein vielversprechender Ansatz, um den Entwicklungsaufwand zu reduzieren. Das gilt insbesondere dann, wenn die zu entwickelnden Programme zusätzliche Einschränkungen einhalten müssen, um z. B. eine statische Interferenz-Analyse möglich zu machen. Zu den wesentlichen Beiträgen dieser Arbeit zählt daher ein Compiler-Werkzeug mit den zugehörigen Methoden zur automatischen Generierung paralleler Echtzeitprogramme aus sequentiellem C-Code und einem vorgegebenen Schedule. In diesem Kapitel werden die dazu nötigen Einzelschritte sowie die zugehörigen Modelle und Zwischendarstellungen genauer betrachtet. Die grundlegenden Konzepte wurden in Teilen bereits in den eigenen Veröffentlichungen [RKB+19] und [RB20b] skizziert und sollen im Folgenden detaillierter dargelegt werden. Die prototypische Implementierung des Compiler-Werkzeugs ist eingebettet in den Kontext der in Abschnitt 3.3.3 beschriebenen ARGO-Werkzeugkette zur automatisierten Parallelisierung harter Echtzeitanwendungen.

Um die Zieldarstellung der parallelisierenden Transformation zu definieren, wird in Abschnitt 4.1 zunächst ein paralleles Programmiermodell für das zu erzeugende C-Programm eingeführt, das in Teilen auf der eigenen Publikation [RMB+18] basiert. Abschnitt 4.2 beschreibt daraufhin die einzelnen Schritte zur Transformation des Codes in eine parallele Zwischendarstellung. Zwei entscheidende Teilschritte des Compiler-Werkzeugs sind die Optimierung von Kommunikation bzw. Synchronisation sowie die heterogene Speicherallokation. Die dabei verwendeten Algorithmen stellen jeweils größere Beiträge dar und werden deshalb in den Kapiteln 6 und 7 gesondert betrachtet.

4.1. Ein paralleles Programmiermodell für harte Echtzeitanwendungen

4.1.1. Zielsetzung und Anforderungen

Ein wesentliches Ziel, das mit dem neuen Programmiermodell verfolgt werden soll, ist die zeitliche Vorhersagbarkeit der darauf aufbauenden parallelen Programme. Dabei sollen explizit auch Hardware-Plattformen ohne garantierte zeitliche Isolation unterstützt werden. Gleichzeitig soll sich der Ansatz auch für gemischt-kritische Systeme sowie ein breites Spektrum von Programmen und Algorithmen eignen. Existierende Programmiermodelle decken diese Anforderungen oft nicht ab, da sie entweder die Vorhersagbarkeit von Interferenzeffekten nicht berücksichtigen, keine parallele Ausführung iterativer Algorithmen erlauben oder durch zeitgesteuerte Schedules nur bedingt für gemischt-kritische Systeme geeignet sind (vgl. Abschnitt 3.1.1).

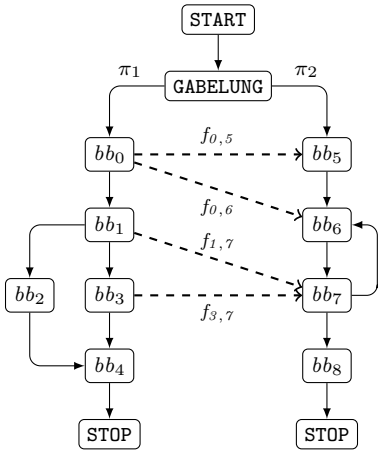


Abbildung 4.1.: Beispielhafter PPG aus zwei Prozessen π_1 und π_2 mit Kontrollfluss- und Synchronisationskanten (Pfeile mit unterbrochenen Linien)

Da das Programmiermodell eine Interferenz-Analyse auf Basis der Synchronisationsstruktur des Programms ermöglichen soll, muss eine akkurate und effiziente MHP-Analyse unterstützt werden. Das MHP-Problem gehört jedoch zu einer Reihe von statischen Analysen für parallele Programme, die im allgemeinen Fall

NP-vollständig sind. Für Synchronisationsmodelle, die auf sogenannten *Rendezvous*-Ereignissen basieren, wurde die NP-Vollständigkeit des MHP-Problems z. B. von Taylor in [109] bewiesen. Darüber hinaus zeigt Ramalingam in [92] für die Klasse der kontext- und synchronisationssensitiven Analysen bei Rendezvous-Synchronisation, dass das Problem im Allgemeinen sogar unentscheidbar ist.

Übertragen auf einen PPG gemäß Definition 2.7 kann die Möglichkeit zu einem solchen Rendezvous durch eine Synchronisationskante $sy_{i,j}$ mit Synchronisationsbedingung $f_{i,j}$ dargestellt werden. Die Synchronisationsbedingung entscheidet dann zur Laufzeit, ob ein Rendezvous zwischen den beteiligten Prozessen stattfindet oder nicht. Abbildung 4.1 veranschaulicht dies am Beispiel eines PPG, der sich zunächst in zwei parallele Prozesse gabelt, die über bedingte Synchronisationskanten miteinander interagieren. Ist z. B. die Bedingung $f_{1,7}$ erfüllt, so kann der Basisblock bb_7 erst nach dem Ende von bb_1 ausgeführt werden. Dies hätte außerdem zur Folge, dass z. B. bb_0 und bb_8 aufgrund ihrer Positionen im Kontrollflussgraphen nicht mehr parallel ausgeführt werden können. Bei nicht erfüllter Bedingung ist eine parallele Ausführung hingegen nicht ausgeschlossen. Lässt sich in solchen Fällen bei einer statischen Analyse nicht sicher bestimmen, ob die Bedingung erfüllt ist, so muss im ungünstigsten Fall von einer parallelen Ausführung und gegenseitiger Interferenz zwischen den Knoten ausgegangen werden. Noch problematischer ist die statische Vorhersage bei der Synchronisationskante von bb_3 zu bb_7, wo die Möglichkeit zur Synchronisation nicht nur von der Bedingung $f_{3,7}$ abhängt, sondern auch von der gewählten Verzweigung bei bb_1. Hinzu kommt, dass bb_7 Teil einer Schleife ist, sodass zur Entwurfszeit u. U. nicht bekannt ist, in welcher Iteration die Synchronisation stattfindet. Selbst für das einfache Beispiel aus der Abbildung ergibt sich also bereits ein relativ komplexes System aus Synchronisationsbedingungen und Kontrollflussverzweigungen, die bei einer MHP-Analyse berücksichtigt werden müssten. Um die Komplexität von MHP-Analysen ohne die übermäßige Verwendung pessimistischer Annahmen überschaubar zu halten, muss daher i. A. die Anzahl der Laufzeitentscheidungen in Bezug auf die Synchronisationsstruktur reduziert werden. Hierzu ist es erforderlich, die PPG-Darstellung des zu analysierenden Programms auf geeignete Weise einzuschränken.

Da Laufzeitentscheidungen (wie z. B. die dynamische Allokation von Speichern) zum Zeitpunkt einer statischen Analyse noch nicht festgelegt sind, stellen sie nicht nur für die MHP-Problematik einen Unsicherheitsfaktor dar. Auch in anderen Teilen der statischen WCET-Analyse muss die Unsicherheit durch pessimistische Annahmen aufgelöst werden, indem jeweils von der Entscheidungsmöglichkeit mit der höchsten Ausführungszeit ausgegangen wird. Im Gegensatz zu Entscheidungen zur Entwurfszeit, wo das Programm auf eine vorteilhafte Entscheidungsmöglichkeit festgelegt werden kann, profitiert die statisch ermittelte WCET deshalb i. A. nicht von zusätzlichen Freiheitsgraden zur Laufzeit.

Ein statisch vorhersagbares Programmiermodell sollte daher möglichst viele Entscheidungen zur Entwurfszeit festlegen.

Neben der Synchronisationsstruktur muss auch die Nutzung von Hardware-Ressourcen im Programm zur Entwurfszeit bekannt sein, um eine Aussage über Interferenz-Kosten treffen zu können. Speziell bei Plattformen mit verteiltem Speicher muss nach Möglichkeit für jeden Speicherzugriff bekannt sein, welches der verteilten Speichersegmente jeweils adressiert wird. Zusammen mit den MHP-Ergebnissen lässt sich dann die Anzahl der parallelen Zugriffe auf eine Speicherkomponente nach oben abschätzen. Diese Information bildet die Grundlage, um die Verzögerungszeiten bei der Arbitrierung und damit letztlich die Interferenz-Kosten bestimmen zu können. Dieselben Anforderungen gelten für die Nutzung von Hardware-Ressourcen zur Kommunikation zwischen den Kernen, die ebenfalls zur Entwurfszeit bekannt sein muss.

Aus den obigen Überlegungen ergeben sich insgesamt die folgenden Anforderungen an das zu spezifizierende parallele Programmiermodell:

1. Die Nutzung von Hardware-Ressourcen durch das parallele Programm muss deterministisch und mittels statischer Analysen bestimmbar sein.
2. Das Programmiermodell soll sowohl Architekturen mit verteiltem Speicher als auch die effiziente Nutzung von gemeinsamen Speichern unterstützen.
3. Die PPG-Darstellung paralleler Programme muss ausreichend stark eingeschränkt werden, um eine effiziente MHP-Analyse zu ermöglichen.
4. Laufzeitentscheidungen sollten vermieden werden, um eine pessimistische Überschätzung der WCET zu verhindern.
5. Die parallele Ausführung von Berechnungen innerhalb verschachtelter Schleifen soll unterstützt werden.
6. Das parallele Programm sollte *Race Conditions* und *Deadlocks* vollständig vermeiden.
7. Die Gesamtlaufzeit des parallelen Programms soll auch im durchschnittlichen Fall von der schnelleren Beendigung einzelner Berechnungen profitieren können (anders als bei zeitgesteuerten Schedules).
8. Die in Abschnitt 3.3.3.5 genannten Einschränkungen müssen eingehalten werden, um die generelle Analysierbarkeit der parallelen Programme sicherzustellen.

4.1.2. Spezifikation des Programmiermodells

Da Laufzeitentscheidungen für die WCET i. A. keine Vorteile mit sich bringen, setzt das vorgeschlagene Programmiermodell kein Betriebssystem und keinen Laufzeit-Scheduler voraus. Die Möglichkeit zur Verwendung eines geeigneten Echtzeitbetriebssystems (RTOS), z. B. zur die Realisierung gemischt-kritischer Systeme, bleibt jedoch bestehen. Das parallele Berechnungsmodell basiert auf *Kahn Process Networks* (kurz KPN, vgl. Abschnitt 2.2.5), deren Prozesse π_k jeweils eins zu eins einem Prozessorkern zugewiesen werden. Neben den Prozessen π_k sind FIFO-Kanäle $FIFO_{k,l} := (\pi_k, \pi_l, \nu)$ mit endlicher Kapazität ν vorgesehen. Um die verwendeten Speicheradressen möglichst deterministisch zu gestalten, muss ein paralleles Programm die Allokation der verfügbaren Speichersegmente in Software verwalten (d. h. ohne Speicher-Virtualisierung oder NUMA-Abstraktion). Hierzu repräsentiert das Programmiermodell die verfügbaren Speichersegmente durch logische Speicherbereiche, die entweder exklusiv von einem Prozess oder gemeinsam von einer definierten Menge von Prozessen verwendet werden können (vgl. [RMB+18]). Abbildung 4.2 veranschaulicht die genannten Basiskomponenten des Programmiermodells und ihre Zusammenhänge. Grundsätzlich muss zur Entwurfszeit bekannt sein, welcher Prozess welche Ressourcen (Speichersegmente und FIFO Kanäle) nutzen kann. Alle Elemente des Programmeiermodells müssen daher statisch alloziert werden.

4.1.2.1. Grundlegende Bestandteile

Die Bestandteile des Programmiermodells bilden eine Hardware-Abstraktionsschicht, die aufgrund der Anforderungen an die Vorhersagbarkeit jedoch vergleichsweise dünn ist. Abbildung 4.3 zeigt ein Beispiel für die Programmstruktur auf Applikationsebene und die Abbildung ihrer Bestandteile auf Hardware-Einheiten. Jede Komponente des Programmiermodells darf dabei ausschließlich eine zur Entwurfszeit bekannte Menge von Hardware-Einheiten verwenden. Um diese Abbildung explizit darzustellen, können die Hardware-Einheiten durch das in Kapitel 5 definierte Plattformmodell beschrieben werden. Aus den jeweiligen Arbitrierungsverfahren lassen sich dann die Interferenz-Kosten bei der Nutzung der logischen Speicher und FIFO-Kanäle im parallelen Programm bestimmen.

Durch den Aufbau des Programmiermodells sind – abhängig von der Zielplattform – grundsätzlich sowohl Nachrichten-basierte Kommunikation über die FIFO-Kanäle als auch Kommunikation über die gemeinsam genutzten Speichersegmente möglich. Hierdurch können z. B. größere Datenfelder in einem gemeinsam genutzten Speicher abgelegt werden, um unnötige Kosten für Nachrichten-basierte Kommunikation zu vermeiden. Für kleinere Datenmengen, die in schnellen privaten Speichern abgelegt sind, kann dagegen weiterhin Nachrichten-basierte

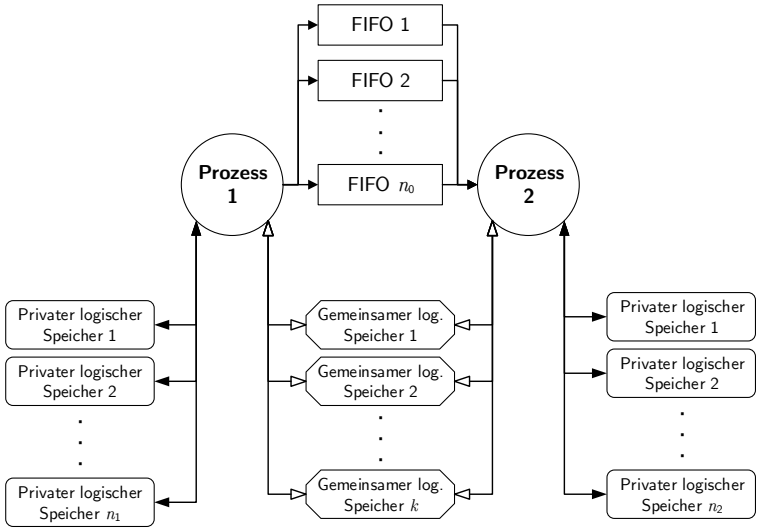


Abbildung 4.2.: Basiskomponenten des Programmiermodells (vgl. [RMB+18])

Kommunikation eingesetzt werden. Auf diese Weise lassen sich u. a. die Zugriffszahlen auf gemeinsame Speicher reduzieren und somit Interferenz-Kosten einsparen.

4.1.2.2. Interaktion zwischen Prozessen

Neben der Nachrichten-basierten Kommunikation können die FIFO-Kanäle auch zur Synchronisation verwendet werden, um z. B. *Race Conditions* bei gemeinsam genutzten Daten zu vermeiden (vgl. Abschnitt 2.2.3). Die Einhaltung der Präzedenzbedingungen lässt sich so für alle Arten von Abhängigkeiten einheitlich durch FIFO-Operationen darstellen. Als Basisoperationen auf den FIFO-Kanälen dienen hierbei die in Abschnitt 2.2.3.1 beschriebenen *Signal/Wait*-Paare sowie Sende- und Empfangsoperationen (*Send/Receive*), die neben Synchronisation auch eine Datenübertragung beinhalten. *Signal/Wait*-Operationen können dabei durch das Senden/Empfangen eines einzelnen Tokens über den FIFO-Kanal realisiert werden. Für Sende-/Empfangsoperationen wird außerdem gefordert, dass die Anzahl der übertragenen Dateneinheiten für jede Operation zur Entwurfszeit bekannt ist. Aufgrund der endlichen Kapazität ν der

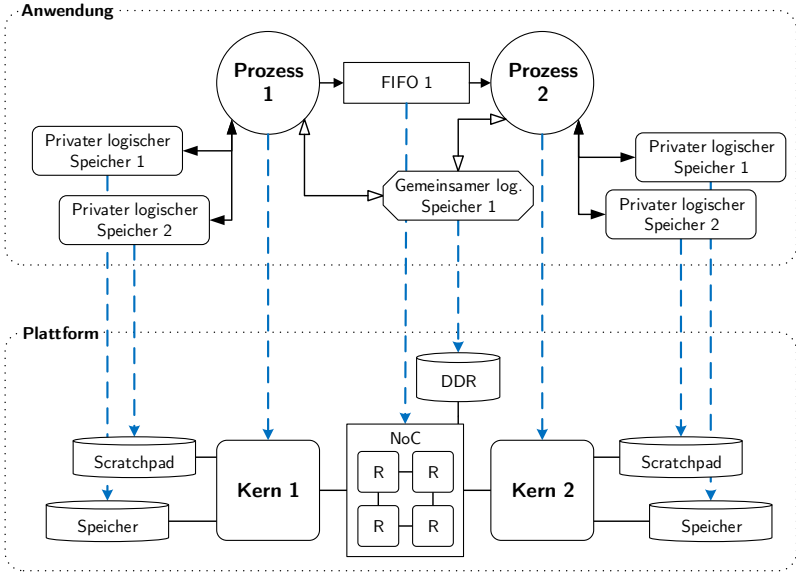


Abbildung 4.3.: Beispielhafte statische Abbildung der Komponenten des Programmiermodells auf die Hardware-Einheiten einer NoC-basierten Hardware-Plattform (vgl. [RtBM18]).

Kanäle $FIFO_{k,l} := (\pi_k, \pi_l, \nu)$ können auch *Send* bzw. *Signal* blockieren, sodass die in Abschnitt 2.2.3.1 beschriebene Mischform aus synchroner und asynchroner Kommunikation vorliegt.

Bei der Verwendung von gemeinsam genutzten Speichern müssen Programmiermodelle ein geeignetes *Speicher-Konsistenzmodell* (englisch *Memory Consistency Model*) definieren. Letzteres legt die Semantik für gemeinsame Speicherbereiche fest und spezifiziert insbesondere, wann Änderungen an gemeinsamen Daten durch einen der Prozesse/Prozessorkerne für einen zweiten Kern sichtbar werden [1]. Das Modell der *sequentiellen Konsistenz* (englisch *Sequential Consistency*) gehört zu den intuitivsten Konsistenzmodellen, bietet aber zugleich wenig Spielraum für Optimierungen. Speicherzugriffe werden bei diesem Modell einzeln und in der Reihenfolge ihrer Ausführung im lokalen Kontrollfluss des Prozesses für die anderen Kerne sichtbar. Um diese Eigenschaften zu erreichen, müssen z. B. Cache-Kohärenzprotokolle zwischen privaten Cache-Stufen vorhanden sein,

sodass alle Schreibzugriffe unverzüglich für die anderen Kerne sichtbar werden (vgl. Abschnitt 2.1.1). Ferner muss das Umsortieren von Speicherzugriffen durch den Compiler ggf. unterbunden werden.

Aufgrund der Einschränkungen bei sequentieller Speicher-Konsistenz, verwendet das im Rahmen dieser Arbeit eingesetzte Programmiermodell ein gelockertes Speicher-Konsistenzmodell (englisch *Relaxed Consistency Model*, siehe [1]). Dabei werden Synchronisationsoperationen explizit in die Speicher-Konsistenz einbezogen. Speicherzugriffe zwischen zwei Synchronisationsoperationen müssen dann keine sequentielle Konsistenz mehr einhalten, sodass sie umsortiert und zusammengefasst werden dürfen. Findet jedoch eine Synchronisation zwischen zwei Prozessen π_k und π_l statt, so müssen alle durch π_k vorgenommenen Änderungen im gemeinsam genutzten Speicher nach der Synchronisation für π_l sichtbar sein. Dies lässt sich erreichen, indem Synchronisationsoperationen mit sogenannten *Speicherbarrieren* (englisch *Memory Barrier*) verknüpft werden. Speicherbarrieren können in die Klassen ACQUIRE (englisch für *Erwerben*) und RELEASE (englisch für *Freigeben*) unterteilt werden (siehe z. B. [1; 47]). Barrieren vom Typ ACQUIRE sorgen dafür, dass nachfolgende Lesezugriffe auf gemeinsame Speicher den jeweils aktuellen Speicherinhalt lesen und nicht etwa veraltete Daten aus lokalen Caches. RELEASE-Barrieren stellen dagegen sicher, dass alle vor der Barriere vorgenommenen Änderungen an den Speicherinhalten im gemeinsamen Speicher sichtbar gemacht werden. Dies kann z. B. erforderlich sein, wenn vorher nur eine Kopie der Daten in einem lokalen Cache bearbeitet wurde, ohne die Cache-Inhalte wieder zurückzuschreiben. Durch den Einsatz eines solchen Konsistenzmodells können z. B. hardwareseitige Cache-Kohärenzprotokolle vermieden werden, indem ACQUIRE-Barrieren lokal zwischengespeicherte Daten im Cache als ungültig markieren, sodass sie aus dem Speicher nachgeladen werden müssen. Bei Caches, die lokale Änderungen nicht sofort in den Hauptspeicher schreiben, müssen außerdem RELEASE-Barrieren eingesetzt werden, um das sofortige Zurückschreiben der Daten zu veranlassen. Speicherbarrieren müssen i. A. auch vom Compiler berücksichtigt werden, um sicherzustellen, dass über Barrieren hinweg keine Speicherzugriffe umsortiert oder gemeinsam genutzte Daten in Prozessorregistern zwischengespeichert werden.

In dem vorgestellten Programmiermodell wird das Speicher-Konsistenzmodell dadurch umgesetzt, dass *Signal*-Operationen mit einer RELEASE-Barriere und *Wait*-Operationen mit einer ACQUIRE-Semantik einhergehen. Signalisiert ein Prozess π_k dann einem anderen Prozess π_l, dass Daten im gemeinsam genutzten Speicher geändert wurden, so werden durch die RELEASE-Semantik von *Signal* alle Speicherzugriffe von π_k sichtbar gemacht. Gleichzeitig sorgt *Wait* durch die ACQUIRE-Semantik dafür, dass π_l im weiteren Verlauf auch die neuesten Daten aus dem Speicher nachlädt. Abbildung 4.4 zeigt am Beispiel eines UML Sequenzdiagramms, wie sich auf diese Weise softwareseitige Cache-Kohärenz

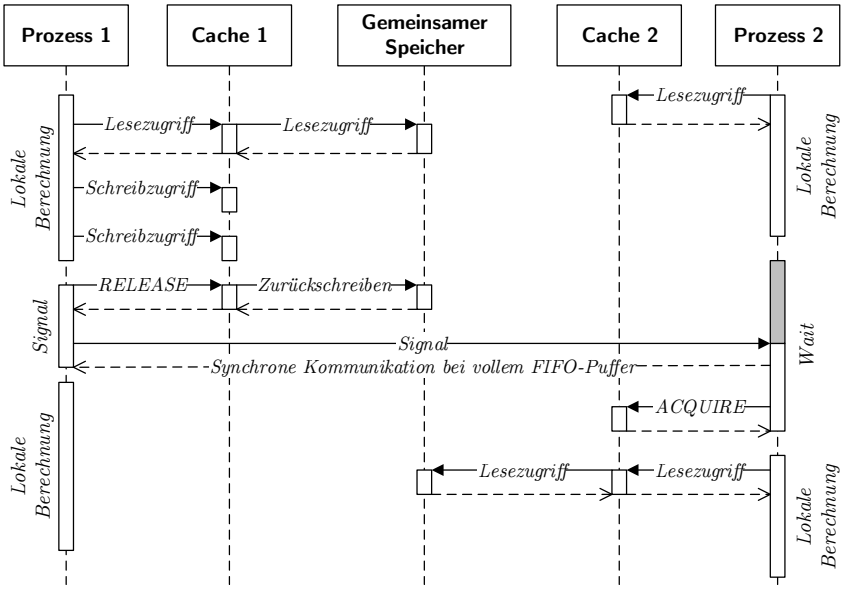


Abbildung 4.4.: *Signal/Wait*-Synchronisation mit ACQUIRE-/RELEASE-Semantik zur Realisierung von Software-gesteuerter Cache-Kohärenz

herstellen lässt, auch wenn die Hardware kein Kohärenzprotokoll bereitstellt. Die Schreibzugriffe von Prozess 1 werden dabei zunächst im lokalen Cache zwischengespeichert, bis die REALEASE-Semantik von *Signal* dafür sorgt, dass die Daten in den Speicher zurückgeschrieben werden. Auf der Seite von Prozess 2 bewirkt die *Wait*-Operation zunächst, dass die Ausführung bis zum Erhalt des Signals von Prozess 1 pausiert wird (in grau dargestellt). Anschließend werden die lokalen Cache-Inhalte von Prozess 2 durch die ACQUIRE-Semantik verworfen, sodass die nachfolgenden Lesezugriffe die aktualisierten Daten direkt aus dem gemeinsamen Speicher nachladen. Da die *Signal/Wait*-Synchronisation allein schon zur Vermeidung von *Race Conditions* benötigt wird, wird das Problem der Cache-Kohärenz durch die Verknüpfung mit den Speicherbarrieren implizit und ohne Zusatzaufwand für den Software-Entwickler mitberücksichtigt.

4.1.2.3. Analysierbarkeit der Synchronisationsstruktur

Zur Beschreibung paralleler Programme auf Basis des hier definierten Programmiermodells durch einen PPG, sollen *Signal/Wait-* und *Send/Receive*-Operationen im Folgenden jeweils durch separate PPG-Knoten dargestellt werden. Wie in Abschnitt 4.1.1 diskutiert, kann der Such-Raum einer statischen MHP-Analyse durch Kontrollflussverzweigungen, Schleifen und Synchronisationsbedingungen signifikant erhöht, und das Analyse-Problem dadurch sogar unentscheidbar werden. Für die PPG-Repräsentation der Prozesse wird daher gefordert, das zur Laufzeit keine Synchronisationsbedingungen vorhanden sein dürfen, sodass $f_{i,j} = 1$ für alle Synchronisationskanten $sy_{i,j}$ gilt. Für FIFO-Kommunikation ist dies (zumindest in Vorwärtsrichtung) implizit erfüllt, da *Wait-* bzw. *Receive*-Operationen stets so lange blockieren, bis eine fest vorgegebene Anzahl an Dateneinheiten empfangen wurde. Das Warten auf andere Prozesse findet also in jedem Fall statt und hängt damit nicht von einer dynamischen Synchronisationsbedingung ab. Da die FIFO-Operationen stets auf genau einem definierten Kanal operieren, ist es sinnvoll, darüber hinaus zu fordern, dass jede dieser Operationen mit genau einer Synchronisationskante verbunden ist. Daraus ergeben sich fest definierte Paare von *Signal/Wait-* und *Send/Receive*-Operationen, die Daten austauschen und Synchronisation herstellen können.

Die begrenzte Kapazität der FIFO-Kanäle und die damit verbundene Möglichkeit, dass auch *Signal-* bzw. *Send*-Operationen blockieren können, stellen hingegen ein Problem dar. Ist in einem FIFO-Kanal keine freie Kapazität vorhanden, so müssen Sendeoperationen warten bis der Empfänger-Prozess durch das Entnehmen von Datenelementen aus dem FIFO-Puffer wieder freien Platz schafft. Dieser Effekt stellt eine zusätzliche Möglichkeit zur Synchronisation dar, deren Zustandekommen i. A. vom Füllstand des FIFO-Kanals zur Laufzeit abhängt. Diese, bei endlichen Speicherkapazitäten kaum vermeidbare, bedingte Synchronisation muss im PPG als implizite Synchronisationskante berücksichtigt werden und widerspricht damit der Forderung $f_{i,j} = 1$. Das Problem lässt sich dadurch lösen, dass diese impliziten Kanten bei der MHP-Analyse nicht berücksichtigt werden. Die Ergebnisse der Analyse bilden dann eine sichere, wenn auch pessimistische Überschätzung der Mengen von PPG-Knoten, die potentiell parallel ablaufen können. Bei der Bestimmung der Synchronisationszeiten im Rahmen einer Mehrkern-WCET-Analyse (vgl. Abschnitt 3.3.3.3) muss die zusätzliche Möglichkeit zur Synchronisation jedoch berücksichtigt werden. Das lässt sich z. B. erreichen, indem für diese Analyse das Modell der synchronen Kommunikation angenommen wird, sodass *Signal/Wait* und *Send/Receive* in beide Richtungen eine Synchronisationskante mit $f_{i,j} = 1$ beinhalten (vgl. Abschnitt 2.2.3.1 und Abbildung 2.9). Durch die Verwendung unterschiedlicher PPG-Darstellungen bei der MHP-Analyse und der Bestimmung von Synchronisationszeiten kann in

beiden Fällen ein pessimistisches und damit sicheres Ergebnis erreicht werden, ohne dabei bedingte Synchronisation berücksichtigen zu müssen.

Für Synchronisation innerhalb von Schleifen und Kontrollflussverzweigungen (z. B. `for`- oder `if`-Blöcke in C) hängt die Häufigkeit bzw. das Zustandekommen von Synchronisation notwendigerweise von Laufzeitbedingungen ab. In dem Beispiel aus Abbildung 4.1 ist die Synchronisation zwischen bb_3 und bb_7 z. B. sowohl von der Kontrollflussverzweigung um bb_3 als auch von der Schleife um bb_7 abhängig. Eine solche Konstruktion ist nicht nur für die statische Analyse ungünstig, sondern führt bei $f_{3,7} = 1$ auch zu einem potentiellen *Deadlock*. Letzteres ist z. B. der Fall, wenn der linke Kontrollfluss den Zweig über bb_2 nimmt, während bb_7 endlos auf Synchronisation mit bb_3 wartet. Um derartige Probleme auszuschließen und die statische Vorhersagbarkeit zu erhöhen, wird im Folgenden gefordert, dass sich beide Seiten der *Signal/Wait*- und *Send/Receive*-Paare stets in einer äquivalenten Iterationsdomäne befinden müssen. Letzteres hat zur Folge, dass beide Seiten in jedem Ausführungsszenario mit gleicher Häufigkeit ausgeführt werden.

Abbildung 4.5 zeigt ein einfaches Beispiel für ein paralleles Programm, das die in diesem Abschnitt aufgestellten Forderungen erfüllt. Das Programm führt 100 Iterationen einer Berechnung aus, die sich aus zwei aufeinanderfolgenden Schritten zusammensetzt. Der zweite Schritt verarbeitet die Ergebnisse des ersten, wird aber in einem anderen Prozess ausgeführt, sodass die Daten vorher mittels *Send/Receive* übertragen werden müssen. Bei dieser als *Software Pipelining* bezeichneten Art der parallelen Verarbeitung kann die Iteration $i+1$ von Schritt 1 parallel zur i-ten Iteration von Schritt 2 berechnet werden.

Wie vom Programmiermodell gefordert, lässt die Synchronisationskante in der PPG-Zwischendarstellung aus Abbildung 4.5b eine eindeutige Zuordnung der *Send*-Operation zu einer zugehörigen *Receive*-Operation zu. Implizite Synchronisationskanten aufgrund der begrenzten FIFO-Kapazität werden hier vernachlässigt, da sie später durch die genannten pessimistischen Annahmen bei der statischen Analyse berücksichtigt werden. Sowohl die *Send*- als auch die *Receive*-Operation befinden sich in einer Schleife mit exakt 100 Iterationen, sodass auch die Forderung nach einer äquivalenten Iterationsdomäne erfüllt ist. Die `if`-Verzweigung enthält dagegen keine Synchronisation, weshalb sie nicht in beiden Prozessen vorhanden sein muss.

Da eine *reduzierbare* PPG-Darstellung gefordert wird (vgl. Abschnitt 3.3.3.5), lässt sich das Programm durch Ausrollen der Schleifen und Entfernen der Rückwärts-Kanten in eine aPPIR gemäß Definition 3.1 überführen. Hierdurch wird die MHP-Analyse und damit auch die statische WCET-Analyse des parallelen Programms mit dem in Abschnitt 3.3.3.3 beschriebenen Software-Werkzeug

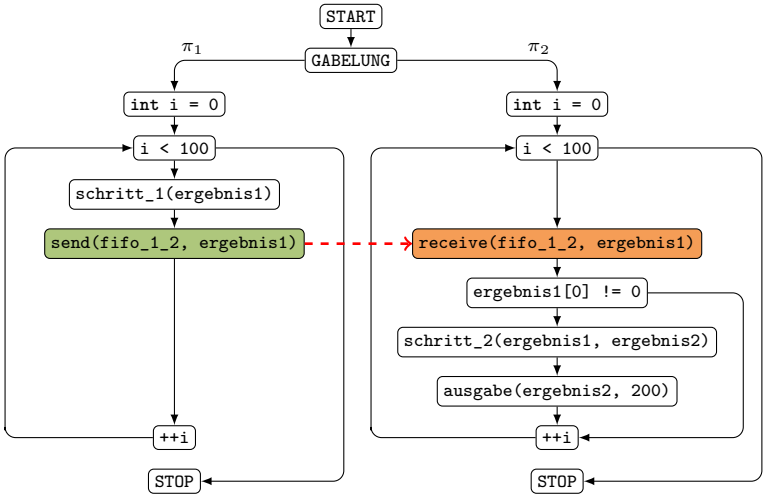
Prozess 1

```
for (int i = 0; i < 100; ++i) {
  int ergebnis1[200];
  schritt_1(ergebnis1);
  send(fifo_1_2, ergebnis1);



}
```

Prozess 2

```
for (int i = 0; i < 100; ++i) {
  int ergebnis1[200];
  int ergebnis2[200];
  receive(fifo_1_2, ergebnis1);
  if (ergebnis1[0] != 0) {
    schritt_2(ergebnis1, ergebnis2);
    ausgabe(ergebnis2, 200);
  }
}
```

(a) C-Code Beispiel



(b) PPG-Zwischendarstellung mit farblich hervorgehobenen Synchronisationsknoten

Abbildung 4.5.: Beispiel für ein paralleles C-Programm auf Basis des vorgestellten Programmiermodells mit dem zugehörigen PPG

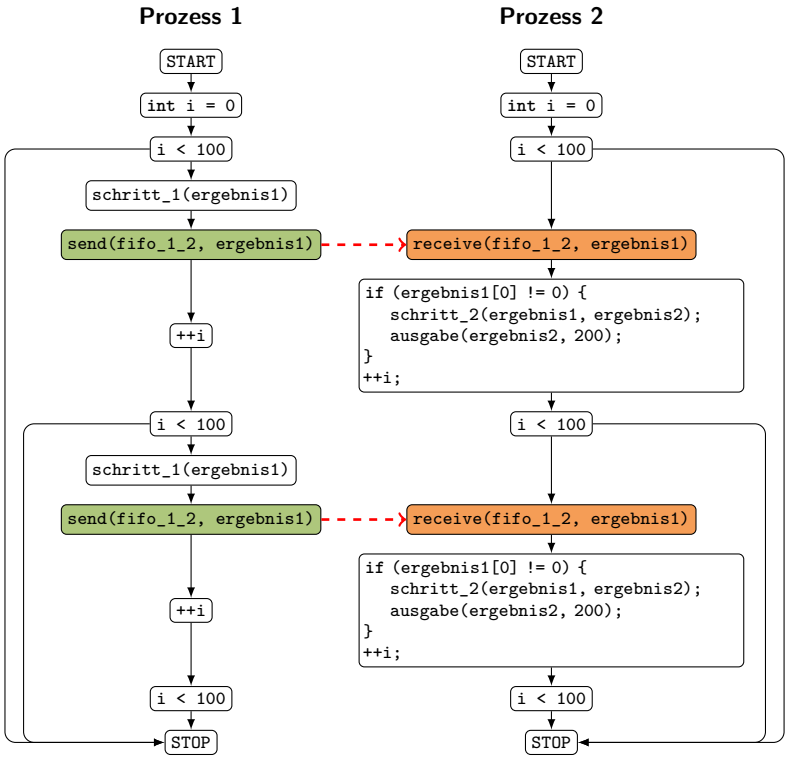


Abbildung 4.6.: Die aPPIR-Zwischendarstellungen des C-Programms aus Abbildung 4.5 mit farblich hervorgehobenen Synchronisationsknoten

ermöglicht. Das Ergebnis dieser Transformation für das Beispiel aus Abbildung 4.5b ist in Abbildung 4.6 dargestellt. Durch das Zusammenfassen möglichst großer SESE-Regionen ohne Synchronisation (vgl. Abschnitt 3.3.3.3) kann die gesamte `if`-Verzweigung zu einem Block zusammengefasst werden. Dank der ausgerollten Schleifen ist zudem gut zu erkennen, dass der zweite Aufruf von `schritt_1` – wie beim Software Pipelining vorgesehen – parallel zum ersten Aufruf von `schritt_2` ablaufen kann. Einen Sonderfall beim Ausrollen der Schleife bildet der Block mit der Schleifenbedingung „`i < 100`“, der dreifach vorhanden ist, um einen azyklischen Kontrollfluss zu erzeugen.

4.1.3. Diskussion

Insgesamt erfüllt das vorgestellte parallele Programmiermodell alle in Abschnitt 4.1.1 aufgelisteten Anforderungen. So werden alle Elemente des Programmiermodells statisch alloziert und zur Entwurfszeit jeweils einer wohldefinierten Menge von Hardware-Einheiten zugewiesen. Diese Entscheidungen finden also nicht zur Laufzeit (Anforderung 4) statt und die genutzten Hardware-Ressourcen lassen sich zur Entwurfszeit ableiten (Anforderung 1). Das Speichermodell bestehend aus logischen Speicherbereichen deckt dabei auch komplexe heterogene Speicherhierarchien aus privaten und gemeinsam genutzten Speicherkomponenten ab (Anforderung 2). Durch die FIFO-basierte Synchronisation kann ein Prozess unmittelbar nach Erhalt der benötigten Daten fortgesetzt werden, sodass die Wartezeit ausschließlich von dem tatsächlichen Sendezeitpunkt und, im Gegensatz zu zeitgesteuerten Schedules, nicht von der WCET abhängt (Anforderung 7). Weiterhin ist Synchronisation in verschachtelten Schleifen möglich, sodass iterative Algorithmen z. B. mittels Software-Pipelining parallelisiert werden können (Anforderung 5). Im Falle einer reduzierbaren PPG-Darstellung, lässt sich das parallele Programm zudem in eine aPPIR überführen, um eine effiziente MHP-Analyse zu ermöglichen (Anforderung 3).

Die Einhaltung der in Abschnitt 3.3.3.5 beschriebenen Einschränkungen sowie die Freiheit von *Race Conditions* oder *Deadlocks* können nicht per Konstruktion des Programmiermodells garantiert werden, sondern müssen bei der Entwicklung konformer Programme berücksichtigt werden. Diese Aspekte sind daher insbesondere bei dem in Abschnitt 4.2 diskutierten Aufbau der automatisierten Code-Generierung für das vorgestellte Programmiermodell zu beachten.

Während viele der häufig genutzten Programmiermodelle auf eine Vereinfachung der (manuellen) Entwicklung paralleler Software und eine definierte Semantik für Laufzeit-Scheduler abzielen, gehen die Zielsetzungen des präsentierten Programmiermodells eher in die entgegengesetzte Richtung. Da die Erzeugung des Codes von automatisierten Werkzeugen übernommen werden soll, maximiert

das Modell die statische Analysierbarkeit der Synchronisationsstruktur anstatt die manuelle Software-Entwicklung komfortabler zu gestalten. Aufgrund des Verzichts auf ein Laufzeitsystem zugunsten einer verbesserten Vorhersehbarkeit entfällt zudem die Notwendigkeit eine Semantik für das Laufzeit-Scheduling bereitzustellen. Programmiermodelle mit vergleichbarer statischer Analysierbarkeit setzen dagegen meist auf zeitliche Isolation, zeitgesteuerte Schedules und/oder feste Zeitfenster für einzelne Tasks. Da bisher nur wenige Arbeiten das Problem der Interferenz-Analysen auf Basis der Synchronisationsstruktur untersucht haben, wurde auch die Frage nach einem dafür geeigneten Programmiermodell bislang kaum betrachtet.

4.2. Erzeugung parallelisierter Programme

Das Ziel der in diesem Abschnitt vorgestellten Code-Transformation besteht darin, eine sequentielle Zwischendarstellung für die Programmiersprache C unter Berücksichtigung eines vorgegebenen Schedules in ein semantisch äquivalentes paralleles Programm umzuwandeln. Es handelt sich also im Wesentlichen um einen Quellcode-zu-Quellcode-Compiler (S2S) mit dem Ziel, einen Schedule im Hinblick auf die WCET möglichst optimal in einem parallelen Programm umzusetzen. Das generierte Zielprogramm muss dabei zu dem im vorherigen Abschnitt 4.1 definierten Programmiermodell kompatibel sein.

Für die S2S-Transformation sind insbesondere die folgenden Anforderungen an das zu generierende Programm von Bedeutung:

- Äquivalente Funktionalität im generierten parallelen Programm und dem sequentiellen Eingabeprogramm
- Vermeidung von *Deadlocks* per Konstruktion
- Vermeidung von *Race Conditions*
- Erhalt der WCET-Analysierbarkeit
- Einhaltung der Eins-zu-eins-Korrespondenz von *Signal* und *Wait* bzw. *Send* und *Receive* Operationen
- Effiziente Verwendung der FIFO-Kommunikationsoperationen
- Effiziente Allokation verteilter und gemeinsam genutzter Speichersegmente in komplexen Speicherhierarchien

Die Einhaltung dieser Anforderungen ermöglicht die Generierung effizienter und funktional korrekter paralleler Programme, deren WCET und Interferenz-Kosten statisch bestimmt werden können. Mit dem Ziel, die durch den Scheduling-Schritt getroffenen Optimierungsentscheidungen zu verfeinern bzw. zu ergänzen, sollen

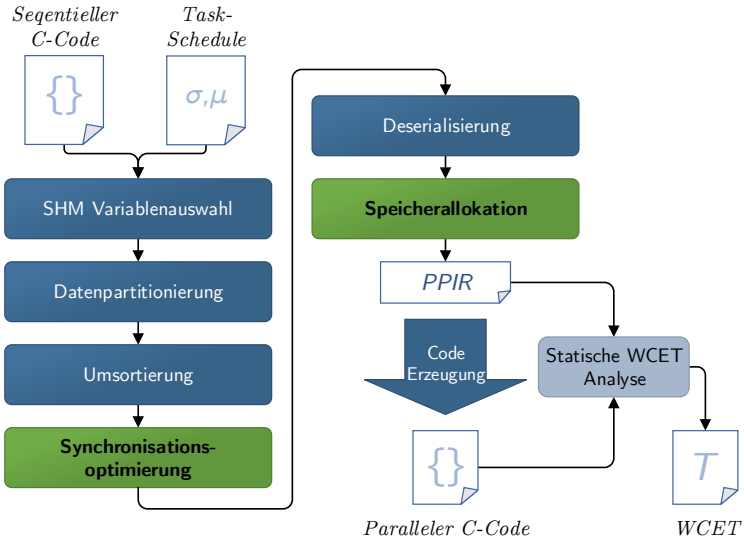


Abbildung 4.7.: Einzelschritte der Transformation in ein paralleles C-Programm (vgl. [RB20b])

neben Transformationsvorgängen explizit auch Optimierungsschritte integriert werden. Dies ermöglicht die gewinnbringende Nutzung von Freiheitsgraden auf Code-Ebene, die mit den beim Scheduling üblichen Task-Graphen nicht abgebildet werden können.

4.2.1. Aufbau des Transformations-Werkzeugs

Abbildung 4.7 zeigt den grundsätzlichen Ablauf des vorgestellten S2S-Compiler-Werkzeugs. Die grün hervorgehobenen Blöcke enthalten Optimierungsalgorithmen, die in den Kapiteln 6 und 7 im Detail betrachtet werden. Im Kontext einer Werkzeugkette zur automatischen Parallelisierung, wie der in Abbildung 3.2 dargestellten ARGO-Werkzeugkette, realisiert der S2S-Compiler die Blöcke „Daten-Management & Synchronisation“ sowie die „Quellcode-Erzeugung“. Um die Nachvollziehbarkeit und Testbarkeit sicherzustellen, findet die Transformation in mehreren Einzelschritten statt, die jeweils eine funktional korrekte Zwischendarstellung (IR) erzeugen. Durch diesen Aufbau kann bei Bedarf in verschiedenen Schritten der Transformation ausführbarer C-Quellcode erzeugt und getestet werden. Abbildung 4.8 veranschaulicht dies anhand eines Beispiels für

ein Eingabe-Programm und der Zwischenergebnisse aus den einzelnen Transformationsschritten. Die Nachvollziehbarkeit der Zwischenergebnisse kann z. B. bei der möglichen Qualifizierung eines solchen Compilers gemäß Sicherheitsstandards wie DO-330 (siehe z. B. [118]) in der Luftfahrt von Vorteil sein.

Das Code-Beispiel aus Abbildung 4.8 realisiert eine *Schnelle Fourier-Transformation* (englisch *Fast Fourier Transform*, kurz FFT), die das komplex-wertige Frequenzspektrum eines diskreten Signals mit bis zu 1024 Abtastpunkten berechnet. Eine FFT-Transformation kann stets als gewichtete Summe zweier FFT-Transformationen mit halber Anzahl an Abtastpunkten dargestellt werden (siehe [80; 31]). Dies lässt sich zur Parallelisierung nutzen, da die beiden kleineren Transformationen (im Code repräsentiert durch die Funktion `fft`) unabhängig sind und daher parallel berechnet werden können. Lediglich das Zusammenführen der Teilergebnisse (im Code durch `combine_fft` dargestellt) muss zentral auf einem Kern ausgeführt werden. In Abbildung 4.8 ist eine entsprechende Zuweisung zu zwei parallelen Prozessen im Quelltext als Kommentar vermerkt. Der Datentyp `cplx_float` stellt dabei eine geeignete Repräsentation für komplexe Zahlen bereit.

Den Ausgangspunkt für den S2S-Compiler bildet ein *sequentieller C-Code* in SSA-Form (vgl. Abschnitt 2.3.4), der die Anforderungen aus Abschnitt 3.3.3.5 erfüllt, sowie ein *Task-Schedule* für die zugehörige HTG-Darstellung. Letzterer ist gegeben durch die Funktionen σ und μ, die die Startzeit bzw. die Prozess-Zuordnung der HTG-Knoten festlegen (vgl. Definition 2.5). Die SSA-Form wird für die Analyse von Abhängigkeiten innerhalb der einzelnen Transformationsschritte verwendet. Um die SSA-Form im fertigen Programm nicht realisieren zu müssen, findet nach dem Teilschritt der Synchronisationsoptimierung eine Rücktransformation statt, die die SSA-Artefakte (z. B. die Φ-Funktionen) wieder entfernt (vgl. Abschnitt 2.3.4). Neben Eingabeprogramm und Schedule greifen die Transformationsschritte außerdem auf eine Beschreibung der Zielplattform auf Basis der in Kapitel 5 vorgestellten ADL zurück.

Ähnlich wie bei der Scheduling-Phase (vgl. Abschnitt 3.3.3), sind auch während der S2S-Transformation noch keine präzisen Informationen zur abschließenden WCET des zu generierenden Programms vorhanden. Die in Abbildung 4.8 dargestellten Optimierungsschritte müssen daher ebenfalls auf *a priori* Schätzungen der WCET-Beiträge einzelner Code-Abschnitte aus einer sequentiellen WCET-Analyse zurückgreifen.

Als Ergebnis des Transformationsprozesses wird zum einen eine PPIR-Zwischendarstellung, einschließlich der aPPIR, und zum anderen der zugehörige parallele C-Code erzeugt. Beides kann von der in Abschnitt 3.3.3.3 beschriebenen statischen WCET-Analyse verwendet werden, um eine WCET-Grenze für das parallele Programm zu ermitteln (vgl. [RKB+19]). Der kompilierte parallele

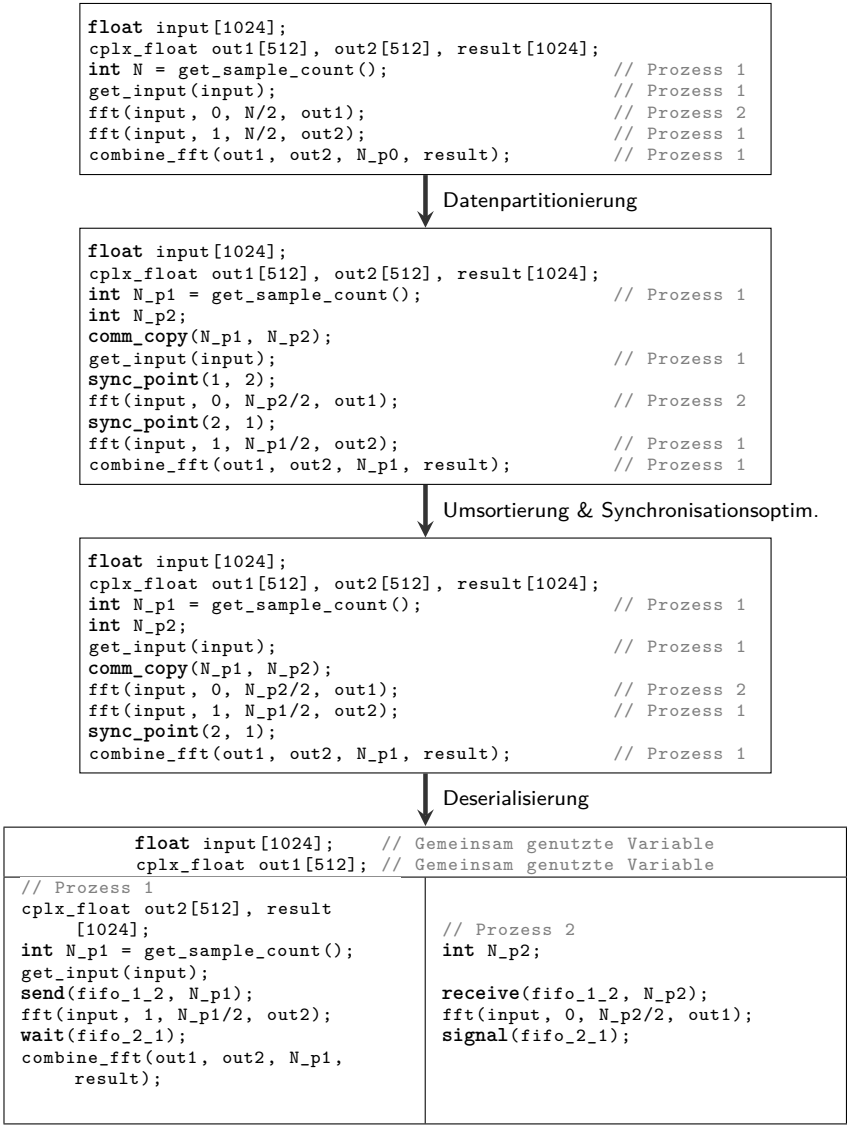


Abbildung 4.8.: Eingabe-Code und Zwischenergebnisse an verschiedenen Stellen des Transformationsprozesses aus Abbildung 4.7

C-Code dient dabei als Grundlage für die WCET-Analysen auf Code-Ebene mit Werkzeugen wie aiT [29]. Die im Rahmen der Transformation durchlaufenen Einzelschritte aus Abbildung 4.7 werden in den folgenden Unterabschnitten näher erläutert.

4.2.1.1. SHM Variablenauswahl

Im ersten Schritt des Transformationsprozesses wird auf Basis der Prozess-Zuordnung μ zunächst über die Kommunikationssemantik der einzelnen Variablen des C-Programms entschieden. Die Menge $\mathcal{V}$ aller Variablen $v_p \in \mathcal{V}$ des C-Programms werden dabei entweder der Menge $\mathcal{V}^s$ der gemeinsam genutzten Variablen oder der Menge $\mathcal{V}^p$ der privaten Variablen zugeteilt. Erstere müssen später in einem geeigneten gemeinsamen logischen Speichersegment abgelegt werden, während letztere in beliebigen Speichern platziert werden können. In dem Beispiel aus Abbildung 4.8 wird davon ausgegangen, dass die Variablen `input` und `out1` für die Platzierung in einem gemeinsamen Speicher ausgewählt wurden.

Um später ein funktionsfähiges paralleles Programm erzeugen zu können, muss die vorgenommene Einteilung der Variablen auf der Zielplattform realisierbar sein. Dazu muss u. a. sichergestellt werden, dass für die gemeinsam genutzten Variablen $v_p \in \mathcal{V}^s$ ein ausreichend großer gemeinsam nutzbarer logischer Speicher auf der Zielplattform realisiert werden kann. Auf diesen Speicher müssen alle Prozesse, die den Inhalt einer darin befindlichen Variablen $v_p \in \mathcal{V}^s$ auslesen oder modifizieren können müssen, Zugriff haben. Um dies zu gewährleisten, ist es sinnvoll, eine vollständige vorläufige Speicherallokation zu berechnen, wobei ein Optimierungsverfahren die Kosten und den Nutzen der beiden Zuweisungsoptionen $\mathcal{V}^p$ und $\mathcal{V}^s$ gegeneinander abwägt. Im Rahmen der vorgestellten Compiler-Transformation verwendet dieser Optimierungsschritt eine Abwandlung des Verfahrens, das später bei der abschließenden *Speicherallokation* (vgl. Abbildung 4.7) Verwendung findet. Das Optimierungsmodell basiert auf ganzzahliger linearer Optimierung (ILP) und wird in Kapitel 7 im Detail vorgestellt.

Die Zuteilung einer Variablen zu der Menge $\mathcal{V}^s$ ist mit der Nebenbedingung verbunden, dass der durch σ und μ definierte Schedule keine WAW- oder WAR-Abhängigkeiten (vgl. Abschnitt 2.2.2) für diese Variable verletzen darf. Andernfalls können *Race Conditions* zur Laufzeit nicht ausgeschlossen werden. Private Variablen in $\mathcal{V}^p$ werden dagegen nur von einem Prozessorkern genutzt, sodass zwar für RAW-Abhängigkeiten Kommunikation zur Datenübertragung nötig ist, bei WAW- und WAR-Abhängigkeiten jedoch nicht zwischen den Kernen synchronisiert werden muss.

Die genannte Nebenbedingung für gemeinsam genutzte Variablen kann grundsätzlich auf zwei Arten gehandhabt werden. Zum einen können WAW- und WAR-Abhängigkeiten im Task-Graphen für alle oder ggf. auch nur für einen ausgewählten Teil der Variablen als Präzedenzbedingungen beim Berechnen des Schedule berücksichtigt werden. Die zweite Option besteht darin, den Schedule ohne Berücksichtigung von WAW- und WAR-Abhängigkeiten zu berechnen, und anschließend zu überprüfen, für welche der Variablen die Nebenbedingung erfüllt ist. Die erste Option hat den Vorteil, dass die ausgewählten Variablen in jedem Fall auf gemeinsame Speicher abgebildet werden können. Dem Scheduler stehen durch die zusätzlichen Abhängigkeiten dann jedoch weniger Freiheitsgrade für die Optimierung der Parallelisierung zur Verfügung. Bei der zweiten Option bleibt es dagegen mehr oder weniger dem Zufall überlassen, ob eine Variable nach dem Scheduling noch für gemeinsam genutzte Speicher in Frage kommt. Die insgesamt sinnvollste Option besteht deshalb darin, eine Vorauswahl $\mathcal{V}^{cand} \subseteq \mathcal{V}$ geeigneter Kandidaten zu selektieren, für die WAW- und WAR-Abhängigkeiten zu berücksichtigen sind. Dies kann entweder vor der Erzeugung des Schedules oder als integraler Bestandteil der Schedule-Optimierung erfolgen. Die Co-Optimierung mit dem Schedule ist unter Optimierungsgesichtspunkten die vielversprechendere Variante. Da das Scheduling-Problem jedoch nicht im Fokus dieser Arbeit liegt, wird im Folgenden von einer Auswahl vor dem Scheduling-Schritt ausgegangen (z. B. nach der Code-Transformation in Abbildung 3.2). Dies kann durch die Definition einer Kostenfunktion erfolgen, die zum Aufbau einer sortierten Liste von Variablen verwendet wird. Aus der Liste können dann so lange die kostengünstigsten Kandidaten ausgewählt werden, bis die Kosten einen Schwellwert überschreiten. Eine einfache Kostenfunktion lässt sich über die Variablengröße und das Verhältnis von RAW-Abhängigkeiten zu den übrigen Abhängigkeiten definieren.

4.2.1.2. Datenpartitionierung

Der Schritt der Datenpartitionierung besteht im Wesentlichen aus drei Teilschritten:

1. Erzeugung privater Kopien aller Variablen $v_p \in \mathcal{V}^p$, deren Daten später in mehreren Prozessen benötigt werden.
2. Das Einfügen von Platzhaltern für spätere Kommunikations- und Synchronisationsoperationen im sequentiellen Kontrollflussgraphen.
3. Das Einfügen von Platzhaltern für Kommunikation bei Kontrollflussabhängigkeiten.

Die Transformation in Teilschritt 1 sorgt dafür, dass jede C-Variable $v_p \in \mathcal{V}^p$ (mit Ausnahme der gemeinsam genutzten Variablen $v_p \in \mathcal{V}^s$) nur von einem der parallelen Prozesse verwendet wird. Wird auf den Wert einer Variablen v_p im Eingabeprogramm von HTG-Tasks $TN_i, TN_j, \ldots$ mit unterschiedlichen Prozess-Zuweisungen $\mu(TN_i) \neq \mu(TN_j) \neq \ldots$ zugegriffen, so muss für jeden der beteiligten Prozesse eine lokale Kopie erstellt werden. Auf diese Weise wird sichergestellt, dass die Kopien in privaten Speichersegmenten abgelegt werden können. Die Übertragung der Daten zwischen den lokalen Kopien erfolgt im späteren parallelen Programm durch Nachrichten-basierte Kommunikation unter Verwendung der *Send*- und *Receive*-Operationen des Programmiermodells. Gemeinsam genutzte Variablen in $\mathcal{V}^s$ bleiben von der Transformation unberührt, da hier per Definition nur eine Instanz der Daten in einem gemeinsam genutzten Speichersegment abgelegt wird. In dem Beispiel aus Abbildung 4.8 ist die C-Variable `N` von Teilschritt 1 der Transformation betroffen. Diese Variable wird in beiden Aufrufen an `fft` verwendet, weshalb die darin gespeicherten Daten von beiden Prozessen benötigt werden. In dem Zwischenergebnis nach der Datenpartitionierung sind mit `N_p1` und `N_p2` deshalb zwei Kopien der Variablen vorhanden.

Durch die Duplikation von privaten Variablen wird der Datenfluss im sequentiellen Programm ggf. unterbrochen. Letzteres geschieht immer dann, wenn einer Definition und einer zugehörigen Verwendung in der SSA-Form unterschiedliche Kopien der ursprünglichen Variablen zugeordnet sind. Teilschritt 2 korrigiert dies, indem eine spezielle Kopieroperation `comm_copy` zwischen Definition und Verwendung eingefügt wird. Diese Operation überträgt die Daten zwischen den lokalen Kopien unterschiedlicher Prozesse und dient damit als Platzhalter für die später benötigte Nachrichten-basierte Kommunikation. In dem Beispiel aus Abbildung 4.8 ist die Definition „`int N = get_sample_count()`“ dem ersten Prozess und die zugehörige Verwendung im ersten Aufruf von `fft` dem zweiten Prozess zugewiesen. Der transformierte Quelltext enthält deshalb eine `comm_copy`-Operation, die den Wert von `N_p1` nach `N_p2` kopiert.

Um die im späteren parallelen Programm benötigte Synchronisation vollständig abzubilden, fügt Teilschritt 2 mit `sync_point` außerdem auch Platzhalter-Operationen für Synchronisation ein. Diese werden für die funktionale Korrektheit der nach wie vor sequentiellen Zwischendarstellung zwar nicht benötigt, ermöglichen jedoch in den darauffolgenden Schritten eine vereinheitlichte Betrachtung von Kommunikation und Synchronisation. Analog zu den Platzhaltern für Kommunikation, sind `sync_point`-Operationen immer dann erforderlich, wenn eine gemeinsam genutzte Variable $v_p \in \mathcal{V}^s$ von HTG-Tasks in unterschiedlichen Prozessen verwendet wird. Zur Vermeidung von *Race Conditions* nach dem Entfernen der SSA-Form im fertigen Programm, müssen neben den SSA-Definitionen und Verwendungen (d. h. RAW-Abhängigkeiten) auch WAW-

```
for (int i = 0; i < 10; ++i) {
  // Inhalt für die Prozesse 1 und 2
}
```

Eingabe

```
for (int i_p1 = 0, i_p2 = 0; i_p1 < 10, i_p2 < 10; ++i_p1, ++i_p2) {
  // Inhalt für die Prozesse 1 und 2
}
```

Ausgabe

(a) Handhabung einfacher Schleifen

```
while (komplexe_Bedingung()) {
  // Inhalt für die Prozesse 1 und 2
}
```

Eingabe

```
while (1) {
  bool b_p1 = komplexe_Bedingung();     // Prozess 1
  comm_copy(b_p1, b_p2);
  if (!b_p1, !b_p2) break;
  // Inhalt für die Prozesse 1 und 2
}
```

Ausgabe

(b) Handhabung komplexer Schleifen

Abbildung 4.9.: Beispiele für die Handhabung von Schleifen bei der Datenpartitionierung

und WAR-Abhängigkeiten berücksichtigt werden. Wenn also zwischen zwei Verwendungen einer Variablen $v_p \in \mathcal{V}^s$ mit unterschiedlicher Prozess-Zuweisung μ eine der genannten Abhängigkeitstypen besteht, so muss eine entsprechende `sync_point`-Operationen eingefügt werden. In dem Beispiel aus Abbildung 4.8 findet dies an zwei Stellen statt, wobei die erste `sync_point`-Operation die Verfügbarkeit der Eingabedaten im gemeinsamen Speicher nach dem Aufruf von `get_input(input)` an Prozess 2 signalisiert. Die zweite Synchronisation informiert Prozess 1 anschließend in die Gegenrichtung darüber, dass die Ergebnisse in `out1` verfügbar sind. Die Argumente von `sync_point` geben dabei jeweils den Index des Ursprungs- und des Zielprozesses an.

In der SSA-Form werden WAW- und WAR-Abhängigkeiten zwar grundsätzlich eliminiert (vgl. Abschnitt 2.3.4), vor der Generierung des fertigen parallelen Programms wird diese Darstellung jedoch wieder entfernt. Neben der SSA-Form ist daher eine separate Analyse zur Detektion dieser Abhängigkeitstypen erforderlich, um eine Grundlage für das Einfügen von Synchronisation bereitzustellen.

Anders als `comm_copy`-Operationen geben die Argumente von `sync_point` nur Aufschluss über die beteiligten Prozesse, nicht jedoch über die zugehörige gemeinsam genutzte Variable. Um den Bezug zwischen `sync_point` und Variablen in späteren Schritten berücksichtigen zu können, werden diese Operationen mit der Semantik einer selektiven *Speicherbarriere* (vgl. Abschnitt 4.1.2) verknüpft. Unter einer *selektiven* Barriere ist dabei eine Speicherbarriere zu verstehen, die lediglich für eine Liste bestimmter Variablen gilt. Hierdurch wird verhindert, dass spätere Compiler-Optimierungen Variablenzugriffe und Platzhalter-Operationen unzulässigerweise vertauschen können.

Das Einfügen der Platzhalter für Kommunikation/Synchronisation für eine Abhängigkeit $(bb_i, bb_j) \in V \times V$ im Kontrollflussgraphen $CFG = (V, E^{cf}, TYP)$ findet bevorzugt unmittelbar nach dem Knoten $bb_i \in V$ statt. Befinden sich die Knoten bb_i und bb_j in unterschiedlichen Iterationsdomänen (d. h. sie befinden sich nicht innerhalb derselben Schleife), so wird die Operation, soweit möglich, außerhalb der Schleifen um die Knoten platziert. Im Modell des HTG entspricht dies einer Position im Ausgabe-Knoten (vgl. Abschnitt 2.3.5) des größten hierarchischen Task-Knotens, der bb_i, nicht aber bb_j als verschachtelten Kind-Knoten enthält.

In Abbildung 4.8 nicht zu sehen ist die Handhabung von Kontrollflussabhängigkeiten in Teilschritt 3 der Transformation. Für diese Abhängigkeiten ist u. U. weitere Kommunikation nötig, um die funktionale Korrektheit des Programms sicherzustellen und den Anforderungen des Programmiermodells hinsichtlich äquivalenter Iterationsdomänen für die späteren *Signal/Wait*- und *Send/Receive*-Paare gerecht zu werden. Konkret sind davon Schleifen und Kontrollflussstrukturen (z. B. `if`-Blöcke in C) betroffen, die entweder eine `comm_copy`/`sync_point`-Operation

oder Blöcke mit Zuweisungen zu unterschiedlichen Prozessen enthalten. Grundsätzlich unterscheidet die Transformation zwischen

1. deterministischen Schleifen mit einer festen Anzahl an Iterationen und
2. Schleifen und Kontrollflussverzweigungen, bei denen die Ausführungshäufigkeit der betroffenen Basisblöcke zur Laufzeit variieren kann.

Abbildung 4.9 zeigt jeweils ein Beispiel für diese beiden Fälle. Bei ersterem handelt es sich um eine Schleife mit einer einfachen Zähl-Variablen, deren Iterationszahl durch statische Analysen zur Entwurfszeit bestimmt werden kann. Die Prüfung der Abbruchbedingung ist dort nur mit geringem Rechenaufwand verbunden, sodass es effizienter ist, die Schleifenstruktur mitsamt der Zähl-Variablen für jeden beteiligten Prozess zu duplizieren. Der resultierende Zwischencode ist beispielhaft in Abbildung 4.9a dargestellt. Im Fall von komplexeren Schleifen oder Kontrollflussverzweigungen ist das Ergebnis der Bedingung nicht zur Entwurfszeit bekannt und zu dessen Bestimmung können umfangreichere Berechnungen nötig sein. In Abbildung 4.9b ist dies durch den Aufruf von `komplexe_Bedingung` dargestellt. In einem solchen Fall ist es sinnvoll, die Bedingung in nur einem Prozess zu evaluieren und das Ergebnis dann an die übrigen Prozesse zu übermitteln. Eine beispielhafte Umsetzung im C-Code nach der Datenpartitionierung ist in Abbildung 4.9b dargestellt.

Insgesamt besteht das Ergebnis der Datenpartitionierung damit aus einem Programm mit sequentiellem Kontrollfluss, dessen Datenfelder in eine Menge gemeinsam genutzter Variablen sowie jeweils eine Menge von privaten Variablen für jeden Prozessorkern partitioniert sind (siehe Abbildung 4.8 für ein Beispiel).

4.2.1.3. Umsortierung

Nachdem die Datenfelder partitioniert wurden, entfallen durch die Duplikation der privaten Variablen alle WAW- und WAR-Abhängigkeiten mit Ausnahme derer, die bei der Generierung des Schedules explizit berücksichtigt wurden. Der Kontrollfluss kann daher in die vom Schedule vorgesehene Reihenfolge gebracht werden, ohne dabei potentiell den Datenfluss zu zerstören. Hierzu werden die zu einem HTG-Knoten gehörigen Code-Abschnitte jeweils anhand der durch σ vorgegebenen Startzeiten in aufsteigender Reihenfolge sortiert. Um die Korrektheit des Ergebnisses zu garantieren (z. B. im Falle eines fehlerhaften Schedules), kann dabei ein Sortierverfahren eingesetzt werden, dass auf paarweiser Vertauschung der Elemente basiert (z. B. das sogenannte Austauschsortieren, im Englischen auch *Bubblesort* genannt). Bei jeder dieser Vertauschungen kann dann nochmals überprüft werden, ob dies zur Verletzung einer Datenabhängigkeit oder Speicherbarriere (wie z. B. die `sync_point`-Operationen) führen würde. Diese zusätzliche

Überprüfung kann im Hinblick auf die Qualifizierbarkeit eines entsprechend aufgebauten Compiler-Werkzeugs von Vorteil sein.

4.2.1.4. Synchronisationsoptimierung

Die Positionen der Platzhalter-Operationen werden bei der Datenpartitionierung mit einem vergleichsweise einfachen Verfahren für jede Abhängigkeit in Isolation ausgewählt. Dieses Vorgehen führt jedoch i. A. zu einem suboptimalen Ergebnis. Dies zeigt sich u. a. in dem Beispiel aus Abbildung 4.8, wo sich nach der Datenpartitionierung ein `sync_point` zwischen den beiden Aufrufen von `fft` befindet. Durch diese Platzierung müsste Prozess 1 vor der lokalen FFT-Berechnung auf die Beendigung der FFT-Funktion in Prozess 2 warten, wodurch die parallele Ausführung der beiden `fft`-Aufrufe verhindert wird. Um solche Probleme zu vermeiden, ist es sinnvoll, die Positionierung der Platzhalter-Operationen in einem zusätzlichen Schritt im Detail zu verfeinern. So wäre es z. B. im Falle der `sync_point`-Operation aus Abbildung 4.8 ohne weiteres möglich, die Operation hinter den zweiten Aufruf von `fft` zu verschieben.

Nach der Umsortierung befindet sich der Kontrollfluss in seiner endgültigen Reihenfolge, sodass die Optimierung der Platzhalter-Operationen nicht mehr auf die HTG-Darstellung zurückgreifen muss. Stattdessen dient der Kontrollflussgraph als Grundlage, wodurch das globale Verschieben von Synchronisation über Iterationsdomänen hinweg ermöglicht wird (Die Beschränkung auf eine einzelne HTG-Ebene entfällt). Insgesamt besteht das Ziel der Optimierung dann darin, unnötigen Kommunikationsaufwand und Wartezeiten durch Synchronisation im späteren parallelen Programm zu vermeiden. Weiteres Optimierungspotential entsteht dadurch, dass die zuvor eingefügten Synchronisationsoperationen aufgrund der isolierten Betrachtung der Abhängigkeiten ggf. auch *redundant* sein können. In Abbildung 4.8 trifft dies nach der Datenpartitionierung z. B. auf die erste der beiden `sync_point`-Operationen zu. Bei geeigneter Platzierung können solche redundanten Operationen ohne Verlust der funktionalen Korrektheit mit einer anderen Operation zusammengefasst werden. In dem Beispiel aus Abbildung 4.8 lässt sich die redundante `sync_point`-Operation z. B. an die Position der Kommunikation `comm_copy(N_p1, N_p2)` verschieben. An dieser Stelle haben beide Operationen dieselbe synchronisierende Wirkung, da Prozess 2 dort allein schon wegen der Kommunikation mit Prozess 1 synchronisiert wird. Beide Operationen lassen sich deshalb zu einer einzelnen zusammenfassen, sodass die redundante `sync_point`-Operation eingespart werden kann.

Der zur Synchronisationsoptimierung eingesetzte Algorithmus wird in Kapitel 6 im Detail beschrieben. An dieser Stelle soll deshalb lediglich das Beispiel aus Abbildung 4.8 zur Veranschaulichung der Ergebnisse angeführt werden. Die

Optimierung ermöglicht dort die parallele Ausführung der beiden `fft`-Aufrufe sowie das Einsparen einer redundanten Synchronisationsoperation.

4.2.1.5. Deserialisierung

Die Deserialisierung erzeugt aus dem noch sequentiellen Kontrollfluss ein parallelisiertes Programm. Ausgehend von der Zuweisungsfunktion μ aus dem Scheduling-Modell werden die einzelnen Basisblöcke den jeweils vorgesehenen parallelen Prozessen zugeordnet. Die Reihenfolge von Blöcken, die demselben Prozess zugewiesen sind, bleibt dabei erhalten, da das sequentielle Programm bereits in den vorangegangenen Schritten umsortiert wurde. Wie in Abbildung 4.8 beispielhaft gezeigt, werden die bisherigen Platzhalter-Operationen in beiden beteiligten Prozessen durch entsprechende *Signal/Wait-* und *Send/-Receive*-Operationen ersetzt. Die dabei verwendeten FIFO-Kanäle werden bei Bedarf instanziiert und können dann von mehreren aufeinanderfolgenden Operationen verwendet werden. Schleifen und Kontrollflussstrukturen, die während der Datenpartitionierung bereits auf die Parallelisierung vorbereitet wurden (vgl. Abbildung 4.9), werden in allen daran beteiligten Prozessen dupliziert. Bei einfachen Schleifen wird dabei in jedem Prozess die jeweils passende Zähl-Variable verwendet. Für das Ausgabeprogramm aus Abbildung 4.9a würde das bedeuten, dass in dem Duplikat für Prozess 2 alle Ausdrücke mit `i_p1` entfallen.

Die Bezeichnung als *Deserialisierung* impliziert bereits, dass es sich bei diesem Schritt um die inverse Transformation zu der in Abschnitt 2.2.4 diskutierten Serialisierung von parallelen Programmen handelt. Das erzeugte Programm ist daher per Definition *serialisierbar*, woraus gemäß [97] die Abwesenheit von Deadlocks folgt. Durch die späte Deserialisierung kann also die Forderung nach der *Abwesenheit von Deadlocks per Konstruktion* erfüllt werden, da die nachfolgenden Schritte der S2S-Transformation keine Änderungen mehr am PPG und der Synchronisationsstruktur vornehmen.

4.2.1.6. Speicherallokation

Die Speicherallokation stellt den letzten größeren Schritt der S2S-Transformation vor der Erzeugung des parallelen C-Codes dar. Das Ziel dieses Schrittes besteht darin, die Zuordnung der Variablen des parallelen Programms zu den verfügbaren logischen Speichersegmenten möglichst optimal festzulegen. Die zuvor festgelegte Menge $\mathcal{V}^s$ der gemeinsam genutzten Variablen bildet dabei eine Nebenbedingung, die die Zuordnung der darin enthaltenen Variablen zu einem geeigneten gemeinsam genutzten Speichersegment erzwingt. In dem Beispiel aus

Abbildung 4.8 müssen z. B. die Variablen `input` und `out1` in einem für Prozess 1 und Prozess 2 zugänglichen logischen Speichersegment abgelegt werden.

Die optimale Belegung der Speicherbereiche wird durch ganzzahlige lineare Optimierung berechnet und bezieht dabei auch die Kosten für Interferenzeffekte mit ein. Das verwendete ILP-Modell wird in Kapitel 7 im Detail vorgestellt. Das Ergebnis der Optimierung sind statische Speicheradressen für alle Variablen, die nicht dem primären Programm-Stack zugeordnet sind. Der Quellcode wird daraufhin so transformiert, dass bei Zugriffen auf die betreffenden Variablen jeweils die zugehörigen Adressbereiche verwendet werden. Durch die Integration dieser festen Adressen in den Quelltext können die Informationen, anders als bei gängigen Allokationsroutinen wie `malloc`, nach der Kompilierung einfacher aus der erzeugten Binärdatei extrahiert werden. Dies erhöht die Präzision der Datenwert-Analyse innerhalb von Werkzeugen zur statischen WCET-Schätzung (vgl. Abschnitt 2.4.1).

4.2.2. Diskussion

Die präsentierte S2S-Transformation erfüllt die zuvor definierten Anforderungen durch eine Abfolge von Einzelschritten mit verzögerter Deserialisierung. Letztere stellt die geforderte Abwesenheit von Deadlocks und die Eins-zu-eins-Korrespondenz von Synchronisationsoperationen sicher. Alle Schritte, die Änderungen an der Synchronisationsstruktur vornehmen, verwenden zunächst Platzhalter-Operationen, die für Paare aus *Signal/Wait* bzw. *Send/Receive* stehen. So bleibt die Eins-zu-eins-Korrespondenz dieser Operationen während der Transformation per Konstruktion erhalten. Um die funktionale Äquivalenz der generierten Programme besser verifizieren zu können, kann nach jedem Teilschritt ein funktionsfähiges C-Programm generiert und getestet werden. *Race Condidions* werden durch das Einbeziehen von WAW- und WAR-Abhängigkeiten bei der Datenpartitionierung vermieden. Die Handhabung von Schleifen während der Transformation stellt sicher, dass die Anzahl der Iterationen unverändert bleibt. Bei komplexen Schleifenbedingungen wie in Abbildung 4.9b, kann die Transformation allerdings dazu führen, dass die Bestimmung von Schleifengrenzen während der WCET-Analyse nur mit Zusatzinformationen aus anderen Prozessen möglich ist. Da solche prozessübergreifenden Analysen aufwändig sind, enthält die Transformation einen Mechanismus, der Informationen zu Schleifengrenzen aus dem Eingabeprogramm über die Einzelschritte hinweg in das parallele Zielprogramm überträgt. Auf Basis dieser Informationen können später entsprechende Annotationen für die statische WCET-Analyse erzeugt werden. Auf diese Weise bleibt die WCET-Analysierbarkeit des Eingabeprogramms über die Transformationsschritte hinweg erhalten.

Die Effizienz der erzeugten parallelen Programme hinsichtlich der FIFO-Kommunikation und der Speichernutzung kann durch die beiden Optimierungsschritte (*Synchronisationsoptimierung* und *Speicherallokation*) im Transformationsablauf verbessert werden. Bedingt durch das *Phase-Ordering Problem* (siehe Abschnitt 2.3.6) verfolgt die S2S-Transformation für beide Optimierungsschritte einen zweistufigen Ansatz. Zunächst wird mit der *SHM Variablenauswahl* und der *Datenpartitionierung* eine vorläufige Speicherallokation bzw. Synchronisationsplatzierung erzeugt, um die notwendigen Vorentscheidungen zu treffen. Erst in den späteren Optimierungsphasen findet auf Basis der zusätzlich verfügbaren Informationen (u. a. des umsortierten CFG bei der *Synchronisationsoptimierung* und der Synchronisationsstruktur bei der *Speicherallokation*) eine abschließende Entscheidung über das Endergebnis statt.

In Vergleich zu der Erzeugung paralleler Programme in existierenden Ansätzen erfordern die Einschränkungen des in Abschnitt 4.1 definierten Programmiermodells einen spezialisierten Ansatz, der die MHP-Analysierbarkeit sicherstellt. Hierzu müssen insbesondere die geforderten Einschränkungen der Synchronisationsstruktur bei der Konstruktion des parallelen Programms berücksichtigt werden. Im Gegensatz zu dem in Abschnitt 3.3.2 vorgestellten Stand der Technik bei der Parallelisierung harter Echtzeit-Software, ermöglicht der vorgestellte Ansatz damit sowohl eine präzise statische Analyse von Interferenzeffekten als auch die feingranulare Parallelisierung iterativer Algorithmen mit Hilfe des HTG. Manuelle Eingriffe durch den Endnutzer sind nicht erforderlich, sodass ein höher Automatisierungsgrad als in den Entwurfsmuster-basierten Ansätzen erreicht wird. Davon unberührt bleibt die Möglichkeit, Vorgaben des Endnutzers, z. B. bei der Auswahl der gemeinsam genutzten Variablen oder der Zuweisung bestimmter Variablen zu bestimmten Speicherbereichen, in den Prozess einfließen zu lassen.

4.3. Fazit

In diesem Kapitel wurde ein S2S-Compiler-Werkzeug zusammen mit einem vorhersagbaren parallelen Programmiermodell für harte Echtzeitsysteme vorgestellt. In Kombination tragen beide Komponenten signifikant zur Lösung einiger offener Problemstellungen bei. Dies betrifft insbesondere:

1. Die Reduktion des Entwicklungsaufwands für parallele Echtzeitprogramme durch Entwurfsautomatisierung.
2. Die Bereitstellung einer Grundlage für die statische Interferenz-Analyse auf Basis von MHP-Informationen.
3. Die Möglichkeit, parallele Echtzeitprogramme ohne größeren Aufwand für verschiedene Zielplattformen zu optimieren und zu evaluieren.

Integriert in eine Werkzeugkette wie ARGO (siehe Abschnitt 3.3.3), ermöglichen die in diesem Kapitel präsentierten Beiträge einen Entwicklungsprozess für parallele Echtzeitprogramme, der kaum aufwändiger ist als die Software-Entwicklung für Einzelkernprozessoren. Der Endanwender kann die Software zunächst als sequentielles C-Programm entwickeln, ohne dabei die Komplexität von Mehrkernprozessoren mit Race Conditions, Kommunikation, Synchronisation und zeitlichen Interferenzen berücksichtigen zu müssen. Diese Aspekte werden auf den nachfolgenden Schritt der Parallelisierung verlagert, wo sie durch die vorgestellte Compiler-Transformation automatisiert berücksichtigt werden können. Produktivitätsverluste in der Software-Entwicklung beim Übergang zu Mehrkernprozessoren können dadurch weitgehend vermieden werden.

Die Eignung der generierten Programme für MHP-basierte Interferenz-Analysen erlaubt es ferner, auf zeitliche Isolation und Techniken wie zeitgesteuerte Schedules zu verzichten. Dadurch kann insbesondere die Effizienz beim Einsatz in gemischt-kritischen Systemen signifikant erhöht werden. Das Programmiermodell lässt sich zudem (in Verbindung mit den Beiträgen aus dem nachfolgenden Kapitel 5) auf einer Vielzahl unterschiedlicher Zielplattformen umsetzen. Hierzu zählen sowohl Mehrkernprozessoren mit gemeinsam genutztem Hauptspeicher als auch Plattformen mit verteilten Speicherhierarchien. Durch die Plattformunabhängigkeit des Ansatzes können Anwenderprogramme weitgehend ohne Berücksichtigung der Plattform entwickelt und später automatisiert für die jeweilige Hardware optimiert werden.

Insgesamt bilden die hier beschriebenen Beiträge damit einen entscheidenden Baustein für neuartige Werkzeugketten, die den Einsatz von Mehrkernprozessoren in harten Echtzeitsystemen effizienter und einfacher gestalten können.

Kapitel 5.

Modellierung zeitkritischer Hardware-/Software-Systeme

Das in Kapitel 4 vorgestellte Compiler-Werkzeug ist plattformunabhängig und bezieht Informationen über die Hardware aus einem generischen Modell der Zielplattform. Die Anforderungen an dieses Modell beinhalten dabei insbesondere, dass es ausreichend detailliert sein muss, um sichere Aussagen über Speicherzugriffszeiten unter Berücksichtigung von Interferenzeffekten ableiten zu können. Dieses Kapitel präsentiert eine entsprechende Lösung bestehend aus einem generischen Plattformmodell, einer Methode zur Berechnung der Zugriffszeiten und einer erweiterten Architekturbeschreibungssprache (ADL) zur strukturierten Beschreibung der Zielplattform. Teile des Modells und der ADL wurden in den eigenen Veröffentlichungen [RB20a; RMB+18; RHD+16] publiziert.

Dieses Kapitel gliedert sich in fünf Abschnitte, wobei der nachfolgende Abschnitt 5.1 zunächst die Ziele und Anforderungen für das zu spezifizierende Plattformmodell definiert. Abschnitt 5.2 beschreibt daraufhin den grundlegenden Aufbau des Modells, bevor Abschnitt 5.3 die statische Berechnung von Speicherzugriffszeiten anhand der modellierten Informationen diskutiert. Die erweiterte ADL wird im darauffolgenden Abschnitt 5.4 eingeführt. Der abschließende Abschnitt 5.5 beendet das Kapitel dann mit einer Diskussion der vorgestellten Beiträge.

5.1. Zielsetzung und Anforderungen

Das Hauptziel des Plattformmodells und der zugehörigen ADL besteht darin, alle Hardware-bezogenen Informationen bereitzustellen, die für die plattformunabhängige Realisierung komplexer Werkzeugketten wie der in Abschnitt 3.3.3 beschriebenen ARGO-Werkzeuge nötig sind. Das schließt insbesondere die Hardware-Details zur Erzeugung von parallelen Programmen, deren Plattformspezifischer Optimierung sowie der Berechnung einer sicheren oberen Grenze für

die WCET auf Mehrkernprozessoren mit ein. Hierbei sollen sowohl Architekturen mit gemeinsam genutzten als auch mit verteilten Speichern unterstützt werden.

Einige Anforderungen an das Plattformmodell ergeben sich aus dem in Abschnitt 4.1.2 spezifizierten Programmiermodell. Die Bestandteile des Programmiermodells müssen jeweils zur Entwurfszeit auf eine Menge von Hardware-Komponenten abgebildet werden können (vgl. Abschnitt 4.1.2.1 und Abbildung 4.3). Dazu ist es erforderlich, die Komponenten auf einer geeigneten Abstraktionsebene durch ein generisches Modell darzustellen. Das schließt sowohl die Prozessorkerne und Speicherkomponenten als auch die dazwischenliegende Chip-interne Kommunikationsinfrastruktur mit ein. Der interne Aufbau der Prozessorkerne ist an dieser Stelle weniger relevant, da die konkrete Mikroarchitektur in gängigen C-Compilern und WCET-Analyse Werkzeugen für Einzelkernprozessoren bereits berücksichtigt wird. Für die hier vorgesehenen Zwecke genügt es daher, die Architektur des Befehlssatzes (ISA) und eine überschaubare Anzahl von zusätzlichen Parametern in das Modell einzubeziehen. Die interne Struktur von Prozessorkernen und Speichern kann dann durch einen solchen Satz von Parametern (z. B. ISA, Speichergrößen, Zugriffslatenzen, etc.) abstrahiert und ansonsten als *Black Box*[1] betrachtet werden. Die Kommunikationsinfrastruktur weist dagegen – vor allem in MPSoCs – oft unterschiedliche, zumeist heterogene, Topologien auf. Insbesondere für die präzise Quantifizierung von Interferenzeffekten in gemeinsam genutzten Komponenten muss dieser Teil der Plattform deshalb relativ detailliert abgebildet werden.

Die wichtigste Anforderung an das Modell der Kommunikationskomponenten ist die Bereitstellung aller zur korrekten Bestimmung von Zugriffszeiten nötigen Informationen. Diese Zugriffszeiten werden im Folgenden auch als *Worst Case Access Times* (WCAT) bezeichnet. Sie wirken sich naturgemäß auf die WCET eines Programms aus und werden von Interferenzeffekten maßgeblich beeinflusst. Sowohl für die logischen Speichersegmente des vorgestellten Programmiermodells als auch für die FIFO-Kanäle muss das Zeitverhalten im ungünstigsten Fall anhand des Plattformmodells vorhergesagt werden können. Dazu müssen einerseits die Routen aller Zugriffe und die dabei durchlaufenen Busse, Router und sonstigen Kommunikationskomponenten (vgl. Abschnitt 2.1.3) aus dem Modell abgeleitet werden. Andererseits muss das Zeitverhalten aller Komponenten entlang der Routen, inklusive der Verzögerungen durch Interferenzeffekte, hinterlegt sein. Dazu ist ein formales Modell der Arbitrierung (vgl. Abschnitt 2.1.3) in den einzelnen Komponenten, sowie eine geeignete Beschreibung der Routenplanung erforderlich.

[1] Beim „Black Box“-Prinzip werden nur die nach außen sichtbaren Merkmale einer Systemkomponente, nicht aber deren innerer Aufbau betrachtet.

Unter Berücksichtigung einiger weiterer Aspekte (siehe auch [RHD+16]), die zur plattformunabhängigen Realisierung komplexer Werkzeugketten benötigt werden, muss das zu spezifizierende Modell der Hardware-Plattform im Wesentlichen die folgenden Aspekte abdecken:

1. Eine detaillierte Beschreibung Chip-interner Kommunikationskomponenten in verschiedenen Ausprägungen (NoCs, Bus-Hierarchien, etc.).
2. Ein detailliertes Modell der Bitbreiten aller Kommunikationsverbindungen, deren Arbitrierungs-Verfahren sowie der Routing-Schemata.
3. Die Modellierung verschiedener Typen von Speicherzugriffen (z. B. Lese-/Schreibzugriffe) und des jeweils anfallenden Datenvolumens, einschließlich der Adress- und Verwaltungsdaten in den Bus- bzw. NoC-Protokollen.
4. Die Modellierung von Adressräumen und deren Abbildung auf die zugehörigen (verteilten) Speicherkomponenten.
5. Eine hinreichend genaue Beschreibung der Caches, einschließlich ihrer Verdrängungsstrategien.
6. Informationen über Anzahl, Art und Befehlssatz aller Prozessorkerne.

5.2. Aufbau des Plattformmodells

Auf Basis der genannten Anforderungen wird im Folgenden ein generisches Plattformmodell spezifiziert. Hierbei werden zunächst die Annahmen und Einschränkungen hinsichtlich der unterstützten Zielplattformen diskutiert. Anschließend werden mit der Definition des Plattform-Komponenten-Graphen, der Arbitrierungsmodelle und der Routenplanung die zentralen Bestandteile zur Modellierung der Topologie einer Rechnerarchitektur vorgestellt.

5.2.1. Annahmen & Einschränkungen

Die einschränkenden Annahmen für die Definition des Plattformmodells resultieren einerseits aus den Anforderungen zur Realisierung einer statischen WCET-Analyse. Andererseits wird auf die Unterstützung einiger, in Echtzeitsystemen oft untypischer Hardware-Funktionen verzichtet, woraus sich weitere Einschränkungen ergeben. Letztere können jedoch durch zukünftige Erweiterungen des Plattformmodells bei Bedarf aufgehoben werden.

Die folgenden Einschränkungen resultieren weitgehend aus den Anforderungen aktueller Werkzeuge zur statischen WCET-Analyse:

Statisch vorhersagbare Prozessorkerne: Gängige WCET-Analyse-Werkzeuge verwenden eigene abstrakte Prozessor-Modelle (siehe [116]), die jedoch nur für bestimmte vorhersagbare Mikroarchitekturen verfügbar sind. Im Folgenden wird daher angenommen, dass die modellierten Plattformen nur Kerne mit solchen Mikroarchitekturen enthalten.

Faire Arbitrierung: Die Forderung nach fairer Arbitrierung stellt v. a. sicher, dass Zugriffe eines Prozessorkerns nicht permanent durch andere Zugriffe blockiert werden können (vgl. Definition 2.1). Andernfalls ließe sich für die blockierten Zugriffe keine endliche WCAT ermitteln.

Statische vorhersagbare Routenplanung: Um die Routen von Zugriffen zur Entwurfszeit ermitteln zu können, ist ein vorhersagbares Verfahren zur Routenplanung (z. B. XY-Routing) erforderlich.

Kein Paketverlust in NoCs: Bei Datenübertragungen über Kommunikationsverbindungen (z. B. NoCs) dürfen keine Datenpakete verloren gehen. Andernfalls müssten einzelne Pakete zur Laufzeit ggf. erneut versendet werden, ohne dass dies zur Entwurfszeit präzise vorhergesagt und bei der Berechnung der WCET berücksichtigt werden kann.

Keine gemeinsam genutzten Caches: Gemeinsam von mehreren Prozessoren genutzte Caches sind kaum vorhersagbar, da nur bei exakter Kenntnis des Programmablaufs auf allen beteiligten Prozessoren eine sichere Aussage über deren Inhalte getroffen werden kann. Aus diesem Grund werden solche Caches nicht unterstützt.

Keine Cache-Kohärenz-Mechanismen: Hardware-gesteuerte Mechanismen zur Erhaltung der Cache-Kohärenz (vgl. Abschnitt 2.1.1) können zu ähnlichen Effekten wie gemeinsam genutzte Caches führen. Je nach Mechanismus werden die Inhalte einzelner Caches auch hier durch mehrere Prozessorkerne beeinflusst. Daher werden im Folgenden Plattformen ohne Kohärenz-Mechanismen vorausgesetzt.

Das in dieser Arbeit entwickelte Plattformmodell setzt außerdem die folgenden zusätzlichen Annahmen voraus:

Bare-Metal Software: Es wird vorausgesetzt, dass die Software ohne ein Betriebssystem mit Speicher-Virtualisierung „bare-metal“ ausgeführt wird. Durch diese Annahme lässt sich aus der (physikalischen) Speicheradresse direkt auf die adressierten Speicherbereiche zurückschließen.

Kommunikation mit FIFO-Charakteristik: Die Reihenfolge übertragener Dateneinheiten aus derselben Quelle darf durch Kommunikationskomponenten nicht permutiert werden. End-zu-End-Kommunikationspfade müssen daher eine FIFO-Charakteristik aufweisen. Dies gilt insbesondere auch

für Speicherzugriffe, die vom Speichersystem nur in der Reihenfolge ihres Auftretens bearbeitet werden dürfen. Andernfalls könnten Speicherzugriffe aus derselben Quelle miteinander interferieren und so z. B. zu Laufzeitanomalien führen.

Arbitrierungs-Verzögerungen durch Latenz und Durchsatz beschreibbar: Die maximal durch Arbitrierung verursachten Verzögerungszeiten müssen sich aus den Parametern Latenz L und Durchsatz DS ableiten lassen. Interferenzeffekte müssen dabei auf diese beiden Parameter beschränkt sein und dürfen in einer einzelnen Komponente nur von der aktuellen Anzahl konkurrierender Zugriffe abhängen (d. h. sie sind eine Funktion von n^{aktiv} aus Abschnitt 2.1.3.1).

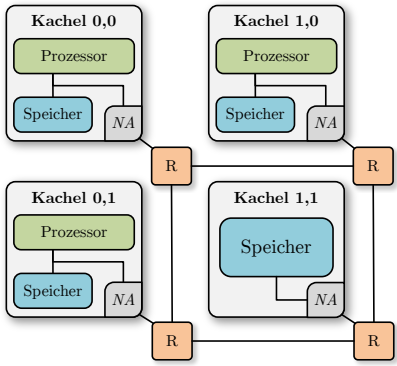
Während sich die ersten beiden Einschränkungen bei Bedarf vergleichsweise einfach durch entsprechende Erweiterungen des Plattformmodells aufheben lassen, ist die Forderung zu den Arbitrierungs-Verzögerungen von grundlegenderer Natur. Für die gängigen Arbitrierungsverfahren TDMA und RR ist sie jedoch erfüllt (vgl. Abschnitt 2.1.3.1).

5.2.2. Plattform-Komponenten-Graph

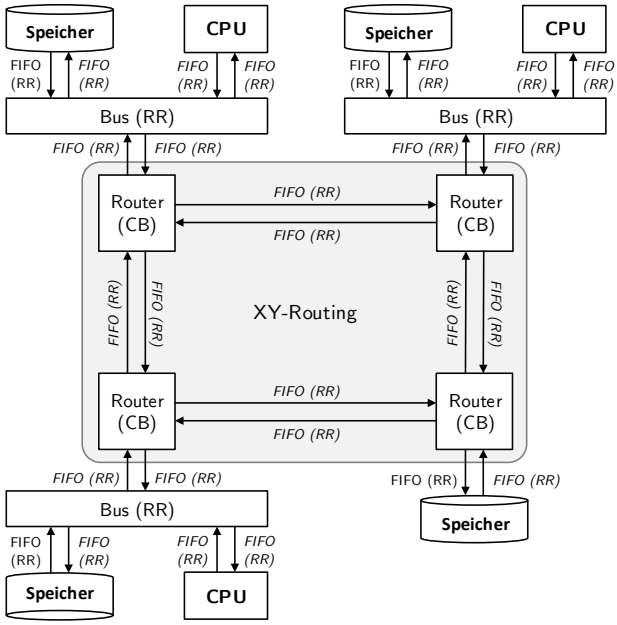
Um die Struktur und Topologie einer Hardware-Plattform im Rahmen des zu spezifizierenden Modells zu beschreiben, wird im Folgenden der sogenannte *Plattform-Komponenten-Graph* definiert:

Definition 5.1 (Plattform-Komponenten-Graph)**.** *Sei V^{cg} eine Menge von komplexen Komponenten einer Hardware-Plattform und E^{cg} eine Menge von Komponenten, die jeweils eine gerichtete Punkt-zu-Punkt-Verbindung zwischen zwei komplexen Komponenten herstellen. Dann heißt der gerichtete Graph $CG = (V^{cg}, E^{cg})$ Plattform-Komponenten-Graph. Für den Plattform-Komponenten-Graphen sei ferner die Menge $\mathcal{C} := V^{cg} \cup E^{cg}$ aller Komponenten der Hardware-Plattform definiert.*

Abbildung 5.1 zeigt ein Beispiel für einen solchen Komponenten-Graphen und die damit modellierte Architektur einer einfachen NoC-basierten Plattform. Für alle Komponenten, die für das Zeitverhalten der Architektur ausschlaggebend sind, muss entweder ein Knoten $u \in V^{cg}$ oder mindestens eine Kante $e \in E^{cg}$ in dem Graphen vorhanden sein. Die Kanten können dabei für einfache Verbindungsleitungen mit Punkt-zu-Punkt-Charakteristik oder FIFO-Puffer mit einer bestimmten Kapazität stehen. Sie werden in Abbildung 5.1 z. B. für die



(a) Eine einfache NoC-basierte Architektur



(b) Plattform-Komponenten-Graph mit Arbitrierungsschemata RR und Crossbar (CB)

Abbildung 5.1.: Beispiele für eine einfache NoC-basierte Architektur und den zugehörigen Plattform-Komponenten-Graphen *CG* (vgl. [RB20a])

NoC-Verbindungen zwischen den Routern verwendet. Einfache ungepufferte Verbindungsleitungen können dabei durch eine FIFO-Verbindung mit der Kapazität 0 modelliert werden. Die Knoten des Graphen repräsentieren dagegen komplexere Komponenten wie Prozessoren, Speicher oder Kommunikationskomponenten mit mehreren Ein- und Ausgängen (z. B. Router in NoCs).

Der vorgestellte Plattform-Komponenten-Graph CG lässt sich grundsätzlich in formale Darstellungen, wie z. B. Warteschlangennetzwerke oder Datenflussgraphen, überführen. Um mit solchen Darstellungen detaillierte Aussagen treffen zu können, muss jedoch das Zugriffsmuster der Software z. B. durch Ankunftsfunktionen formal dargestellt werden (vgl. Abschnitt 3.1.2.2). Ohne Begrenzung der Datenraten in der Hardware ist das Zugriffsmuster der in Abschnitt 3.3.3.3 definierten Code-Segmente in der aPPIR-Darstellung jedoch im Allgemeinen nicht bekannt. Im Folgenden wird daher nicht auf diese formalen Darstellungen zurückgegriffen, sondern ein spezialisiertes Modell entwickelt, dass lediglich die Maximalzahl von Speicherzugriffen $N^{mem}(cs, m)$ pro Code-Segment als bekannt voraussetzt.

Komplexere Hardware-Architekturen, wie die in Abbildung 5.1a dargestellte NoC-Architektur, sind in der Regel hierarchisch aufgebaut. So enthalten die Kacheln aus der Abbildung z. B. jeweils eine Reihe von internen Unterkomponenten. Um aus der konkreten Architektur einen geeigneten nicht-hierarchischen Komponenten-Graphen zu erzeugen, muss die hierarchische Struktur u. U. aufgebrochen werden. In der Architektur aus Abbildung 5.1 ist es z. B. sinnvoll, nicht die Kacheln als Ganzes, sondern deren Unterkomponenten in den Komponenten-Graphen aufzunehmen, sodass auch das Zeitverhalten der lokalen Speicher und Bus-Systeme abgebildet wird. Die geeigneten Hierarchieebenen und Komponenten für den Komponenten-Graphen eines Plattformmodells müssen i. A. abhängig vom Verhalten der konkreten Plattform durch einen Endbenutzer identifiziert werden. Dabei muss sichergestellt werden, dass das Zeit- und Kommunikationsverhalten durch das Modell korrekt wiedergegeben wird. Komponenten wie die Netzwerk-Adapter aus Abbildung 5.1a (NA in der Abbildung) können z. B. wie in Abbildung 5.1b durch einfache FIFO-Verbindungen modelliert werden. Je nach Implementierung der Netzwerk Adapter kann jedoch auch ein komplexeres Modell, ggf. unter Berücksichtigung interner Unterkomponenten, erforderlich sein.

Abbildung 5.2 zeigt das UML-Klassendiagramm einer möglichen Umsetzung des Plattform-Komponenten-Graphen als Klassenstruktur, um die Relationen der einzelnen Bestandteile zu veranschaulichen. Neben der eigentlichen Graphenstruktur und den unterschiedlichen Knotentypen sind in dem abgebildeten Modell weitere Zusatzinformationen hinterlegt. Diese werden in den folgenden Unterabschnitten näher diskutiert. Für die tatsächliche Umsetzung wird der

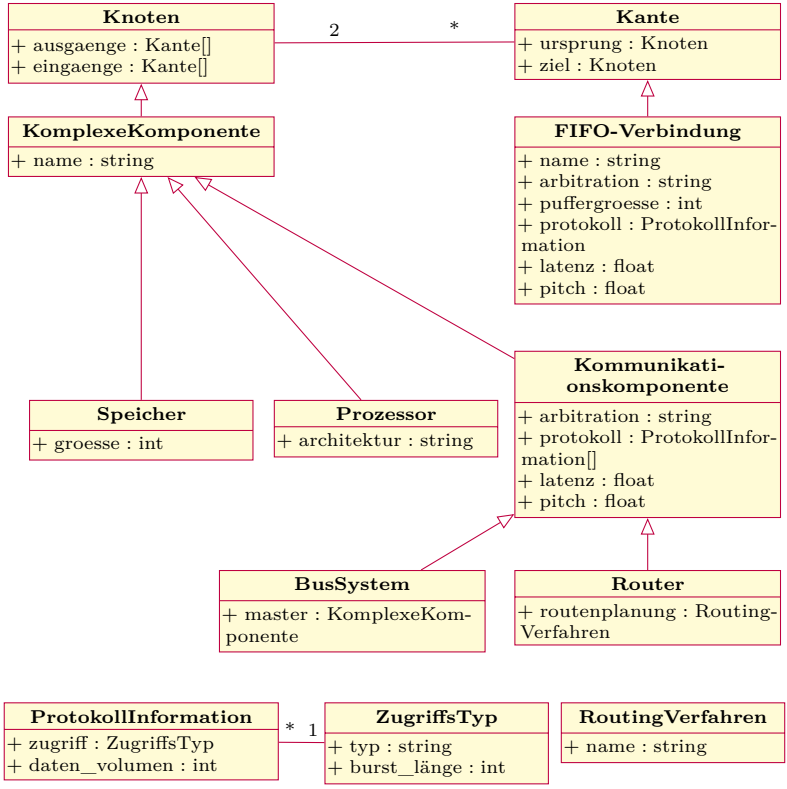


Abbildung 5.2.: UML-Klassendiagramm für eine mögliche Realisierung des Plattform-Komponenten-Graphen

Graph jedoch nicht mit dem dargestellten Modell, sondern zusammen mit einer Reihe weiterer Attribute durch die in Abschnitt 5.4 definierte Architekturbeschreibungssprache realisiert.

5.2.2.1. Arbitrierungsmodelle

Die Arbitrierung beim Zugriff auf gemeinsam genutzte Hardware-Komponenten bestimmt unmittelbar die durch Interferenzeffekte verursachten Verzögerungen für die beteiligten Prozessorkerne (vgl. Abschnitt 2.1.3.1). Das Arbitrierungsschema muss deshalb, wie in Abbildung 5.2 dargestellt, für alle Kommunikationskomponenten im Plattformmodell hinterlegt werden. Jeder Komponente $c \in \mathcal{C}$ wird dazu entweder ein konkretes Arbitrierungsschema oder das Verhalten einer *Crossbar* (kurz CB, im deutschen auch als *Kreuzschiene* bezeichnet) zugeordnet. In dem Beispiel aus Abbildung 5.1b ist diese Information jeweils in Klammern angegeben. Die Punkt-zu-Punkt-Verbindungen sind dort mit RR Arbitrierung realisiert, während die NoC-Router CB-Verhalten aufweisen. In letzteren findet keine Arbitrierung statt, da sich Zugriffe in Crossbars Interferenz-frei kreuzen können. Diese Kombination aus Crossbar und FIFO-Verbindungen bildet das Verhalten einfacher NoC-Router hinreichend genau ab, da deren interne Architektur typischerweise aus einer zentralen Crossbar und FIFO-basierten Eingangspuffern besteht (siehe z. B. [58]).

Ähnlich wie in Kapitel 2, Abschnitt 2.1.3.1 und in typischen Datenfluss-basierten NoC-Modellen (vgl. Abschnitt 3.1.2.2) werden Arbitrierungseinheiten im Folgenden durch die Parameter Latenz L und Durchsatz DS charakterisiert. Grundsätzlich werden faire Arbitrierungsverfahren unterstützt, bei denen diese beiden Größen über die Zeit konstant sind oder nur von der Anzahl n^{aktiv} der aktiven Eingänge an der Arbitirerungseinheit abhängen. Dies trifft insbesondere für die in Abschnitt 2.1.3.1 eingeführten Arbitrierungsverfahren TDMA und RR sowie die zugehörigen Gleichungen (2.1) bis (2.4) mit deren Latenz und Durchsatz zu.

Um später eine einheitlichere Formulierung der Gleichungen zu erhalten, wird im Folgenden nicht der Durchsatz, sondern die Zeitspanne zwischen zwei aufeinanderfolgenden Dateneinheiten betrachtet. Diese Größe wird in Anlehnung an den SHIM-Standard [111] auch als *Pitch* P bezeichnet, der als inverser Durchsatz wie folgt definiert ist:

Definition 5.2 (Pitch)**.** *Der Pitch P bezeichnet den inversen Wert des Durchsatzes DS aus Definition 2.2 mit*

$$P := \frac{1}{DS} \ .$$

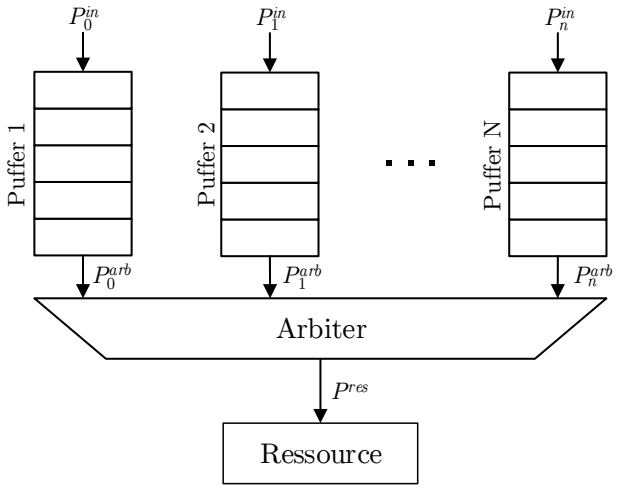


Abbildung 5.3.: Arbitrierungseinheit mit Pitches an den Verbindungsleitungen (vgl. [RB20a])

Um die Arbitrierung innerhalb einer Komponente $c \in \mathcal{C}$ des Komponenten-Graphen zu modellieren, wird die in Kapitel 2 vorgestellte Struktur aus Abbildung 2.5 zugrunde gelegt. Die Länge der Eingangspuffer ergeben sich dabei aus den Längen der modellierten FIFO-Puffer, die jeweils durch eine Kante $e \in E^{cg}$ des CG repräsentiert werden. Wie in Abbildung 5.3 dargestellt, haben die Verbindungsleitungen vor und nach dem eigentlichen Arbiter im Allgemeinen verschiedene Pitches bzw. Durchsätze. Die Berechnungsvorschrift für die Pitches $P_j^{arb}(c, n^{aktiv})$ an den Eingängen des Arbiters geht für die Arbitrierungsverfahren TDMA und RR unmittelbar aus den in Kapitel 2 eingeführten Gleichungen (2.1) bzw. (2.2) hervor. Anders als in den genannten Gleichungen soll im Folgenden statt der Arbitrierungsperiode p_i die Anzahl der aktiven Eingänge n^{aktiv} als offener Parameter betrachtet werden. Mit einem solchen Ansatz lassen sich alle Arbitrierungsverfahren abbilden, deren Latenzen/Pitches nur von n^{aktiv} abhängen oder, wie bei TDMA, konstant sind.

Durch die dazwischenliegenden Puffer können die Pitches $P_j^{arb}(c, n^{aktiv})$ und $P_j^{in}(c, n^{aktiv})$ im Allgemeinen voneinander abweichen, zumindest solange die Puffer nicht vollständig gefüllt sind. Verfügt ein Puffer noch über freie Kapazitäten, so können selbst für $P_j^{arb}(c, n^{aktiv}) = 0$ kurzzeitig noch Daten mit

$P_j^{in}(c, n^{aktiv}) > 0$ am Eingang entgegengenommen werden. Einige mathematische Modelle, wie z. B. das *Netzwerk-Kalkül* (vgl. Abschnitt 3.1.2.2), sind in der Lage, solche kurzzeitigen transienten Effekte abzubilden. Dazu muss jedoch die zeitliche Verteilung des Datenaufkommens bzw. der Speicherzugriffe hinreichend genau bekannt sein. Letztere kann in realen Programmen jedoch mit jeder Ausführung des Programmcodes stark variieren (z. B. in Abhängigkeit der Cache-Inhalte), sodass sich bei der statischen Berechnung der WCET zur Entwurfszeit kaum eine sinnvolle Aussage treffen lässt. Das zu spezifizierende WCET-zentrierte Plattformmodell löst dieses Problem durch die pessimistische Annahme, dass die Puffer stets vollständig gefüllten sind. Unter dieser Annahme kann nur dann ein neues Datum am Eingang eines Puffers aufgenommen werden, wenn ein anderes Datum aus dem Puffer an den Arbiter weitergegeben wird. Daraus folgt $P_j^{arb}(c, n^{aktiv}) = P_j^{in}(c, n^{aktiv})$ und die beiden Pitches lassen sich zu einem effektiven Eingangs-Pitch

$$P_j^{eff}(c, n^{aktiv}) := P_j^{arb}(c, n^{aktiv}) = P_j^{in}(c, n^{aktiv}) \tag{5.1}$$

zusammenfassen.

Zur Berechnung der Latenzen $L_j^{eff}(c, n^{aktiv})$, die eine Dateneinheit vom Erreichen eines der Eingangspuffer in Abbildung 5.3 bis zu ihrer Bearbeitung in der gemeinsam genutzten Ressource benötigt, werden im Folgenden leere Puffer angenommen. Diese Annahme ist i. A. optimistisch und kann daher nur unter geeigneten Voraussetzungen zur Berechnung einer WCET herangezogen werden. Die späteren Betrachtungen in Abschnitt 5.3 zeigen jedoch, dass diese Voraussetzungen für die vorgesehene Verwendung der berechneten Latenz-Werte erfüllt sind. Trifft eine Dateneinheit am Eingang eines leeren Puffers ein, so bleibt sie nur so lange im Puffer, bis der Arbiter den Zugriff zu der gemeinsam genutzten Ressource freigibt. Für die Arbitrierungsverfahren TDMA und RR entspricht $L_j^{eff}(c, n^{aktiv})$ dann L^{TDMA} bzw. L^{RR} aus den Gleichungen (2.3) und (2.4) in Kapitel 2.

Während das Plattformmodell grundsätzlich beliebige faire Arbitrierungsschemata unterstützt, zählt RR in der Praxis zu den am häufigsten in NoCs oder Bus-Systemen eingesetzten fairen Verfahren. Deshalb sollen an dieser Stelle noch die expliziten Gleichungen zur Berechnung von $L_j^{eff}(c, n^{aktiv})$ und $P_j^{eff}(c, n^{aktiv})$ für RR angeführt werden:

$$L_j^{eff}(c, n^{aktiv}) = T^{ZS}(c) \cdot n^{aktiv}, \tag{5.2}$$

$$P_j^{eff}(c, n^{aktiv}) = n^{aktiv} \cdot P^{res}(c). \tag{5.3}$$

Dies folgt unmittelbar aus den Gleichungen (2.2) und (2.4), wobei $P^{res}(c)$ für den zu DS^{res} gehörigen Pitch steht. Außerdem hängen im Kontext des *CG*

alle Größen von der betrachteten Komponente c ab. Die Anzahl der aktiven Eingänge n^{aktiv} ist jeweils so zu wählen, dass sie dem Maximum der aktuellen Arbitrierungsperiode p_i und der darauffolgenden Arbitrierungsperiode p_{i+1} entspricht, um dem Term $\max\left(n^{aktiv}(p_i), n^{aktiv}(p_{i+1})\right)$ in Gleichung (2.4) zu genügen.

Für Latenz und Pitch sieht das Modell aus Abbildung 5.2 jeweils einen Parameter in den Kommunikationskomponenten und FIFO-Verbindungen vor. Diese können z. B. im Fall von RR-Arbitrierung genutzt werden, um die Werte für $T^{ZS}(c)$ und $P^{res}(c)$ zu hinterlegen. Mit Hilfe des Feldes für das Arbitrierungsschema kann eine passende Implementierung dann für einen spezifischen Wert von n^{aktiv} die effektiven Werte $L_j^{eff}(c, n^{aktiv})$ und $P_j^{eff}(c, n^{aktiv})$ berechnen.

5.2.3. Speicherzugriffe und Routenplanung

In NoCs oder Bus-Hierarchien sind i. A. mehrere Komponenten mit eigenen Arbitrierungseinheiten an der Bearbeitung eines Speicherzugriffs beteiligt. In NoCs gibt es zudem unterschiedliche Routen, die ein Zugriff potentiell durchlaufen kann. Insbesondere in Netzwerken mit Paketvermittlung (vgl. Abschnitt 2.1.3) müssen neben den Nutzdaten außerdem noch zusätzliche Protokoll-Informationen übertragen werden. Um die letztendliche Zugriffszeit bestimmen zu können, müssen all diese Aspekte auf geeignete Weise in dem zu spezifizierenden Plattformmodell abgebildet werden.

Im Allgemeinen werden Speicherzugriffe durch eine Ursprungskomponente $c \in \mathcal{C}$ (typischerweise ein Prozessorkern oder eine DMA-Einheit) unter Verwendung einer lokalen Speicheradresse gestartet. Die Speicheradresse verweist auf eine Speicherkomponente $m_l \in \mathcal{C}$, an die der Zugriff übertragen werden muss. Die bei der Übertragung involvierten Arbitrierungs-Stufen ergeben sich dann aus dem Pfad bzw. der Route, die der Speicherzugriff im Plattform-Komponenten-Graphen CG durchläuft. Eine solche Route $R_{kl} := (c_k, \ldots, c_l)$ für ein gegebenes Paar (c_k, m_l) aus einer Ursprungskomponente $c_k \in \mathcal{C}$ und einer Speicherkomponente $m_l := c_l \in \mathcal{C}$ muss gemäß der Anforderungen an die Hardware-Plattform zur Entwurfszeit bekannt sein.

Zur Bestimmung des Verlaufs einer Route R_{kl} ist für Router-Komponenten in dem Modell aus Abbildung 5.2 eine Strategie zur Routenplanung hinterlegt. Typischerweise erstreckt sich jede Instanz eines solchen Routing-Verfahrens über mehrere Komponenten, wie es in dem Beispiel aus Abbildung 5.1 für das XY-Routing innerhalb des NoC dargestellt ist. Auf Basis dieser Informationen wird der Verlauf von R_{kl} nach dem folgenden Schema bestimmt:

1. Ermittlung des kürzesten Pfades von c_k nach m_l in CG.

2. Korrektur des Verlaufs dieses Pfades innerhalb aller Regionen des CG, denen ein spezifisches Routing-Verfahren zugeordnet ist.

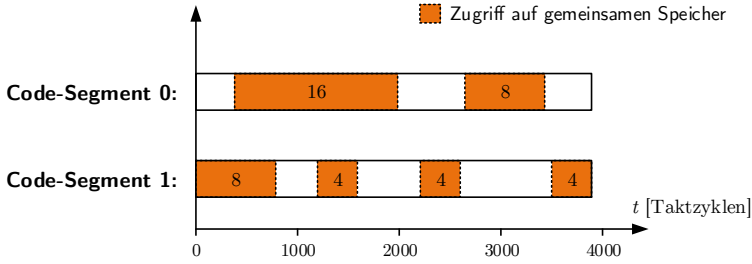
Bei der Erstellung eines Plattformmodells ist darauf zu achten, dass der nach diesem Verfahren bestimmte Pfad mit der tatsächlichen Route R_{kl} der Speicherzugriffe übereinstimmt. Im Fall von Lesezugriffen ist außerdem zu beachten, dass die Speicherkomponente m_l zusätzlich eine Antwort über eine zweite Route R_{lk} senden muss. Der Verlauf dieser Route kann i. A. von R_{kl} abweichen, die Routen für diese Rückkanäle lassen sich aber ansonsten analog zu den „vorwärts" gerichteten Routen behandeln.

Entlang einer Route R_{kl} müssen die darüber abgewickelten Zugriffe in den beteiligten Komponenten $c \in R_{kl}$ einem der Eingänge/Eingangspuffer j der zugehörigen Arbitrierungseinheit zugeordnet werden. Im Folgenden wird angenommen, dass jede Route R_{kl}, die die Komponenten $c \in R_{kl}$ enthält, einem festen Eingangspuffer $j = \mathfrak{J}(c, R_{kl})$ zugewiesen ist. Auf diese Weise wird sichergestellt, dass alle Routen in jeder Komponente durch die faire Arbitrierung gleichbehandelt werden. In NoCs findet eine solche Zuordnung zu den Puffer-Ressourcen z. B. in Netzwerken mit *Leitungsvermittlung* (englisch *Circuit Switching*) oder *virtueller Leitungsvermittlung* statt (vgl. Abschnitt 2.1.3). In diesem Fall entspricht eine Route R_{kl} genau einem Circuit, bei dessen Aufbau die Hardware alle nötigen Puffer- und Arbitrierungs-Ressourcen reserviert.

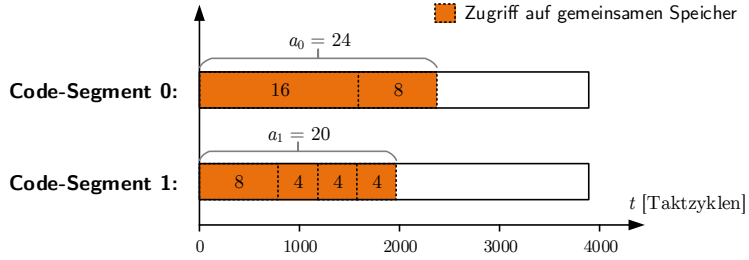
Allen Kommunikationskomponenten und FIFO-Verbindungen ist in Abbildung 5.2 ein Feld mit Protokoll-Informationen für verschiedene Zugriffstypen zugeordnet. Der Zugriffstyp gibt dabei an, ob es sich um eine Leseanfrage, Lese-Antwort oder einen Schreibzugriff handelt, und wie lange die dabei übermittelte Sequenz aus Datenwörtern (im englischen auch *Burst* genannt) ist. Ersteres wird in der Klasse `ZugriffsTyp` aus Abbildung 5.2 durch den Parameter `typ` und Letzteres durch `burst_länge` spezifiziert. Zu jedem Paar aus Zugriffstyp und Kommunikations- bzw. FIFO-Komponente geben die Protokoll-Informationen dann das anfallende Datenvolumen, einschließlich der Protokoll-Daten an.

5.3. Statische Berechnung von Speicherzugriffszeiten

Das in Abschnitt 5.2 vorgestellte Plattformmodell stellt alle nötigen Hardware-Informationen bereit, um sichere obere Grenzen für Speicherzugriffszeiten zu ermitteln. Aufgrund von Interferenzeffekten können die Zugriffszeiten nicht in Isolation berechnet werden, sondern müssen auch die parallelen Vorgänge auf anderen Prozessorkernen berücksichtigen. Zur Gewinnung der dazu nötigen



(a) Typische Verteilung



(b) Verteilung im ungünstigsten Fall

Abbildung 5.4.: Mögliche Verteilungen für die Speicherzugriffe zweier Code-Segmente über die Zeit (vgl. [RB20a])

Informationen wird im Folgenden auf die in Abschnitt 3.3.3.3 beschriebene „May-Happen-in-Parallel“-Analyse (MHP-Analyse) der aPPIR-Darstellung zurückgegriffen. Diese ordnet jedem Code-Segment cs_i eine Menge von Code-Segmenten $MHP(cs_i)$ zu, die potentiell parallel zu cs_i ausgeführt werden können. Ferner ist für jedes Segment cs_i der zugehörige Prozessorkern sowie die maximale Anzahl der darin ausgeführten Speicherzugriffe $N^{mem}(cs_i, m_l)$ pro Speicherkomponente m_l bekannt[2]. Systemweit können dann während der Ausführungszeit des Code-Segments cs_i alle Code-Segmente aus der Menge

$$\mathcal{PS}(cs_i) := MHP(cs_i) \cup \{cs_i\} \tag{5.4}$$

[2]Für die tatsächliche Umsetzung muss auch der Typ der Speicherzugriffe aus dem Modell in Abbildung 5.2 berücksichtigt werden. Zur Vereinfachung der Notation wird dies jedoch in den formalen Betrachtungen vernachlässigt.

potentiell auf einem der Prozessorkerne ausgeführt werden. Die Menge $\mathcal{PS}(cs_i)$ kann somit als eine Art kernübergreifender Ausführungskontext für cs_i angesehen werden. Das Argument von $\mathcal{PS}$ wird aus Gründen der besseren Lesbarkeit im Folgenden meist unterdrückt.

Aus den vorhandenen Informationen zu den Code-Segmenten geht zwar die Anzahl der Speicherzugriffe hervor, über ihre zeitliche Verteilung lässt sich jedoch keine genaue Aussage treffen. Zur Laufzeit können im Allgemeinen bei jeder Ausführung eines Code-Segments unterschiedliche Verteilungen auftreten. Abbildung 5.4 veranschaulicht das anhand zweier beispielhafter Verteilungen der Zugriffe für zwei Code-Segmente cs_0 und cs_1 auf einen gemeinsam genutzten Speicher. Während beide Verteilungen $a_0 = 24$ Zugriffe für cs_0 und $a_1 = 20$ Zugriffe für cs_1 aufweisen, gibt es bei der Anzahl der gleichzeitig auftretenden Zugriffe signifikante Unterschiede. In dem Szenario aus Abbildung 5.4a gibt es nur acht solcher Zugriffe, wohingegen in Abbildung 5.4b die maximal mögliche Anzahl von $a_1 = 20$ parallel ablaufenden Zugriffen erreicht wird. Da solche parallelen Zugriffe auf einen gemeinsamen Speicher in den Arbitrierungseinheiten kollidieren, bestimmt ihre Anzahl maßgeblich die verursachten Interferenz-Kosten. Ohne genaue Kenntnis der zeitlichen Verteilung kann eine sichere Aussage zur Interferenz jedoch nur unter Verwendung eines pessimistischen Szenarios wie in Abbildung 5.4b getroffen werden. Da ein Zugriff in cs_0 bei fairer Arbitrierung mit maximal einem Zugriff von cs_1 kollidieren kann (vgl. Abschnitt 2.1.3.1), stellt die maximale zeitliche Überlappung in diesem Szenario gleichzeitig einen sicheren oberen Grenzwert dar. Die maximal möglichen Interferenz-Kosten für cs_0 aus Abbildung 5.4 lassen sich daher unter der Annahme von 20 konfliktbehafteten und vier konfliktfreien Zugriffen berechnen.

5.3.1. Arbitrierungs-Szenarien

Im allgemeinen Fall müssen nicht nur zwei, sondern $|\mathcal{PS}|$ Code-Segmente, und nicht nur ein gemeinsamer Speicher, sondern alle Komponenten c entlang einer Route R_{kl} berücksichtigt werden. Hierzu ermittelt man zunächst für jede Komponente $c \in \mathcal{C}$ die maximale Anzahl $a_j(c, \mathcal{PS})$ aller Dateneinheiten, die am Arbitrierungs-Eingang j während der Ausführung der Code-Segmente in $\mathcal{PS}$ anfallen können. Ähnlich wie in Abbildung 5.4 werden die Argumente von a_j zur besseren Lesbarkeit im Folgenden unterdrückt. Die Werte der Variablen a_j ergeben sich unter Berücksichtigung der Routen R_{kl} und der zugeordneten Arbiter-Eingänge $j = \mathfrak{J}(c, R_{kl}, \mathcal{PS})$ dann aus der Anzahl $N^{mem}(cs_i, m_l)$ der Speicherzugriffe aller Code-Segmente $cs_i \in \mathcal{PS}$. Dazu wird aus $N^{mem}(cs_i, m_l)$ das Datenaufkommen $\overline{N}(R_{kl}, \mathcal{PS})$ pro Route R_{kl} ermittelt. Da entlang einer verzweigungsfreien Route keine Daten hinzukommen oder verloren gehen dürfen

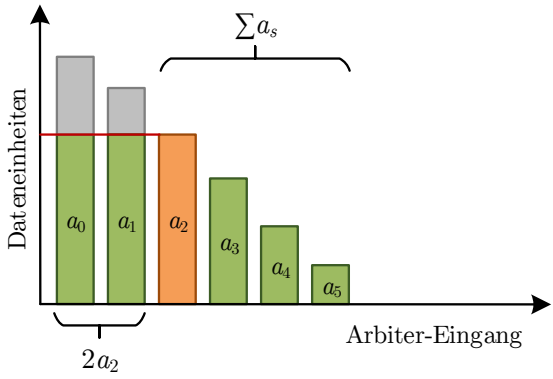


Abbildung 5.5.: Datenaufkommen an den Arbiter-Eingängen einer Komponente c mit den Termen für die Interferenz-Kosten von Eingang 2 (vgl. [RB20a])

(vgl. Abschnitt 5.2.1), entspricht dieser Wert auch gleichzeitig der Auslastung der zugehörigen Arbiter-Eingänge:

$$a_{\mathfrak{J}(c,R_{kl},\mathcal{PS})} = \overline{N}(R_{kl},\mathcal{PS}) \qquad \forall c \in R_{kl}\,. \tag{5.5}$$

Hierbei hängt $\mathfrak{J}(c,R_{kl},\mathcal{PS})$ explizit von $\mathcal{PS}$ ab, da die Nummerierung der Eingänge im Folgenden, je nach Kontext $\mathcal{PS}$, permutiert werden muss. Diese Annahme bezieht sich jedoch nur auf die logischen Indizes der Eingänge, während die von einer Route R_{kl} belegten Hardware-Ressourcen über die Programmlaufzeit hinweg unverändert bleiben können.

Abbildung 5.5 zeigt ein Beispiel für die Verteilung der Datenaufkommen a_j an den einzelnen Arbiter-Eingängen einer Komponente c. Die Überlappung der dargestellten Balken entspricht dort genau der maximalen Anzahl kollidierender Zugriffe bei einer zeitlichen Verteilung wie der aus Abbildung 5.4b. In dem Beispiel soll nun angenommen werden, dass die Speicherzugriffe des zu untersuchenden Code-Segments cs_i dem Arbiter-Eingang 2 zugeordnet sind. Zur Vereinfachung der grafischen und der zugehörigen formalen Darstellung wird dabei angenommen, dass die (logischen) Indizes j nach dem Datenaufkommen a_j in absteigender Reihenfolge sortiert sind, d. h. $a_j > a_s \implies j < s$. Anhand der Lastverteilung für die Arbiter-Eingänge müssen zur Bestimmung der Zugriffszeiten die zugehörigen effektiven Latenzen $L_j^{\mathit{eff}}(c, n^{aktiv})$ und Pitches $P_j^{\mathit{eff}}(c, n^{aktiv})$ bestimmt werden. Da diese beiden Größen i. A. von der Anzahl n^{aktiv} der aktuell

aktiven Arbiter-Eingänge abhängen, muss bei ungleich verteiltem Datenaufkommen zwischen verschiedenen Fällen unterschieden werden. Betrachtet man den Arbiter-Eingang $j = 2$ aus Abbildung 5.5, so können aufgrund der fairen Arbitirerung maximal a_5 der a_2 Dateneinheiten mit einer Dateneinheit an Eingang 5 interferieren. Da die verbleibenden Eingänge $j < 5$ ein höheres Datenaufkommen als Eingang 5 aufweisen, interferieren die ersten a_5 der a_2 Dateneinheiten im ungünstigsten Fall außerdem auch mit allen übrigen Eingängen. Für die Arbitirerung dieser a_5 Dateneinheiten gilt also $n^{aktiv} = 6$. Im Anschluss an diese Daten folgt an dem untersuchten Eingang 2 eine Anzahl von $a_4 - a_5$ Dateneinheiten, die nicht mehr mit Eingang 5, aber nach wie vor mit den Eingängen $j < 5$ interferieren können. Letztere müssen also mit $n^{aktiv} = 5$ berücksichtigt werden. Dieses Schema kann bis zum Erreichen von $j = 2$ fortgesetzt werden. Ab dann ändert sich die Situation, da die a_2 Elemente in Eingang 2 bei fairer Arbitrierung nicht mit mehr als a_2 Dateneinheiten aus den Eingängen 1 und 2 interferieren können. Die in Abbildung 5.5 grau dargestellten Anteile der zugehörigen Balken spielen für den Pitch und die Latenz an Eingang 2 deshalb keine Rolle mehr.

Aus dem beschriebenen Schema folgt, dass die a_2 Dateneinheiten und die zugehörigen Speicherzugriffe i. A. mit unterschiedlichen Latenzen und Pitches beaufschlagt werden müssen. In die WCET des analysierten Code-Segments geht am Ende jedoch nur die akkumulierte Zugriffszeit aller Speicherzugriffe ein. Es muss daher nicht explizit zwischen den einzelnen Dateneinheiten unterschieden werden. Die Pitches und Latenzen können deshalb über den gesamten Ausführungskontext $\mathcal{PS}(cs_i)$ hinweg gemittelt werden, ohne dass sich die Summe der Zugriffszeiten in cs_i verändert. Zur expliziten Darstellung der mittleren Pitches $P_j^{avg}(c, \mathcal{PS})$ und Latenzen $L_j^{avg}(c, \mathcal{PS})$ wird im Folgenden (wie bereits im vorherigen Abschnitt 5.2) das RR-Arbitrierungsverfahren zugrunde gelegt. Da die Gleichungen (5.2) und (5.3) linear bezüglich n^{aktiv} sind, können die durchschnittlichen Latenzen/Pitches mit Hilfe des Mittelwertes $\hat{n}_j^{aktiv}(c, \mathcal{PS})$ über die zeitabhängigen Werte von n^{aktiv} bestimmt werden. Dieser Mittelwert ergibt sich durch Verallgemeinerung des am Beispiel von Abbildung 5.5 diskutierten Schemas zu

$$\hat{n}_j^{aktiv}(c, \mathcal{PS}) \quad = \quad \frac{1}{a_j} \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right), \tag{5.6}$$

wobei n^{max} die maximale Anzahl von Arbiter-Eingängen bezeichnet. Wie in Abbildung 5.5 dargestellt, stehen die beiden Terme innerhalb der Klammern jeweils für die Eingänge mit höherem Datenaufkommen als a_j („$j \cdot a_j$“) bzw. die Eingänge mit gleichem oder niedrigerem Datenaufkommen, einschließlich a_j selbst („$\sum a_s$“). Bei grafischer Betrachtung in Abbildung 5.5 stellen diese Terme die Gesamtlänge aller Balken abzüglich der grau dargestellten Bereiche dar. Eine detaillierte Herleitung von Gleichung (5.6) findet sich in Anhang A.1.

Durch Einsetzen von $\hat{n}_j^{aktiv}(c, \mathcal{PS})$ in die Gleichungen (5.2) und (5.3) ergeben sich dann die folgenden Berechnungsvorschriften für die gesuchten Latenzen und Pitches:

$$L_j^{avg}(c, \mathcal{PS}) = \frac{T^{ZS}(c)}{a_j} \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right), \tag{5.7}$$

$$P_j^{avg}(c, \mathcal{PS}) = \frac{P^{res}(c)}{a_j} \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right). \tag{5.8}$$

Für den Fall des Rundlaufverfahrens mit einer einzelnen Arbitrierungseinheit kommen Nowotsch et al. in [78] bei einer ähnlichen Betrachtung zu einer vergleichbaren Formulierung.

5.3.2. Latenz und Pitch für ganze Routen

Die Gleichungen (5.7) und (5.8) modellieren die Latenz und den Pitch während der Ausführung der Code-Segmente $cs_i \in \mathcal{PS}$ für eine einzelne Komponente. Zur Bestimmung der letztendlichen Zugriffszeiten müssen diese Größen für ganze Routen R_{kl} bekannt sein. Die Ende-zu-Ende-Latenz $L^{rt}\big(R_{kl}, \mathcal{PS}\big)$ einer Route R_{kl} ergibt sich aus der Summe der Latenzen aller Einzelkomponenten:

$$L^{rt}\big(R_{kl}, \mathcal{PS}\big) = \sum_{c \in R_{kl}} L^{avg}_{\mathfrak{J}(c, R_{kl}, \mathcal{PS})}\big(c, \mathcal{PS}\big). \tag{5.9}$$

Die Verwendung der durchschnittlichen Latenzen L^{avg} ist hier möglich, da die Durchschnittsbildung über die Dateneinheiten beim späteren Aufsummieren der Latenzen L^{rt} wieder kompensiert wird (siehe nachfolgender Abschnitt 5.3.3).

Im Gegensatz zur Latenz ergibt sich der Pitch/Durchsatz einer Reihenschaltung von Komponenten nicht aus der Summe über alle Komponenten, sondern wird – unter Vernachlässigung transienter Effekte – durch die Komponente mit dem niedrigsten Durchsatz bzw. höchsten Pitch festgelegt. Damit ergibt sich der Zusammenhang

$$\hat{P}^{rt}\big(R_{kl}, \mathcal{PS}\big) = \max_{c \in R_{kl}} \left\{ P^{avg}_{\mathfrak{J}(c, R_{kl}, \mathcal{PS})}\big(c, \mathcal{PS}\big) \right\}. \tag{5.10}$$

Hier liefern die durchschnittlichen Werte P^{avg} aufgrund der max-Operation jedoch nur unter bestimmten Annahmen einen sicheren oberen Grenzwert. Aus

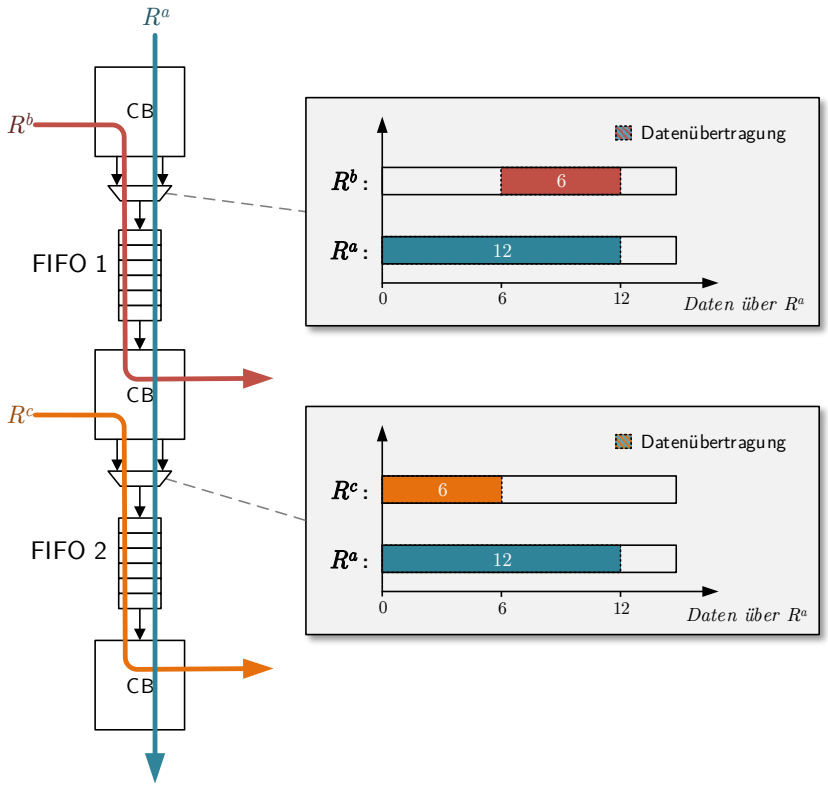


Abbildung 5.6.: Beispiel für den Ausschnitt eines Komponenten-Graphen mit drei Routen R^a, R^b, R^c und dem zeitlichen Verlauf des Datenaufkommens an den Arbitern

diesem Grund wird im Folgenden zwischen $\hat{P}^{rt}$ für den gemittelten Wert und P^{rt} für den oberen Grenzwert ohne zusätzliche Annahmen unterschieden. Anders als bei der Latenz kann die Summe in der Durchschnittsbildung aus Gleichung (5.8) i. A. nicht mit dem max-Operator vertauscht werden. Es macht also einen Unterschied, ob zuerst der Durchschnitt über die Dateneinheiten auf Komponenten-Ebene gebildet wird, oder ob die Pitches für die gesamte Route berechnet werden, bevor die Durchschnittsbildung erfolgt. Abbildung 5.6 veranschaulicht die zugrundeliegende Situation anhand dreier Crossbar-Komponenten (CB) und zweier FIFO-Puffer. Das Beispiel ähnelt dem Modell, dass in Abbildung 5.1b für den NoC-Teil der Plattform verwendet wird. In der dargestellten Situation soll nun der Pitch $P^{rt}(R^a, \mathcal{PS})$ der Route R^a unter der Annahme eines Datenaufkommens von 12 Dateneinheiten für R^a und jeweils 6 Dateneinheiten für R^b und R^c bestimmt werden. Aus den abgebildeten zeitlichen Überlappungen der Datentransfers auf den einzelnen Routen ergibt sich der durchschnittliche Pitch P^{avg} an den Eingängen der beiden Arbiter gemäß Gleichung (5.8) dann in beiden Fällen zu $1,5 \cdot P^{res}$.

Die Berechnung von $\hat{P}^{rt}(R^a, \mathcal{PS})$ anhand der Durchschnittswerte berücksichtigt jedoch noch nicht, dass sich die zeitlichen Verteilungen des Datenaufkommens an den beiden FIFOs u. U. ungünstig überlagern können. Eine solche Situation ist in den beiden Diagrammen in Abbildung 5.1b dargestellt. Geht man von dem Extremfall mit einer Pufferlänge von 0 für beide FIFOs aus, so kann der Arbiter von FIFO 1 nur dann eine Dateneinheit übertragen, wenn FIFO 2 die Dateneinheit unmittelbar weiterleiten kann. Die maximalen Pitches beider Komponenten sind in dem Fall also stets dieselben. Für die ersten 6 Dateneinheiten von R^a wird der Pitch P^{rt} der Route demnach durch R^c und für die zweiten 6 Dateneinheiten durch R^b auf jeweils $2 \cdot P^{res}$ festgelegt (unter der Annahme, dass beide Arbiter den gleichen Basis-Pitch P^{res} haben). Wird die Puffergröße hingegen auf 6 festgelegt, so können die ersten 6 Dateneinheiten mit vollem Durchsatz P^{res} in den Puffer von FIFO 1 geschrieben werden, ohne dabei von dem niedrigeren Durchsatz in FIFO 2 beeinflusst zu werden. Für die zweiten 6 Dateneinheiten sinkt dann der Durchsatz in FIFO 1 aufgrund der Interferenz mit R^b, während er an FIFO 2 ansteigt. Dadurch leert sich der erste Puffer wieder langsam, weswegen der zugehörige Arbiter nicht durch den Rückstau ausgebremst wird. Im Mittel stimmt der Durchsatz in diesem Fall dann mit dem Ergebnis $1,5 \cdot P^{res}$ aus Gleichung (5.10) überein.

An dem diskutierten Beispiel wird deutlich, dass die Pufferlängen für die zeitliche Entkopplung der Pitches der einzelnen Komponenten eine entscheidende Rolle spielen. Gleichung (5.10) gilt dabei nur unter der Annahme, dass es keine transienten Rückstau-Effekte durch die zeitliche Verteilung des Datenaufkommens gibt. Diese Voraussetzung muss anhand der Pufferlängen und des Datenaufkommens durch eine (pessimistische) Füllstands-Analyse im Einzelnen verifiziert

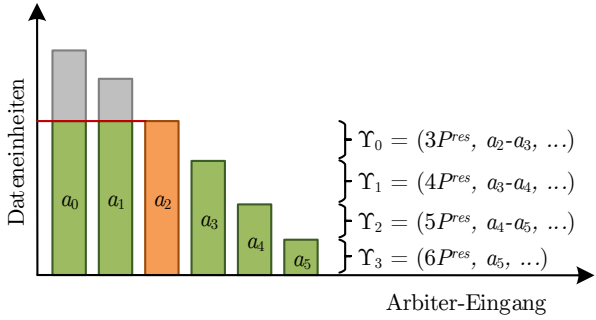


Abbildung 5.7.: Einteilung der Dateneinheiten an Eingang 2 in Intervalle Υ_i mit gleichem Pitch

werden. Zu diesem Zweck kann z. B. die Methode des Netzwerk-Kalküls (siehe Abschnitt 3.1.2.2) unter Verwendung der ungünstig Überlagerten zeitlichen Lastverteilungen gemäß Abbildung 5.6 als Ankunftsfunktionen herangezogen werden.

Falls auf die Verwendung des Netzwerk-Kalküls verzichtet werden soll, oder wenn die Puffer-Kapazitäten generell nicht ausreichen, so muss die pessimistische Variante unter Annahme einer Pufferlänge von 0 zugrunde gelegt werden. Um $P^{rt}\big(R_{kl}, \mathcal{PS}\big)$ in diesem Fall bestimmen zu können, soll im Folgenden ein entsprechender Algorithmus als Ersatz für Gleichung (5.10) hergeleitet werden. Wie zuvor beschrieben, ergibt sich im Allgemeinen an jeder Komponente c mit n^{max} zusammenlaufenden Routen R_{kl} eine Lastverteilung nach dem Schema von Abbildung 5.5. Aus diesen allgemeinen Verteilungen muss nun, wie für das vereinfachte Beispiel aus Abbildung 5.6, die für den Pitch ungünstigste Überlagerung ermittelt werden. In den Einzelverteilungen lassen sich die Dateneinheiten der betrachteten Route in verschiedene Intervalle mit einer festen Anzahl aktiver Arbiter-Eingänge n^{aktiv} einteilen. In Abbildung 5.5 wären dies z. B. die in Abschnitt 5.3.1 diskutierten a_5 der a_2 Dateneinheiten mit $n^{aktiv} = 6$ oder die $a_4 - a_5$ Dateneinheiten mit $n^{aktiv} = 5$. Mit $j = \mathfrak{J}(c, R_{kl}, \mathcal{PS})$ lassen sich für jede Komponente im Allgemeinen $n^{max} - j$ solcher Intervalle mit unterschiedlichen Pitches identifizieren. Im Folgenden sei ein solches Intervall durch das Tupel $\Upsilon_i = (P^{eff}, \alpha, c, R_{kl}, \mathcal{PS})$, bestehend aus dem effektiven Pitch P^{eff}, der Anzahl α der abgedeckten Dateneinheiten sowie der aktuellen Komponente c, der Route R_{kl} und dem Ausführungskontext $\mathcal{PS}$, beschrieben. Dabei ist α

definiert durch die jeweilige Anzahl der $a_{\mathfrak{J}(c,R_{kl},\mathcal{PS})}$ Dateneinheiten des aktuellen Arbiter-Eingangs, die denselben Wert für n^{aktiv} aufweisen. Abbildung 5.7 veranschaulicht dies an einem Beispiel für eine einzelne Komponente c.

Algorithmus 5.1 Algorithmus zur Bestimmung von $P^{rt}\big(R_{kl},\mathcal{PS}\big)$

Input: Route R_{kl}, Kontext $\mathcal{PS}$
Output: Pitch P^{rt}

$IL \leftarrow \emptyset$
for all $c \in R_{kl}$ **do**
  for $s = \mathfrak{J}(c, R_{kl}, \mathcal{PS})$ **to** $n^{max} - 2$ **do**
    $\Upsilon \leftarrow ((s+1) \cdot P^{res}(c),\ a_s - a_{s+1},\ c,\ R_{kl},\ \mathcal{PS})$
    $IL \leftarrow IL \cup \{\Upsilon\}$
  end for
  $\Upsilon \leftarrow (n^{max} \cdot P^{res}(c),\ a_{n^{max}-1},\ c, R_{kl},\ \mathcal{PS})$
  $IL \leftarrow IL \cup \{\Upsilon\}$
end for

$IL \leftarrow$ SortInDescendingOrder(IL, P^{eff})
$n \leftarrow 0$
$P^{acc} \leftarrow 0$

for $i = 0$ **to** $|IL| - 1$ **do**
  $(P^{eff}, \alpha, c, R_{kl}, \mathcal{PS}) \leftarrow IL[i]$
  if $n + \alpha > \overline{N}(R_{kl}, \mathcal{PS})$ **then**
    $P^{acc} \leftarrow P^{acc} + P^{eff} \cdot \big(\overline{N}(R_{kl}, \mathcal{PS}) - n\big)$
    break
  else
    $P^{acc} \leftarrow P^{acc} + \alpha * P^{eff}$
    $n \leftarrow n + \alpha$
  end if
end for

$P^{rt} \leftarrow {}^{P^{acc}}\!/_{\overline{N}(R_{kl},\mathcal{PS})}$

Die ungünstigste Überlagerung entlang einer Route R_{kl} entsteht dann, wenn die $\overline{N}(R_{kl}, \mathcal{PS})$ Dateneinheiten auf der Route vollständig durch eine Aneinanderreihung der teuersten Intervalle Υ_i, d. h. der Intervalle mit dem höchsten Pitch, überdeckt werden. Dabei sind die Intervalle aller Komponenten $c \in R_{kl}$ der Route zu berücksichtigen. Der Durchschnitt der Pitches aus dieser Überlagerung kann dann für die Berechnung eines sicheren Grenzwerts für die Verzögerungszeiten

der Gesamtheit aller $\overline{N}(R_{kl}, \mathcal{PS})$ Dateneinheiten verwendet werden. Algorithmus 5.1 zeigt einen Algorithmus zur Berechnung von $P^{rt}\big(R_{kl}, \mathcal{PS}\big)$ auf Basis dieser Überlegungen. Darin werden zunächst alle Intervalle Υ in einer Schleife über die Komponenten $c \in R_{kl}$ ermittelt und zu einer Liste *IL* hinzugefügt. Die Berechnung von P^{eff} und α folgt dabei dem Schema aus Abbildung 5.7. Die Liste *IL* wird anschließend mittels „SortInDescendingOrder“ in absteigender Reihenfolge nach P^{eff} sortiert, sodass die teuersten Intervalle am Anfang stehen. In einer Schleife über diese Liste werden die teuersten Pitches P^{eff} dann so lange in P^{acc} akkumuliert, bis alle $\overline{N}(R_{kl}, \mathcal{PS})$ Dateneinheiten abgedeckt sind. Der daraus resultierende Mittelwert der Pitches stellt schließlich den mittleren Pitch P^{rt} für die ungünstigste Überlagerung der Einzelverteilungen dar.

5.3.3. Bestimmung der Zugriffszeit

Mit der Latenz L^{rt} und dem Pitch P^{rt} aus Gleichung (5.9) bzw. Algorithmus 5.1 und/oder Gleichung (5.10) lässt sich die Performanz einer Route für den Ausführungskontext $\mathcal{PS}$ im ungünstigsten Fall beschreiben. Um daraus die Wartezeit eines Prozessorkerns auf den Abschluss eines Speicherzugriffs abzuleiten, muss auch der Zugriffstyp (d.h. Lese- oder Schreibzugriff) berücksichtigt werden. Bei Schreibzugriffen ist ferner zwischen sogenannten *Non-posted Writes* und *Posted Writes* zu unterscheiden. Bei letzteren muss nach dem Start eines Schreibzugriffs nicht erst auf dessen Beendigung gewartet werden, bevor weitere Transaktionen initiiert werden können. Lesezugriffe sowie *non-posted Writes* blockieren den Bus bzw. die Kommunikationsverbindung dagegen bis zum Abschluss der Transaktion. In einfachen Prozessorarchitekturen wird die Ausführung weiterer Instruktionen während des Wartens auf einen solchen Speicherzugriff pausiert. Die entsprechende Wartezeit geht damit direkt in die WCET der betroffenen Software ein.

Im Fall von *posted Writes* muss lediglich die Schreibanfrage gesendet werden, bevor die Ausführung der Software fortgesetzt werden kann. Die Anzahl der für diese Anfrage anfallenden Dateneinheiten, einschließlich der Protokollinformationen, sei im Folgenden mit n^{p+} bezeichnet[3]. Die Wartezeit D^p eines Prozessorkerns auf einzelne *posted Writes* wird dann durch das Senden der Anfrage mit einem Durchsatz von $1/P^{rt}(R_{kl}, \mathcal{PS})$ bestimmt und ergibt sich zu

$$D^p\big(R_{kl}, \mathcal{PS}\big) = n^{p+} \cdot P^{rt}\big(R_{kl}, \mathcal{PS}\big)\,. \tag{5.11}$$

[3]Im Allgemeinen ist der Umfang der Protokoll-Informationen in dem Modell aus Abbildung 5.2 entlang einer Route nicht notwendigerweise konstant. Zugunsten einer vereinfachten formalen Darstellung wird dies jedoch an dieser Stelle vernachlässigt. Die Umsetzung im Rahmen der in Abschnitt 5.4 spezifizierten ADL bezieht diesen Freiheitsgrad dagegen mit ein.

Bei Zugriffen vom Typ *non-posted* wird eine Anfrage über R_{kl} i. A. von der Speicherkomponente m_l mit einer Antwort über R_{lk} beantwortet. Der Zugriff setzt sich also aus einer Anfrage mit n^{np+} und einer Antwort mit n^{np-} Dateneinheiten (einschließlich Protokollinformationen) zusammen. Die Verzögerungszeit beinhaltet zunächst – wie schon bei *posted Writes* – das Versenden der Anfrage mit dem Durchsatz $1/P^{rt}(R_{kl}, \mathcal{PS})$ und zusätzlich noch den Empfang der Antwort mit dem Durchsatz $1/P^{rt}(R_{lk}, \mathcal{PS})$. Da die Anfrage erst nach einer Latenz von $L^{rt}\big(R_{kl}, \mathcal{PS}\big)$ bei der Speicherkomponente ankommt und auch die Antwort zusätzlich um $L^{rt}\big(R_{lk}, \mathcal{PS}\big)$ verzögert wird, fließen diese beiden Latenzen ebenfalls in die Wartezeit ein. Insgesamt ergibt sich die Verzögerungszeit $D^{np}\big(R_{kl}, \mathcal{PS}\big)$ für Zugriffe vom Typ *non-posted* damit zu

$$D^{np}\big(R_{kl}, \mathcal{PS}\big) = n^{np+} \cdot P^{rt}\big(R_{kl}, \mathcal{PS}\big) + n^{np-} \cdot P^{rt}\big(R_{lk}, \mathcal{PS}\big) + L^{rt}\big(R_{kl}, \mathcal{PS}\big) + L^{rt}\big(R_{lk}, \mathcal{PS}\big)\,. \quad (5.12)$$

Die Latenzen in dieser Gleichung basieren auf der in Abschnitt 5.2.2.1 getroffenen Annahme leerer FIFO-Puffer im Netzwerk. Demnach werden keine Rückstau-Effekte innerhalb einer Route berücksichtigt. Da ein Prozessor auf die Beendigung eines Zugriffs vom Typ *non-posted* warten muss, bevor ein weiterer gestartet werden kann, sind alle mit dem ersten Zugriff verbundenen Datenübertragungen beim Start einer neuen Transaktion bereits abgeschlossen. Die Annahme leerer Puffer für die betroffenen Routen R_{kl} und R_{lk} trifft folglich zu[4].

Ausgehend von den Speicherzugriffszeiten aus den Gleichungen (5.11) und (5.12) kann deren Beitrag zur lokalen WCET eines Code-Segments cs_i ermittelt werden. Letzterer entspricht der Summe $\overline{D}(cs_i)$ der Verzögerungszeiten aller Speicherzugriffe in cs_i, die wie folgt berechnet werden kann:

$$\overline{D}(cs_i) = \sum_{\forall m_l} \Big[N^{mem,np}(cs_i, m_l) \cdot D^{np}\big(R_{kl}, \mathcal{PS}(cs_i)\big) + N^{mem,p}(cs_i, m_l) \cdot D^{p}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \Big] \quad \text{mit} \quad R_{kl} = \big(\mu^{cs}(cs_i), \ldots, m_l\big)\,. \quad (5.13)$$

Hierbei bezeichnet $\mu^{cs} : \mathcal{CS} \mapsto \mathcal{C}$, in Analogie zu der Zuweisungsfunktion μ in Schedules, den Prozessorkern c_k, auf dem cs_i ausgeführt wird. Damit entspricht R_{kl} der Route, die von cs_i verwendet wird, um auf m_l zuzugreifen. Die in

[4] Eventuell vorangegangene *posted Writes* können nach wie vor Puffer-Ressourcen belegen, sodass ggf. noch deren Latenz als zusätzliche Wartezeit beaufschlagt werden muss.

Kapitel 3 eingeführte Anzahl $N^{mem}(cs_i, m_l)$ der Speicherzugriffe pro Code-Segment und Speicher wird dabei in die Anzahl $N^{mem,p}$ der *posted Writes* und die Anzahl $N^{mem,np}$ der übrigen Speicherzugriffe unterteilt. Für den Fall, dass zur Entwurfszeit nicht bekannt ist, ob ein Zugriff als *posted Write* ausgeführt wird, kann D^{np} anstelle von D^{p} als pessimistische Abschätzung verwendet werden.

5.4. Erweiterte Architekturbeschreibungssprache für Echtzeitsysteme

Das in Abschnitt 5.2 präsentierte Modell des Komponenten-Grahpen *CG* definiert einen Grundbestand an Informationen, die insbesondere für die statische Berechnung der Zugriffszeiten erforderlich sind. Einige der in Abschnitt 5.1 genannten Aspekte, insbesondere die Beschreibung der Adressräume, ist im *CG* jedoch noch nicht berücksichtigt. Um die Informationen des *CG* strukturiert darzustellen und die fehlenden Aspekte zu ergänzen, wird im Folgenden eine erweiterte Architekturbeschreibungssprache für Echtzeitsysteme vorgestellt (siehe auch [RHD+16]). Die Spezifikation baut auf dem existierenden ADL-Standard *„Software-Hardware Interface for Multi-Many-Core“* [111], kurz SHIM, auf und ergänzt ihn um einige zur Darstellung des *CG* nötige Informationen.

5.4.1. Existierende ADL als Basis für Erweiterungen

In Abschnitt 3.1.2.1 wurden mit AADL, EAST-ADL, SHIM und UML-MARTE vier ADLs genannt, die für die Darstellung der nötigen Informationen in Frage kommen. Keiner dieser Kandidaten wurde jedoch für die Modellierung von Interferenzeffekten mit einem für das vorgestellte Modell ausreichenden Detailgrad konzipiert (vgl. Abschnitt 3.1.2.1). Daher ist für jede der genannten ADLs eine entsprechende Erweiterung erforderlich, um die zu Beginn des Kapitels festgelegte Zielsetzung zu erreichen. Als Grundlage für die Realisierung einer solchen erweiterten ADL wird in dieser Arbeit die SHIM-Spezifikation verwendet. Ausschlaggebend für diese Auswahl sind die folgenden Hauptvorteile von SHIM im Vergleich zu den genannten Alternativen:

Fokus auf die Schnittstellen von Hardware & Software: SHIM fokussiert sich, wie schon am Namen der Spezifikation ersichtlich, auf die Schnittstelle zwischen Hardware und Software in Mehrkernprozessoren. Dies entspricht weitestgehend den Schwerpunkten, die auch für die Interferenz-Analyse relevant sind. So befassen sich auch die bisher präsentierten Modelle weitgehend mit der Interaktion von Software und Hardware in Mehrkernprozessoren, insbesondere hinsichtlich des Zeitverhaltens.

Gute Erweiterbarkeit: SHIM basiert auf der weit verbreiteten Auszeichnungssprache *Extensible Markup Language* (XML) [117], die auf gute Erweiterbarkeit ausgelegt ist. Dadurch können die zu spezifizierenden Erweiterungen mit den in XML vorgesehenen Erweiterungsmechanismen, wie z. B. den sogenannten *XML-Schemata* (auch als *XML Schema Definition*, kurz XSD, bezeichnet), aufgebaut werden. Dadurch bleibt die Kompatibilität mit existierenden XML-basierten Programmbibliotheken und Software-Werkzeugen erhalten.

Kompatibles Modell zur Performanz-Beschreibung: Die Beschreibung von Performanz-Aspekten in SHIM basiert auf der Latenz L und dem Pitch P. Diese beiden Parameter werden auch in dem zuvor präsentierten Modell zur statischen Berechnung von Zugriffszeiten verwendet.

Gute Übereinstimmung mit den konkreten Anforderungen: Von den sechs in Abschnitt 5.1 formulierten wesentlichen Anforderungen werden die letzten vier (Anforderung 3 bis Anforderung 6) durch die SHIM-Spezifikation bereits weitgehend erfüllt. Die darüber hinaus benötigten Erweiterungen beschränken sich daher hauptsächlich auf die zur Darstellung des *CG* relevanten Anforderungen 1 und 2.

Diese Arbeit geht von der SHIM-Spezifikation in Version 1.0 [111] aus, über die im Folgenden zunächst ein Überblick gegeben wird, bevor Abschnitt 5.4.2 die spezifischen Erweiterungen präsentiert.

5.4.1.1. Übersicht über die SHIM-Spezifikation

Das Modell einer Plattform-Architektur in SHIM gliedert sich in die folgenden drei Hauptaspekte:

1. die Beschreibung der Komponenten-Hierarchie (sog. *ComponentSets*),
2. die Beschreibung der Adressräume (sog. *AddressSpaceSets*) und
3. die Beschreibung der Kommunikationsverbindungen (sog. *CommunicationSets*).

Erstere bildet eine Hierarchie aus *ComponentSets*, in die die Beschreibungen von Prozessorkernen, Speichern und komplexeren Kommunikationskomponenten eingebettet sind. Eine vereinfachte UML-basierte Darstellung des zugrundeliegenden Modells ist in Abbildung 5.8 als Klassendiagramm abgebildet. Bei den Haupt-Komponenten unterscheidet das ADL-Format grundsätzlich zwischen *MasterComponents* und *SlaveComponents*, wobei letztere im Wesentlichen für Speicherkomponenten verwendet werden. Bei *MasterComponents* entscheidet

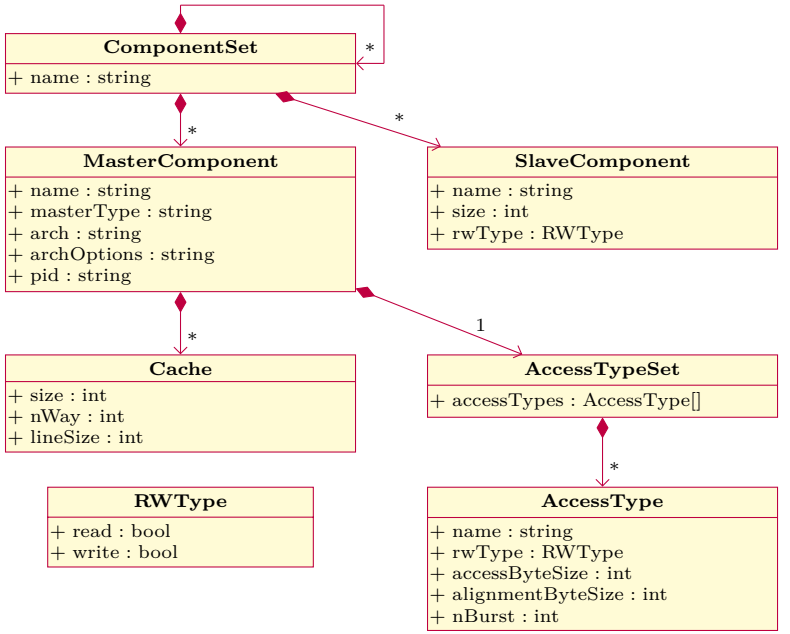


Abbildung 5.8.: Vereinfachtes UML-Klassendiagramm zur SHIM Komponenten-Hierarchie

das Feld *masterType*, ob es sich um eine Prozessoreinheit, eine Übertragungseinheit oder eine sonstige Komponente handelt. *MasterComponents* können zudem Caches enthalten und besitzen eine Liste aus verschiedenen, von der Komponente unterstützten *AccessTypes*. Jeder *AccessType* stellt Informationen über die Art eines Speicherzugriffs, einschließlich der Anzahl der Bytes pro Zugriff, der Länge eines *Bursts* und der Klassifikation in Lese- oder Schreibzugriffe, bereit.

Das SHIM-Modell für Adressräume besteht im Wesentlichen aus *AddressSpaces*, die in eine Menge überlappungsfreier *SubSpaces* unterteilt sind. Die *SubSpaces* stellen dabei jeweils einen zusammenhängenden Speicheradressbereich, bestehend aus einer Start- und einer Endadresse, dar. Über sogenannte *MasterSlaveBindings* können für jeden *SubSpace* eine Reihe von Paaren aus einer *MasterComponent* und einer *SlaveComponent* spezifiziert werden. Diese Paare definieren jeweils eine Speicherkomponente, die von der zugehörigen *MasterComponent* über den betrachteten *SubSpace* angesprochen werden kann.

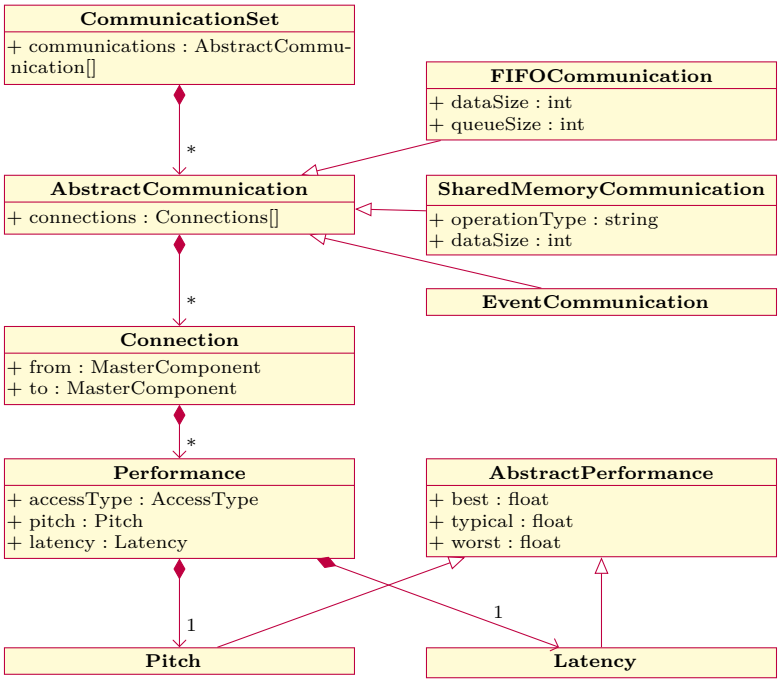


Abbildung 5.9.: Vereinfachtes UML-Klassendiagramm zum Kommunikationsmodell von SHIM

Der dritte Aspekt dient der Beschreibung von Kommunikation in der Plattform und erlaubt die Modellierung verschiedener Arten von Kommunikationsverbindungen zwischen *MasterComponents*. Abbildung 5.9 zeigt eine vereinfachte Darstellung dieses Modells als UML-Klassendiagramm. Kommunikation wird darin durch den Typ *AbstractCommunication* repräsentiert, der mit den *Connections* eine Menge von gerichteten Punkt-zu-Punkt-Verbindungen zwischen jeweils zwei *MasterComponents* zusammenfasst. Jeder *Connection* sind wiederum eine Menge von *Performance*-Elementen zugeordnet, die jeweils die Performanz der Verbindung hinsichtlich Latenz und Pitch für einen bestimmten *AccessType* beschreiben. Sowohl Pitch als auch Latenz werden durch *AbstractPerformance* jeweils für den günstigsten („best“), durchschnittlichen („typical“) und ungünstigsten („worst“) Fall angegeben. Im Hinblick auf den Typ *AbstractCommunication* sieht der SHIM-Standard eine Reihe von Spezialisierungen vor, zu denen u. a.

die dargestellten Typen *FIFOCommunication*, *SharedMemoryCommunication* und *EventCommunication* zählen.

5.4.2. Erweiterung der ADL Spezifikation

Ein Vergleich des in Abbildung 5.2 dargestellten Referenz-Modells für den Plattform-Komponenten-Graphen mit dem Modellierungsansatz von SHIM in den Abbildungen 5.8 und 5.9 offenbart einige Parallelen. Im Einzelnen lassen sich viele der benötigten Informationen in beiden Darstellungen fast Eins-zu-Eins wiederfinden:

- Die gerichteten Punkt-zu-Punkt-Verbindungen (*Connections*) aus dem SHIM-Kommunikationsmodell lassen sich auch als Kanten eines gerichteten Graphen interpretieren und bilden damit eine Entsprechung zu der Klasse *Kante* aus Abbildung 5.2.
- Die *AccessTypes* aus SHIM lassen sich auf die Klasse *ZugriffsTyp* in Abbildung 5.2 abbilden.
- Die *Performance*-Elemente für Latenz und Pitch in SHIM decken neben dem ungünstigsten Fall auch den typischen und den günstigsten Fall ab. Sie stellen damit eine Obermenge der im Referenz-Modell aus Abbildung 5.2 enthaltenen Informationen bereit.
- Speicherkomponenten im Referenz-Modell können direkt auf *SlaveComponents* in SHIM abgebildet werden.
- Die Attribute von *FIFOCommunication* stellen, ebenso wie der Typ *FIFO-Verbindung* aus dem Referenz-Modell, die nötigen Informationen über die Größe der FIFO-Puffer bereit.

Zur Darstellung aller für die Interferenz-Analyse benötigten Informationen fehlen im SHIM-Standard jedoch insbesondere noch die folgenden Aspekte:

Zu hoher Abstraktionsgrad im Kommunikationsmodell: Die in SHIM vorgesehenen Kommunikationsmechanismen, z. B. die Klassen *SharedMemoryCommunication* für Kommunikation über gemeinsame Speicher oder *EventCommunication* für das Versenden von Ereignissen, sind eher auf der Abstraktionsebene des Programmiermodells angesiedelt. Sie repräsentieren damit vorwiegend die Software-Sicht, ohne die Implementierung auf der Hardware-Seite im Detail zu beschreiben. Für die statische Analyse von Speicherzugriffszeiten muss dieses Modell neu interpretiert und erweitert werden, um tatsächliche Hardware-Einheiten wie Busse oder Hardware-basierte FIFO-Puffer darstellen zu können.

Fehlende Verknüpfung von SlaveComponents & Kommunikation: Durch das abstraktere Verständnis von Kommunikation sind Speicherkomponenten in SHIM nicht in das durch die *Connections* aufgebaute Kommunikationsnetzwerk eingebunden. Eine *Connection* kann ausschließlich *MasterComponents* verbinden, während die für Speicher verwendeten *SlaveComponents* außen vor bleiben.

Fehlende Information zur Arbitrierung: Das Performanz-Modell auf Basis der *Performance*-Elemente bezieht mit den *AccessTypes* lediglich die Zugriffstypen als offene Parameter für Latenzen und Pitches ein. Das Arbitrierungs-Schema und die damit verbundene Abhängigkeit der Performanz von der Anzahl kollidierender Datenübertragungen n^{aktiv} (vgl. Abschnitt 5.2.2.1) wird jedoch nicht berücksichtigt.

Fehlende Information zur Routenplanung: SHIM beinhaltet keine Informationen über die Routenplanung in komplexen Kommunikationsverbindungen, wie z. B. NoCs.

Keine Beschreibung von Protokollinformationen: SHIM stellt keine Möglichkeit bereit, um die für Protokollinformationen anfallenden Dateneinheiten bei Anfragen/Antworten im Rahmen von Lese-/Schreibzugriffen zu spezifizieren. Insbesondere in NoCs ist diese Information erforderlich, um das Datenaufkommen präzise nach oben abschätzen zu können.

Um die fehlenden Aspekte zu ergänzen, erweitert das im Rahmen dieser Arbeit vorgestellte ADL-Schema die SHIM-Spezifikation vor allem im Bereich des Kommunikationsmodells. Die im Folgenden skizzierten Modifikationen lassen sich in ein erweitertes XSD Schema integrieren, wobei die Rückwärts-Kompatibilität zu SHIM erhalten bleibt (siehe auch [RHD+16]). Das im Vergleich zu Abbildung 5.8 erweiterte Komponenten-Modell ist in Abbildung 5.10 dargestellt. Die wesentliche Anpassung besteht darin, dass *MasterComponents* und *SlaveComponents* eine gemeinsame Basisklasse haben, die später als Knoten des Plattform-Komponenten-Graphen betrachtet werden kann. Bei den Caches wurde außerdem das Attribut *replacementStrategy* ergänzt, um die zur Bestimmung der WCET benötigte Cache-Verdrängungsstrategie (vgl. Abschnitt 2.1) angeben zu können.

Die umfangreichsten Erweiterungen werden im Kommunikationsmodell benötigt, um die oben genannten fehlenden Informationen zu ergänzen. Abbildung 5.11 zeigt das gegenüber Abbildung 5.9 entsprechend erweiterte Kommunikationsmodell. Damit verbunden ist die bereits erwähnte Neuinterpretation der von *AbstractCommunication* abgeleiteten Kommunikationselemente, die nun tatsächliche Hardware-Einheiten repräsentieren. Dementsprechend wird z. B. mit Elementen vom Typ *FIFOCommunication* und deren *Connections* eine Menge von tatsächlich in der Hardware vorhandenen FIFO-Puffern modelliert. Eine Änderung an den *Connections* erlaubt es außerdem, zwei Komponenten vom

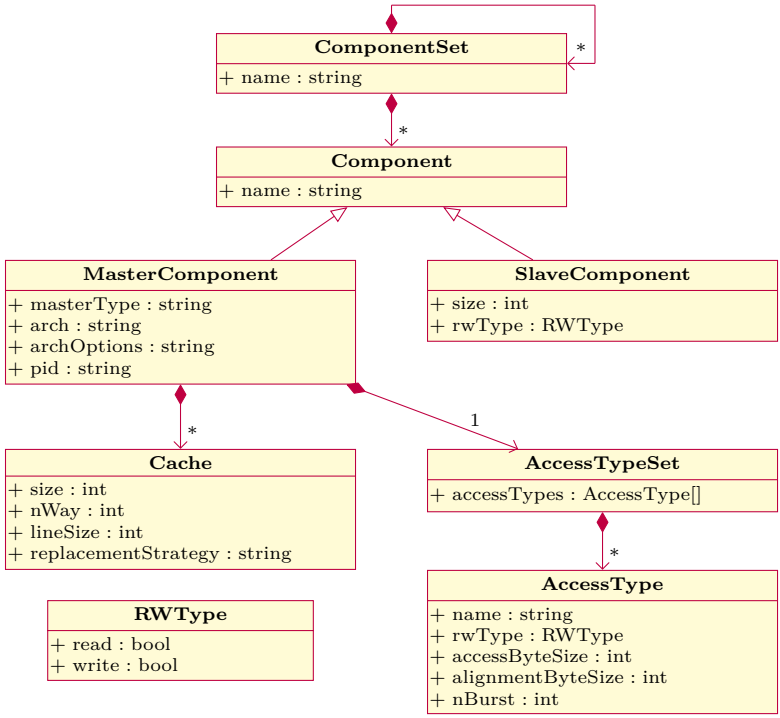


Abbildung 5.10.: Vereinfachtes UML-Klassendiagramm zur Komponenten-Hierarchie der erweiterten ADL

(neu hinzugefügten) Typ *Component* zu verbinden, sodass auch *SlaveComponents* über einen FIFO-Puffer angebunden werden können. Da Bus-Systeme meist keine Punkt-zu-Punkt-Verbindungen darstellen, sondern i. A. mehr als zwei Bus-Teilnehmer verbinden, wird mit *BusCommunication* außerdem eine weitere Klasse von Kommunikationselementen eingeführt. Im Gegensatz zu den in SHIM vorhandenen Elementen besteht eine *BusCommunication* nicht aus einzelnen *Connections*, sondern wird durch zwei Mengen aus daran angeschlossenen Komponenten beschrieben. Eine dieser Mengen enthält dabei die Master-Komponenten, die u. a. neue Bus-Transaktionen starten können, und die andere die übrigen Slave-Komponenten. Mit dieser Erweiterung bieten die beiden Klassen *FIFOCommunication* und *BusCommunication* dann die Möglichkeit, einen Plattform-Komponenten-Graphen vollständig abzubilden.

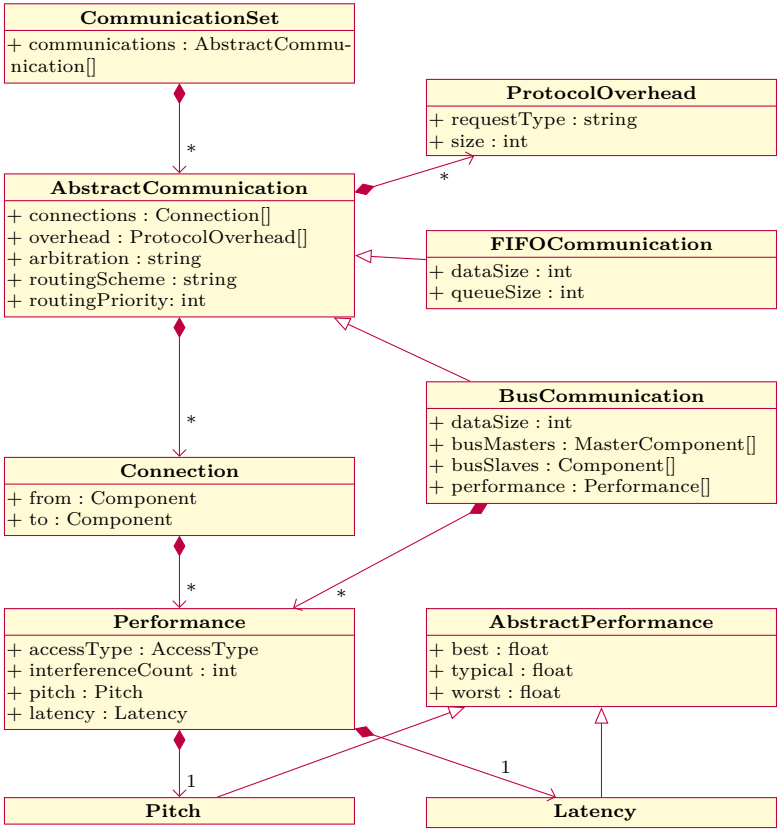


Abbildung 5.11.: Vereinfachtes UML-Klassendiagramm zum Kommunikationsmodell der erweiterten ADL

Zusätzliche Felder in *AbstractCommunication* erlauben es außerdem, die relevanten Aspekte der Arbitrierung, Routenplanung und Kommunikationsprotokolle in die Plattform-Beschreibung einzubeziehen. Hinsichtlich der Arbitrierung lässt sich mit dem Feld „arbitration“ explizit ein Arbitrierungs-Schema angeben. Unter Verwendung des Feldes „interferenceCount“ in den *Performance*-Elementen kann das Zeitverhalten, z. B. bei weniger verbreiteten Arbitrierungsmethoden, auch frei definiert werden. Hierzu können mehrere *Performance*-Elemente für denselben Zugriffstyp erstellt werden, wobei der Parameter „interferenceCount“ angibt, für welche Anzahl kollidierender Datentransfers n^{aktiv} die Angaben zu Latenz und Pitch jeweils gültig sind. Auf diese Weise lässt sich im Prinzip jeder beliebige Zusammenhang zwischen n^{aktiv} und den Latenzen/Pitches modellieren. Im Hinblick auf die Routenplanung ermöglichen es die Felder „routingScheme“ und „routingPriority“ das Verfahren zur Routenplanung bzw. eine Routing-Priorität vorzugeben. Beide Parameter gelten jeweils für alle *Connections*, die derselben übergeordneten *AbstractCommunication* zugeordnet sind. Die Priorität dient bei der Berechnung des kürzesten Pfades gemäß Abschnitt 5.2.3 als Kantengewicht und erlaubt damit eine flexible Anpassung der Routenplanung. Zur Beschreibung der erforderlichen Datenmenge für das Kommunikationsprotokoll, erhält *AbstractCommunication* das neue Feld „overhead“, dass auf eine Menge von Elementen vom Typ *ProtocolOverhead* verweist. Das Feld „requestType“ gibt dabei an, ob die jeweilige Angabe für eine *Schreibanfrage*, *Leseanfrage* oder die *Antwort auf eine Leseanfrage* gelten soll.

Insgesamt ermöglichen die skizzierten Erweiterungen, insbesondere durch die Änderungen am Kommunikationsmodell, eine ausreichend detaillierte Beschreibung der Hardware, um den in Abschnitt 5.3 beschriebenen Ansatz zur Interferenz-Analyse anwenden zu können. Da alle zusätzlichen Felder im XSD-Schema als optional markiert sind, bleibt die Rückwärtskompatibilität zu SHIM gewahrt.

5.5. Diskussion & Abgrenzung

In diesem Kapitel wurden mit einem generischen Plattformmodell und einem zugehörigen ADL-Schema die Voraussetzungen für eine plattformunabhängige Interferenz-Analyse zur Bestimmung der WCET paralleler Programme geschaffen. Das schließt insbesondere die formale Darstellung der Kommunikationsinfrastruktur durch den Komponenten-Graphen *CG* sowie eine darauf aufbauende Methode zur Berechnung von sicheren Grenzen für Speicherzugriffszeiten unter Berücksichtigung von Interferenzeffekten in Mehrkernprozessoren ein. Zur Parametrierung dieser Modelle und Berechnungsmethoden wird ein ADL-Schema verwendet, das darüber hinaus noch weitere Informationen für die Realisierung komplexer Werkzeugketten bereitstellt. Hierbei wird mit SHIM [111] ein

verbreiteter Standard als Grundlage verwendet, um die Wiederverwendbarkeit von Plattform-Beschreibungen und die Interoperabilität mit SHIM-basierten Software-Werkzeugen zu maximieren.

Zur Berechnung der Interferenz-bedingten Verzögerungen bei Speicherzugriffen verwendet der Ansatz die Ergebnisse einer MHP-Analyse (vgl. Abschnitt 3.3.3) und bezieht damit die Synchronisationsstruktur des parallelen Programms explizit mit ein. Verglichen mit existierenden Ansätzen, die Interferenzen anhand der Task-Laufzeiten begrenzen, eignet sich diese Lösung besser für gemischt-kritische Systeme (vgl. Abschnitt 3.1.1). Anders als die in Abschnitt 3.1.1 genannten Interferenz-Analysen ist der vorgestellte Ansatz durch das generische Hardware-Modell außerdem plattformunabhängig. Im Gegensatz zu Ansätzen wie [24], die die Synchronisationsstruktur an den Schedule anpassen, wird keine zusätzliche Synchronisation benötigt und das Programmiermodell muss weniger stark eingeschränkt werden.

Durch die kombinierte Betrachtung der hardwareseitigen Kommunikationsinfrastruktur und der softwareseitigen Synchronisationsstruktur schließt der vorgestellte Ansatz die Lücke zwischen statischen Software-Analysen und den in Abschnitt 3.1.2.2 diskutierten mathematischen Modellen für Chip-interne Netzwerke. Während letztere zwar garantierte Netzwerk-Latenzen bestimmen können, lassen sich die dafür benötigten Zugriffsmuster i. A. nicht ohne zusätzliche Annahmen (wie z. B. eine hardwareseitige Steuerung der Datenrate) aus einer statischen Programmanalyse gewinnen. Der vorgestellte Ansatz nutzt dagegen eine pessimistische Abschätzung der Zugriffsmuster auf Basis der MHP-Ergebnisse und kommt deshalb ohne solche Annahmen aus.

Ein Vergleich der präsentierten erweiterten ADL mit typischen ADL-basierten Plattformmodellen zeigt deutliche Unterschiede im Abstraktionsgrad der Hardware-Beschreibung. Für hardwarenahe Analysen, wie der detaillierten Interferenz-Berechnung aus Abschnitt 5.3, bieten existierende ADLs ohne zusätzliche Erweiterungen kein ausreichend präzises Modell. Um die in Abschnitt 5.1 formulierten Anforderungen erfüllen zu können, erweitert die vorgestellte ADL-Spezifikation daher mit SHIM eine verbreitete ADL für Mehrkernprozessoren.

Insgesamt bietet der vorgestellte Ansatz damit erstmals eine umfassende Lösung, die die Synchronisationsstruktur zur Begrenzung der Interferenz-Kosten nutzt, sowohl Hardware- als auch Software-Aspekte berücksichtigt und dabei weitgehend plattformunabhängig ist. Die Beiträge dieses Kapitels stellen insbesondere auch einen entscheidenden Baustein für die Realisierung des in Abschnitt 3.3.3.3 beschriebenen Ansatzes zur statischen WCET-Analyse für Mehrkernprozessoren bereit.

Kapitel 6.

Kommunikationsoptimierung für parallele Echtzeitprogramme

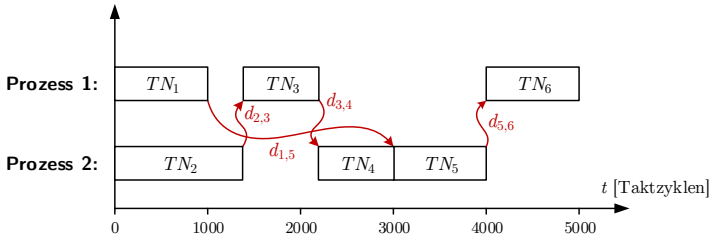
Kommunikation und Synchronisation sind in parallelen Programmen von entscheidender Bedeutung für den Gewinn, der durch die Parallelisierung erzielt werden kann. Das in Kapitel 4 vorgestellte Compiler-Werkzeug sieht deshalb einen gesonderten Schritt zur Optimierung der Synchronisationsstruktur des erzeugten Programms vor (vgl. Abschnitt 4.2.1.4). In diesem Kapitel sollen die zugehörigen Methoden und Optimierungsverfahren entwickelt und im Detail beschrieben werden. Das Hauptaugenmerk liegt dabei auf dem Kriterium der WCET, wobei der Ansatz vor dem Hintergrund gemischt-kritischer Systeme auch zur Optimierung der Rechenleistung im Durchschnittsfall verwendet werden kann. Die weiteren Betrachtungen basieren auf dem in Kapitel 4 vorgestellte Programmiermodell für parallele Echtzeitprogramme und führen dementsprechend sowohl Kommunikation als auch Synchronisation auf einfache FIFO-Operationen zurück. Wie schon in Abschnitt 2.2.3 kann Kommunikation dabei als Spezialfall von Synchronisation mit zusätzlicher Übertragung von Nutzdaten angesehen werden.

Herkömmliche Ansätze zur automatischen Parallelisierung betrachten die Optimierung von Kommunikation und Synchronisation meist als Teil des Scheduling-Problems (vgl. Abschnitt 3.3). Während es grundsätzlich sinnvoll ist, diesen Aspekt beim Scheduling zu berücksichtigen, lässt sich das volle Optimierungspotential i. A. nicht vollständig mit den dabei verwendeten Task-Graphen-Modellen abbilden. Hinzu kommt, dass das Scheduling von nicht-trivialen Task-Graphen oft ohnehin schon ein komplexes Optimierungsproblem darstellt. Ein präziseres Modell von Kommunikation und Synchronisation, insbesondere bei gleichzeitiger Berücksichtigung von Interferenz-Kosten, kann daher schnell zu nicht mehr handhabbaren Problemgrößen führen. Ein Beispiel aus der Literatur ist der ILP-basierte Scheduler von Rouxel et al. [94], mit dem selbst bei kleineren (nicht-hierarchischen) Task-Graphen eine beträchtliche Rechenzeit zur Lösung des Problems benötigt wird. Der in dieser Arbeit verfolgte mehrstufige Ansatz

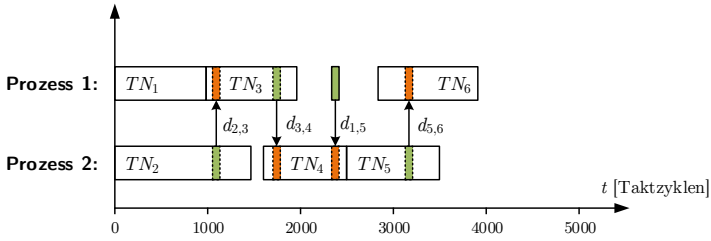
vermeidet derartige Probleme mit der Komplexität des Scheduling-Schritts, ohne dabei auf die feingranulare Optimierung von Kommunikation und Synchronisation in einem nachgelagerten Schritt verzichten zu müssen. Die wesentlichen Konzepte zur Kommunikationsoptimierung auf Quellcode-Ebene sind auch Gegenstand der eigenen Publikation [RB20b].

Dieses Kapitel beginnt mit der Definition der Ziele in Abschnitt 6.1 und leitet anschließend in Abschnitt 6.2 die konkrete Problemstellung ab. Die Abschnitte 6.3 und 6.4 stellen daraufhin ein Lösungsverfahren für die Problemstellung vor, bevor Abschnitt 6.5 das Kapitel mit einer Diskussion der präsentierten Methode abschließt.

6.1. Zielsetzung



(a) Beispiel für einen generierten Schedule



(b) Beispielhafte Realisierung des Schedules

Abbildung 6.1.: Beispielhafter Schedule mit Task-Konten TN_i und Datenabhängigkeiten $d_{i,j}$ sowie einer mögliche Realisierung (vgl. [RB20b])

Den Ausgangspunkt für die feingranulare Optimierung der Synchronisationsstruktur bildet ein gegebener Schedule gemäß Definition 2.5. Dieser bezieht sich im Allgemeinen auf die HTG-Darstellung des Programms, setzt sich aber auf jeder Hierarchieebene aus Schedules für einfache Task-Graphen aus Task-Knoten TN_i und Abhängigkeiten $d_{i,j}$ zusammen. Abbildung 6.1a zeigt erneut den beispielhaften Schedule aus Abschnitt 2.2, wie er z. B. für eine Hierarchieebene des HTG generiert worden sein könnte (Abhängigkeiten zwischen Tasks auf demselben Kern werden zur besseren Darstellung unterdrückt). In einem solchen Schedule gibt es offene Freiheitsgrade, die sich für die feingranulare Optimierung nutzen lassen. Deutlich wird dies z.B. an der Datenabhängigkeit $d_{1,5}$, für die der Zeitpunkt der Kommunikation/Synchronisation im Intervall zwischen 1000 und 3000 Taktzyklen frei gewählt werden kann. Die Nebenbedingungen für gültige Schedules aus Definition 2.5 erzwingen außerdem, dass sich zwei Task-Knoten TN_i und TN_j zeitlich nicht überlappen dürfen, wenn zwischen ihnen eine Abhängigkeit $d_{i,j}$ existiert. Betrachtet man die Task-Knoten jedoch nicht mehr als atomar und bezieht den internen Kontrollflussgraphen mit ein, so können die Endpunkte der Abhängigkeiten auf Basisblock-Ebene lokalisiert werden. Innerhalb des CFG kann eine Synchronisation für $d_{i,j}$ dann ggf. schon vor Ende von TN_i bzw. nach Beginn von TN_j eingefügt werden. Die dadurch entstandenen Überlappungen zwischen abhängigen Tasks können zur Verkürzung der Gesamtlaufzeit bzw. der WCET des parallelen Schedules beitragen.

Durch die effiziente Ausnutzung von Freiheitsgraden könnte sich der in Abbildung 6.1a dargestellte Schedule beispielsweise wie in Abbildung 6.1b realisieren lassen. Unter einer Realisierung ist hierbei eine Zwischendarstellung des Programms zu verstehen, in der die Synchronisationsoperationen im Kontrollflussgraphen platzierten wurden. Die Realisierung muss den gegebenen Schedule hinsichtlich der Zuweisung von Tasks zu Prozessorkernen sowie der Reihenfolge von Tasks auf demselben Kern umsetzen.

In der Realisierung aus Abbildung 6.1b konnte die Gesamtdauer für die parallele Ausführung der sechs Tasks gegenüber dem ursprünglichen Schedule deutlich verringert werden. Durch Verschieben von Synchronisationsoperationen in die Tasks hinein schließen sich teilweise Lücken im Schedule, innerhalb derer ein Prozessorkern sonst unbeschäftigt wäre. Die gezeigte Realisierung ist außerdem für FIFO-basierte Kommunikationsmodelle geeignet, da die Sendeoperationen und deren zugehörige Empfangsoperationen in der gleichen Reihenfolge abgearbeitet werden. Dadurch, dass Sende- und Empfangsoperationen bei Bedarf auch zur gleichen Zeit stattfinden können, ist die Realisierung weiterhin auch für den Einsatz synchroner Kommunikation geeignet (vgl. Abschnitt 2.2.3). Diese Eigenschaft stellt eine notwendige Randbedingung für die Optimierung der Kommunikation dar, um die Kompatibilität mit dem in Kapitel 4 beschriebenen Programmiermodell zu erhalten. Andernfalls könnte z. B. die zu Abhängigkeit

$d_{1,5}$ gehörige Sendeoperation an das Ende von TN_1 positioniert werden, während die Empfangsoperation an den Anfang von TN_5 platziert wird. Damit hätten die Sendeoperationen für $d_{1,5}$ und $d_{3,4}$ eine andere Reihenfolge als die zugehörigen Empfangsoperationen und könnten deshalb nicht auf demselben FIFO-Kanal operieren.

In ihrer Wirkung können sowohl Kommunikation als auch Synchronisation durch Operationen auf FIFO-Kanälen dargestellt werden. In ihrer Semantik gibt es jedoch signifikante Unterschiede. Während Synchronisation dazu dient, die Reihenfolge bestimmter Operationen auf unterschiedlichen Prozessorkernen festzulegen, ist Kommunikation in erster Linie zur Datenübertragung vorgesehen (vgl. Abschnitt 4.1.2). Der Synchronisationseffekt ist dabei lediglich eine notwendige Nebenwirkung. Im Gegensatz zu Kommunikationsoperationen kann eine Synchronisationsoperation deshalb *redundant* sein, sofern benachbarte Synchronisations- oder Kommunikationsoperationen denselben ordnungserhaltenden Effekt haben. Falls also z. B. die Abhängigkeit $d_{1,5}$ aus Abbildung 6.1 nur Synchronisation erfordert, wäre die zugehörige Laufzeitoperation redundant, da die Kommunikation/Synchronisation für die Abhängigkeit $d_{3,4}$ bereits dafür sorgt, dass TN_5 stets nach TN_1 ausgeführt wird. Die Operation könnte daher ohne Verlust der funktionalen Äquivalenz entfernt werden.

Im Gegensatz zur Schedule-Optimierung auf Basis des HTG soll die zu realisierende Kommunikationsoptimierung den gesamten Kontrollflussgraphen berücksichtigen, ohne dabei auf eine einzelne HTG-Ebene beschränkt zu sein. Dies ermöglicht unter anderem die Verschiebung von Kommunikation/Synchronisation über verschachtelte Schleifenhierarchien hinweg oder das Erkennen und Entfernen redundanter Synchronisationsoperationen in unterschiedlichen Schleifen.

Insgesamt besteht das Ziel der Kommunikationsoptimierung dann darin, die genannten Freiheitsgrade so zu nutzen, dass die WCET des parallelen Programms unter Einhaltung des gegebenen Schedules minimiert wird. Die daraus resultierende Problemstellung wird im nachfolgenden Abschnitt genauer herausgearbeitet.

6.2. Problemstellung

Die Problemstellung der Kommunikationsoptimierung lässt sich als Platzierungsproblem für Kommunikations- und Synchronisationsoperationen im Kontrollflussgraphen des zu parallelisierenden Programms definieren. Aus den genannten Zielsetzungen ergeben sich dabei im Wesentlichen drei Optimierungskriterien:

1. Die Positionen der Synchronisationsoperationen sollten Lücken im Schedule, die mit Untätigkeit der Prozessorkerne verbunden sind, nach Möglichkeit vermeiden bzw. verkleinern.
2. Bei Programmen mit Schleifen sollten Synchronisationsoperationen so platziert werden, dass sie möglichst selten ausgeführt werden müssen. Synchronisation innerhalb häufig durchlaufener Schleifen sollte also vermieden werden.
3. Soweit zulässig, soll für eine Menge von redundanten Synchronisationsoperationen dieselbe Position gewählt werden, um das spätere Zusammenfassen zu einer Gesamtoperation zu ermöglichen.

Die Erhaltung der korrekten Funktionalität des generierten Programms sowie die Kompatibilität zu dem in Kapitel 4 spezifizierten Programmiermodell erfordern außerdem die folgenden Randbedingungen:

- Die Einhaltung von Abhängigkeiten,
- die Erzeugung einer Synchronisationsstruktur, die auch für synchrone Kommunikation geeignet ist und
- die vollständige Vermeidung von *Deadlocks* in dem erzeugten parallelen Programm.

Die letzten beiden Punkte werden – wie in Kapitel 4 diskutiert – durch den Optimierungsschritt nicht tangiert, sofern er vor der Deserialisierung des sequentiellen Programms stattfindet. Um dies zu gewährleisten, wird das zu lösende Optimierungsproblem im Folgenden auf Basis der sequentiellen Zwischendarstellung nach dem Schritt der *Datenpartitionierung* (vgl. Abschnitt 4.2) formuliert. Ein entsprechendes Beispielprogramm ist in Abbildung 6.2 und der zugehörige Kontrollflussgraph $CFG = (V, E^{cf}, TYP)$ in Abbildung 6.3 abgebildet. Wie dargestellt, sollen Synchronisationsoperationen (einschließlich Kommunikation) dabei, im Gegensatz zu den regulären CFG-Knoten gemäß Definition 2.6, als separate Knoten $s \in \mathcal{S} \subset V$ vom neu hinzugefügten Typ $SYNC$ betrachtet werden. Dabei entspricht

$$\mathcal{S} := \left\{ bb_i \in V : \ TYP(bb_i) = SYNC \right\} \tag{6.1}$$

der Menge aller Synchronisationsknoten des CFG. Aus der Zuweisung von Task-Knoten zu Prozessen (vgl. Definition 2.5) im Schedule lässt sich für die jeweiligen Knoten $bb_i \in V \setminus \mathcal{S}$ des CFG außerdem eine analoge Zuweisung $\mu^{bb} : V \mapsto \mathcal{P}$ zu einem Prozess π_k ermitteln. Dies schließt die Synchronisationsknoten $s \in \mathcal{S}$ explizit aus, da sie sich auf beide beteiligte Prozesse auswirken und damit keine eindeutige Zuweisung haben. In Abbildung 6.2 ist die Zuweisungs-Information μ^{bb} als Kommentar im Quellcode annotiert.

```
double sum_a = 0, sum_b_p0 = 0, sum_b_p1 = 0;

// ------- Prozess 0 -------
for (int i = 0; i < 100; ++i) {
    vect_a[i] = abs(vect_a[i] - reference[i]);
    sum_a += vect_a[i];
}

// ------- Prozess 1 -------
for (int j = 0; j < 100; ++j) {
    vect_b[j] = abs(vect_b[j] - reference[j]);
    sync_point(1, 0);
    sum_b_p1 += vect_b[j];
    comm_copy(sum_b_p1, sum_b_p0);
}

// ------- Prozess 0 -------
if (sum_a < sum_b_p0) {
    function(vect_a,vect_b);
} else {
    function(vect_b,vect_a);
}
```

Abbildung 6.2.: Beispielprogramm nach dem Umsortierunsschritt der S2S-Transformation (vgl. [RB20b])

Synchronisationsoperationen und damit die Synchronisationsknoten $s \in \mathcal{S}$ dienen vor allem dazu, die Einhaltung von Abhängigkeiten bzw. die Ausführungsreihenfolge bestimmter Blöcke des CFG Prozessor-übergreifend zu erzwingen (vgl. Abschnitt 2.2.3). Es gibt daher i. A. für jeden Synchronisationsknoten s eine Menge $\mathcal{B}^-(s) \subset V$ von Blöcken mit Zuweisung zu einem Prozess π_k, die garantiert vor dem Beginn eines beliebigen anderen Blocks aus einer zweiten Menge $\mathcal{B}^+(s) \subset V$ mit Zuweisung zu π_l ausgeführt werden müssen. Für sinnvolle Synchronisationsoperationen gilt hierbei $\pi_k \neq \pi_l$ und $\mathcal{B}^-(s) \cap \mathcal{B}^+(s) = \emptyset$. In Abbildung 6.3 sind diese Mengen für den hervorgehobenen `sync_point` graphisch dargestellt. Die Synchronisationsoperation stellt in diesem Beispiel sicher, dass die Lesezugriffe innerhalb des `if`-Blocks auf die gemeinsam genutzte Variable `vect_b` erst nach der Zuweisung des Variablenwertes durch den anderen Prozess erfolgen können.

Eine Synchronisation s mit den zugehörigen Mengen $\mathcal{B}^-(s)$ und $\mathcal{B}^+(s)$ erfüllt ihre Funktion nur dann, wenn jeder Kontrollflusspfad zwischen beliebigen Paaren (a, b) aus Ausgangsknoten $a \in \mathcal{B}^-(s)$ und Zielknoten $b \in \mathcal{B}^+(s)$ durch eine Instanz von s verläuft. Andernfalls gibt es eine mögliche Ausführung des Programms durch einen nicht abgedeckten Kontrollflusspfad, in dem die Synchronisationsbedingung nicht erzwungen wird. In einem solchen Fall wären Probleme wie *Race Conditions* nicht mehr ausgeschlossen. Bei der Optimierung

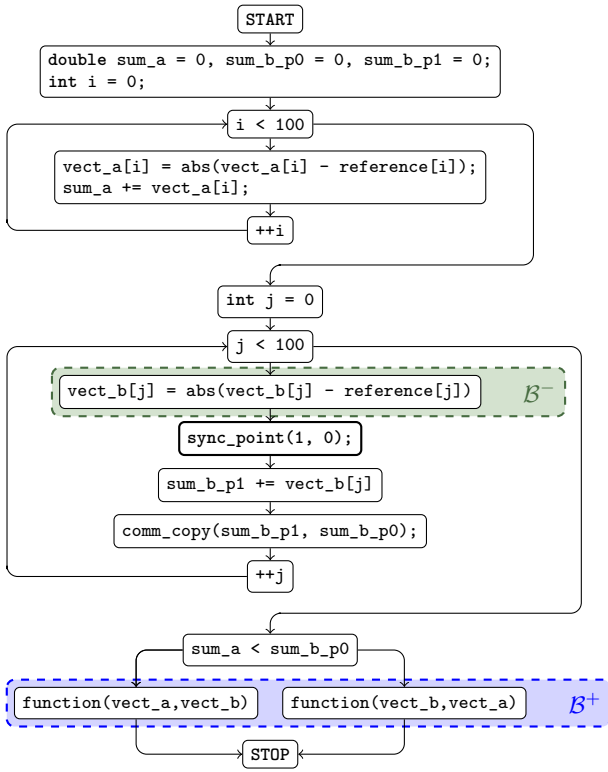


Abbildung 6.3.: Kontrollflussgraph des Beispielprogramms aus Abbildung 6.2

der Position des Synchronisationsknotens im CFG muss dieses Kriterium also eingehalten werden, um eine mögliche Verletzung von Abhängigkeiten zu vermeiden (Die erste der oben genannten Randbedingungen). Auf dieser Basis kann dann das gesuchte Platzierungsproblem für Synchronisationsoperationen formuliert werden.

Im Folgenden sollen die Kanten $e \in E^{cf}$ des Kontrollflussgraphen als mögliche neue Positionen der Synchronisationsoperation in Betracht gezogen werden. Da die Synchronisation nur auf den Pfaden zwischen einem $a \in \mathcal{B}^-(s)$ und einem $b \in \mathcal{B}^+(s)$ nötig ist, genügt es dabei nur die entsprechende Teilmenge $\mathcal{VP}(s) \subseteq E^{cf}$ der für s gültigen Positionen zu berücksichtigen. $\mathcal{VP}(s)$ lässt sich z. B. durch mehrfache Anwendung des Verfahrens der *Tiefensuche* (englisch *Depth-First*

Search, kurz DFS, siehe [61]) für jeden der Knoten in $\mathcal{B}^-(s)$ als Startpunkt bestimmen. Sobald das Suchverfahren einen Knoten $b \in \mathcal{B}^+(s)$ antrifft, werden alle Kanten des durchlaufenen Pfades zu $\mathcal{VP}(s)$ hinzugefügt. Als Kostenfunktion für die Optimierung kann für die Kanten $e \in \mathcal{VP}(s)$ jeweils ein Kantengewicht $w(e)$ definiert werden. Die Position des Synchronisationsknotens kann dann durch das Lösen des folgenden Platzierungsproblems optimiert werden:

Definition 6.1 (Platzierungsproblem für Synchronisation)**.** *Gegeben sei ein Kontrollflussgraph $CFG = (V, E^{cf}, TYP)$, ein Synchronisationsknoten s mit den zugehörigen Mengen $\mathcal{B}^-(s)$ und $\mathcal{B}^+(s)$ sowie die Menge $\mathcal{VP}(s)$ der für s gültigen Positionen. Das Platzierungsproblem für Synchronisation besteht dann darin, eine Teilmenge $POS(s) \in \mathcal{VP}(s)$ zu finden, die das Optimierungsproblem*

$$\min_{POS(s) \subseteq \mathcal{VP}(s)} \left\{ \sum_{\forall e \in POS(s)} w(e) \right\}$$

unter der Nebenbedingung

$$\Big|\omega \cap POS(s)\Big| \geq 1 \qquad \forall a \in \mathcal{B}^-(s),\ \forall b \in \mathcal{B}^+(s),\ \forall \omega \in \mathcal{W}(a,b)$$

optimal löst. Hierbei bezeichnet $\mathcal{W}(a,b)$ die Menge aller gerichteten Kantenfolgen in CFG, die in a beginnen und in b enden.

Die Gesamtkosten für eine Lösung $POS(s)$ dieses Optimierungsproblems werden im Folgenden durch

$$K\big(\, POS(s) \,\big) := \sum_{\forall e \in POS(s)} w(e) \tag{6.2}$$

beschrieben.

Die Kostenfunktion w sollte dabei so gewählt werden, dass die drei eingangs definierten Optimierungskriterien durch die Minimierung erfüllt werden. Für die ersten beiden Kriterien ist dies relativ einfach zu realisieren. Das Zusammenfassen redundanter Synchronisationsoperationen (Kriterium 3) lässt sich jedoch mit dem Problem aus Definition 6.1 nicht direkt beschreiben, da die Formulierung nur einen einzelnen Synchronisationsknoten berücksichtigt. In Abschnitt 6.4 soll jedoch eine Erweiterung vorgestellt werden, mit der sich auch letzterer Fall auf eine Kombination aus mehreren Platzierungsproblemen gemäß Definition 6.1 zurückführen lässt.

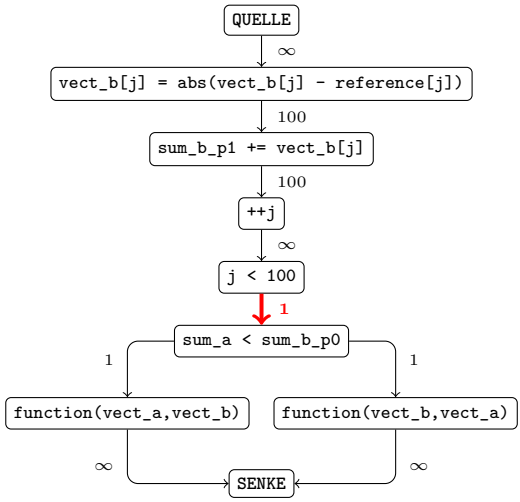


Abbildung 6.4.: Zur Platzierung der `sync_point`-Operation aus Abbildung 6.3 verwendeter Graph $G^{st}(s)$ (vgl. [RB20b])

6.3. Lösungsverfahren für einzelne Synchronisationsknoten

Für die Lösung des Optimierungsproblems aus Definition 6.1 ist lediglich eine Teilmenge $\mathcal{VP}(s)$ der Kanten des CFG von Bedeutung. Es bietet sich daher an, nur den Subgraphen $G(s) = (V^{sub}(s), \mathcal{VP}(s))$ mit diesen Kanten und den damit verbundenen Knoten $V^{sub}(s)$ zu betrachten. Aus der Nebenbedingung für gültige Lösungen $POS(s)$ in Definition 6.1 folgt dann unmittelbar, dass der Graph $G(s)$ durch das Entfernen aller Kanten $e \in POS(s)$ in mindestens zwei nicht-zusammenhängende Komponenten unterteilt wird. Die erste dieser Komponenten enthält mindestens die Knoten $\mathcal{B}^-(s)$ und die zweite mindestens die Knoten $\mathcal{B}^+(s)$. Da jeder Pfad zwischen diesen Komponenten mindestens eine Kante $e \in POS(s)$ enthalten muss, gibt es nach deren Entfernung keinen Zusammenhang mehr. Das Optimierungsproblem lässt sich damit auf das Problem eines minimalen Kantenschnitts zwischen den Knoten in $\mathcal{B}^-(s)$ und denen in $\mathcal{B}^+(s)$ zurückführen. Bestehen $\mathcal{B}^-(s)$ und $\mathcal{B}^+(s)$ jeweils aus genau einem Element, so ist das Problem äquivalent zu dem in Abschnitt 2.3.6.2 beschriebenen Problem des *minimalen s-t-Schnitts*. Deutlich wird dies u. a. auch daran, dass das Minimierungsproblem in Definition 6.1 praktisch identisch zu dem in Gleichung (2.6) ist. Das Optimierungsproblem kann in diesem Sonderfall also direkt mit Algorithmen wie dem *Edmonds-Karp Algorithmus* gelöst werden.

Auch für den allgemeinen Fall, dass $\mathcal{B}^-(s)$ und $\mathcal{B}^+(s)$ mehr als ein Element enthalten, lässt sich ein passendes Problem des *minimalen s-t-Schnitts* konstruieren. Hierzu wird auf Basis von $G(s)$ ein erweiterter Graph $G^{st}(s)$ konstruiert, der für das Beispiel der `sync_point`-Operation aus Abbildung 6.3 in Abbildung 6.4 dargestellt ist. Synchronisationsknoten sind dabei nicht Bestandteil der Knotenmenge von $G^{st}(s)$ und werden deshalb in Abbildung 6.4 vernachlässigt, da zunächst nur die individuelle Platzierung einzelner Synchronisationsknoten betrachtet werden soll. Die wichtigste Ergänzung in $G^{st}(s)$ gegenüber $G(s)$ sind zwei zusätzliche Konten, die im Folgenden als „Quelle“ α und „Senke“ β bezeichnet werden (vgl. Abbildung 6.4). Weiterhin werden zusätzliche Kanten

$$(\alpha, v^-) \text{ für } \forall v^- \in \mathcal{B}^-(s) \quad \text{und} \quad (v^+, \beta) \text{ für } \forall v^+ \in \mathcal{B}^+(s)$$

hinzugefügt, um alle Elemente in $\mathcal{B}^-(s)$ mit α und alle Elemente in $\mathcal{B}^+(s)$ mit β zu verbinden. Die Gewichte für diese Kanten werden zu $w(e) = \infty$ gesetzt, um zu verhindern, dass sie Teil eines minimalen Schnitts mit endlichem Gewicht werden können. Auf diese Weise wird erreicht, dass die Knoten $v^- \in \mathcal{B}^-(s)$ stets zusammen mit α auf einer Seite des Schnitts sind, während β zusammen mit den Knoten $v^+ \in \mathcal{B}^+(s)$ auf der anderen liegt. Falls ein s-t-Schnitt zwischen α und β durch Kanten mit endlichem Gewicht existiert, dann lässt sich das Problem aus Definition 6.1 folglich mit Verfahren wie dem *Edmonds-Karp Algorithmus* optimal lösen. Für eine entsprechende Lösung $POS(s)$ kann anschließend eine Instanz/Kopie von s an jeder Kante $e \in POS(s)$ des minimalen Schnitts eingefügt werden, während der ursprüngliche Knoten s entfernt wird. Die Funktionalität des Programms bleibt dadurch erhalten, da der Kantenschnitt die Nebenbedingung aus Definition 6.1 per Konstruktion einhält.

In dem konkreten Beispiel aus Abbildung 6.4 wurden die Ausführungshäufigkeiten $n^{cf}(e)$ der Kontrollflusskanten e als Gewichtung $w(e)$ herangezogen. Außerdem wurden Gewichte von $w(e) = \infty$ verwendet, um syntaktisch in C nicht umsetzbare Positionen für Synchronisationsoperationen auszuschließen. Dies ist z. B. zwischen den Blöcken „`++j`“ und „`j < 100`“ der Fall, da innerhalb des Schleifenkopfes der `for`-Schleife keine Synchronisationsoperation eingefügt werden kann. Die optimale Lösung für das Code-Beispiel ist dann ein Kantenschnitt durch die hervorgehobene Kante mit Gewicht 1 gegeben. Im Vergleich zur ursprünglichen Position der `sync_point`-Operation, reduziert die Umplatzierung auf Basis dieser Lösung die Ausführungshäufigkeit von 100 auf 1.

6.3.1. Kostenfunktion

Neben der Reduktion der Ausführungshäufigkeit soll die Optimierung nach Möglichkeit auch Lücken im Schedule schließen können (vgl. Abschnitt 6.2).

Hierzu wird, im Gegensatz zu Abbildung 6.4, eine entsprechend erweiterte Gewichtung $\hat{w}(e,s)$ der Kanten von $G^{st}(s)$ eingeführt. Lücken im Schedule äußern sich in Programmen, die auf dem Programmiermodell aus Kapitel 4 basieren, durch Wartezeiten bei blockierenden Synchronisationsoperationen wie *Wait* und *Receive*. Werden diese Wartezeiten minimiert, so können auch die Lücken reduziert werden. Im ungünstigsten Ausführungsfall sind die WCET-Werte beim Erreichen einer Synchronisationsoperation in den beteiligten Prozessen ausschlaggebend für die Wartezeiten. Für einen Synchronisationsknoten s sei der sendende Prozess mit *Signal*-Semantik im Folgenden durch $\pi^-(s)$ und der empfangende Prozess mit *Wait*-Semantik durch $\pi^+(s)$ gegeben. Für die möglichen Positionen $e \in \mathcal{VP}(s)$ können die Ankunftszeiten $WCET^{sync}\left(e, \pi^+(s)\right)$ und $WCET^{sync}\left(e, \pi^-(s)\right)$, an denen die entsprechende Kontrollflusskante im ungünstigsten Fall von den Prozessen $\pi^+(s)$ bzw. $\pi^-(s)$ erreicht wird i. A. unterschiedlich sein. Erreicht $\pi^-(s)$ die Kante e vor $\pi^+(s)$, so entsteht in $\pi^-(s)$ eine Wartezeit von mindestens

$$\Delta T^- \;:=\; WCET^{sync}\left(e, \pi^+(s)\right) \;-\; WCET^{sync}\left(e, \pi^-(s)\right)\,, \tag{6.3}$$

da die *Wait*-Operation auf das zugehörige *Signal* warten muss. Berücksichtigt man zudem die Möglichkeit zur impliziten Synchronisation durch volle FIFO-Puffer, so muss die Synchronisation im ungünstigsten Fall auch in die entgegengesetzte Richtung als blockierend betrachtet werden (vgl. Abschnitt 4.1.2). Unter dieser Annahme führt eine Differenz zwischen $WCET^{sync}\left(e, \pi^+(s)\right)$ und $WCET^{sync}\left(e, \pi^-(s)\right)$ in jedem Fall zu einer Wartezeit von

$$\Delta T \;=\; \left| WCET^{sync}\left(e, \pi^+(s)\right) \;-\; WCET^{sync}\left(e, \pi^-(s)\right) \right| \tag{6.4}$$

in einem der beteiligten Prozesse.

Um die Gesamtkosten $\hat{w}(e,s)$ für die Platzierung eines Synchronisationsknotens s an der Kante e zu definieren, müssen die Kriterien der Ausführungshäufigkeit $n^{cf}(e)$ und der zu erwartenden Wartezeiten ΔT geeignet gewichtet werden. Dazu bietet es sich an, die Ausführungshäufigkeit $n^{cf}(e)$ mit einem Faktor für den konstanten WCET-Beitrag bei der Ausführung der Synchronisationsoperation zu gewichten, um die Kosten für beide Aspekte in Zeiteinheiten ausdrücken zu können. Auf Basis dieser Grundidee kann dann die folgende Kostenfunktion definiert werden:

$$\begin{aligned}\hat{w}(e,s) \;:=\; & n^{cf}(e) \cdot \left(\tau^d \cdot n^{comm}(s) + \tau^c\right) \\ & + \left| WCET^{sync}\left(e, \pi^+(s)\right) \;-\; WCET^{sync}\left(e, \pi^-(s)\right) \right|.\end{aligned} \tag{6.5}$$

Dabei steht $n^{comm}(s)$ für die Anzahl der Dateneinheiten, die im Zuge der Operation s über den FIFO-Kanal übertragen werden müssen. Die freien Parameter τ^d und τ^c können abhängig von der Zielplattform und der konkreten Implementierung der FIFO-Kommunikation so gewählt werden, dass sie näherungsweise die WCET für die Übertragung einer Dateneinheit (Parameter τ^d) bzw. eine konstante WCET für den Aufruf der Kommunikationsfunktion (Parameter τ^c) abbilden.

Es bleibt noch zu erörtern, wie die Ankunftszeiten $WCET^{sync}(e, \pi_k)$ der Kontrollflusskanten konkret bestimmt werden können. Wie in Abschnitt 4.2 beschrieben, stehen zum Zeitpunkt der Optimierung lediglich *a priori* Schätzungen der WCET-Beiträge einzelner Programmteile zur Verfügung. Aus diesen Informationen muss eine (i. A. grobe) Schätzung $WCET^{bb}(bb_i)$ der WCET einzelner Kontrollflussknoten abgeleitet werden. Dies kann z. B. durch approximative Verfeinerung der für das Scheduling verwendeten WCET-Werte von den Task-Knoten auf die zugehörigen Basisblöcke des CFG erreicht werden. Die gesuchten Werte $WCET^{sync}(e, \pi_k)$ ergeben sich dann aus der Länge der längsten zyklenfreien Knotenfolge im (reduzierbaren) Kontrollflussgraphen, die vom Start-Knoten zu der Kante e führt. Zur Bestimmung der Länge der Knotenfolgen für den jeweiligen Prozess π_k sind die Knoten bb_i dabei, abhängig von ihrer Prozess-Zuweisung $\mu^{bb}(bb_i)$, jeweils mit

$$W(bb_i, \pi_k) := \begin{cases} WCET^{bb}(bb_i) \cdot n^{exec}(bb_i) & \mu^{bb}(bb_i) = \pi_k \\ 0 & \mu^{bb}(bb_i) \neq \pi_k \end{cases} \tag{6.6}$$

zu gewichten. Die WCET wird hier mit der Ausführungshäufigkeit $n^{exec}(bb_i)$ der Knoten gewichtet, sodass Zyklen/Schleifen im CFG bei der Bestimmung des längsten Pfades nur einmal durchlaufen werden müssen. Hierzu ist noch anzumerken, dass i. A. auch die Synchronisationsoperationen, u. a. durch die jeweils verursachten Wartezeiten, zur Länge dieses Pfades beitragen können. Diese Effekte können mit Hilfe der Gleichungen (6.4) bzw. (6.5) in die Berechnungen einbezogen werden. Durch die individuelle Umplatzierung einer Synchronisationsoperation ändern sich allerdings ggf. die Ankunftszeiten aller nachfolgenden Kanten. Werden die Synchronisationsoperationen also schrittweise nacheinander neu platziert, so muss die Kostenfunktion nach jedem Schritt aktualisiert werden. Dadurch entsteht jedoch eine Abhängigkeit der Ergebnisse von der Reihenfolge, in der die Umplatzierung durchgeführt wird. Neben dem Ziel, redundante Synchronisation zu entfernen, ist dies einer der Hauptgründe für die im Folgenden beschriebene Erweiterung zur Co-Optimierung mehrerer Synchronisationsoperationen.

6.4. Co-Optimierung mehrerer Synchronisationsknoten

Um mehrere potentiell redundante Synchronisationsoperationen zusammenzufassen, müssen diese

1. an eine gemeinsame, für beide Operationen gültige Position verschoben und
2. durch eine funktional äquivalente Einzeloperation ersetzt werden können.

Für eine Menge $M := \{s_0, \ldots, s_k\}$ von Synchronisationsoperationen ist Ersteres genau dann möglich, wenn mindestens eine Position $e \in E^{cf}$ für alle Operationen gültig ist:

$$\mathcal{VP}(s_0) \cap \ldots \cap \mathcal{VP}(s_k) \neq \emptyset\,. \tag{6.7}$$

Das zweite Kriterium ist erfüllt, wenn alle Operationen dasselbe Paar von Prozessen $\pi^-(s_k)$ und $\pi^+(s_k)$ betreffen und wenn ihre kollektive Funktionalität durch eine geeignete Einzeloperation abgedeckt werden kann. Letzteres gilt zum einen für reine Synchronisationsoperationen (*Signal/Wait*) und zum anderen für Kommunikationsoperationen (*Send/Receive*), die dieselben Daten übertragen (Letzteres kann z. B. auftreten, wenn die ursprünglichen Positionen zweier Kommunikationsoperationen in alternativen Kontrollflussverzweigungen liegen). Außerdem lassen sich auch Synchronisationsoperationen mit geeigneten Kommunikationsoperationen verbinden, da Kommunikation in dem vorgestellten Programmiermodell stets auch Synchronisation impliziert.

6.4.1. Platzierungsproblem für mehrere Synchronisationsknoten

Um alle Synchronisationsknoten in M gemeinsam an einer optimalen Position zu platzieren, kann der im vorigen Abschnitt beschriebene Ansatz für die individuelle Platzierung auf Basis des minimalen Schnitts auf mehrere Operationen erweitert werden. Hierzu ist es zunächst sinnvoll eine gemeinsame Menge $\mathcal{VP}(M)$ von gültigen Positionen zu definieren:

$$\mathcal{VP}(\{s_0, \ldots, s_k\}) := \mathcal{VP}(s_0) \cup \ldots \cup \mathcal{VP}(s_k)\,. \tag{6.8}$$

Analog dazu lassen sich kombinierte Mengen $\mathcal{B}^-(M)$ und $\mathcal{B}^+(M)$ der Ausgangs- bzw. Zielknoten wie folgt definieren:

$$\mathcal{B}^-(\{s_0, \ldots, s_k\}) := \mathcal{B}^-(s_0) \cup \ldots \cup \mathcal{B}^-(s_k)\,, \tag{6.9}$$

$$\mathcal{B}^+(\{s_0, \ldots, s_k\}) := \mathcal{B}^+(s_0) \cup \ldots \cup \mathcal{B}^+(s_k)\,. \tag{6.10}$$

Mit $\mathcal{VP}(M)$, $\mathcal{B}^-(M)$ und $\mathcal{B}^+(M)$ ist es möglich, den Graphen G^{st} aus dem vorherigen Abschnitt für eine Menge M aus mehreren Synchronisationsknoten zu verallgemeinern:

Definition 6.2 (Subgraph zur Synchronisationsplatzierung)**.** *Gegeben sei ein Kontrollflussgraph $CFG = (V, E^{cf}, TYP)$, eine Menge $M := \{s_0, \ldots, s_k\}$ von Synchronisationsknoten, die Menge $\mathcal{VP}(M) \subseteq E^{cf}$ aller für mindestens ein $s \in M$ gültigen Positionen und die vereinigten Mengen $\mathcal{B}^-(M)$ bzw. $\mathcal{B}^+(M)$ der Ausgangs- bzw. Zielknoten. Der Subgraph für die kombinierte Synchronisationsplatzierung aller $s \in M$ ist dann definiert durch*

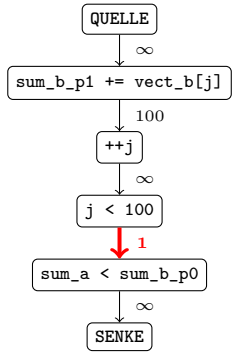
$$G^{st}(M) := \left(V^{st}, E^{st}, TYP\right) ,$$

mit

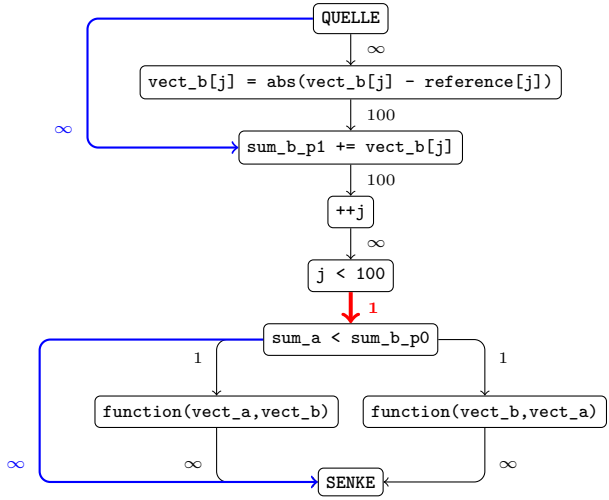
$$\begin{aligned} E^{st} :=& \mathcal{VP}(M) \cup \left\{(\alpha, v^-) :\ v^- \in \mathcal{B}^-(M)\right\} \\ & \cup \left\{(v^+, \beta) :\ v^+ \in \mathcal{B}^+(M)\right\} , \\ V^{st} :=& \{\alpha, \beta\} \cup \left\{v \in V :\ (v, a) \in E^{st},\ a \in V\right\} \\ & \cup \left\{v \in V :\ (a, v) \in E^{st},\ a \in V\right\} . \end{aligned}$$

Da $\mathcal{B}^-(M)$ und $\mathcal{B}^+(M)$ jeweils die entsprechenden Mengen aller Einzeloperationen $s \in M$ in sich vereinigen, erfüllt ein endlicher Kantenschnitt $POS(M)$ zwischen diesen Mengen (bzw. der Quelle α und der Senke β) nach wie vor die Nebenbedingung aus Definition 6.1 für jede der Einzeloperationen. Enthält M mehr als eine Synchronisationsoperation, so muss ein solcher Schnitt jedoch nicht mehr notwendigerweise existieren. Dieser Fall tritt z. B. dann ein, wenn die Mengen $\mathcal{B}^-(M)$ und $\mathcal{B}^+(M)$ durch die Vereinigung in Gleichung (6.9) bzw. (6.10) nicht mehr disjunkt sind. Die Nebenbedingung aus Definition 6.1 kann dann nicht mehr eingehalten werden, da alle Knoten, die sowohl in $\mathcal{B}^-(M)$ als auch in $\mathcal{B}^+(M)$ enthalten sind, mit der leeren Kantenfolge $\emptyset$ einen Pfad ohne Synchronisationsoperation zwischen $\mathcal{B}^-(M)$ und $\mathcal{B}^+(M)$ bilden.

Zur Veranschaulichung des Subgraphen $G^{st}(M)$ für mehrere Synchronisationsoperationen kann erneut das Beispielprogramm aus den Abbildungen 6.2 und 6.3 herangezogen werden. Falls die Menge $M = \{s_0\}$ ausschließlich die `sync_point`-Operation s_0 aus diesem Programm enthält, so entspricht $G^{st}(M)$ dem zuvor eingeführten Beispiel aus Abbildung 6.4. Analog dazu kann für die `comm_copy`-Operation s_1 der zugehörige individuelle Subgraph $G^{st}(\{s_1\})$



(a) Zur Platzierung der `comm_copy`-Operation verwendeter Graph $G^{st}(\{s_1\})$



(b) Kombinierter Subgraph $G^{st}(\{s_0, s_1\})$ zur gemeinsamen Platzierung beider Synchronisationsoperationen aus Abbildung 6.3

Abbildung 6.5.: Beispiel zur kombinierten Platzierung des `sync_point` s_0 und der `comm_copy`-Operation s_1 aus Abbildung 6.3

konstruiert werden. Letzterer ist in Abbildung 6.5a mit einer farblich hervorgehobenen Kante für den minimalen Schnitt dargestellt. Aus den Einzelgraphen zur individuellen Platzierung aus Abbildung 6.4 und Abbildung 6.5a resultiert dann der kombinierte Graph $G^{st}(\{s_0, s_1\})$, wie er in Abbildung 6.5b dargestellt ist. Die blau hervorgehobenen Kanten mit Gewicht $w(e) = \infty$ entstehen durch die Vereinigung der Mengen $\mathcal{B}^-$ und $\mathcal{B}^+$. Sie verlaufen parallel zu anderen Kanten mit endlichem Gewicht, sodass letztere nicht mehr Bestandteil eines endlich-gewichteten Kantenschnitts zwischen Quelle und Senke sein können. Einige für s_0 gültige Positionen sind damit für die gemeinsame Platzierung von $M = \{s_0, s_1\}$ nicht mehr gültig, sodass sich der Lösungsraum im Vergleich zur individuellen Platzierung verkleinert. Für das vorliegende Beispiel betrifft diese Einschränkung jedoch nur Kanten, die auch bei individueller Platzierung nicht optimal sind. Die kombinierte Lösung fällt daher mit dem Optimum der individuellen Lösungen zusammen. Da die beiden Operationen außerdem zu einer einzelnen `comm_copy`-Operation mit äquivalenter Funktionalität zusammengefasst werden können, repräsentiert die gemeinsame Platzierung außerdem die kostengünstigere Lösung. Im Vergleich zum ursprünglichen Programm lässt sich der Aufwand für Kommunikation/Synchronisation durch das Optimierungsverfahren damit von zwei Operationen, die jeweils 100 Mal ausgeführt werden, auf eine Operation mit nur einer Ausführung reduzieren.

6.4.2. Exploration des Lösungsraums

Mit der Berechnung des minimalen Schnitts durch einen Graphen $G^{st}(M)$ lässt sich eine gemeinsame Position für eine vorgegebene Menge M von Synchronisationsoperationen bestimmen. Aus der Menge aller Synchronisationsoperationen $\mathcal{S}$ müssen dazu allerdings zunächst geeignete Teilmengen $M \subseteq \mathcal{S}$ ausgewählt werden. Eine Rolle spielt dabei zum einen das zu Beginn von Abschnitt 6.4 beschriebene Kriterium, dass die Einzeloperationen grundsätzlich für das Zusammenfassen zu einer Gesamtoperation geeignet sein müssen. Zum anderen sollten die gewählten Teilmengen die Menge $\mathcal{S}$ bei geringen Gesamtkosten vollständig überdecken, um ein möglichst optimales Gesamtergebnis zu erhalten. Im Folgenden wird dieses Optimierungsproblem in der Form eines *Graph-Clustering-Problems* (siehe z. B. [98]) formuliert und ein heuristisches Verfahren zu dessen approximativer Lösung vorgestellt. Dabei soll die Definition des *Graph-Clustering* aus [98] zugrunde gelegt werden, die unter diesem Begriff die optimale Gruppierung der Knoten eines Graphen unter Berücksichtigung seiner Kantenstruktur versteht.

Die Grundlage für die Formulierung als Clustering-Problem bildet der sogenannte *kombinierte Lösungsgraph* $J = (V^{joint}, E^{joint})$:

Definition 6.3 (Kombinierter Lösungsgraph)**.** *Sei $\mathcal{S}$ eine Menge von Synchronisationsknoten und $V^{joint} \subset \mathrm{Pot}(\mathcal{S})$ eine Menge $\{M_0, \ldots, M_n\}$ von disjunkten Teilmengen von $\mathcal{S}$, die $\mathcal{S}$ vollständig überdecken*[1]*. Dann heißt der ungerichtete Graph $J = (V^{joint}, E^{joint})$ kombinierter Lösungsgraph für $\mathcal{S}$, falls eine Kante $\{M_i, M_j\} \in E^{joint}$ genau dann existiert, wenn*

- *für die gültigen Positionen $\mathcal{VP}(M_i) \cap \mathcal{VP}(M_j) \neq \emptyset$ gilt und*
- *die Synchronisationsoperationen $s \in M_i \cup M_j$ zu einer einzelnen Operation zusammengefasst werden können.*

Die Knoten $M \in V^{joint}$ dieses Graphen repräsentieren dann jeweils eine Menge von Synchronisationsoperationen, die gemeinsam platziert werden können. Dabei stehen die ungerichteten Kanten $\{M_i, M_j\} \in E^{joint}$ für die Möglichkeit, die Knoten M_i und M_j zu einem Cluster zusammenzufassen, um alle Synchronisationsoperationen in $M_i \cup M_j$ in einem späteren Schritt gemeinsam platzieren zu können. Die in Definition 6.3 gestellten Forderungen an die Kanten folgen aus Gleichung (6.7) und den genannten Voraussetzungen für das Zusammenfassen von Synchronisationsknoten.

Auf Basis des kombinierten Lösungsgraphen J lässt sich ein konstruktives Clustering-Verfahren zur iterativen Optimierung der zugehörigen Knotenmengen V^{joint} entwickeln. Hierzu wird zunächst ein initialer Lösungsgraph J erzeugt, dessen $|\mathcal{S}|$ Knoten jeweils genau eine Synchronisationsoperation $s \in \mathcal{S}$ enthalten (d. h. $\forall M \in V^{joint} : |M| = 1$). Für jeden Knoten $M \in V^{joint}$ dieses Graphen kann mit Hilfe des minimalen s-t-Schnitts durch $G^{st}(M)$ eine gültige Platzierung der Operationen mit Gesamtkosten $K\big(POS(M)\big)$ berechnet werden. Für jede Kante $e^{joint} := \{M_i, M_j\} \in E^{joint}$ besteht dann die Möglichkeit, die adjazenten Knoten zusammenzufassen und mit Hilfe von $G^{st}(M_i \cup M_j)$ eine kombinierte Lösung mit den Kosten $K\big(POS(M_i \cup M_j)\big)$ zu ermitteln. Es können also entweder zwei individuelle Lösungen mit den Gesamtkosten

$$K\big(POS(M_i)\big) + K\big(POS(M_j)\big)$$

oder eine kombinierte Lösung mit den Kosten

$$K\big(POS(M_i \cup M_j)\big)$$

[1] $\mathrm{Pot}(X)$ bezeichnet die *Potenzmenge* von X.

verwendet werden. Jeder Kante e^{joint} lässt sich dann ein Gewicht

$$w^{joint}(e^{joint}) := K\big(POS(M_i)\big) + K\big(POS(M_j)\big) - K\big(POS(M_i \cup M_j)\big) \quad (6.11)$$

zuordnen, das die Kostenersparnis bzw. Erhöhung durch die Verwendung der kombinierten Lösung im Vergleich zu den individuellen Lösungen beschreibt.

Um die Entscheidung über das Zusammenfassen zweier adjazenter Knoten des initialen Lösungsgraphen J zu treffen, eignen sich Optimierungsverfahren aus der Klasse der *Greedy-Algorithmen*. Letztere zeichnen sich dadurch aus, dass sie bei jeder Entscheidungsmöglichkeit stets die zum aktuellen Zeitpunkt kostengünstigste Option wählen. Für das konkrete Problem bedeutet das, dass stets zuerst die Kante mit dem höchsten Gewicht $w^{joint}(e^{joint})$ ausgewählt wird, um die zugehörigen Knoten dann zusammenzufassen. Algorithmus 6.1 enthält eine entsprechende Umsetzung dieses Ansatzes. Der Algorithmus beginnt mit dem Aufbau des initialen Graphen J gemäß der Regeln in Definition 6.3, wobei die Ermittlung der Kanten durch die Funktion BerechneKantenmenge dargestellt wird. Da Synchronisationsknoten, die im CFG weit voneinander entfernt sind, oft keine gültige gemeinsame Position haben, ist J meist in mehrere zusammenhängende Komponenten Z unterteilt. Zwischen diesen Komponenten existiert per Definition keine Kante, sodass sie auch nicht für die Bildung gemeinsamer Cluster in Frage kommen. Die einzelnen Komponenten, die in Algorithmus 6.1 durch die Funktion ZusammenhaengendeKomponenten aufgelistet werden, können deshalb unabhängig voneinander optimiert werden. Letzteres geschieht in der äußeren **for**-Schleife des Algorithmus, die über alle zusammenhängenden Komponenten iteriert.

Der Algorithmus sortiert die Liste $ZK \subseteq V^{joint}$ der zusammenhängenden Komponenten vor ihrer Verwendung mit Hilfe der Funktion SortiereNachCFGPosition nach der mittleren Position aller darin enthaltenen Synchronisationsknoten innerhalb des CFG. Diese Reihenfolge basiert auf der *Topologischen Sortierung* der Knoten im reduzierbaren Kontrollflussgraphen unter Vernachlässigung von rückwärtsgerichteten Kontrollflusskanten. Auf diese Weise werden zusammenhängende Komponenten, deren Synchronisationsknoten im Programmablauf als erste erreicht werden, in der Optimierungsschleife auch zuerst bearbeitet. Dieses Vorgehen dient der Lösung des in Abschnitt 6.3.1 erwähnten Problems, dass die Umplatzierung von Synchronisationsknoten auch die Schätzung der Ankunftszeiten $WCET^{sync}$ aller nachfolgenden Kontrollflusskanten beeinflusst. Indem die im CFG tendenziell früher erreichten zusammenhängenden Komponenten bzw. Synchronisationsknoten zuerst optimiert werden, können die dadurch entstandenen Änderungen der Ankunftszeiten bei der anschließenden Platzierung späterer

Synchronisationsknoten berücksichtigt werden. Algorithmus 6.1 ruft hierzu in jeder Iteration der äußeren Schleife die Funktion BerechneAnkunftszeiten auf, die die Werte von $WCET^{sync}$ für alle betroffenen Kontrollflusskanten neu berechnet. Bei der in Abschnitt 6.3.1 beschriebenen Berechnung des längsten zyklenfreien Pfades zu der betreffenden Kontrollflusskante müssen hierzu die Wartezeiten für die Synchronisation an den betroffenen Kanten $e \in POS(s)$ für $\forall s \in \mathcal{S}$ berücksichtigt werden.

Nachdem die Ankunftszeiten $WCET^{sync}$ aktualisiert wurden, berechnet der Algorithmus in der Funktion BerechneKantengewichte zunächst die Kantengewichte $w^{joint}(e^{joint})$ aller Kanten e^{joint} des kombinierten Lösungsgraphen J gemäß Gleichung (6.11). Hierzu muss vorher für jeden Knoten und jede Kante der zugehörige Subgraph G^{st} aufgebaut und z. B. mit Hilfe des Edmonds-Karp Algorithmus ein minimaler s-t-Schnitt bestimmt werden. Auf Basis der resultierenden Kantengewichte wird innerhalb der aktuell bearbeiteten zusammenhängenden Komponente Z nach dem *Greedy*-Verfahren die Kante $e^{max} = \{M_i, M_j\}$ mit dem höchsten Gewicht ausgewählt. Das durch e^{max} verbundene Knotenpaar wird daraufhin zu einem einzelnen Knoten zusammengefasst, sofern das Kantengewicht $w^{joint}(e^{max})$ dafür einen positiven Gewinn verspricht (d. h. $w^{joint}(e^{max}) > 0$). Das Zusammenfassen der beiden Knoten $M_i \in e^{max}$ und $M_j \in e^{max}$ ist in Algorithmus 6.1 durch die Funktion FasseKnotenZusammen dargestellt. Die Mengen V^{joint} und Z werden dabei wie folgt durch neue ersetzt:

$$V^{joint,neu} = \left(V^{joint} \cup \left\{M_i \cup M_j\right\}\right) \setminus M_i \setminus M_j\,, \tag{6.12}$$

$$Z^{neu} = \left(Z \cup \left\{M_i \cup M_j\right\}\right) \setminus M_i \setminus M_j\,. \tag{6.13}$$

Ferner berechnet die Funktion unter Verwendung der Kriterien aus Definition 6.3 eine neue Kantenmenge $E^{joint,neu}$. Nach dem Zusammenfassen der Knoten aktualisiert der Algorithmus erneut die Kantengewichte und sucht die nächste Kante mit höchstem Gewinn in Z. In der inneren **while**-Schleife wird der Clustering-Schritt so lange wiederholt, bis in Z keine Kante mehr ein positives Gewicht aufweist.

Ist dieses Abbruchkriterium für eine zusammenhängende Komponente Z erfüllt, so werden die zu den Knoten $M \in V^{joint}$ gehörigen minimalen Kantenschnitte als Lösungen $POS(s)$ für die Synchronisationsknoten s übernommen. Sobald das Verfahren für alle zusammenhängenden Komponenten von J abgeschlossen ist, sind mit der Funktion $POS(s)$ die optimierten Positionen sämtlicher Synchronisationsknoten des Programms gegeben. Durch das Entfernen der ursprünglichen Synchronisationsknoten s und das Platzieren einer Kopie/Instanz

Algorithmus 6.1 Algorithmus zur Optimierung der Synchronisationsstruktur

Input: Kontrollflussgraph $CFG = (V, E^{cf}, TYP)$,
Synchronisationsknoten $\mathcal{S} := \{v \in V : TYP(v) = SYNC\}$,
Mögliche Positionen $\mathcal{VP} : \mathcal{S} \mapsto \text{Pot}(E^{cf})$
Output: Gewählte Positionen $POS : \mathcal{S} \mapsto \text{Pot}(E^{cf})$

$POS(s) \leftarrow \varnothing, \quad \forall\, s \in \mathcal{S}$
$V^{joint} \leftarrow \{ M \subset \mathcal{S} : |M| = 1 \}$
$E^{joint} \leftarrow$ BerechneKantenmenge(V^j)
$J \leftarrow (V^{joint}, E^{joint})$
$ZK \leftarrow$ ZusammenhaengendeKomponenten(J)
SortiereNachCFGPosition(ZK)
for $i = 0$ **to** $|CP| - 1$ **do**
  $WCET^{sync} \leftarrow$ BerechneAnkunftszeiten(CFG, POS)
  $Z \leftarrow ZK[i]$
  $w^{joint} \leftarrow$ BerechneKantengewichte(Z)
  $e^{max} \leftarrow \arg\max_{e \in E^{joint}} \{w^{joint}(e)\}$
  while $w^{joint}(e^{max}) > 0$ **do**
    $(V^{joint}, E^{joint}, Z) \leftarrow$ FasseKnotenZusammen(e^{max}, J, Z)
    $w^{joint} \leftarrow$ BerechneKantengewichte(Z)
    $e^{max} \leftarrow \arg\max_{e \in E^{joint}} \{w^{joint}(e)\}$
  end while
  for all $M \in Z$ **do**
    for all $s \in M$ **do**
      $POS(s) \leftarrow$ Kantenschnitt(M)
    end for
  end for
end for

von s an jeder Kontrollflusskante $e^{cf} \in POS(s)$ lässt sich das Ergebnis auf die Zwischendarstellung des Programms übertragen.

6.4.2.1. Komplexität des Lösungsverfahrens

Die Komplexität von Algorithmus 6.1 wird im Wesentlichen durch das wiederholte Lösen von Problemen des minimalen s-t-Schnitts innerhalb der Aufrufe von BerechneKantengewichte bestimmt. Wird der Edmonds-Karp Algorithmus für die Berechnung eines s-t-Schnitts durch einen Graphen $G = (V, E)$ verwendet, so beträgt die Komplexität jeweils $\mathcal{O}(|V| \cdot |E|^2)$. Eine effiziente Implementierung von Algorithmus 6.1 puffert die Ergebnisse für alle Kanten und Knoten von J in einem Zwischenspeicher, sodass ein bereits bekannter s-t-Schnitt in BerechneKantengewichte nicht erneut berechnet werden muss.

Für jede zusammenhängende Komponente im initialen Graphen J muss zunächst ein minimaler s-t-Schnitt für jeden Knoten $M \in V^{joint}$ und jede Kante $e \in E^{joint}$ berechnet werden. Für alle Komponenten zusammen fallen also $|S| + |E^{joint}|$ Kantenschnitte an. Werden zwei durch eine Kante verbundene Knoten zusammengefasst, so entspricht der s-t-Schnitt des neuen Knotens dem der bisherigen Kante, weshalb er dann nicht erneut berechnet werden muss. Zusätzliche s-t-Schnitte müssen hingegen für diejenigen Kanten im aktualisierten Graphen J berechnet werden, die den neu entstandenen zusammengefassten Knoten mit anderen Knoten verbinden. Im ungünstigsten Fall existiert eine solche Kante zu jedem der anderen Knoten in der zusammenhängenden Komponente Z, sodass der Edmonds-Karp Algorithmus nach jedem Clustering-Schritt maximal $|Z|$ Mal ausgeführt wird. Durch jeden Clustering-Schritt reduziert sich $|Z|$ außerdem um 1, sodass für nachfolgende Schritte weniger Berechnungsaufwand nötig ist.

In dem für die Rechenzeit des Algorithmus ungünstigsten Fall besteht der initiale Graph J aus einer einzigen zusammenhängenden Komponente. Für die darin enthaltenen Knoten und Kanten werden zunächst $|\mathcal{S}| + |E^{joint}|$ s-t-Schnitte berechnet. Im Anschluss an den n-ten Clustering-Schritt kommen dann jeweils maximal $|\mathcal{S}| - 2 - n$ Kantenschnitt-Berechnungen hinzu, wobei die maximale Zahl von Clustering-Schritten auf $|\mathcal{S}| - 1$ begrenzt ist. Für $|\mathcal{S}| \geq 2$ ergeben sich daraus insgesamt

$$|\mathcal{S}| + |E^{joint}| + \sum_{n=0}^{|\mathcal{S}|-2} \Big\{ |\mathcal{S}| - 2 - n \Big\} \tag{6.14}$$

Kantenschnitt-Probleme. Da J per Definition ein einfacher Graph ist, kann die Menge E^{joint} nicht mehr als $|\mathcal{S}|^2$ Elemente enthalten. Die Komplexität von

Algorithmus 6.1 hinsichtlich der Anzahl der Kantenschnitt-Probleme lässt sich damit für eine Menge $\mathcal{S}$ von Synchronisationsknoten durch

$$\mathcal{O}\left(|\mathcal{S}|^2\right) \tag{6.15}$$

beschreiben.

In realen Programmen tritt der für obige Abschätzung verwendete Fall einer einzigen zusammenhängenden Komponente und einer großen Zahl von Kanten $|E^{joint}|$ in J kaum auf. Andernfalls müssten sich praktisch sämtliche Synchronisationsoperationen des Programms zu einer einzigen Einzeloperation zusammenfassen lassen, was bestenfalls für sehr kleine Programme zu erwarten ist.

6.5. Diskussion & Abgrenzung

In diesem Kapitel wurde ein Ansatz zur optimierten Platzierung einer gegebenen Menge von Synchronisationsoperationen in einem Kontrollflussgraphen vorgestellt. Das Verfahren erweitert bzw. ergänzt die herkömmliche Optimierung von Schedules, indem es eine feingranulare Neuplatzierung unabhängig von den Task-Grenzen ermöglicht. Dadurch wird eine globale Optimierung über Schleifen und Kontrollflussverzweigungen hinweg unterstützt. Anders als bei Ansätzen wie dem hierarchischen Task-Graphen wird der gesamte CFG als zusammenhängender Lösungsraum betrachtet, ohne dass einzelne Schleifen oder Hierarchieebenen isoliert voneinander optimiert werden müssen. Durch diesen Ansatz können dann z. B. Synchronisationsoperationen unter bestimmten Voraussetzungen aus Schleifen bzw. stark zusammenhängenden Komponenten des CFG herausgezogen werden, um ihre Ausführungshäufigkeit zu reduzieren. Weiterhin ist es möglich, redundante Synchronisationsoperationen aus unterschiedlichen Iterationsdomänen an eine gemeinsame Postion zu verschieben und sie dort zu einer kombinierten Operation zusammenzufassen.

Der vorgestellte Ansatz ist speziell auf das in Kapitel 4 eingeführte Programmiermodell und die zugehörige Compiler-Transformation ausgerichtet und eignet sich daher vor allem für die WCET-optimierte Software-Parallelisierung. Mit entsprechend angepassten Kostenmetriken kann das Konzept aber grundsätzlich auch zur Verbesserung der Performanz im durchschnittlichen Fall verwendet werden. Für Software ohne Echtzeitanforderungen ist das zugrundeliegende Programmiermodell jedoch vergleichsweise restriktiv und die Ziele der vorgestellten statischen Optimierung (insbesondere das Vermeiden von Wartezeiten bei Synchronisation) können dort ggf. auch durch geeignete Laufzeit-Systeme erreicht werden. Aus diesem Grund bleiben statisch vorhersagbare Echtzeitprogramme weiterhin der hauptsächliche Anwendungsbereich des vorgestellten Verfahrens.

Für Echtzeitanwendungen gibt es im Stand der Technik aus Kapitel 3 bisher keine vergleichbaren Ansätze, die auch Kontrollflussgraphen mit Zyklen unterstützen. Die in Abschnitt 3.3.2 diskutierten Arbeiten zur Parallelisierung von Echtzeitprogrammen basieren entweder auf azyklischen Graphen und bewegen sich eher im Bereich der Schedule-Optimierung (z. B. [24; 73; 82]), oder sie überlassen die (manuelle) Optimierung auf Quellcode-Ebene dem Endnutzer (z. B. [33; 103]).

Kapitel 7.

Heterogene Speicherverwaltung für parallele Echtzeit-Software

Sowohl das in Kapitel 4 präsentierte Programmiermodell als auch das Plattformmodell aus Kapitel 5 unterstützen Hardware-Plattformen mit einer Kombination aus verteilten und gemeinsam genutzten Speichersegmenten. Diese Speichersegmente können i. A. *heterogen* hinsichtlich ihrer Größen, Speicheranbindungen, Zugriffsgeschwindigkeiten und der darauf zugreifenden Prozessorkerne sein. Aus Gründen der statischen Vorhersagbarkeit bei der WCET-Analyse, insbesondere im Hinblick auf Interferenzeffekte, muss die Belegung der Speicher bzw. der logischen Speichersegmente des Programmiermodells zur Entwurfszeit festgelegt werden. Hierzu ist in der Compiler-Transformation aus Kapitel 4 ein gesonderter Schritt zur Speicherallokation vorgesehen, der die Belegung der verfügbaren Speicher festlegt und optimiert (vgl. Abschnitt 4.2). Da die Speicherallokation einen entscheidenden Faktor für die Anzahl kollidierender Speicherzugriffe in gemeinsam genutzten Ressourcen darstellt, ist es naheliegend, die Kosten von Interferenzeffekten in dem zugehörigen Optimierungsmodell zu berücksichtigen. In diesem Kapitel soll daher ein Verfahren zur Optimierung der Speicherallokation entwickelt werden, das das Interferenzmodell aus Kapitel 5 explizit mit einbezieht.

Die Berücksichtigung von Interferenzeffekten bei der Speicherallokation stellt im Vergleich zu herkömmlichen Ansätzen eine wesentliche Erweiterung des Standes der Technik dar (vgl. Abschnitt 3.2.1). Durch die Verwendung des generischen Plattformmodells aus Kapitel 5 lässt sich das im Folgenden vorgestellte Verfahren außerdem auf eine breite Palette von Zielplattformen, einschließlich NoC-basierter Vielkernprozessoren, anwenden. Die Beiträge aus dem Bereich der Speicherverwaltung sind auch Gegenstand der eigenen Veröffentlichung [RB20a] und sollen im Folgenden detaillierter vorgestellt werden.

Die Gliederung dieses Kapitels besteht aus der Definition der Ziele in Abschnitt 7.1, gefolgt von der Definition der Problemstellung in Abschnitt 7.2.

Daran schließt in Abschnitt 7.3 die Beschreibung eines vorbereitenden Optimierungsschritts an, der der in Abschnitt 7.4 vorgestellten Methode zur Speicherallokation vorangestellt ist. Abschnitt 7.5 schließt das Kapitel dann mit einer Diskussion des vorgestellten Ansatzes ab.

7.1. Zielsetzung

Das Hauptziel der in dieser Arbeit entwickelten Speicherallokationsmethode besteht darin, die Speichernutzung in parallelen Echtzeitprogrammen für eine Rechnerarchitektur mit komplexen verteilten Speicherhierarchien zu optimieren. Dabei gilt es sowohl ein breites Spektrum an Mehrkernprozessorsystemen zu unterstützt als auch die Interferenz-Kosten niedrig zu halten. In diesem Zusammenhang sollen insbesondere die folgenden Aspekte berücksichtigt werden:

1. Die optimierte Verteilung der Daten eines parallelen Programms auf nahezu beliebige heterogene Speicherhierarchien.
2. Die Unterstützung von privaten sowie gemeinsam von mehreren Prozessen genutzten Variablen.
3. Berücksichtigung der Kosten von Interferenzeffekten.
4. Weitgehende Unabhängigkeit von der konkreten Zielplattform.

Im Vergleich zu den in Abschnitt 3.2.1 diskutierten Ansätzen ist diese Zielsetzung weniger auf die Allokation schneller Scratchpad-Speicher als Ersatz für Hardware-Caches ausgerichtet. Vielmehr steht die effiziente Nutzung heterogener und verteilter Speicherhierarchien im Vordergrund. Die klassische Aufteilung in Hauptspeicher und Scratchpad-Speicher, wie sie in vielen Ansätzen zur Scratchpad-Allokation vorausgesetzt wird, ist dort i. A. nicht gegeben. Um die genannten Ziele zu erreichen, bedarf es einer allgemeineren Allokationsmethode, die nicht auf der Annahme einer spezifischen Speicherarchitektur beruht. Neben den Eigenschaften der Speicherkomponenten an sich, sollte ein WCET-basierter Ansatz für Mehrkernprozessoren dabei insbesondere auch die Speicheranbindung und die darin auftretenden Interferenzeffekte mit einbeziehen.

Da die Ziele dieser Arbeit auf Mehrkernprozessoren mit heterogenen Speicherhierarchien ausgerichtet sind, soll ein *statisches* Allokationschema den dynamischen Ansätzen vorgezogen werden (vgl. Abschnitt 3.2.1). Die Bezeichnung als „*dynamische*“ Allokation ist in diesem Kontext nicht zu verwechseln mit den dynamischen *Heap-Speichern*, die in C z. B. von den Standardfunktionen `malloc` und `free` implementiert werden. Im Vergleich zu statischen Ansätzen haben dynamische Allokationsverfahren für die vorgesehene Anwendung in Mehrkernprozessoren einige entscheidende Nachteile. Dazu zählen konkret:

Zusätzliche Speicherzugriffe: Die dynamische Speicherallokation erlaubt explizit das Verschieben von Variablen zwischen den Speichersegmenten. Im Vergleich zu einer rein statischen Zuweisung entstehen dadurch zusätzliche Speicherzugriffe sowohl im Quell- als auch im Ziel-Speichersegment. In Mehrkernprozessoren erhöht sich dadurch i. A. die Last auf den Kommunikationsverbindungen und damit auch die durch Interferenzen hervorgerufenen Verzögerungen für alle betroffenen Prozessorkerne.

Fragmentierung: Dynamische Allokation ist grundsätzlich von dem Problem der Fragmentierung des Adressraums betroffen. Wird eine vergleichsweise kleine Variable in einen anderen Speicher verschoben, so wird im bisherigen Speicher i. A. nur ein entsprechend kleiner Adressbereich frei. Ohne die Verschiebung weiterer Variablen lässt sich dieser Adressbereich dann nicht mehr für Variablen mit höherem Speicherbedarf verwenden. Mit der Zeit kann es deshalb zu Situationen kommen, in denen ein Speichersegment nicht mehr vollständig belegt werden kann, da kein zusammenhängender Adressbereich mit ausreichender Länge zur Verfügung steht.

Höhere Komplexität des Optimierungsproblems: In größeren Mehrkernprozessoren mit verteilten Speichern führt die erhöhte Zahl von Speicherbereichen in Verbindung mit den zusätzlich anfallenden Interferenz-Kosten im Vergleich zu Einzelkernprozessoren bereits zu einem signifikant komplexeren Allokationsproblem. Wenn außerdem noch das dynamische Verschieben von Variablen zur Laufzeit berücksichtigt werden soll, steigt die Komplexität des Problems weiter an. Darüber hinaus ist zu erwarten, dass diese zusätzliche Komplexität aufgrund der beiden zuvor genannten Nachteile nur einen geringen Gegenwert mit sich bringt.

Die dadurch begründete Wahl eines statischen Verfahrens bedeutet indes nicht, dass Speicherbereiche, die nach dem Ende der Lebensspanne einer Variablen frei werden, im weiteren Programmverlauf ungenutzt bleiben müssen. Die Wiederverwendung von freiwerdendem Speicher gehört im Rahmen des zu entwickelnden statischen Allokationsverfahrens deshalb ebenfalls zu den wesentlichen Zielen.

7.2. Problemstellung

Im Kontext der in Kapitel 4 vorgestellten Compiler-Transformation besteht die grundsätzliche Problemstellung der Speicherallokation darin, die Variablen des Programms so auf die Speichersegmente abzubilden, dass die WCET des gesamten parallelen Programms minimiert wird. Das Optimierungskriterium der WCET muss dabei durch ein geeignetes Modell geschätzt werden, da abschließende Information erst nach der Kompilierung und WCET-Analyse des parallelisierten C-Codes verfügbar sind. Die Variablen $v \in \mathcal{V}$ des Programms sind nach dem Schritt der *SHM Variablenauswahl* bereits in eine Menge $\mathcal{V}^p$ von privaten und eine Menge $\mathcal{V}^s$ von gemeinsam genutzten Variablen unterteilt (vgl. Abschnitt 4.2). Die Variablen in $\mathcal{V}^p$ bzw. $\mathcal{V}^s$ müssen dann für die Dauer ihrer Lebensspanne einem privaten logischen Speichersegment bzw. einem gemeinsam genutzten logischen Speichersegment des Programmiermodells aus Kapitel 4 zugewiesen werden. Diese Speichersegmente werden später jeweils statisch auf eine Speicherkomponente $m_l \in \mathcal{M} \subset \mathcal{C}$ des Plattformmodells aus Kapitel 5 abgebildet, wobei $\mathcal{M}$ die Menge aller Speicherkomponenten im Plattform-Komponenten-Graphen *CG* (vgl. Definition 5.1) repräsentiert. Im Folgenden soll die Abstraktionsebene der logischen Speichersegmente zur Vereinfachung der Notation vernachlässigt, und von einer direkten Zuweisung der Variablen zu den Speicherkomponenten ausgegangen werden. Bei der konkreten Realisierung des Allokationsverfahrens sollte i. A. jedoch nicht auf die Ebene der logischen Speicher verzichtet werden, sodass z. B. die Aufteilung eines gemeinsam genutzten Hauptspeichers in private logische Speichersegmente weiterhin möglich bleibt.

Die Speicherallokation operiert im Rahmen der Compiler-Transformation aus Abschnitt 4.2 auf einer parallelisierten Zwischendarstellung des Programms, die sich wahlweise durch einen parallelen Programmgraphen (PPG) oder einen azyklischen PPIR Graphen (aPPIR) darstellen lässt (vgl. Definition 2.7 bzw. Definition 3.1). Bei einem statischen Allokationsansatz besteht das Grundproblem dann darin, eine feste Zuweisung $\mu^v : \mathcal{V} \mapsto \mathcal{M}$ von Variablen auf Speicherkomponenten zu ermitteln. In Echtzeit-Software muss die Zuweisungsfunktion μ^v außerdem zur Entwurfszeit bekannt sein, um bei der statischen Analyse zuverlässige Aussagen über die Zugriffszeiten treffen zu können.

Im Folgenden soll das Allokationsproblem zunächst in der von Avissar et al. [4] verwendeten Formulierung dargestellt werden. Im Gegensatz zu den meisten Ansätzen zur Allokation von Scratchpad-Speichern ist diese Formulierung für eine beliebige Zahl von Speichersegmenten geeignet. Die Autoren betrachten in [4] zwar nur Einzelkernprozessoren, der grundsätzliche Aufbau des Allokationsproblems lässt sich jedoch auch auf Mehrkernprozessoren übertragen. Die

Zuweisungsfunktion μ^v wird dabei durch einen Vektor $\mathcal{I}$ aus Entscheidungsvariablen

$$I_{pl} := \begin{cases} 1 & \text{für } \mu^v(v_p) = m_l \\ 0 & \text{für } \mu^v(v_p) \neq m_l \end{cases} \tag{7.1}$$

mit den zulässigen Werten 1 und 0 beschrieben. Für ein Paar $(v_p, m_l) \in \mathcal{V} \times \mathcal{M}$ aus Variable und Speicherkomponente gibt I_{pl} dann an, ob die Variable diesem Speicher zugewiesen ist ($I_{pl} = 1$) oder nicht ($I_{pl} = 0$). Da eine Variable v_p nur genau einem Speicher zugewiesen werden darf, muss weiterhin

$$\sum_{\forall m_l \in \mathcal{M}} I_{pl} \;=\; 1 \qquad \forall\, v_p \tag{7.2}$$

gelten.

Gültige Zuweisungsfunktionen μ^v müssen außerdem die Nebenbedingung erfüllen, dass einem Speicher m_l zu keinem Zeitpunkt mehr Variablen zugewiesen sein dürfen, als es dessen Speicherkapazität $|m_l|$ erlaubt. Unter Vernachlässigung der Möglichkeit, Speicherbereiche außerhalb der Lebensspanne einer Variablen wiederzuverwenden, lässt sich diese Nebenbedingung durch

$$\sum_{\forall v_p} I_{pl} \cdot |v_p| \;\leq\; |m_l| \qquad \forall m_l \tag{7.3}$$

beschreiben. Dabei bezeichnet $|v_p|$ den zur Speicherung von v_p benötigten Speicherplatz.

7.2.1. Wiederverwendung von Speicherplatz bei Stack-Variablen

Um den Speicherplatz $|v_p|$ nach dem Ablauf der Lebensdauer einer Variablen v_p auf dem Programm-Stack wiederverwenden zu können, schlagen Avissar et al. [4] eine Erweiterung der Nebenbedingung aus Gleichung (7.3) vor. Als Voraussetzung müssen zunächst geeignete Basiseinheiten für den Programm-Stack definiert werden. Da diese Arbeit auf der Programmiersprache C basiert, sollen im Folgenden die „{…}“-Blöcke, die im C-Code jeweils einen möglichen Gültigkeitsbereich für Stack-Variablen darstellen, als Basiseinheiten verwendet werden. Aus diesen Gültigkeitsbereichen lässt sich dann ein gerichteter Graph G^{stack} wie folgt definieren:

Definition 7.1 (Stack-Graph)**.** *Sei $g_i \in V^{stack}$ eine Menge von Gültigkeitsbereichen für Variablen („{…}“-Blöcke) in einem C-Programm. Dann heißt der*

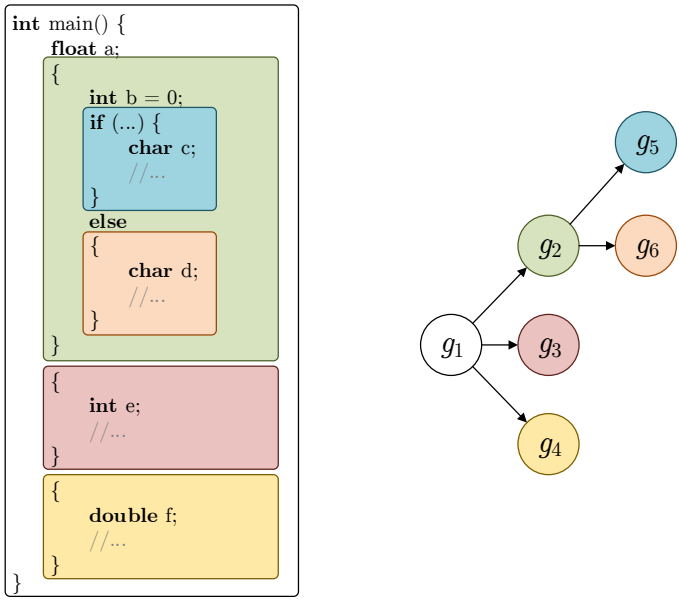


Abbildung 7.1.: Verschachtelte Gültigkeitsbereiche in C mit zugehörigem Graphen G^{stack}

gerichtete Graph $G^{stack} = \left(V^{stack}, E^{stack}\right)$ *Stack-Graph des Programms, falls die Kante* $(g_i, g_j) \in E^{stack}$ *genau dann existiert, wenn*

- *der Gültigkeitsbereich* g_j *Bestandteil des Gültigkeitsbereiches* g_i *ist oder*
- g_j *der größte Gültigkeitsbereich einer C-Funktion ist, die innerhalb von* g_i *aufgerufen wird.*

In jedem der Gültigkeitsbereiche g_i kann eine Menge von Stack-Variablen definiert sein, sodass g_i auch als eine Menge $g_i \subseteq \mathcal{V}$ von Variablen interpretiert werden kann. Ist die in Abschnitt 3.3.3.5 gestellte Forderung, dass keine rekursiven Funktionsaufrufe vorhanden sind, für ein C-Programm erfüllt, so gibt es in G^{stack} außerdem keine gerichteten Zyklen.

Abbildung 7.1 veranschaulicht den Graphen G^{stack} für ein einfaches Quellcode-Beispiel. Darin liegen z. B. die Variablen `b` und `e` niemals gleichzeitig im Stack-Speicher, was sich in G^{stack} daran zeigt, dass zwischen den zugehörigen Gültigkeitsbereichen g_2 und g_3 kein gerichteter Pfad existiert. In diesem Fall genügt es

also, nur den Speicherplatz für ein Datum vom Typ `int` zu reservieren, und ihn dann für beide Variablen zu verwenden. In dem dargestellten Graphen G^{stack} steht der Knoten g_1 für den größten Gültigkeitsbereich in der Einstiegs-Funktion des Programms (in C typischerweise die Funktion `main`) und markiert damit den Quellknoten $q \in V^{stack}$. Ist G^{stack} zyklenfrei, so lassen sich außerdem eine Reihe von Senken $\zeta_i \in V^{stack}$ ermitteln, die keine untergeordneten Gültigkeitsbereiche mehr enthalten. In Abbildung 7.1 sind das die Knoten g_3, g_4, g_5 und g_6. Durchläuft der Kontrollfluss einen Basisblock innerhalb des Gültigkeitsbereiches g_i, so muss der Programm-Stack zu diesem Zeitpunkt die Variablen $v_p \in g_i$ sowie die Variablen aller Gültigkeitsbereiche auf dem Pfad von der Quelle q zu g_i enthalten. In den Senken ζ_i erreicht die Größe des Programms-Stack dann ein lokales Maximum, da es dort keine weiteren untergeordneten Gültigkeitsbereiche zur Verlängerung dieses Pfades mehr gibt. Betrachtet man den aufsummierten Speicherbedarf aller Variablen eines Gültigkeitsbereiches g_i als dessen Knotengewicht $w^{var}(g_i)$, so ergibt sich das globale Maximum der Stack-Größe aus der längsten aller Knotenfolgen von der Quelle q zu einer der Senken ζ_i.

Mit Hilfe von G^{stack}, seiner Quelle q und den Senken ζ_i lässt sich die Nebenbedingung aus Gleichung (7.3) für die Stack-basierte Speicherverwaltung neu formulieren (vgl. [4]):

$$\sum_{\omega \in \mathcal{W}(q,\zeta_i)} \sum_{g_i \in \omega} \sum_{v_p \in g_i} I_{pl} \cdot |v_p| \quad \leq \quad |m_l| \qquad \forall m_l, \zeta_i \,. \tag{7.4}$$

Dabei bezeichnet $\mathcal{W}(q, \zeta_i)$ die Menge aller Pfade in G^{stack}, die in q beginnen und in ζ_i enden. Globale Variablen, die nicht auf dem Stack abgelegt werden, können in dieser Formulierung als Bestandteil des Quellknotens q betrachtet werden. Die Verwendung unterschiedlicher Speicherkomponenten impliziert außerdem, dass das Programm nach der Speicherallokation nicht nur einen primären Programm-Stack, sondern i. A. eine Stack-basierte Speicherstruktur in jedem Speichersegment $m_l \in \mathcal{M}$ enthält.

7.2.2. Definition des Allokationsproblems

Unter Verwendung von Gleichung (7.4) lässt sich das Problem der Speicherallokation für eine beliebige Zielfunktion dann wie folgt definieren:

Definition 7.2 (Problem der Speicherallokation)**.** *Sei $Y(\mathcal{I})$ eine skalarwertige Kostenfunktion, die von den Entscheidungsvariablen I_{pl} abhängt, und G^{stack} der*

Stack-Graph des Programms mit der zugehörigen Funktion $\mathcal{W}(q, \zeta_i)$. Dann ist das Optimierungsproblem der Speicherallokation durch

$$\min_{\forall \mathcal{I}} \left\{ Y(\mathcal{I}) \right\} \qquad \textit{mit}$$

$$\sum_{\forall m_l \in \mathcal{M}} I_{pl} \;=\; 1 \qquad \forall\, v_p \;,$$

$$\sum_{\omega \in \mathcal{W}(q,\zeta_i)} \sum_{g_i \in \omega} \sum_{v_p \in g_i} I_{pl} \cdot |v_p| \;\leq\; |m_l| \qquad \forall m_l, \zeta_i$$

gegeben.

Diese Problemstellung lässt sich grundsätzlich auch auf Mehrkernprozessoren übertragen. Dazu muss bei der Zuweisung der Variablen $v_p \in \mathcal{V}$ sichergestellt werden, dass alle Prozessorkerne, die die Daten in v_p benötigen, auch Zugriff auf das entsprechende Speichersegment haben. Dies lässt sich dadurch erreichen, dass Entscheidungsvariablen I_{pl} nur dann in das Problem einbezogen werden, wenn die besagte Bedingung für die zugehörige Zuweisungsoption erfüllt ist.

Da beide Nebenbedingungen aus Definition 7.2 bezüglich der ganzzahligen Entscheidungsvariablen I_{pl} *linear* sind, lässt sich das Optimierungsproblem mit den Methoden der ganzzahligen linearen Optimierung (ILP) lösen, sofern auch die Kostenfunktion Y linear ist (vgl. Definition 2.8). Avissar et al. [4] schlagen zu diesem Zweck eine einfache lineare Kostenfunktion für Einzelkernprozessoren auf Basis einer konstanten Zugriffszeit für jedes Speichersegment vor. Für die Anwendung auf Mehrkernprozessoren und Echtzeitsystemen ist eine solche Kostenmetrik jedoch weniger geeignet, da sie insbesondere Interferenzeffekte nicht berücksichtigt.

Im Folgenden soll deshalb auf Basis der Problemstellung aus Definition 7.2 ein neuer Ansatz zur WCET-optimierten Speicherallokation für Mehrkernprozessoren vorgestellt werden. Dabei wird das Interferenz-Modell aus Kapitel 5 in eine ILP-Formulierung überführt, um das Allokationsproblem unter Berücksichtigung von Interferenzen mittels ILP optimal lösen zu können. Vor der eigentlichen Optimierung findet jedoch ein Vorverarbeitungsschritt statt, um die Wiederverwendung von Speicherplatz beim Verlassen der Gültigkeitsbereiche von Stack-Variablen zu verbessern. Im nachfolgenden Abschnitt wird deshalb zunächst der letztgenannte Schritt vorgestellt, bevor sich Abschnitt 7.4 mit der eigentlichen Speicherallokation befasst.

7.3. Vorbereitung der Programmstruktur

Die Möglichkeit zur Wiederverwendung von Speicherplatz bei der Speicherallokation basiert auf der Nebenbedingung aus Gleichung (7.4) und damit dem Stack-Graphen gemäß Definition 7.1. Um möglichst niedrige Stack-Größen zu erreichen, ist es i. A. sinnvoll, für jede Variable $v_p \in \mathcal{V}$ einen möglichst kleinen Gültigkeitsbereich zu wählen. Hierzu können im Wesentlichen zwei Vorgehensweisen verfolgt werden:

1. Das Verschieben von Variablen in kleinere Gültigkeitsbereiche, die im Programm bereits existieren.
2. Das Einfügen zusätzlicher Gültigkeitsbereiche bzw. „{…}“-Blöcke an geeigneten Stellen.

Im Rahmen dieser Arbeit wird eine Kombination aus beiden Vorgehensweisen eingesetzt, um die Lebensspannen der Stack-Variablen möglichst stark reduzieren zu können. Das Verschieben einer Variablen v_p in den kleinstmöglichen existierenden Gültigkeitsbereich lässt sich relativ einfach realisieren. Hierzu wird im Graphen G^{stack} der kleinste Gültigkeitsbereich $g_i \in V^{stack}$ ermittelt, der sämtliche Basisblöcke mit Zugriffen auf v_p umfasst. Das Einfügen neuer Gültigkeitsbereiche ist dagegen mit Abwägungen verbunden, da sich „{…}“-Blöcke in C zur Wahrung des Sack-Prinzips im Quellcode nicht überlappen dürfen. Die Konstruktion überlappungsfreier neuer Gültigkeitsbereiche stellt daher i. A. ein Optimierungsproblem dar, für das im Folgenden ein Lösungsverfahren vorgestellt wird.

Vor dem Einfügen neuer Gültigkeitsbereiche werden die Variablen zunächst, wie beschrieben, jeweils in den kleinstmöglichen existierenden Gültigkeitsbereich verschoben. Anschließend kann das besagte Optimierungsproblem für jeden existierenden Gültigkeitsbereich g_i in G^{stack} separat formuliert werden. Gesucht ist dann eine Menge aus neuen, überlappungsfreien Gültigkeitsbereichen, die g_i auf geeignete Weise unterteilen. Hierzu werden die n Kind-Knoten des zu g_i gehörigen „{…}“-Blocks im abstrakten Syntaxbaum (AST, vgl. Abschnitt 2.3.1) mit einem Index $k \in \{0, 1, \ldots, n-1\}$ der Reihe nach aufgelistet. Jeder Variablen v_p im Gültigkeitsbereich g_i kann dann auf Basis dieses Index eine Lebensspanne $\lambda(v_p) = [a, b]$ zugeordnet werden. Letztere definiert in der Liste der Kind-Knoten von g_i mit $k = a$ den ersten und mit $k = b$ den letzten Knoten, der auf die Daten in v_p zugreift.

Abbildung 7.2 zeigt ein Quellcode-Beispiel mit den zugehörigen Lebensspannen $\lambda(v_p)$ der einzelnen Variablen. Es handelt sich um einen Auszug aus einer einfachen Implementierung des *Canny-Algorithmus* [12] zur Kantendetektion in Graustufenbildern. Der Algorithmus wandelt ein Eingabebild in mehreren

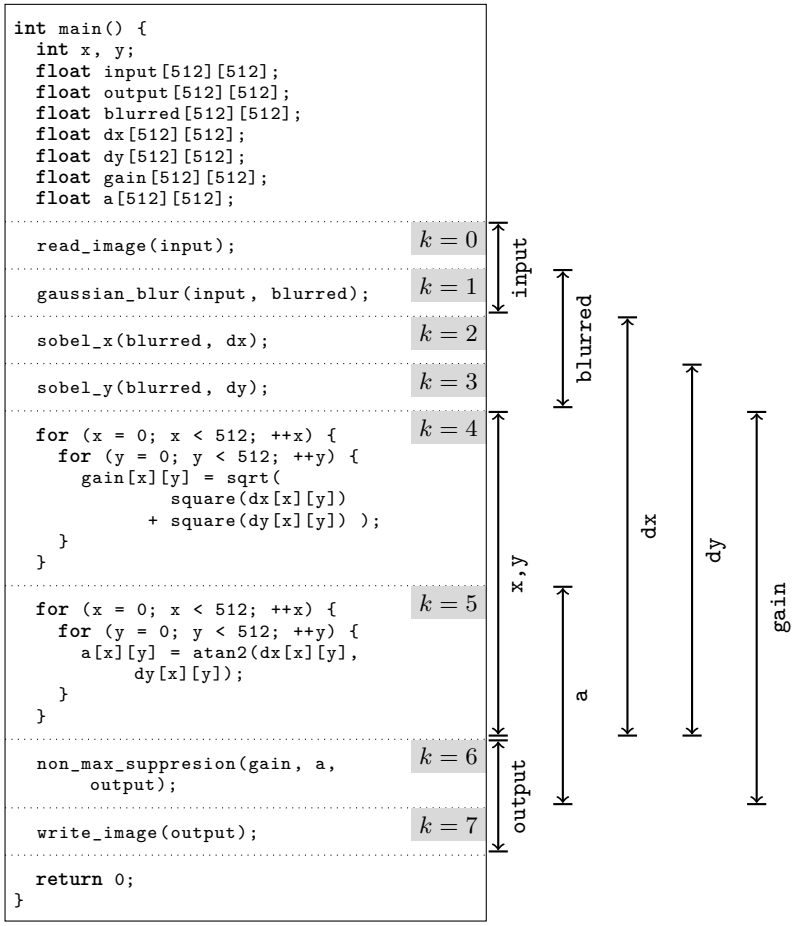


Abbildung 7.2.: Code-Beispiel für einen Kantendetektor nach Canny mit grafisch dargestellten Lebensspannen λ der Variablen

```
// ...

{
  float input[512][512];
  read_image(input);
  gaussian_blur(input, blurred);
}

// ...

{
  float output[512][512];
  non_max_suppresion(gain, a, output);
  write_image(output);
}

// ...
```

Abbildung 7.3.: Reduzierte Gültigkeitsbereiche für die Variablen `input` und `output` aus Abbildung 7.2

Schritten in ein Kantenbild um, wobei die zu einer Kante gehörigen Pixel des Eingabebildes einen hohen Grauwert im Ausgabebild erhalten. In dem Beispiel-Code sind dem obersten Gültigkeitsbereich der `main`-Funktion $n = 8$ relevante AST-Knoten untergeordnet, die in der Abbildung von $k = 0$ bis $k = 7$ durchnummeriert sind. Die Lebensspannen der in `main` deklarierten Stack-Variablen sind auf der rechten Seite von Abbildung 7.2 grafisch dargestellt. In dieser Darstellung ist direkt zu erkennen, dass sich z. B. die Lebensspannen der Variablen `input` mit $\lambda = [0, 1]$ und `output` mit $\lambda = [6, 7]$ nicht überlappen. Erstellt man für diese beiden Variablen jeweils einen neuen „{…}“-Block, der nur den Code der Knoten in $[0, 1]$ bzw. $[6, 7]$ enthält, so können beide Variablen in diese kleineren Gültigkeitsbereiche verschoben werden. Das Ergebnis dieser Änderungen ist in Abbildung 7.3 skizziert. Die beiden Variablen befinden sich dann in orthogonalen Gültigkeitsbereichen und belegen deshalb niemals zur gleichen Zeit Speicher auf dem Stack. Der maximale Speicherbedarf des Programm-Stack kann also um die Größe einer dieser Variablen verringert werden.

Während die Situation bei den Variablen `input` und `output` des Beispielprogramms vergleichsweise einfach ist, können sich die Lebensspannen zweier Variablen im Allgemeinen auch überlappen. Bei `dx` und `dy` aus Abbildung 7.2 ist die Lebensspanne $\lambda(v_y)$ von `dy` z. B. eine Teilmenge der Lebensspanne $\lambda(v_x)$ von `dx`. In diesem Fall könnten zwei neue, verschachtelte Gültigkeitsbereiche eingefügt werden, sodass auch diese beiden Variablen nur während ihrer Lebensspanne Speicher auf dem Stack belegen. Anders sieht die Situation aus, wenn sich zwei Lebensspannen $\lambda(v_p)$ und $\lambda(v_q)$ überlappen, aber keine Teilmengen voneinander

sind. In Abbildung 7.2 trifft dies z. B. auf `input` und `blurred` zu. Es ist dann nicht mehr möglich, Stack-basierte Gültigkeitsbereiche zu finden, in denen beide Variablen nicht länger als ihre Lebensspanne gültig bleiben. In solchen Fällen gibt es also einen Konflikt zwischen zwei Variablen, dessen Anwesenheit durch die Bedingung

$$\theta(v_p, v_q) \;:=\; \big(\lambda(v_p) \cap \lambda(v_q) \neq \emptyset\big) \wedge \big(\lambda(v_p) \not\subseteq \lambda(v_q)\big) \wedge \big(\lambda(v_p) \not\subseteq \lambda(v_q)\big) \tag{7.5}$$

beschrieben werden kann. Weiterhin kann mit

$$\Theta(v_p) := \big\{ v_q \in g_i : \; \theta(v_p, v_q) \big\} \tag{7.6}$$

die Menge aller mit v_p in Konflikt stehenden Variablen definiert werden.

Existiert ein Konflikt $\theta(v_p, v_q)$ zwischen v_p und v_q, so muss eine der beiden Variablen in den nächstgrößeren Gültigkeitsbereich $\Gamma(\{v_p, v_q\})$, der mit $\big(\lambda(v_p) \cup \lambda(v_q)\big) \subseteq \Gamma(\{v_p, v_q\})$ die Lebensspannen beider Variablen umfasst, verschoben werden. Im Folgenden wird angenommen, dass $\Gamma(\{v_p, v_q\})$ dabei entweder der Lebensspanne $\lambda(v_r)$ einer dritten Variablen mit $\big(\lambda(v_p) \cup \lambda(v_q)\big) \subseteq \lambda(v_r)$ oder der gesamten Spanne $[0, n-1]$ entspricht. Demnach muss dann eine der beiden Variablen `input` und `blurred` aus dem Beispiel in Abbildung 7.2 in den Gültigkeitsbereich $[0, 7]$ verschoben werden. In dem konkreten Fall ist es sinnvoller, wenn die Variable `blurred` den größeren Gültigkeitsbereich erhält, sodass eine zusätzliche Überlappung von `input` mit `dx` und `dy` vermieden wird. Im allgemeinen Fall stehen jedoch mehr als zwei Variablen in Konflikt zueinander, sodass der nächstgrößere Gültigkeitsbereich Γ mit $\Gamma\big(\Theta(v_p)\big)$ für eine Menge $\Theta(v_p)$ aus mehreren in Konflikt stehenden Variablen verallgemeinert werden muss. Die Definition von $\Gamma\big(\Theta(v_p)\big)$ ergibt sich dann analog zu dem Fall mit zwei Variablen als eine Obermenge der Vereinigung aller Lebensspannen der Variablen in $\Theta(v_p)$.

Die optimale Wahl der Gültigkeitsbereiche stellt i. A. ein Optimierungsproblem dar, zu dessen Lösung die genannten binären Entscheidungen für alle Konflikte $\theta(v_p, v_q)$ unter Minimierung der Stack-Größe festgelegt werden müssen. Wie bei der Speicherallokation lässt sich dieses Problem durch ein ILP-Modell darstellen und optimal lösen. Hierzu wird für jede Variable $v_p \in g_i$ eine ILP-Entscheidungsvariable

$$\delta_p := \begin{cases} 1 & v_p \text{ wird in den Gültigkeitsbereich } \Gamma\big(\Theta(v_p)\big) \text{ verschoben} \\ 0 & v_p \text{ erhält einen eigenen Gültigkeitsbereich mit } \lambda(v_p) \end{cases} \tag{7.7}$$

definiert. Die ILP-Variablen entscheiden also darüber, ob der Gültigkeitsbereich einer Variablen v_p im Vergleich zu ihrer Lebensspanne $\lambda(v_p)$ vergrößert werden

muss oder nicht. Da im Falle eines Konflikts $\theta(v_p, v_q)$ mindestens eine der beteiligten Variablen in einen nicht-minimalen Gültigkeitsbereich verschoben werden muss, gelten außerdem die Nebenbedingungen

$$\delta_p + \delta_q \geq 1 \qquad \forall p, q : \ \theta(v_p, v_q)\,. \tag{7.8}$$

Als Kostenfunktion dient die maximale Stack-Größe $\mathfrak{S}$, die durch

$$\begin{aligned} \mathfrak{S} \ = \ \max_{v_p \in g_i} \left\{ \sum_{v_q \in \Lambda(v_p)} (1 - \delta_q) \cdot |v_q| + \sum_{v_q \in \Lambda^\Gamma(v_p)} \delta_q \cdot |v_q| \right\} \\ \text{mit } \Lambda(v_p) := \left\{ v_q \in g_i : \ \lambda(v_p) \subseteq \lambda(v_q) \right\} \\ \text{und } \Lambda^\Gamma(v_p) := \left\{ v_q \in g_i : \ \lambda(v_p) \subseteq \Gamma\big(\Theta(v_q)\big) \right\} \end{aligned} \tag{7.9}$$

gegeben ist[1]. Beim Betreten des neuen Gültigkeitsbereichs einer beliebigen Variable v_p muss der Stack per Definition bereits alle Variablen v_q mit einem größeren Gültigkeitsbereich als $\lambda(v_p)$ enthalten. Die Mengen dieser Variablen, einschließlich v_p selbst, sind in Gleichung (7.9) durch $\Lambda(v_p)$ bzw. $\Lambda^\Gamma(v_p)$ gegeben. Dabei liegt für $\Lambda(v_p)$ die Annahme des minimalen Gültigkeitsbereiches zugrunde, während $\Lambda^\Gamma(v_p)$ die entsprechende Menge für den Fall eines vergrößerten Gültigkeitsbereiches darstellt. Die Gewichtung der Variablengrößen $|v_q|$ mit den Entscheidungsvariablen δ_q sorgt in den beiden Summen dafür, dass die entsprechende binäre Entscheidung in die Berechnung des Speicherbedarfs für den Stack eingeht. Zusammen geben die beiden Summen dann die Stack-Größe unmittelbar zu Beginn (und aufgrund des Stack-Prinzips auch am Ende) der Lebensspanne der betrachteten Variablen v_p an. Durch die Bildung des Maximalwerts über alle Variablen v_p muss notwendigerweise auch der Fall enthalten sein, in welchem die Stack-Größe das gesuchte Maximum $\mathfrak{S}$ annimmt.

Während die Nebenbedingung in Gleichung (7.8) linear ist, kann der max-Operator aus Gleichung (7.9) in einem ILP-Modell nicht direkt realisiert werden. Da das gesuchte Maximum $\mathfrak{S}$ jedoch gleichzeitig die Kostenfunktion des Optimierungsproblems bildet, muss eine entsprechende ILP-Variable in der optimalen Lösung den minimal möglichen Wert annehmen. Durch eine Reihe von Nebenbedingungen lässt sich dann sicherstellen, dass dieses Minimum für keine der Variablen $v_p \in g_i$ kleiner als die Stack-Größe zu Beginn der jeweiligen Lebensspanne $\lambda(v_p)$ sein kann. Daraus ergibt sich dann das folgende ILP-Problem:

[1] Da nur ein Gültigkeitsbereich g_i in Isolation optimiert wird, kann der Speicherbedarf eventueller Stack-Variablen außerhalb von g_i an dieser Stelle vernachlässigt werden.

Definition 7.3 (ILP-Problem zur Optimierung der Gültigkeitsbereiche)**.** *Gegeben seien die Variablen v_p, deren Lebensspannen $\lambda(v_p)$, die vergrößerten Gültigkeitsbereiche $\Gamma\big(\Theta(v_p)\big)$ und die zugehörigen Mengen $\Theta(v_p)$ der mit v_p in Konflikt stehenden Variablen v_q mit $v_q \in \Theta(v_p) \Leftrightarrow \theta(v_p, v_q)$. Ferner sei mit dem Vektor Δ je eine Entscheidungsvariable δ_p für jede Variable v_p definiert. Das ILP-Problem zur Optimierung der Gültigkeitsbereiche im Knoten g_i des Stack-Graphen ist dann gegeben durch*

$$\min_{\forall \Delta} \{\, \mathfrak{S} \,\}$$

unter den Nebenbedingungen

$$\delta_p + \delta_q \;\geq\; 1 \qquad \forall p, q :\; \theta(v_p, v_q)\,,$$

$$\mathfrak{S} \;\geq\; \left\{ \sum_{v_q \in \Lambda(v_p)} (1 - \delta_q) \cdot |v_q| + \sum_{v_q \in \Lambda^{\Gamma}(v_p)} \delta_q \cdot |v_q| \right\} \qquad \forall v_p \in g_i\,.$$

Dabei gilt

$$\Lambda(v_p) := \Big\{ v_q \in g_i :\; \lambda(v_p) \subseteq \lambda(v_q) \Big\} \quad \textit{und}$$
$$\Lambda^{\Gamma}(v_p) := \Big\{ v_q \in g_i :\; \lambda(v_p) \subseteq \Gamma\big(\Theta(v_q)\big) \Big\}\,.$$

Durch das Lösen des Optimierungsproblems aus Definition 7.3 für jeden existierenden Knoten des Graphen G^{stack} kann die maximale Stack-Größe des Gesamtprogramms bei einer gleichbleibenden Menge von Variablen reduziert werden. Es empfiehlt sich, diesen Schritt vor der eigentlichen Speicherallokation anzuwenden, um das Optimierungspotential hinsichtlich der zweiten Nebenbedingung aus Definition 7.2 zu erhöhen.

7.4. ILP-Modell zur Speicherallokation

Mit Definition 7.2 sind die notwendigen Nebenbedingungen zur Bestimmung einer korrekten Lösung des Allokationsproblems mittels ILP gegeben. Es fehlt jedoch noch eine lineare Kostenfunktion Y, für die im Folgenden mehrere Varianten vorgestellt werden sollen. Die Grundlage dafür bildet das in Kapitel 5 eingeführte Modell der Zielplattform.

7.4.1. Kostenfunktion ohne Interferenz-Modell

Unter Vernachlässigung von Interferenzeffekten lässt sich die Summe aller Speicherzugriffszeiten in Abhängigkeit der Entscheidungsvariablen I_{pl} relativ einfach als lineare Kostenfunktion darstellen. Die in den Gleichungen (5.11) und (5.12) vorgestellten Berechnungsvorschriften für die Speicherzugriffszeiten führen ohne Interferenz zu konstanten Ergebnissen. Dadurch entfällt die Abhängigkeit vom Ausführungskontext bzw. den parallel ablaufenden Code-Segmenten $\mathcal{PS}$. Die resultierenden Interferenz-freien Zugriffszeiten für Speicherzugriffe vom Typ *posted* bzw. *non-posted* sollen im folgenden mit $D^{p}\big(R_{kl}, \emptyset\big)$ bzw. $D^{np}\big(R_{kl}, \emptyset\big)$ bezeichnet werden. Der Ausführungskontext $\mathcal{PS}$ wird dabei durch die leere Menge $\emptyset$ ersetzt, um die Unabhängigkeit von parallel ablaufenden Berechnungen auszudrücken.

Zur Berechnung der Anzahl der Speicherzugriffe $N^{mem,p}(cs_i, m_l)$ beziehungsweise $N^{mem,np}(cs_i, m_l)$ pro Code-Segment (vgl. Gleichung (5.13)) müssen die Zugriffe auf alle Variablen mit einer Zuweisung zu dem Speichersegment m_l berücksichtigt werden. Mit der Anzahl $N^{var,p}(cs_i, v_p)$ der *posted Writes* im Code-Segment cs_i auf die Variable v_p und der korrespondierenden Anzahl $N^{var,np}(cs_i, v_p)$ der Zugriffe vom Typ *non-posted* führt das zu den folgenden Gleichungen:

$$N^{mem,p}\big(cs_i, m_l\big) \;=\; \sum_{v_p \in \mathcal{V}} I_{pl} \cdot N^{var,p}\big(cs_i, v_p\big)\,, \tag{7.10}$$

$$N^{mem,np}\big(cs_i, m_l\big) \;=\; \sum_{v_p \in \mathcal{V}} I_{pl} \cdot N^{var,np}\big(cs_i, v_p\big)\,. \tag{7.11}$$

Unter Verwendung von Gleichung (5.13) lässt sich dann mit der Summe aller Verzögerungen durch Speicherzugriffe eine Zielfunktion Y für den Interferenz-freien Fall definieren:

$$\begin{aligned} Y\left(\mathcal{I}\right) \;=\; \sum_{cs_i \in \mathcal{CS}} \sum_{v_p \in \mathcal{V}} \sum_{\forall m_l} I_{pl} \cdot \Big\{ & D^{np}\big(R_{kl}, \emptyset\big) \cdot N^{var,np}\big(cs_i, v_p\big) \\ & +\; D^{p}\big(R_{kl}, \emptyset\big) \cdot N^{var,p}\big(cs_i, v_p\big) \Big\} \\ & \text{mit} \quad R_{kl} = \big(\mu^{cs}(cs_i), \ldots, m_l\big)\,. \end{aligned} \tag{7.12}$$

Diese Zielfunktion ähnelt stark der von Avissar et al. [4] vorgeschlagenen Kostenmetrik. Da $D^{p}(R_{kl}, \emptyset)$ und $D^{np}(R_{kl}, \emptyset)$ Konstanten sind, ist $Y(\mathcal{I})$ linear und das Optimierungsproblem lässt sich mittels ILP optimal lösen. Beim Übergang zu interferenzabhängigen Zugriffsverzögerungen $D^{p}(R_{kl}, \mathcal{PS})$ und $D^{np}(R_{kl}, \mathcal{PS})$ ist

dies jedoch nicht mehr gegeben, da die in D^p bzw. D^{np} enthaltenen Interferenz-Kosten von der Allokation anderer Variablen und damit i. A. von $\mathcal{I}$ abhängig sind. Aus diesem Grund sind umfassendere Erweiterungen dieses einfachen Optimierungsmodells nötig, um eine lineare Kostenfunktion $Y^{if}(\mathcal{I})$ zu definieren, die auch Interferenzeffekte korrekt abbildet. Vor der detaillierten Herleitung dieser Erweiterungen in Abschnitt 7.4.3 soll im Folgenden zunächst noch eine Variante des ursprünglichen Allokationsproblems für gemeinsam genutzte Variablen eingeführt werden.

7.4.2. Allokation gemeinem genutzter Speicher

Die vorgestellte Kostenfunktion Y lässt sich durch eine geringfügige Erweiterung des Optimierungsproblems aus Definition 7.2 auch für die Allokation gemeinsam genutzter Variablen im ersten Schritt der Compiler-Transformation aus Abschnitt 4.2 verwenden. Die Programmdarstellung ist in diesem Schritt noch sequentiell und die Variablen $v_p \in \mathcal{V}$ müssen in eine Menge $\mathcal{V}^s$ aus gemeinsam genutzten Variablen und eine Menge $\mathcal{V}^p$ aus privaten Variablen unterteilt werden (vgl. Abschnitt 4.2.1.1). Da für alle Variablen $v_p \in \mathcal{V}^p$ im Schritt der Datenpartitionierung mehrere private Kopien angelegt werden, muss sich der dadurch erhöhte Speicherbedarf auch im ILP-Modell widerspiegeln.

Ein entsprechend angepasstes ILP-Problem muss zunächst die Entscheidung zwischen gemeinsamer und privater Nutzung einer Variablen v_p geeignet abbilden. Dazu werden für jedes Paar $(v_p, m_l) \in \mathcal{V} \times \mathcal{M}$ aus Variable und Speicherkomponente nicht nur eine, sondern zwei Entscheidungsvariablen I_{pl}^{priv} und I_{pl}^{shared} definiert. Beide sind analog zu I_{pl} definiert, wobei I_{pl}^{priv} die Allokation als private Kopie und I_{pl}^{shared} die Allokation als gemeinsam genutzte Variable repräsentiert.

Im Folgenden soll außerdem die Anzahl der Prozesse/Prozessorkerne, die auf die Inhalte der Variablen $v_p \in \mathcal{V}$ zugreifen, durch $n^{\pi}(v_p)$ gegeben sein. Wird die Variable der Menge $\mathcal{V}^p$ zugeordnet, so müssen später $n^{\pi}(v_p)$ private Kopien angelegt werden. Die Nebenbedingung

$$\sum_{\forall m_l \in \mathcal{M}} I_{pl} \;=\; 1 \qquad \forall\, v_p$$

aus Definition 7.2 berücksichtigt das jedoch noch nicht, weshalb sie entsprechend abgeändert werden muss. Diese Änderung, in Verbindung mit der Unterscheidung zwischen I_{pl}^{priv} und I_{pl}^{shared}, lässt sich dann wie folgt in einem erweiterten Optimierungsproblem realisieren:

Definition 7.4 (Allokationsproblem mit gemeinsam genutzten Variablen)**.** *Sei $Y(\mathcal{I})$ eine skalarwertige Kostenfunktion, die von den Entscheidungsvariablen I_{pl}^{priv} und I_{pl}^{shared} abhängt, und $n^{\pi}(v_p)$ die Anzahl der Prozessorkerne, die Zugriff auf die Inhalte von v_p benötigen. Dann ist das Optimierungsproblem der Speicherallokation mit gemeinsam genutzten Variablen durch*

$$\min_{\forall \mathcal{I}} \left\{ Y(\mathcal{I}) \right\} \qquad mit$$

$$\sum_{\forall m_l \in \mathcal{M}} \left[I_{pl}^{priv} + n^{\pi}(v_p) \cdot I_{pl}^{shared} \right] = n^{\pi}(v_p) \qquad \forall\, v_p \;,$$

$$\sum_{\omega \in \mathcal{W}(q,\zeta_i)} \sum_{g_i \in \omega} \sum_{v_p \in g_i} |v_p| \cdot \left[I_{pl}^{priv} + I_{pl}^{shared} \right] \leq |m_l| \qquad \forall m_l, \zeta_i$$

gegeben.

Die erste Nebenbedingung in diesem Problem kann nur dann erfüllt werden, wenn entweder genau eine Entscheidungsvariable I_{pl}^{shared} den Wert 1, oder genau $n^{\pi}(v_p)$ Entscheidungsvariablen I_{pl}^{priv} den Wert 1 haben. Die Bedingung drückt damit genau die gewünschte Auswahl zwischen $n^{\pi}(v_p)$ privaten Kopien (in potentiell unterschiedlichen Speichern) oder einer gemeinsam genutzten Instanz der Variablen v_p aus. Sollte eine gemeinsame Verwendung eines bestimmten Speichers m_l in der Hardware-Architektur nicht möglich sein, so können alle zugehörigen Entscheidungsvariablen I_{pl}^{shared} von Anfang an auf 0 gesetzt werden.

Für das Optimierungsproblem aus Definition 7.4 ist es generell sinnvoll, die Entscheidungsvariablen I_{pl}^{priv} und I_{pl}^{shared} mit entsprechenden Koeffizienten in die Kostenfunktion $Y(\mathcal{I})$ eingehen zu lassen. Diese Koeffizienten können genutzt werden, um die Kosten für zusätzliche WAW- oder WAR-Abhängigkeiten (vgl. Abschnitt 4.2.1.1) bei gemeinsamer Nutzung der Variablen bzw. den Zusatzaufwand für Kommunikation bei privaten Kopien geeignet abzubilden. Bedingt durch das *Phase-Ordering Problem* (vgl. Abschnitt 2.3.6) basieren die in dieser Arbeit verwendeten Werte auf einer *a priori* Schätzung der, zum Zeitpunkt der Optimierung noch unbekannten, Kostenfaktoren.

Das erweiterte Optimierungsproblem in Definition 7.4 kann grundsätzlich auch als Basis für das im nächsten Unterabschnitt vorgestellte ILP-basierte Interferenz-Modell verwendet werden. Da die Entscheidung über gemeinsam genutzte Variablen in dem Complier-Werkzeug aus Kapitel 4 jedoch bereits vor der abschließenden Speicherallokation getroffen wird, wird in der vorliegenden Arbeit

kein Gebrauch von dieser Möglichkeit gemacht. Zugunsten einer übersichtlicheren Notation basieren die nachfolgenden Betrachtungen deshalb wieder auf Definition 7.2.

7.4.3. Kostenfunktion mit Interferenz-Modell

Das Interferenzmodell aus Kapitel 5 ermöglicht die Beschreibung von Interferenzeffekten für ein generisches Modell der Zielplattform, enthält jedoch auch einige Nichtlinearitäten, weshalb es nicht ohne Weiteres in ein ILP-Modell überführt werden kann. Insbesondere sind die Gleichungen (5.7) und (5.8) nichtlinear bezüglich der Variablen a_j, die ihrerseits von den Allokationsentscheidungen $\mathcal{I}$ abhängen. Hinzu kommt die in Abschnitt 5.3.1 getroffene Annahme, dass die Indizes j in absteigender Reihenfolge nach a_j sortiert sind. Außerdem kann die Berechnung der Pitches, entweder als $\hat{P}^{rt}(R_{kl}, \mathcal{PS})$ mittels Gleichung (5.10) oder durch Algorithmus 5.1, nicht ohne weiteres durch lineare Gleichungen ausgedrückt werden. Insbesondere die Variante mit Algorithmus 5.1 ist durch ILP-Modelle kaum zu realisieren und soll deshalb zugunsten des einfacheren Modells aus Gleichung (5.10) nicht weiterverfolgt werden. Selbst wenn die Voraussetzungen für Gleichung (5.10) (siehe Abschnitt 5.3.1) nicht erfüllt sein sollten, so bietet sie doch eine hinreichend genaue Näherung für den Zweck der Optimierung.

7.4.3.1. Auflösen der Nichtlinearitäten

Im Folgenden sollen zunächst die Nichtlinearitäten in den Gleichungen (5.7) und (5.8) durch geeignetes Umformen beseitigt werden. Setzt man Gleichung (5.7) in Gleichung (5.9) ein, so ergibt sich

$$
\begin{aligned}
&L^{rt}\big(R_{kl}, \mathcal{PS}\big) \;= \\
&\quad \tfrac{1}{\overline{N}(R_{kl},\mathcal{PS})} \cdot \sum_{c \in R_{kl}} T^{ZS}(c) \cdot \left[\overline{N}(R_{kl}, \mathcal{PS}) \cdot \mathfrak{J}(c, R_{kl}, \mathcal{PS}) \;+ \sum_{s=\mathfrak{J}(c,R_{kl},\mathcal{PS})}^{n^{max}-1} a_s \right] \\
&\qquad\qquad = \tfrac{1}{\overline{N}(R_{kl},\mathcal{PS})} \cdot \sum_{c \in R_{kl}} \overline{L}_{\mathfrak{J}(c,R_{kl},\mathcal{PS})}(c, \mathcal{PS})\,. \qquad (7.13)
\end{aligned}
$$

In dieser Formulierung wurde $a_{\mathfrak{J}(c,R_{kl},\mathcal{PS})}$ unter Verwendung von Gleichung (5.5) durch das Datenaufkommen $\overline{N}(R_{kl}, \mathcal{PS})$ der Route ersetzt und der entstandene Vorfaktor $1/\overline{N}(R_{kl}, \mathcal{PS})$ vor die Summe verschoben. Da der Vorfaktor zur Durchschnittsbildung dient, verbleibt dann anstelle der mittleren Latenz $L_j^{avg}(c, \mathcal{PS})$

die akkumulierte Gesamtlatenz $\overline{L}_j(c, \mathcal{PS})$ innerhalb der Summe. Die Gleichungen für die Pitches mit dem max-Operator können auf die gleiche Weise umgeformt werden, wobei analog zu $\overline{L}_j(c, \mathcal{PS})$ der akkumulierte Pitch $\overline{P}_j(c, \mathcal{PS})$ definiert werden kann:

$$\overline{L}_j(c, \mathcal{PS}) \;=\; T^{ZS}(c) \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right), \tag{7.14}$$

$$\overline{P}_j(c, \mathcal{PS}) \;=\; P^{res}(c) \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right). \tag{7.15}$$

Diese akkumulierten Größen repräsentieren die Summen aller Latenzen bzw. Pitches für die Speicherzugriffe des Arbiter-Eingangs j während der parallelen Ausführung von $\mathcal{PS}$. Die Gleichungen enthalten keine Nichtlinearität bezüglich a_j mehr, es muss jedoch nach wie vor die Reihenfolge der Indizes j eingehalten werden. Analog zu den Latenzen/Pitches der Einzelkomponenten $c \in R_{kl}$ können mit $\overline{L}^{rt}(R_{kl}, \mathcal{PS})$ und $\overline{P}^{rt}(R_{kl}, \mathcal{PS})$ auch akkumulierte Werte für die Latenz bzw. den Pitch ganzer Routen definiert werden. Dabei muss der Vorfaktor $1/\overline{N}(R_{kl}, \mathcal{PS})$ in die Berechnungsvorschriften für die Verzögerungszeiten aus den Gleichungen (5.11) und (5.12) verschoben werden. Auch hier können mit $\overline{D}^{p}(R_{kl}, \mathcal{PS})$ und $\overline{D}^{np}(R_{kl}, \mathcal{PS})$ wieder akkumulierte Größen gebildet werden, die den Vorfaktor nicht mehr enthalten. Die Verwendung der akkumulierten Größen ist im Kontext der Speicherallokation nicht hinderlich, da letztendlich ohnehin die Summe aller Verzögerungen $\overline{D}(cs_i)$ im Code-Segment cs_i für die Zielfunktion ausschlaggebend ist.

Zur Berechnung von $\overline{D}(cs_i)$ unter Verwendung der akkumulierten Größen $\overline{D}^{p}(R_{kl}, \mathcal{PS})$ und $\overline{D}^{np}(R_{kl}, \mathcal{PS})$ kann Gleichung (5.13) wie folgt umgestellt werden:

$$\begin{aligned} \overline{D}(cs_i) \;=\; \sum_{\forall m_l} \Bigg[& q \cdot \frac{N^{mem,np}(cs_i, m_l)}{\overline{N}\big(R_{kl}, \mathcal{PS}(cs_i)\big)} \cdot \overline{D}^{np}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \\ & + \;\; \frac{N^{mem,p}(cs_i, m_l)}{\overline{N}\big(R_{kl}, \mathcal{PS}(cs_i)\big)} \cdot \overline{D}^{p}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \Bigg] \\ & \text{mit} \quad R_{kl} = \big(\mu^{cs}(cs_i), \ldots, m_l\big). \end{aligned} \tag{7.16}$$

In dieser Gleichung hängen sowohl $N^{mem,np}(cs_i, m_l)$ als auch $\overline{N}(R_{kl}, \mathcal{PS}(cs_i))$ von den Allokationsentscheidungen $\mathcal{I}$ ab. Die Gleichung ist deshalb nur dann

linear, wenn sämtliche Terme mit einer der Variablen I_{pl} aus den beiden Brüchen gekürzt werden können. Im allgemeinen Fall müssen bei Zugriffen vom Typ *non-posted* weitere Terme für die rückwärtsgerichtete Route R_{lk} berücksichtigt werden. Da auf den vorwärts und rückwärtsgerichteten Routen jeweils die gleiche Zahl von Zugriffen abgewickelt wird (Anfragen und die zugehörigen Antworten), unterscheiden sich die Lasten $\overline{N}(R_{lk}, \mathcal{PS}(cs_i))$ und $\overline{N}(R_{kl}, \mathcal{PS}(cs_i))$ nur um einen konstanten Faktor ${}^{n^{np-}}/_{n^{np+}}$, der von dem Datenaufkommen für einzelne Anfragen und Antworten abhängt. Zur Vereinfachung der Notation wird dies in Gleichung (7.16) durch den konstanten Faktor q dargestellt.

Um Gleichung (7.16) weiter zu vereinfachen soll im Folgenden angenommen werden, dass die Anzahl $N^{mem,np}(cs_i, m_l)$ der Zugriffe vom Typ *non-posted* und die Anzahl $N^{mem,p}(cs_i, m_l)$ der *posted Writes* in einem näherungsweise festen Verhältnis zueinander stehen. Mit einer Konstanten $r \in [0, 1]$ können diese Größen dann in Abhängigkeit der Gesamtzahl $N^{mem}(cs_i, m_l)$ aller Zugriffe wie folgt dargestellt werden:

$$\begin{aligned} N^{mem,np}(cs_i, m_l) &\simeq r \cdot N^{mem}(cs_i, m_l)\,, \\ N^{mem,p}(cs_i, m_l) &\simeq (1 - r) \cdot N^{mem}(cs_i, m_l)\,. \end{aligned} \tag{7.17}$$

Die Annahme eines konstanten r stellt deshalb nur eine Näherung dar, weil sich der Anteil der *posted Writes* i. A. von Variable zu Variable unterscheidet. Ferner ist r auch vom Speichersegment abhängig, da *posted Writes* ggf. nicht für alle Speichersegmente $m_l \in \mathcal{M}$ der Zielplattform unterstützt werden. Zum Zweck der Speicherallokation muss r also durch eine möglichst zutreffende Konstante approximiert werden.

Mit der genannten Annahme kann Gleichung (7.16) wie folgt vereinfacht werden:

$$\begin{aligned} \overline{D}(cs_i) \;=\; \sum_{\forall m_l} \frac{N^{mem}(cs_i, m_l)}{\overline{N}\big(R_{kl}, \mathcal{PS}(cs_i)\big)} \cdot \Big[\, & q \cdot r \;\cdot\; \overline{D}^{np}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \\ & +\; (1 - r) \;\cdot\; \overline{D}^{p}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \Big] \\ & \text{mit} \quad R_{kl} = \big(\mu^{cs}(cs_i), \ldots, m_l\big)\,. \end{aligned} \tag{7.18}$$

Um den Bruch in Gleichung (7.18) kürzen zu können, muss zunächst das Datenaufkommen $\overline{N}(R_{kl}, \mathcal{PS})$ entlang der Routen R_{kl} in Abhängigkeit der Allokationsentscheidung $\mathcal{I}$ ermittelt werden. Gemäß Gleichung (5.5) entspricht der Wert $\overline{N}(R_{kl}, \mathcal{PS})$ direkt der Auslastung $a_{\mathfrak{J}(c, R_{kl}, \mathcal{PS})}$ der Arbiter-Eingänge. Mit

der Anzahl n^{p+} bzw. n^{np+} der Dateneinheiten je Zugriff (vgl. Abschnitt 5.3.3) ergibt sich dafür der lineare Zusammenhang

$$\overline{N}(R_{kl}, \mathcal{PS}) \;=\; a_{\mathfrak{J}(c,R_{kl},\mathcal{PS})} \;= \sum_{cs_i \in \Omega_k} \sum_{v_p \in \mathcal{V}} I_{pl} \cdot \left[n^{p+} \cdot N^{var,p}(cs_i, v_p) \;+\; n^{np+} \cdot N^{var,np}(cs_i, v_p) \right]$$
$$\text{mit} \quad \Omega_k = \left\{ cs_i \in \mathcal{PS} : \; \mu^{cs}(cs_i) = c_k \right\}. \quad (7.19)$$

Die innere Summe addiert das Datenaufkommen für Zugriffe auf alle Variablen v_p gewichtet mit der Entscheidungsvariablen I_{pl}, die angibt, ob v_p dem Speicher m_l (der Zielkomponente von R_{kl}) zugewiesen wurde. Wird die Variable einem anderen Speicher zugewiesen, so ist $I_{pl} = 0$ und es entsteht kein Datenaufkommen für R_{kl}. Da das gesuchte Datenaufkommen nur durch Speicherzugriffe des Prozessorkerns c_k (dem Ausgangspunkt der Route R_{kl}) zustande kommen kann, berücksichtigt die äußere Summe nur diejenigen Code-Segmente $cs_i \in \mathcal{PS}$, die auf c_k ausgeführt werden. Das Datenaufkommen der rückwärtsgerichteten Routen R_{lk} vom Speicher zum Prozessorkern kann analog unter Verwendung der für die Antworten anfallenden Dateneinheiten n^{np-} ermittelt werden.

Durch Substitution mit Hilfe der Gleichungen (7.10) und (7.11) kann Gleichung (7.19) auch in Abhängigkeit der Zugriffszahlen $N^{mem,p}$ und $N^{mem,np}$ dargestellt werden:

$$\overline{N}(R_{kl}, \mathcal{PS}) \;=\; a_{\mathfrak{J}(c,R_{kl},\mathcal{PS})} \;= \sum_{cs_i \in \Omega_k} \left[n^{p+} \cdot N^{mem,p}\big(cs_i, m_l\big) \;+\; n^{np+} \cdot N^{mem,np}\big(cs_i, m_l\big) \right]$$
$$\text{mit} \quad \Omega_k = \left\{ cs_i \in \mathcal{PS} : \; \mu^{cs}(cs_i) = c_k \right\}. \quad (7.20)$$

Außerdem ergibt sich ein Sonderfall, wenn die Route R_{kl} mit dem Prozessorkern c_k beginnt, auf dem mit $\mu^{cs}(cs_i) = c_k$ auch das Code-Segment cs_i des betrachteten Ausführungskontexts $\mathcal{PS}(cs_i)$ ausführt wird. In diesem Fall enthält die Menge Ω_k dann mit cs_i nur genau ein Code-Segment (vgl. Definition von $\mathcal{PS}$ in Gleichung 5.4). In Gleichung (7.20) ist diese Voraussetzung per Definition von R_{kl} und $\mathcal{PS}$ erfüllt, sodass die Summe über Ω_k entfallen kann. Darauf aufbauend

lässt sich $\overline{N}(R_{kl}, \mathcal{PS})$ in dem Bruch aus Gleichung (7.20) unter Verwendung von Gleichung (7.17) kürzen und es folgt:

$$\overline{D}(cs_i) \;=\; \sum_{\forall m_l} \left[\frac{q \cdot r \;\cdot\; \overline{D}^{np}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \;+\; (1-r) \;\cdot\; \overline{D}^{p}\big(R_{kl}, \mathcal{PS}(cs_i)\big)}{r \;\cdot\; n^{np+} + (1-r) \;\cdot\; n^{p+}} \right]$$
$$\text{mit} \quad R_{kl} = \big(\mu^{cs}(cs_i), \ldots, m_l\big)\,.$$

Zur weiteren Vereinfachung können die darin verwendeten Konstanten außerdem durch

$$\tau^{np} \;:=\; \frac{q \cdot r}{r \;\cdot\; n^{np+} + (1-r) \;\cdot\; n^{p+}}\,,$$
$$\tau^{p} \;:=\; \frac{1-r}{r \;\cdot\; n^{np+} + (1-r) \;\cdot\; n^{p+}}$$

substituiert werden. Die ILP-kompatible Berechnungsvorschrift für die akkumulierten Zugriffszeiten lässt sich dann schließlich wie folgt formulieren:

$$\overline{D}(cs_i) \;=\; \sum_{\forall m_l} \Big[\tau^{np} \;\cdot\; \overline{D}^{np}\big(R_{kl}, \mathcal{PS}(cs_i)\big) + \tau^{p} \;\cdot\; \overline{D}^{p}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \Big]$$
$$\text{mit} \quad R_{kl} = \big(\mu^{cs}(cs_i), \ldots, m_l\big)\,. \tag{7.21}$$

Die Größen $\overline{D}^{np}$ und $\overline{D}^{p}$ können durch die oben beschriebenen Anpassungen in den Gleichungen (5.11) und (5.12) aus den akkumulierten Pitches $\overline{P}^{rt}$ und Latenzen $\overline{L}^{rt}$ wie folgt berechnet werden:

$$\begin{aligned} \overline{D}^{np}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \;&=\; n^{np+} \cdot \overline{P}^{rt}\big(R_{kl}, \mathcal{PS}\big) \;+\; n^{np-} \cdot \overline{P}^{rt}\big(R_{lk}, \mathcal{PS}\big) \\ &\quad + \overline{L}^{rt}\big(R_{kl}, \mathcal{PS}\big) \;+\; \overline{L}^{rt}\big(R_{lk}, \mathcal{PS}\big)\,, \qquad (7.22) \\ \overline{D}^{p}\big(R_{kl}, \mathcal{PS}(cs_i)\big) \;&=\; n^{p+} \cdot \overline{P}^{rt}\big(R_{kl}, \mathcal{PS}\big)\,. \qquad (7.23) \end{aligned}$$

Der direkten Verwendung der Brechungsvorschrift für $\overline{D}(cs_i)$ aus Gleichung (7.21) in einer ILP-Zielfunktion steht noch das Problem der Reihenfolge der Indizes j in den Gleichungen (7.14) und (7.15) entgegen. Da die Reihenfolge der Indizes sicherstellt, dass $a_s \leq a_j$ für alle $s \geq j$ gilt, kann a_s in diesen Gleichungen mit

$a_s = \min(a_s, a_j)$ substituiert werden. Da außerdem $a_s \geq a_j$ für alle $s < j$ gilt, kann der Term $j \cdot a_j$ in beiden Gleichungen durch

$$j \cdot a_j \;=\; \sum_{s=0}^{j-1} \min(a_s, a_j)$$

ersetzt werden. Insgesamt gilt in den Gleichungen (7.14) und (7.15) daher

$$\begin{aligned} j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \quad &= \quad \sum_{s=0}^{j-1} \min(a_s, a_j) \;+\; \sum_{s=j}^{n^{max}-1} \min(a_s, a_j) \\ &= \quad \sum_{s=0}^{n^{max}-1} \min(a_s, a_j)\,. \end{aligned}$$

Mit Hilfe des min-Operators wird der Term also unabhängig von der Reihenfolge der Indizes j und die Gleichungen lassen wie folgt umformen:

$$\overline{L}_j(c, \mathcal{PS}) \;=\; T^{ZS}(c) \cdot \sum_{s=0}^{n^{max}-1} \min(a_s, a_j)\,, \tag{7.24}$$

$$\overline{P}_j(c, \mathcal{PS}) \;=\; P^{res}(c) \cdot \sum_{s=0}^{n^{max}-1} \min(a_s, a_j)\,. \tag{7.25}$$

Diese Zusammenhänge sind zwar unabhängig von der Reihenfolge der Indizes, enthalten mit dem min-Operator aber nach wie vor eine Nichtlinearität. Auch die Berechnung der Pitches $\hat{P}^{rt}(R_{kl}, \mathcal{PS})$ nach Gleichung (5.10) enthält einen max-Operator und ist damit nichtlinear. Sowohl min- als auch max-Operatoren können in ganzzahligen linearen Optimierungsproblemen jedoch mit Hilfe zusätzlicher ILP-Variablen und Nebenbedingungen realisiert werden.

7.4.3.2. Formulierung als ganzzahliges lineares Optimierungsproblem

Die oben hergeleiteten Gleichungen bilden das in Kapitel 5 vorgestellte Interferenz-Modell fast vollständig in einer ILP-kompatiblen Form ab. Lediglich die Näherung aus Gleichung (7.17) und – in manchen Anwendungsfällen – die Verwendung von Gleichung (5.10) anstelle von Algorithmus 5.1 können kleinere Abweichungen vom exakten Modell verursachen. Im Folgenden sollen die Gleichungen schließlich in die Form eines ILP-Problems, bestehend aus Variablen, Nebenbedingungen und einer Zielfunktion, überführt werden. Das Basisproblem

aus Definition 7.2 bleibt dabei erhalten und wird um weitere Variablen und Nebenbedingungen ergänzt.

Die Allokationsentscheidungen $\mathcal{I}$ fließen in der Berechnungsvorschrift aus Gleichung (7.21) nur noch in die Variablen a_j bzw. $\overline{N}(R_{kl}, \mathcal{PS})$ ein. Für eine vorwärtsgerichtete Route R_{kl} ist der lineare Zusammenhang in Abhängigkeit der Entscheidungsvariablen I_{pl} durch Gleichung (7.19) gegeben. Die Berechnung für rückwärtsgerichtete Routen R_{lk} ist analog und wird daher im Folgenden vernachlässigt. Die von $\mathcal{I}$ abhängigen Variablen a_j gehen in dem hergeleiteten Kostenmodell ausschließlich in die Gleichungen (7.24) und (7.25) ein. Um den darin verwendeten min-Operator in einem ILP-Modell abbilden zu können, werden zusätzliche ILP-Variablen $A_{sj} \in \mathbb{R}$ mit

$$A_{sj} \mathrel{\hat{=}} \min(a_s, a_j)$$

eingeführt. Außerdem wird für jede Variable A_{sj} eine 0/1-Hilfsvariable $\Psi_{sj} \in \{0, 1\}$ sowie eine hinreichend große Konstante $\epsilon \in \mathbb{R}$ definiert. Die als reelle Zahlen definierten Variablen nehmen zwar grundsätzlich ganzzahlige Werte an, die Definition als Element von $\mathbb{R}$ drückt in diesem Fall aber aus, dass sie im ILP-Problem nicht notwendigerweise als ganzzahlige Variablen behandelt werden müssen. Dadurch kann das Problem, falls es der Laufzeit des Lösungsverfahrens zugutekommt, auch mit den Methoden der gemischt-ganzzahligen linearen Optimierung gelöst werden. Die ganzzahligen Werte ergeben sich in der optimalen Lösung dann automatisch aus den vorgegebenen Nebenbedingungen und Konstanten.

Da die Variablen A_{sj} mit positivem Koeffizienten in die Kostenfunktion eingehen, führen die Nebenbedingungen

$$A_{sj} \geq a_s - \epsilon \cdot \Psi_{sj} \quad \text{und} \quad A_{sj} \geq a_j + \epsilon \cdot \Psi_{sj} - \epsilon \tag{7.26}$$

in der optimalen Lösung zu demselben Ergebnis wie der min-Operator. Die Konstante ϵ muss dazu größer als die maximal möglichen Werte für a_s und a_j sein. Ein detaillierterer Beweis dafür, dass diese Nebenbedingungen den min-Operator realisieren, findet sich in Anhang A.2. Bei optimaler Wahl des Variablenwerts gibt Ψ_{sj} dann an, ob a_s ($\Psi_{sj} = 0$) oder a_j ($\Psi_{sj} = 1$) der kleinere der beiden Werte ist.

Für jedes Paar aus zwei Routen R_{kl} und R_{qr}, die über mindestens eine gemeinsame Komponente verlaufen, folgen mit Hilfe von Gleichung (7.19) dann die ausgeschriebenen Nebenbedingungen:

$$\forall \mathcal{PS},\, c \in R_{kl} \cap R_{qr}:$$

$$\begin{aligned} A_{sj} \;\geq\; & \sum_{cs_i \in \Omega_k} \sum_{v_p \in \mathcal{V}} I_{pl} \cdot \left[n^{p+} \cdot N^{var,p}(cs_i, v_p) \;+\; n^{np+} \cdot N^{var,np}(cs_i, v_p) \right] \\ & - \epsilon \cdot \Psi_{sj} \\ A_{sj} \;\geq\; & \sum_{cs_i \in \Omega_q} \sum_{v_p \in \mathcal{V}} I_{pr} \cdot \left[n^{p+} \cdot N^{var,p}(cs_i, v_p) \;+\; n^{np+} \cdot N^{var,np}(cs_i, v_p) \right] \\ & + \epsilon \cdot \Psi_{sj} - \epsilon \end{aligned}$$

mit

$$\begin{aligned} j \;&= \mathfrak{J}(c, R_{kl}, \mathcal{PS})\,, \\ s \;&= \mathfrak{J}(c, R_{qr}, \mathcal{PS})\,, \\ \Omega_k \;&= \left\{ cs_i \in \mathcal{PS} :\; \mu^{cs}(cs_i) = c_k \right\}. \end{aligned} \tag{7.27}$$

Unter Verwendung der Variablen A_{sj} geht aus den Gleichungen (7.24) und (7.25) die Formulierung

$$\overline{L}_j(c, \mathcal{PS}) \;=\; T^{ZS}(c) \cdot \sum_{s=0}^{n^{max}-1} A_{sj}\,, \tag{7.28}$$

$$\overline{P}_j(c, \mathcal{PS}) \;=\; P^{res}(c) \cdot \sum_{s=0}^{n^{max}-1} A_{sj} \tag{7.29}$$

hervor. Die Latenzen werden gemäß Gleichung (5.9) bei der Berechnung der Gesamtlatenz einer Route R_{kl} aufsummiert:

$$\overline{L}^{rt}(R_{kl}, \mathcal{PS}) \;=\; \sum_{c \in R_{kl}} \left\{ T^{ZS}(c) \cdot \sum_{s=0}^{n^{max}-1} A_{sj} \right\}. \tag{7.30}$$

Die Variablen A_{sj} gehen damit linear in die Berechnung der Zugriffsverzögerungen $\overline{D}(cs_i)$ in Gleichung (7.21) ein. Bei den Pitches folgt hingegen aus Gleichung (5.10) der Zusammenhang

$$\overline{P}^{rt}(R_{kl}, \mathcal{PS}) \;=\; \max_{c \in R_{kl}} \left\{ \overline{P}_{\mathfrak{J}(c, R_{kl}, \mathcal{PS})}(c, \mathcal{PS}) \right\}, \tag{7.31}$$

der aufgrund der max-Operation erneut durch zusätzliche Variablen und Nebenbedingungen realisiert werden muss. Dazu wird $\overline{P}^{rt}(R_{kl}, \mathcal{PS}) \in \mathbb{R}$ als zusätzliche ILP-Variable in das Modell aufgenommen. Auch diese Variable geht mit positivem Koeffizienten in die Kostenfunktion ein, sodass die optimale Lösung den Wert minimiert. Mit Hilfe der Nebenbedingungen

$$\overline{P}^{rt}\big(R_{kl}, \mathcal{PS}\big) \;\geq\; \overline{P}_{\mathfrak{J}(c,R_{kl},\mathcal{PS})}(c, \mathcal{PS}) \qquad \forall c \in R_{kl} \tag{7.32}$$

lässt sich verhindern, dass der Variablenwert einen der Pitches $\overline{P}_{\mathfrak{J}(c,R_{kl},\mathcal{PS})}(c, \mathcal{PS})$ der individuellen Komponenten unterschreitet. In der optimalen Lösung realisieren diese Nebenbedingungen damit den max-Operator. In Abhängigkeit von A_{sj} ergeben sich daraus die Nebenbedingungen

$$\overline{P}^{rt}\big(R_{kl}, \mathcal{PS}\big) \;\geq\; P^{res}(c) \cdot \sum_{s=0}^{n^{max}-1} A_{sj} \qquad \forall c \in R_{kl}\,. \tag{7.33}$$

Die ILP-Variablen für $\overline{P}^{rt}(R_{kl}, \mathcal{PS})$ und die linearen Zusammenhänge für die Latenzen $\overline{L}^{rt}(R_{kl}, \mathcal{PS})$ können dann direkt in die Gleichungen (7.22) und (7.23) eingesetzt werden, um ein lineares Modell für die Verzögerungszeiten $\overline{D}(cs_i)$ zu erhalten. Wie bereits bei der Definition der Interferenz-freien Zielfunktion in Gleichung (7.12) kann die abschließende Kostenfunktion $Y^{if}(\mathcal{I})$ für das Interferenzmodell dann durch die Summe aller Speicherzugriffszeiten aus Gleichung (7.21) gebildet werden:

$$Y^{if}(\mathcal{I}) \;:=\; \sum_{cs_i \in \mathcal{CS}} \overline{D}(cs_i)\,. \tag{7.34}$$

Die Tabellen 7.1 und 7.2 geben eine Übersicht über das gesamte ILP-Modell zur Zielfunktion Y^{if}. Tabelle 7.1 listet dabei alle Variablen und Konstanten des Modells auf, während Tabelle 7.2 die vollständig ausformulierte Zielfunktion und die über Definition 7.2 hinausgehenden Nebenbedingungen angibt.

7.4.4. Schätzung des kritischen Pfades

Mit den bisher definierten Kostenfunktionen Y und Y^{if} aus den Gleichungen (7.12) und (7.34) kann die Summe der durch Speicherzugriffe verursachten Verzögerungen im ungünstigsten Ausführungsfall minimiert werden. Die Positionen der Zugriffe im parallelen Programmgraphen (PPG) werden dabei allerdings nicht berücksichtigt. Wenn die WCET des parallelen Programms minimiert

Variablen	
$I_{pl} \in \{0,1\}$	0/1-Variable, die angibt, ob die Variable v_p dem Speicher m_l zugewiesen wird
$A_{sj} \in \mathbb{R}$	Variable zur Darstellung der Terme $A_{sj} \hat{=} \min(a_s, a_j)$
$\Psi_{sj} \in \{0,1\}$	Hilfsvariable, die bei der Bestimmung von $\min(a_s, a_j)$ angibt, ob $a_j < a_s$ gilt
$\overline{P}^{rt}(R_{kl}, \mathcal{PS}) \in \mathbb{R}$	Variable für den Pitch einer Route R_{kl}

Konstanten	
$\lvert v_p \rvert$	Speicherbedarf der Variablen v_p
$\lvert m_l \rvert$	Kapazität der Speicherkomponente m_l
$T^{ZS}(c)$	Dauer eines Zeitschlitzes der RR-Arbitrierung für die Komponente c
$P^{res}(c)$	Basis-Pitch der Komponente c
n^{p+}	Dateneinheiten für *posted Writes* auf der vorwärtsgerichteten Route R_{kl}
n^{np+}	Dateneinheiten für Zugriffe vom Typ *non-posted* auf der vorwärtsgerichteten Route R_{kl}
n^{np-}	Dateneinheiten für Zugriffe vom Typ *non-posted* auf der rückwärtsgerichteten Route R_{lk}
τ^{p}	Gewichtungsfaktor für die Zugriffszeiten der *posted Writes*
τ^{np}	Gewichtungsfaktor für die Zugriffszeiten der Zugriffe vom Typ *non-posted*
$N^{var,p}(cs_i, v_p)$	Anzahl der *posted Writes* im Code-Segment cs_i auf die Variable v_p
$N^{var,np}(cs_i, v_p)$	Anzahl der Zugriffe vom Typ *non-posted* im Code-Segment cs_i auf die Variable v_p
ε	Eine hinreichend große Zahl

Tabelle 7.1.: Variablen und Konstanten des ILP-Modells zur Speicherallokation mit Interferenz-Kosten

ZIELFUNKTION

$$Y^{if} \;=\; \sum_{cs_i \in \mathcal{CS}} \sum_{\forall m_l} \Big[\tau^{np} \cdot \overline{D}^{np}\Big(R_{kl}, \mathcal{PS}(cs_i)\Big) \;+\; \tau^{p} \cdot n^{p+} \cdot \overline{P}^{rt}\Big(R_{kl}, \mathcal{PS}(cs_i)\Big) \Big]$$

mit $\quad R_{kl} = \Big(\mu^{cs}(cs_i), \ldots, m_l\Big)\,,$

$$\overline{D}^{np}\Big(R_{kl}, \mathcal{PS}\Big) \;=\; n^{np+} \cdot \overline{P}^{rt}\Big(R_{kl}, \mathcal{PS}\Big) \;+\; n^{np-} \cdot \overline{P}^{rt}\Big(R_{lk}, \mathcal{PS}\Big) + \overline{L}^{rt}\Big(R_{kl}, \mathcal{PS}\Big) \;+\; \overline{L}^{rt}\Big(R_{lk}, \mathcal{PS}\Big)\,,$$

$$\overline{L}^{rt}(R_{kl}, \mathcal{PS}) \;=\; \sum_{c \in R_{kl}} \left\{ T^{ZS}(c) \cdot \sum_{s=0}^{n^{max}-1} A_{sj} \right\}$$

NEBENBEDINGUNGEN

$\forall \mathcal{PS},\, R_{kl},\, R_{qr},\, c \in R_{kl} \cap R_{qr}$:

$$A_{sj} \;\geq\; \sum_{cs_i \in \Omega_k} \sum_{v_p \in \mathcal{V}} I_{pl} \cdot \Big[n^{p+} \cdot N^{var,p}(cs_i, v_p) + n^{np+} \cdot N^{var,np}(cs_i, v_p) \Big] - \epsilon \cdot \Psi_{sj}\,,$$

$$A_{sj} \;\geq\; \sum_{cs_i \in \Omega_q} \sum_{v_p \in \mathcal{V}} I_{pr} \cdot \Big[n^{p+} \cdot N^{var,p}(cs_i, v_p) + n^{np+} \cdot N^{var,np}(cs_i, v_p) \Big] + \epsilon \cdot \Psi_{sj} - \epsilon$$

mit $\quad j \;=\; \mathfrak{J}(c, R_{kl}, \mathcal{PS})\,,$

$s \;=\; \mathfrak{J}(c, R_{qr}, \mathcal{PS})\,,$

$\Omega_k \;=\; \Big\{ cs_i \in \mathcal{PS} :\; \mu^{cs}(cs_i) = c_k \Big\}$

$\forall c \in R_{kl}$:

$$\overline{P}^{rt}\Big(R_{kl}, \mathcal{PS}\Big) \;\geq\; P^{res}(c) \cdot \sum_{s=0}^{n^{max}-1} A_{s,\mathfrak{J}(c,R_{kl},\mathcal{PS})}$$

Tabelle 7.2.: Zielfunktion und zusätzlich zu Definition 7.2 erforderliche Nebenbedingungen des ILP-Modells mit Interferenz-Kosten

werden soll, so sind jedoch vor allem der kritische Pfad im PPG (d. h. der längste Pfad hinsichtlich der Ausführungszeiten der einzelnen Knoten) und die darauf befindlichen Speicherzugriffe ausschlaggebend. So muss z. B. bei Kontrollflussverzweigungen im ungünstigsten Fall stets von dem langsamsten Kontrollflusszweig ausgegangen werden, während schnellere Zweige nicht in die WCET eingehen. Die Laufzeit von Speicherzugriffen auf schnelleren Zweigen hat damit keinen Einfluss auf die WCET, solange sich der Verlauf des kritischen Pfades durch die Zugriffszeiten nicht verändert. Um dieser Tatsache Rechnung zu tragen, beziehen einige Ansätze zur Allokation von Scratchpad-Speichern, wie z. B. der von Suhendra et al. [106], den kritischen Pfad im Kontrollflussgraphen bei der Speicherallokation mit ein. Hierzu wird die Berechnung des kritischen Pfades in das Optimierungsmodell integriert, um den Einfluss der Allokationsentscheidungen auf den Verlauf dieses Pfades zu modellieren. Dadurch lässt sich z. B. der Fall abdecken, dass ein zunächst kürzerer Kontrollflusszweig durch zusätzliche Verzögerungen bei den Speicherzugriffen zu dem Zweig mit der längsten WCET wird.

Im Folgenden wird der von Suhendra et al. [106] für sequentielle Software vorgeschlagene Ansatz zur Berechnung des kritischen Pfades mittels ILP für Mehrkernprozessoren erweitert. Anstelle des Kontrollflussgraphen eines sequentiellen Programms wird dabei die aPPIR-Darstellung (vgl. Definition 3.1) eines parallelisierten Programms auf Basis des vorgestellten Programmiermodells verwendet. Analog zu dem von Suhendra et al. [106] verwendeten CFG ohne rückwärtsgerichtete Kanten hat die aPPIR den Vorteil, dass sie keine Zyklen enthält. Der kritische Pfad kann dann als der längste Pfad in dem entsprechenden azyklischen Graphen definiert werden.

Wie bei der WCET-Analyse für Mehrkernprozessoren in Abschnitt 3.3.3.3, muss für die Startzeiten $\sigma(cs_i)$ und Endzeiten $\varepsilon(cs_i)$ jedes Code-Segments cs_i der aPPIR gemäß Gleichung (3.1) der folgende Zusammenhang gelten:

$$\sigma(cs_i) \;=\; \max_{cs_j \in adj^-(cs_i)} \Big(\varepsilon(cs_j) \Big) \,.$$

Dabei bezeichnet $adj^-(cs_i)$ die Menge aller Code-Segmente cs_j, die unmittelbare Vorgängerknoten von cs_i sind. Im Rahmen des Optimierungsproblems zur Speicherallokation ist es sinnvoll, die WCET der Code-Segmente cs_i in den durch Speicherzugriffe verursachten Anteil $\overline{D}(cs_i)$ und einen ausschließlich für das Ausführen von Prozessor-Instruktionen anfallenden Anteil $WCET^{comp}(cs_i)$ zu unterteilen. Zwischen den Start- und Endzeiten eines bestimmten Code-Segments besteht dann der Zusammenhang

$$\varepsilon(cs_i) \;=\; \sigma(cs_i) \;+\; WCET^{comp}(cs_i) \;+\; \overline{D}(cs_i) \,. \tag{7.35}$$

In Verbindung mit Gleichung (3.1) ergibt sich daraus

$$\sigma(cs_i) \;=\; \max_{cs_j \in adj^-(cs_i)} \Big(\sigma(cs_j) \;+\; WCET^{comp}(cs_j) \;+\; \overline{D}(cs_j) \Big) . \tag{7.36}$$

Durch rekursives Auswerten dieser Gleichung lässt sich in einem azyklischen Graphen wie der aPPIR die Länge des längsten Pfades zum Knoten cs_i ermitteln. Die WCET des Gesamtprogramms wird dann von dem längsten und damit kritischen Pfad in der gesamten aPPIR bestimmt. Wenn $\mathcal{Z}$ die Menge der Senken der aPPIR, d.h. die Menge aller Knoten ohne ausgehende Kanten, bezeichnet, dann ist die WCET des kritischen Pfades durch

$$WCET^{crit} \;=\; \max_{cs_j \in \mathcal{Z}} \Big(\sigma(cs_j) \;+\; WCET^{comp}(cs_j) \;+\; \overline{D}(cs_j) \Big) \tag{7.37}$$

gegeben.

Die max-Operatoren in den Gleichungen (7.36) und (7.37) können in einem ILP-Modell, analog zu der ILP-Darstellung für die Pitches $\overline{P}^{rt}(R_{kl}, \mathcal{PS})$, durch zusätzliche Nebenbedingungen realisiert werden. Dazu werden $\sigma(cs_i)$ und $WCET^{crit}$ jeweils als zusätzliche ILP-Variablen ($\in \mathbb{R}$) definiert. Da die WCET-Werte sinnvollerweise mit positivem Koeffizienten in die Zielfunktion eingehen, lassen sich die Zusammenhänge dieser Variablen durch die Nebenbedingungen

$$\sigma(cs_i) \;\geq\; \sigma(cs_j) \;+\; WCET^{comp}(cs_j) \;+\; \overline{D}(cs_j) \qquad \forall cs_i \in \mathcal{CS}, \forall cs_j \in adj^-(cs_i) \tag{7.38}$$

und

$$WCET^{crit} \;\geq\; \sigma(cs_j) \;+\; WCET^{comp}(cs_j) \;+\; \overline{D}(cs_j) \qquad \forall cs_j \in \mathcal{Z} \tag{7.39}$$

abbilden.

Mit Hilfe der ILP-Variablen $WCET^{crit}$ lässt sich neben Y und Y^{if} eine weitere mögliche Kostenfunktion $Y^{crit}(\mathcal{I})$ für das Optimierungsproblem aus Definition 7.2 formulieren:

$$Y^{crit}(\mathcal{I}) \;:=\; WCET^{crit} . \tag{7.40}$$

Diese Kostenfunktion minimiert die WCET des parallelen Programms und baut ansonsten direkt auf dem zuvor eingeführten ILP-Modell mit Interferenz-Kosten auf. Entscheidend für einen erfolgreichen Einsatz der Zielfunktion $Y^{crit}(\mathcal{I})$ ist jedoch die Verfügbarkeit von akkuraten Werten für die Konstanten

$WCET^{comp}(cs_j)$, die die WCET einzelner Code-Segmente ohne Speicherzugriffszeiten angeben. Hier muss, ähnlich wie bei der Kommunikationsoptimierung in Kapitel 6, im Allgemeinen auf *a priori* Werte aus einer sequentiellen WCET-Analyse zurückgegriffen werden (siehe auch Abschnitt 4.2).

7.5. Diskussion & Abgrenzung

In diesem Kapitel wurden mehrere ILP-basierte Optimierungsmodelle vorgestellt, die zur effizienteren Speichernutzung in parallelisierten Echtzeitprogrammen beitragen können. Dazu zählen insbesondere ein Ansatz zur Speicherallokation unter Berücksichtigung von Interferenzeffekten in Mehrkernprozessoren und ein vorbereitender Schritt zur Optimierung von Stack-Variablen. Für das Problem der Speicherallokation (siehe Definition 7.2) wurden mit Y aus Gleichung (7.12), Y^{if} aus Gleichung (7.34) und Y^{crit} aus Gleichung (7.40) außerdem drei verschiedene Kostenmodelle aufgestellt, die bei der nachfolgenden Evaluation in Kapitel 8 verglichen werden sollen.

Die vorgestellten Ansätze unterstützen nahezu beliebige heterogene Speicherhierarchien, während das Plattformmodell aus Kapitel 5 die Plattformunabhängigkeit gewährleistet. Im Kontext der parallelisierenden Compiler-Transformation aus Kapitel 4 stellen die vorgestellten Optimierungsmethoden umfassende Lösungen für die Teilschritte der *SHM Variablenauswahl* und der abschließenden *Speicherallokation* bereit. Durch die Formulierung als ganzzahlige lineare Optimierungsprobleme können mit Hilfe geeigneter Werkzeuge außerdem beweisbar optimale Lösungen erzeugt werden (vgl. Abschnitt 2.3.6).

Verglichen mit den in Abschnitt 3.2.1 diskutierten existierenden Ansätzen zur Speicherallokation in Echtzeitprogrammen bildet der vorgestellte Optimierungsansatz als einziger die Kosten für Interferenzeffekte bei Speicherzugriffen in Mehrkernprozessoren ab. Ferner beschränkt sich der vorgestellte Ansatz nicht auf Scratchpad-Speicher als Ersatz für Caches, sondern betrachtet den allgemeineren Fall von heterogenen und verteilten Speicherhierarchien, die mit der herkömmlichen Unterteilung in Scratchpad- und Hauptspeicher i. A. nicht zutreffend beschrieben werden können. Im Gegensatz zum Stand der Technik in der WCET-optimierten Speicherallokation eignet sich der vorgestellte Ansatz außerdem für ein breiteres Spektrum von Hardware-Architekturen und ist dank des generischen Plattformmodells unabhängig von einer spezifischen Zielplattform. Auch bei dem Umfang der Unterstützung für Mehrkernprozessoren, einschließlich der Möglichkeit, die Speichernutzung unterschiedlicher Kerne gegeneinander abzuwägen, gehen die präsentierten Konzepte deutlich weiter als die meisten vorherigen Arbeiten für harte Echtzeitsysteme (vgl. Abschnitt 3.2.1).

Kapitel 8.

Evaluation und Ergebnisse

In diesem Kapitel werden die Beiträge dieser Arbeit, einschließlich des vorgestellten Compiler-Werkzeugs und der Optimierungen für Kommunikationsplatzierung und Speicherallokation, experimentell evaluiert. Die prototypische Implementierung des Werkzeugs wird dazu in die in Abschnitt 3.3.3 beschriebene ARGO-Werkzeugkette zur automatischen Parallelisierung von Echtzeit-Software integriert. Auf diese Weise können sequentielle Testanwendungen in Scilab/Xcos oder C implementiert, durch die Werkzeugkette parallelisiert und die Ergebnisse mit dem ursprünglichen C-Programm verglichen werden. Die daraus resultierenden experimentellen Ergebnisse wurden in Teilen auch in den eigenen Publikationen [RB20a; RB20b; RKB+19] veröffentlicht.

Dieses Kapitel beginnt in Abschnitt 8.1 mit einer Beschreibung der bei den Experimenten verwendeten Zielplattformen, Testapplikationen und Bewertungsmethoden. Darauf folgt eine allgemeine Evaluation der vorgestellten Compiler-Transformation in Abschnitt 8.2. In den Abschnitten 8.3 und 8.4 werden detailliertere Ergebnisse zur Kommunikationsoptimierung, der Nutzung gemeinsam genutzter Variablen und der Speicherallokation vorgestellt. Abschnitt 8.5 schließt das Kapitel daraufhin mit einem Fazit ab.

8.1. Aufbau der Experimente

Beim Einsatz der ARGO-Werkzeugkette wurde der in Abschnitt 3.3.3.1 beschriebene Schritt der Code-Transformationen in den Experimenten dieses Kapitels nicht verwendet. Zugunsten einer besseren Reproduzierbarkeit wurde stattdessen eine Reihe von Testanwendungen verwendet, die auch ohne Code-Transformationen gut zu parallelisieren sind. Dadurch bleiben die präsentierten Ergebnisse unabhängig von den Entscheidungen der Code-Transformationen und eignen sich damit besser zur Bewertung der in dieser Arbeit vorgestellten Optimierungsschritte. Weiterhin wird auf die Durchführung mehrerer Iterationen durch die iterative Optimierungssteuerung verzichtet, da sie vorwiegend

den generierten Schedule beeinflusst, für die Beiträge dieser Arbeit aber eine untergeordnete Rolle spielt.

Die in Kapitel 5 beschriebene Architekturbeschreibungssprache (ADL) wird dazu verwendet, Ergebnisse für verschiedene Konfigurationen der Zielplattform zu gewinnen. Auf diese Weise kann zum einen die Plattformunabhängigkeit der Compiler-Werkzeuge und zum anderen das Plattformmodell an sich demonstriert werden. Im Folgenden sollen die darauf basierenden Plattformkonfigurationen, die verwendeten Testanwendungen sowie die zur Bewertung der Ergebnisse genutzten Methoden und Metriken näher beschrieben werden.

8.1.1. Architektur der Zielplattformen

Grundlage für die experimentelle Evaluation bildet eine Reihe von verschiedenen Konfigurationen einer NoC-basierten Zielplattform, die mit Hilfe der ADL aus Kapitel 5 modelliert wurden. Die verwendeten Plattform-Varianten basieren auf den in [RMB+18; 40; 41] beschriebenen Bausteinen der *InvasIC*-Plattform. Die Bezeichnung *InvasIC* ist abgeleitet von dem Paradigma des *Invasiven Computing* in Mehrkernprozessoren, das es Berechnungen erlaubt, dynamisch auf benachbarte Prozessoren zu expandieren (siehe [110]). Da dieses Paradigma für statische Analysierbarkeit weniger geeignet ist, kommt im Rahmen dieser Arbeit die in [RMB+18] vorgestellte Teilmenge der InvasIC-Plattform mit dem Fokus auf statischer Vorhersagbarkeit zum Einsatz. Auf Software-Seite wird das invasive Programmierparadigma dabei durch das in Kapitel 4 vorgestellte Programmiermodell ersetzt.

Die Kommunikation innerhalb der Evaluationsplattform erfolgt über ein NoC mit Unterstützung für *virtuelle Leitungsvermittlung* (vgl. Abschnitt 2.1.3). Die Eigenschaften und Komponenten des NoC werden in der Arbeit von Heißwolf [40] im Detail beschrieben. Die *Circuits* der virtuellen Leitungsvermittlung werden auch als *Guaranteed Service Channels* (GS-Channel) bezeichnet, da die Hardware nach dem erfolgreichen Aufbau eines Channels eine maximale Latenz und einen Mindestdurchsatz garantiert. Verlaufen mehrere dieser Channels über dieselbe NoC-Verbindung, so wird bei der Arbitrierung das Rundlaufverfahren (RR) eingesetzt. Die Topologie des NoC ist in Abbildung 8.1 dargestellt und besteht aus einem zweidimensionalen Gitter mit *XY-Routing*. Die Größe des NoC ist konfigurierbar und wird in dieser Arbeit entweder auf 2x2 oder 3x3 Kacheln festgelegt.

Die Kacheln in der ersten Reihe verfügen, je nach Konfiguration, über eine Anbindung an einen externen Speicher mit mindestens 1 GB Speicherkapazität. Dieser kann mit Hilfe von GS-Channels auch über das NoC angesprochen

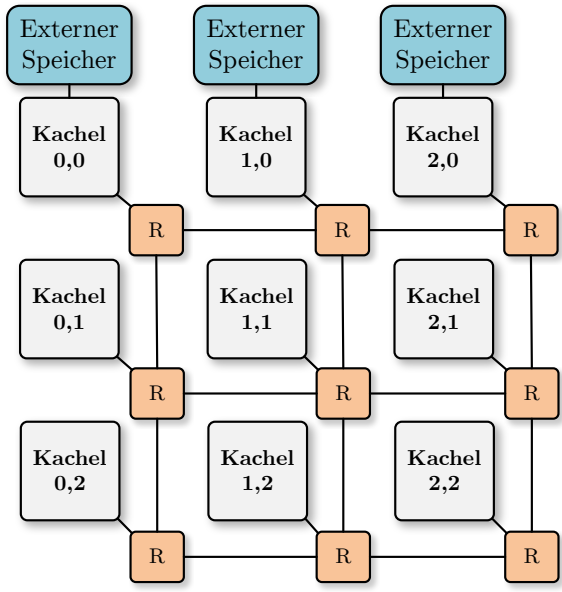
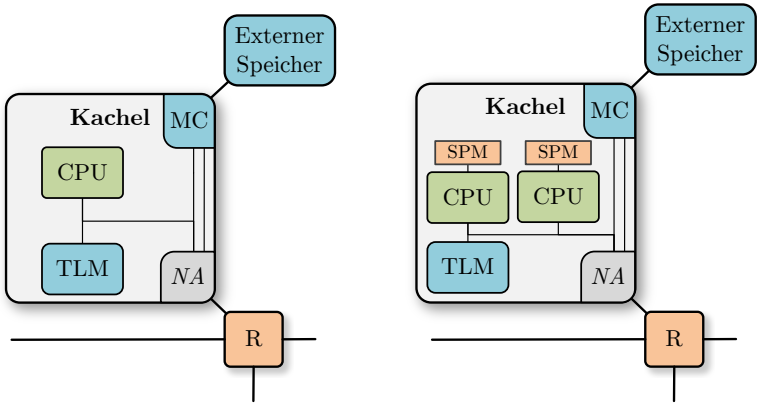


Abbildung 8.1.: Aufbau des NoC der Evaluationsplattform (vgl. [RB20a])

werden. Für die externen Speicher werden im Folgenden zwei verschiedene Konfigurationen verwendet:

1. Ein globaler externer Speicher auf den die Prozessoren aus allen Kacheln Zugriff haben.
2. Je ein externer Speicher für jede Kachel in der ersten Zeile des NoC-Gitters. Die Kerne in den Kacheln haben jeweils Zugriff auf die beiden nächstgelegenen externen Speicher.

Für die interne Architektur der Kacheln werden ebenfalls zwei Varianten (a) und (b) verwendet, die in Abbildung 8.2a bzw. Abbildung 8.2b dargestellt sind. Beide Konfigurationen verwenden Prozessorkerne (CPU) auf Basis der „LEON3“-Architektur (siehe [15]). Diese von der Firma Cobham plc. vertriebene Prozessorarchitektur implementiert den SPARC-V8 Befehlssatz (Kurzform von „Scalable Processor ARChitecture“) und kann mit oder ohne eine dezidierte Hardware-Einheit für Fließkommaberechnungen (englisch *Floating Point Unit*, kurz FPU) instanziiert werden. Neben den Prozessoren verfügt jede Kachel über einen Kachel-internen Speicher, den sogenannten *Tile-Local Memory* (TLM),



(a) Konfig. ohne Scratchpad-Speicher (b) Konfig. mit Scratchpad-Speichern

Abbildung 8.2.: Verwendete Konfigurationen für die Kacheln der Evaluationsplattform (vgl. [RB20a])

mit einer Größe von 512 KiB. Der TLM enthält Instruktionen, den Programm-Stack und weitere Adressbereiche, die von den Programmen frei verwendet werden können. Alle Komponenten einer Kachel sind intern über einen AHB-Bus (*Advanced High-performance Bus*) verbunden, der bei geeigneter Auslegung statisch vorhersagbar ist (siehe [49]). Zur Anbindung an das NoC enthält jede Kachel einen Netzwerk-Adapter (NA), der eine Register-Schnittstelle für den Aufbau und die Nutzung von GS-Channels bereitstellt. Kacheln, die an einen externen Speicher angebunden sind, enthalten zudem einen *Speicher-Controller* (englisch *Memory Controller*, kurz MC), der an den AHB-Bus und über eine gesonderte Verbindung an den lokalen Netzwerk-Adapter angebunden ist. Durch die separate Verbindung können Speicherzugriffe aus dem NoC direkt an den externen Speicher weitergegeben werden, ohne dabei Interferenzen auf dem lokalen AHB-Bus zu erzeugen.

Die beiden Kachel-Typen aus Abbildung 8.2 unterscheiden sich einerseits in der Anzahl der Kerne pro Kachel und andererseits in der Verfügbarkeit von schnellen Scratchpad-Speichern. Die Kacheln der Variante (a) haben nur einen Prozessorkern, der Zugriff auf den lokalen TLM hat. Variante (b) enthält dagegen zwei Prozessorkerne, die zusätzlich zum TLM noch über jeweils einen lokal angebundenen Scratchpad-Speicher (SPM) mit einer Kapazität von 1 KiB verfügen. Durch den Verzicht auf Caches wird auch kein Cache-Kohärenzprotokoll benötigt, um beiden Kernen die kohärente Nutzung der gemeinsamen Daten im

Name	NoC-Größe	Kachelkonfiguration	Anzahl ext. Speicher
2x2a/1	2x2	(a) 1 Kern	1
2x2a/2	2x2	(a) 1 Kern	2
2x2b/1	2x2	(b) 2 Kerne + Scratchpad	1
2x2b/2	2x2	(b) 2 Kerne + Scratchpad	2
3x3a/1	3x3	(a) 1 Kern	1
3x3a/3	3x3	(a) 1 Kern	3

Tabelle 8.1.: Verwendete Konfigurationen der NoC-basierten Zielplattform

TLM zu ermöglichen. Aus demselben Grund werden auch die Daten im externen Speicher in beiden Kachel-Typen nicht in einem Cache vorgehalten. Lediglich für Instruktionen, die sich im Programmverlauf nicht ändern, verfügt jeder Kern in beiden Kachelvarianten über einen L1 Instruktions-Cache, der in Abbildung 8.2 nicht dargestellt ist.

Unter Verwendung der genannten Konfigurationsoptionen kommen bei der nachfolgenden Evaluation die in Tabelle 8.1 aufgelisteten Plattform-Varianten zum Einsatz. Die Konfigurationen werden im Folgenden unter der Bezeichnung aus der ersten Spalte geführt. Alle Prozessorkerne, Kacheln und das NoC werden in sämtlichen Konfigurationen mit einer Taktfrequenz von 100 MHz betrieben. Die Software wird *bare-metal* ohne zusätzliches Laufzeit-Scheduling auf der Plattform ausgeführt. Für Kommunikation und Synchronisation kommt eine vorhersagbare und leichtgewichtige Bibliothek zum Einsatz, die die in Kapitel 4 beschriebenen Bestandteile (FIFO-Kanäle, *Signal-*, *Wait-*, *Send-* und *Receive-*Funktionen) des Programmiermodells implementiert. Die FIFO-Kanäle werden dabei im Wesentlichen auf die GS-Channels des NoC abgebildet. Insgesamt ist die Plattform so konfiguriert, dass das gesamte Datenaufkommen im NoC, einschließlich Kommunikation, Synchronisation und Speicherzugriffe, über die GS-Verbindungen abgewickelt wird.

Die in Tabelle 8.1 aufgelisteten Konfigurationen lassen sich auf Basis existierender Hardware-Blöcke grundsätzlich auf einem *Field Programmable Gate Array* (FPGA) prototypisch realisieren. Der auf diese Weise realisierbaren Plattform-Größe sind jedoch durch die begrenzten Ressourcen der FGPA-Hardware Grenzen gesetzt. Um die Ausführungszeit im durchschnittlichen Fall zu bestimmen, steht deshalb im Folgenden nur eine Plattform-Variante mit 2x2 NoC als FPGA-Prototyp zur Verfügung. Die LEON3-Kerne der FPGA-Implementierung enthalten außerdem keine Hardware-Einheit für Fließkomma-Berechnungen. Da der Schwerpunkt dieser Arbeit auf harten Echtzeitsystemen liegt, ist die Evaluation im durchschnittlichen Fall von nachrangiger Bedeutung. Die für Echtzeitsysteme relevante statische WCET-Analyse ist nicht auf eine vollständige

Applikation	Beschreibung
FFT	Schnelle Fourier-Transformation mit 32 768-Punkten
Canny512	512x512 Pixel Canny-Kantendetektor
Hough512	512x512 Pixel Canny-Kantendetektor gefolgt von einer Hough-Transformation
Hough320	320x320 Pixel Canny-Kantendetektor gefolgt von einer Hough-Transformation
TAWS	*Terrain Awareness and Warning System* für Luftfahrzeuge
k-Means	k-Means Clustering-Algorithmus mit 4096 3D-Vektoren und 4 Mittelwerten

Tabelle 8.2.: Liste der Testanwendungen mit Kurzbeschreibung

Implementierung der Plattform angewiesen und benötigt daher keinen FPGA-Prototypen. Die funktionale Korrektheit der parallelen Software kann bei Bedarf z. B. auch durch die Ausführung der Programme auf gängigen Prozessoren aus dem Endverbraucher-Bereich verifiziert werden.

8.1.2. Parallelisierte Testapplikationen

Zur Evaluation der vorgestellten Compiler-Werkzeuge und Optimierungsansätze kommen die in Tabelle 8.2 aufgelisteten Testanwendungen zum Einsatz. Die Anwendungen basieren auf Implementierungen der folgenden fünf Algorithmen:

Schnelle Fourier-Transformation (FFT): Der FFT-Algorithmus (siehe z. B. [80; 31]) berechnet das komplexwertige Frequenzspektrum eines diskreten Signals (siehe auch Abschnitt 4.2). In vereinfachter Form wurde der Algorithmus bereits in Kapitel 4, Abbildung 4.8 als Beispiel zur Erläuterung des Transformations-Werkzeugs eingeführt. Im Gegensatz zu dieser vereinfachten Variante wird die zur Evaluation verwendete FFT in eine größere Zahl von Teil-Transformationen unterteilt, um die Parallelisierbarkeit zu verbessern. Dazu wird das Eingangssignal zunächst in acht Teile aufgeteilt, für die dann jeweils eine partielle FFT berechnet wird. Die dabei erzeugten Zwischenergebnisse werden dann so lange paarweise zusammengeführt, bis das Gesamtergebnis vorliegt.

Kantendetektor nach Canny: Der *Kantendetektor nach Canny* (siehe [12]) wurde in Kapitel 7, Abbildung 7.2 ebenfalls bereits als Beispiel verwendet. Der Algorithmus wandelt ein Graustufenbild in mehreren Schritten in ein Kantenbild um. Zur Parallelisierung kann das Eingabebild horizontal und/oder vertikal in verschiedene Teilbilder aufgeteilt werden. Diese

Teilbilder lassen sich parallel zueinander in Kantenbilder umwandeln und anschließend wieder zu einem Gesamtbild zusammensetzen.

Hough-Transformation: Die *Hough-Transformation* (siehe [44; 25]) ist ein Verfahren zur Erkennung geometrischer Formen, wie z. B. Kreise oder Geraden, in Graustufenbildern. Das Verfahren kann u. a. auch auf Kantenbilder, wie sie der Kantendetektor nach Canny erzeugt, angewendet werden. Auf diese Weise lässt sich z. B. eine parametrierte Darstellung der Form $d = x \cdot \cos(\alpha) + y \cdot \sin(\alpha)$ für alle geradlinigen Kanten aus dem Kantenbild extrahieren. Dabei stehen d für die Distanz der Geraden zum Koordinaten-Ursprung und α für den Winkel der Normalen zur x-Achse. Das Grundprinzip der Hough-Transformation besteht dann darin, Punkte einer Geraden in dem von x und y aufgespannten Ursprungsraum in den durch α und d aufgespannten Parameterraum zu projizieren. Für den Parameterraum wird eine zweidimensionale Graustufen-Pixelmatrix, auch *Hough-Matrix* genannt, angelegt und in jeder Zelle mit dem Wert 0 initialisiert. Für jedes Pixel (x, y), das im Kantenbild auf einer Kante liegt, werden alle möglichen Geraden $d = x \cdot \cos(\alpha) + y \cdot \sin(\alpha)$ durch den Punkt (x, y) identifiziert und die zugehörigen Koordinaten (α, d) im Parameterraum bestimmt. Für jede dieser Koordinaten (α, d) wird daraufhin die zugehörige Matrix-Zelle in der Hough-Matrix um eins erhöht. Nach dem Abschluss des Verfahrens stehen die lokalen Maxima in der Hough-Matrix dann für die diejenigen Geraden, die im Ursprungsbild die größte Anzahl von Kantenpixeln in sich vereinen. Zur Parallelisierung der Hough-Transformation kann die Hough-Matrix in Teilmatrizen für verschiedene Winkelbereiche unterteilt werden. Für jeden Winkelbereich kann die Evaluation der möglichen Geraden durch die Pixel (x, y) des Ursprungsbildes dann parallel erfolgen.

Terrain Awareness and Warning System: Ein „Terrain Awareness and Warning System“ (TAWS) ist ein elektronisches Sicherheitssystem in der Luftfahrt, das automatisierte Warnsignale für die Piloten eines Luftfahrzeugs bei gefährlicher Annäherung an das Terrain erzeugt. Dazu werden diverse Flug-Parameter, wie z. B. die Sinkrate oder die Flughöhe, überwacht und bewertet. Für Details zu dem verwendeten Algorithmus und dessen Implementierung sei an dieser Stelle auf [RKB+19; 76] verwiesen. Es handelt sich grundsätzlich um eine weitgehend Datenfluss-basierte Anwendung mit zahlreichen Kontrollfluss-Verzweigungen aber vergleichsweise geringem Rechenaufwand. Für die Parallelisierung wurden keine gesonderten Anpassungen vorgenommen, sodass nur Berechnungen, die bereits unabhängig voneinander sind, auf unterschiedlichen Prozessoren ausgeführt werden können.

k-Means-Algorithmus: Bei dem k-Means-Algorithmus handelt es sich um ein Verfahren, das eine Menge von Datenpunkten bzw. Vektoren in eine vorgegebene Anzahl von k Clustern gruppiert (siehe z. B. [86]). Ausgehend von einer anfänglichen Wahl der k namensgebenden Mittelwerte, werden in einem iterativen Verfahren alle Vektoren dem jeweils nächstgelegenen Mittelwert zugeordnet. Anschließend werden die Mittelwerte für die nächste Iteration jeweils anhand der zugeordneten Vektoren neu berechnet. Um die statische Analysierbarkeit sicherzustellen, wird das Verfahren in der hier verwendeten Implementierung nach 100 Iterationen abgebrochen. Zur Parallelisierung des Algorithmus wird die Menge der Vektoren in mehrere Teilmengen unterteilt. Für jede der Teilmengen können dann jeweils eine Zuordnung zu den Mittelwerten sowie neue partielle Mittelwerte parallel auf verschiedenen Kernen berechnet werden. Vor Beginn der nächsten Iteration werden diese Zwischenergebnisse auf einem einzelnen Prozessorkern gesammelt und zu einem Gesamtergebnis vereint.

Die in Tabelle 8.2 aufgeführten Testanwendungen setzen jeweils einen dieser Algorithmen um. Dabei wird die Hough-Transformation stets zusammen mit dem Kantendetektor nach Canny eingesetzt, um ein geeignetes Kantenbild als Eingabe zu erzeugen. Insgesamt decken die Anwendungsfälle mit Kontrollfluss-lastigen Algorithmen wie dem TAWS, verschiedenen Operationen aus dem Bereich der Bildverarbeitung, iterativen Optimierungsverfahren wie dem k-Means-Algorithmus sowie dem schrittweisen Zusammenführen parallel berechneter Teilergebnisse in der FFT ein Spektrum von verschiedenen Anwendungsmustern ab. Auf dieser Basis kann das im Rahmen der vorliegenden Arbeit vorgestellte Compiler-Werkzeug für eine Reihe von verschiedenen Parallelisierungs-Szenarien experimentell bewertet werden.

Für alle Testanwendungen ist eine plattformunabhängige sequentielle Implementierung vorhanden, die mit den in Abschnitt 3.3.3 beschriebenen ARGO-Werkzeugen parallelisiert wird. Alle Schleifen sind so ausgelegt, dass die Schleifengrenzen durch eine automatisierte Analyse der Zwischendarstellung ermittelt werden können. Wie zu Beginn des Kapitels erwähnt, wird der Schritt der Code-Transformation im Folgenden ausgelassen, da die Anwendungen bereits auf gute Parallelisierbarkeit ausgelegt sind. Das ursprüngliche Programm durchläuft dann nur den Scheduling-Schritt, die parallelisierende Transformation, die Quellcode-Erzeugung sowie die Mehrkern-WCET-Analyse (und bei Bedarf das Scilab/Xcos Front-End). Im Scheduling-Schritt wird dabei in allen Fällen eine HEFT-LA Heuristik (vgl. Abschnitt 3.3.3.2 und [RKB+19; ATK+18]) mit geringfügiger Randomisierung als Lösungsverfahren eingesetzt. Bei der Speicherallokation mit dem Ansatz aus Kapitel 7 kommen zu Vergleichszwecken die unterschiedlichen ILP-Kostenfunktionen Y aus Gleichung (7.12), Y^{if} aus Gleichung (7.34) und Y^{crit} aus Gleichung (7.40) zum Einsatz.

Zur Erzeugung von Binärcode für die LEON3-Prozessoren wird in allen Fällen eine Variante der *GNU Compiler Collection* (GCC, siehe z. B. [102]) mit GCC-Basisversion 7.2.0 verwendet. Dabei kommt die Optimierungsstufe 0 (Option „-O0“) zum Einsatz, da viele der gängigen Compiler-Optimierungen zwar für den durchschnittlichen Anwendungsfall vorteilhaft sind, der statischen Analysierbarkeit und der WCET jedoch abträglich sein können (vgl. [27]).

8.1.3. Metriken und Bewertungsmethoden

Das Zielkriterium der in dieser Arbeit vorgestellten Beiträge ist die Minimierung der WCET des parallelisierten Programms (siehe Abschnitt 1.2). Zur Bewertung der Qualität einer Parallelisierung im Hinblick auf dieses Kriterium und den ursprünglichen sequentiellen Quellcode eignet sich daher der WCET-Speedup SP^w. Analog zum allgemeinen Speedup aus Gleichung (2.5) ist letzterer mit der sequentiellen WCET-Grenze $WCET^{seq}$ und der parallelen Entsprechung $WCET^{par}$ durch

$$SP^w := \frac{WCET^{seq}}{WCET^{par}} \tag{8.1}$$

gegeben.

Zur Bestimmung der sequentiellen Grenze $WCET^{seq}$ wird das kommerziell erhältliche Werkzeug aiT (siehe [29]) in der Variante für die LEON3-Architektur eingesetzt. Als Prozessoreinheit wird derjenige Kern der Zielplattform ausgewählt, der die insgesamt schnellste Speicheranbindung an die externen Speicher aufweist. Letzteres trifft typischerweise auf Kerne zu, die sich in einer der Kacheln mit lokalem Speicher-Controller befinden. Die WCET-Analyse in aiT wird dann so konfiguriert, dass sie von festen Speicherzugriffszeiten ohne Interferenzen ausgeht. Diese Zugriffszeiten können mit Hilfe des Plattformmodells unter Annahme des Interferenz-freien Falls bestimmt werden.

Die Ergebnisse für $WCET^{par}$ werden mit Hilfe der Mehrkern-WCET-Analyse aus der ARGO-Werkzeugkette (siehe Abschnitt 3.3.3.3) bestimmt. Da diese ebenfalls auf aiT basiert, ist die Analysemethode auf Kernebene identisch zur Berechnung der sequentiellen WCET, sodass ein direkter Vergleich der resultierenden Werte möglich wird. Bei der WCET-Analyse für Mehrkernprozessoren wird auch die Anzahl der Speicherzugriffe pro Code-Segment mit aiT ermittelt (vgl. Abschnitt 3.3.3.3). Hierbei unterscheidet das Werkzeug gegenwärtig nicht zwischen *posted Writes* und den anderen Zugriffstypen, weshalb im Rahmen der experimentellen Evaluation jeweils vom pessimistischen Fall eines Zugriffs vom Typ *non-posted* ausgegangen wird. Im Hinblick auf die FIFO-basierte Kommunikation wird ferner angenommen, dass Kommunikationsoperationen – abgesehen

von den damit verbundenen Speicherzugriffen – keine weiteren Interferenzen verursachen.

Sowohl bei der sequentiellen als auch der parallelen WCET-Analyse wird zunächst mit Hilfe eines C-Compilers eine Binärdatei mit dem Maschinencode des Programms erzeugt und anschließend analysiert. Allerdings kann eine WCET-Analyse auf Maschinencode-Ebene u. U. nicht alle nötigen Informationen, wie z. B. Schleifengrenzen und Speicheradressen, automatisch ermitteln (vgl. Abschnitt 2.4.1). Die automatisierte Gewinnung dieser Informationen wird z. B. dadurch erschwert, dass die Daten einer lokalen Variablen aus dem ursprünglichen C-Code im Maschinencode in verschiedenen Registern zwischengespeichert, bei Funktionsaufrufen auf dem Stack gesichert und danach wieder in andere Register eingelesen werden können. Dies kann u. a. die Nachverfolgung des Datenflusses für Schleifen-Indizes verhindern, sodass keine zuverlässige Aussage über Schleifengrenzen mehr möglich ist. Im Allgemeinen sind daher zusätzliche Annotationen nötig, die die fehlenden Informationen auf Maschinencode-Ebene ergänzen. Diese müssen oftmals manuell durch den Endnutzer erstellt werden, was typischerweise ein zeitaufwändiges und fehleranfälliges Unterfangen ist.

Da der Quellcode im vorliegenden Fall automatisch generiert wird, können viele der fehlenden Informationen automatisiert aus den Zwischendarstellungen des C-Quellcodes gewonnen werden. Im Quellcode lassen sich z. B. Schleifengrenzen oft deutlich einfacher extrahieren, da dort meist mit strukturierten Schleifenkonstrukten (wie z. B. `for`-Schleifen), eindeutig identifizierbaren Variablen und wohldefinierten Datentypen gearbeitet wird. Durch eine Analyse des Quellcodes lassen sich die Schleifengrenzen zwar weitgehend automatisiert bestimmen, allerdings muss diese Information noch auf den Maschinencode übertragen werden. Ohne weitreichende Kenntnisse bzw. Garantien zum Verhalten des verwendeten C-Compilers, einschließlich der darin durchgeführten Schleifen-Transformationen, ist dies jedoch eine nicht-triviale Aufgabe. Die korrekte und sichere Verknüpfung der Informationen aus Quellcode und Maschinencode ist deshalb auch ein wesentlicher Beweggrund für die in Abschnitt 3.2 beschriebenen Forschungsarbeiten an WCET-optimierten C-Compilern. Da die Umsetzung eines solchen Compilers für die LEON3-Architektur den Rahmen dieser Arbeit sprengen würde, nutzt die nachfolgende Evaluation den oben genannten GCC-basierten Compiler. Durch das Einfügen von Assembler-Sprungmarken (Labels) bei der automatisierten Code-Erzeugung lässt sich das Problem der Annotation von Schleifengrenzen auch für den GCC lösen. Dazu werden Labels in jeder Schleife im Quellcode platziert und mit einer passenden Annotation für die Schleifengrenze versehen. Zumindest auf der niedrigsten Optimierungsstufe erweist sich dieses Verfahren für den eingesetzten Compiler als zuverlässig, da die Schleifen in diesem Fall nicht signifikant transformiert werden.

Neben den Schleifengrenzen können bei der WCET-Analyse auch die Adressbereiche von Speicherzugriffen im Maschinencode u. U. nicht zuverlässig ermittelt werden. Nach dem Schritt der Speicherallokation werden im Quellcode zwar feste Speicheradressen für die Variablen hinterlegt, trotzdem ist es nicht immer möglich, die Adressen wieder aus dem Maschinencode zurückzugewinnen. Das gilt insbesondere für Zugriffe auf mehrdimensionale Felder (Arrays), die im Maschinencode typischerweise durch indirekte Speicherzugriffe realisiert sind. Bei solchen Zugriffen wird die konkrete Adresse erst zur Laufzeit aus dem Array-Index berechnet und kann deshalb bei der statischen Analyse nur dann ermittelt werden, wenn auch der Wertebereich des Index bekannt ist.

Anders als bei den Schleifengrenzen führen fehlende Adressbereiche in einer WCET-Analyse mit aiT nicht zu ungültigen, sondern nur zu stark pessimistischen Ergebnissen. Ist eine Speicheradresse nicht bekannt, so kann von dem pessimistischen Fall des langsamsten Speichersegments ausgegangen werden, sodass die resultierenden WCET-Grenzwerte nach wie vor sicher sind. Zur Evaluation des Speicherallokations-Verfahrens aus Kapitel 7 ist eine solche pessimistische Überschätzung jedoch nicht sinnvoll, da sie die Ergebnisse beträchtlich verfälschen kann. Da das Optimierungsverfahren die Variablen statisch einem Speichersegment zuordnet, sind auf Quellcode-Ebene alle verwendeten Adressbereiche bekannt. Anders als bei den Schleifengrenzen gibt es (ohne einen spezialisierten C-Compiler) für diesen Fall jedoch keine zuverlässige Möglichkeit, die Information automatisiert auf den Maschinencode zu übertragen. Für die nachfolgende Evaluation wird daher auf eine der folgenden beiden Herangehensweisen zurückgegriffen:

1. Es werden keine Adressbereiche annotiert, sodass die Zugriffszeiten ggf. pessimistisch überschätzt und Allokationsentscheidungen durch die WCET-Analyse nicht erkennt werden.
2. Für die Berechnung der Interferenz-Kosten wird die Anzahl der Speicherzugriffe $N^{mem}(cs_i, m_l)$ pro Code-Segment und Speicherkomponente anhand der Variablenzugriffe im Quellcode geschätzt und nicht mit aiT bestimmt. Da C-Compiler typischerweise darauf ausgelegt sind, die Anzahl der Speicherzugriffe im Vergleich zum Quellcode zu reduzieren (z. B. durch das Puffern von Zwischenergebnissen in Registern), ist zu erwarten, dass diese Schätzung insgesamt eher pessimistisch ist. Ohne alle internen Details des Compilers zu berücksichtigen, lässt sich dies jedoch i. A. nicht sicher nachweisen. Die resultierenden WCET-Werte sollten aber dennoch hinreichend akkurat sein, um repräsentative Ergebnisse zur Bewertung der Speicherallokation zu gewinnen.

Methode 2 kommt in erster Linie bei der Evaluation des Verfahrens zur Speicherallokation in Abschnitt 8.4 zum Einsatz. Dazu ist anzumerken, dass die

Anwendbarkeit der evaluierten Optimierungsmethoden auf tatsächliche Echtzeitanwendungen durch die Schätzung nach Methode 2 nicht beeinträchtigt wird. Durch die manuelle Analyse und das händische Erstellen entsprechender Annotationen für die kompilierte Binärdatei lassen sich grundsätzlich auch sichere WCET-Werte bestimmen, die nicht von pessimistischer Überschätzung betroffen sind. Während manuelle Annotationen, z. B. bei der Entwicklung zertifizierbarer Echtzeit-Software (sowohl für Einzelkern- als auch Mehrkernprozessoren), sinnvoll und oft nicht zu vermeiden sind, ist dies für eine experimentelle Evaluation in größerem Stil jedoch kaum praktikabel. Für die Zukunft bergen neuere Entwicklungen im Bereich WCET-optimierter C-Compiler das Potential, den Transfer von Informationen vom Quellcode zum Maschinencode weitgehend zu automatisieren, um manuelle Annotationen vermeiden zu können.

8.2. Bewertung der Compiler-Transformation

In diesem Abschnitt wird die in Kapitel 4 vorgestellte Compiler-Transformation sowohl hinsichtlich des WCET-Speedups als auch der Ausführungszeiten im durchschnittlichen Fall bewertet. Außerdem werden die Ergebnisse mit einem Basisszenario verglichen, in dem die WCET auf Mehrkernprozessoren unter der Annahme von zeitlicher Isolation ohne Interferenz-Analyse ermittelt wird.

8.2.1. Vergleich der WCET mit gemessenen Ausführungszeiten

Für den Vergleich der WCET mit den gemessenen Ausführungszeiten im durchschnittlichen Fall kommt der beschriebene FPGA-Prototyp mit 2x2 NoC und LEON3-Kernen ohne FPU sowie die TAWS-Testanwendung zum Einsatz. Die zugehörigen Ergebnisse für das sequentielle TAWS-Programm auf einem Kern und die parallelisierte Entsprechung sind in Tabelle 8.3 aufgelistet. Durch die Parallelisierung für vier Kerne ohne FPU wird darin ein WCET-Speedup SP^w von 1,77 erreicht.

Die parallelisierende Transformation zielt zwar schwerpunktmäßig auf die Optimierung der WCET ab, erreicht beim TAWS jedoch auch für die gemessene Ausführungszeit einen Speedup von 1,28. Der durchschnittliche Speedup ist niedriger als der WCET-Speedup, was sich neben der WCET-orientierten Optimierungsstrategie auch durch die insgesamt niedrigere Ausführungszeit erklären lässt: Im durchschnittlichen Fall kommt es beim TAWS zu einem schlechteren Verhältnis von Berechnungsaufwand zu Kommunikationsaufwand, da die Anzahl der Kommunikationsoperationen im Gegensatz zur Ausführungszeit konstant bleibt (Es gibt keine Kommunikation innerhalb von Schleifen mit

Szenario	WCET	Gemessene Zeit
Sequentieller Code ohne FPU	7788 μs	672 μs
Paralleler Code ohne FPU (Speedup)	4410 μs (1,77)	524 μs (1,28)
Sequentieller Code mit FPU	484 μs	–

Tabelle 8.3.: WCET und gemessene Ausführungszeiten für den TAWS-Anwendungsfall (vgl. [RKB+19])

variablen Schleifengrenzen). Der Kommunikationsaufwand weicht deshalb im durchschnittlichen Fall weniger stark vom ungünstigsten Ausführungsfall ab.

Ohne Hardware-FPU fügt der C-Compiler verschiedene Software-Routinen für die Fließkommaberechnungen in den Ausgabecode ein, wodurch die Performanz des TAWS deutlich beeinträchtigt wird. Die WCET ist davon typischerweise besonders betroffen, da die eingefügten Software-Routinen verschiedene Schleifen enthalten, deren Grenzen stark von den konkreten Eingabedaten abhängen. Im ungünstigsten Fall sind deshalb i. A. deutlich höhere Schleifengrenzen zu erwarten als bei der Ausführung mit durchschnittlichen Eingabedaten. Um den Unterschied zwischen einer Hardware-FPU und den Software-basierten Fließkommaoperationen quantifizieren zu können, enthält Tabelle 8.3 auch die WCET-Grenze des sequentiellen Codes für einen LEON3-Kern mit aktivierter Hardware-FPU. Die Ergebnisse zeigen, dass sich die sequentielle WCET des TAWS durch die Verwendung der Software-basierten Fließkommaoperationen bereits um mehr als Faktor 16 erhöht. Da die anderen Testanwendungen aus Abschnitt 8.1.2 noch deutlich mehr Fließkommaberechnungen enthalten und der Effekt demnach noch stärker wäre, wird bei diesen Testfällen auf eine Evaluation ohne FPU verzichtet.

Insgesamt zeigen die Ergebnisse für die TAWS-Anwendung, dass die Parallelisierung sowohl für die WCET als auch die Ausführungszeit im durchschnittlichen Fall einen Speedup $SP > 1$ erreichen kann. Für die vier eingesetzten Prozessorkerne ist der erreichte Gewinn zwar relativ niedrig, dies ist jedoch größtenteils der vergleichsweise schlecht parallelisierbaren Testanwendung geschuldet. Unter den in Tabelle 8.2 aufgeführten Anwendungen weist das TAWS insgesamt die mit Abstand geringste Rechenkomplexität auf. Die Verbesserung der WCET um Faktor 1,77 zeigt daher vor allem, dass sich die Parallelisierung selbst in solchen Fällen als gewinnbringend erweisen kann. Über die Effizienz des Verfahrens zur Speicherallokation lassen sich aus den Ergebnissen für das TAWS hingegen kaum Rückschlüsse ziehen, da die lokalen TLMs ausreichen, um alle Daten der Anwendung aufzunehmen.

8.2.2. Vergleich mit dem Fall ohne Interferenz-Analyse

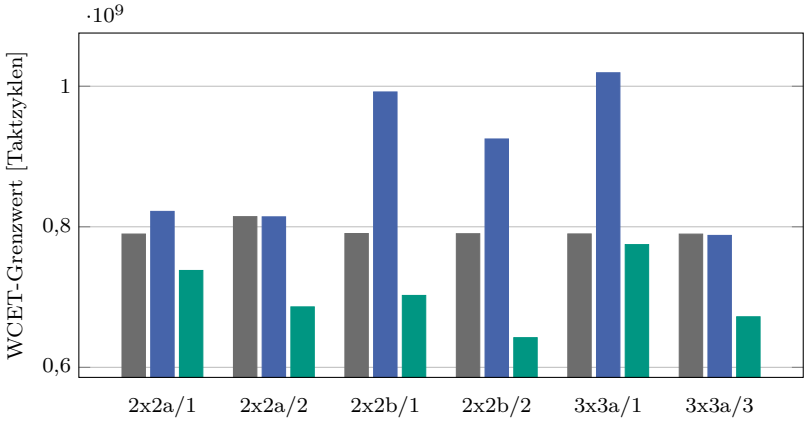
Ein wesentlicher Vorteil durch die Kombination des Plattformmodells aus Kapitel 5 mit dem Programmiermodell aus Kapitel 4 besteht darin, dass Interferenzeffekte präziser eingegrenzt werden können. Um diesen Vorteil quantifizieren zu können, vergleicht die nachfolgende Testreihe den vorgestellten Ansatz mit den WCET-Grenzwerten, die sich ohne die MHP-basierte Interferenz-Analyse ergeben. Hierzu kommen die beiden Testanwendungen *FFT* und *Hough320* zum Einsatz, da sie einen vergleichsweise hohen Bedarf an externem Speicher haben, wodurch auch mit Interferenz-Kosten in signifikantem Umfang zu rechnen ist. Die Bestimmung der Speicherzugriffszahlen erfolgt auf Basis von Informationen aus dem Quellcode (Methode 2 aus Abschnitt 8.1.3). Die Kommunikation zwischen den Prozessoren wird in dieser Testreihe auf Nachrichten beschränkt, sodass keine gemeinsam genutzten Variablen vorhanden sind.

Abbildung 8.3 zeigt die WCET-Grenzwerte, die für diese Applikationen und verschiedene Plattformkonfigurationen (vgl. Tabelle 8.1) durch statische Analysen gewonnen wurden. Die Balken entsprechen dabei jeweils einem der folgenden drei WCET-Werte:

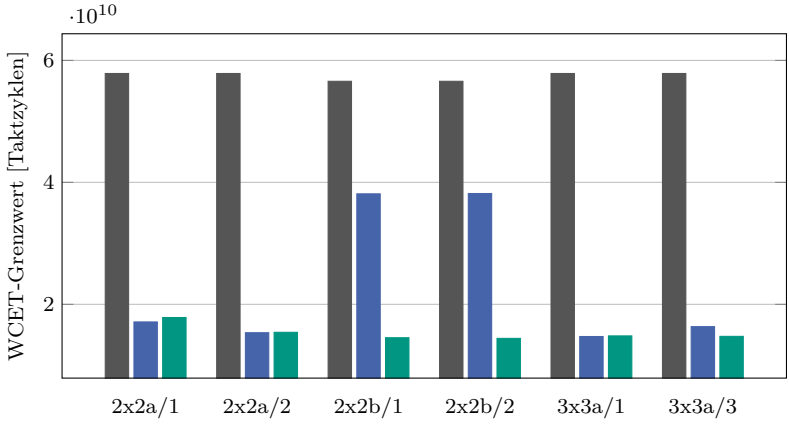
Sequentielle WCET: Die WCET-Grenze des sequentiellen C-Programms vor der Parallelisierung auf einem Kern der jeweiligen Zielplattform.

Parallele WCET ohne Interferenz-Analyse: Die WCET-Grenze des parallelisierten Programms ohne MHP-basierte Interferenz-Analyse. Die Berechnung der Interferenz-Kosten erfolgt unter der Annahme, dass jede vom Programm genutzte Route R_{kl} im Plattform-Komponenten-Graphen (vgl. Abschnitt 5.2.3) permanent ausgelastet ist. Da die kontextabhängige Anzahl $\overline{N}(R_{kl}, \mathcal{PS})$ der Zugriffe pro Route ohne die MHP-Analyse i. A. nicht mehr bekannt ist, werden die belegten Eingänge der Arbitrierungseinheiten pessimistisch durch die Zahl der darüber verlaufenden Routen R_{kl} abgeschätzt. Dieses Vorgehen ist vergleichbar mit Ansätzen, die auf zeitliche Isolation mittels TDMA setzen (vgl. Abschnitt 3.1). Im Gegensatz zu letzteren wird die Struktur des parallelen Codes aber nicht völlig ignoriert, da die im Programm verwendeten Routen R_{kl} nach wie vor berücksichtigt werden. Im Hinblick auf die Speicherallokation kommt die ILP-Kostenfunktion Y aus Gleichung (7.12) zum Einsatz, sodass Interferenzeffekte auch dort nicht mit einbezogen werden.

Parallele WCET mit Interferenz-Analyse: Die parallele WCET-Grenze mit aktiver MHP-Analyse und vollständiger Nutzung des Interferenz-Modells aus Kapitel 5. Die Speicherallokation wird mit der ILP-Kostenfunktion Y^{if} aus Gleichung (7.34) berechnet.



(a) FFT Anwendung



(b) Hough320 Anwendung

Abbildung 8.3.: Sequentielle WCET (▮▮) sowie parallele WCET mit deaktivierter (▮▮) und aktivierter Interferenz-Analyse (▮▮) (vgl. [RB20a])

Abbildung 8.3a zeigt die Ergebnisse für die FFT-Anwendung, in denen die aktivierte Interferenz-Analyse in allen Fällen zu einer deutlich reduzierten und damit präziseren parallelen WCET-Schätzung führt. Bei der *Hough320*-Anwendung in Abbildung 8.3b gibt es hingegen auf den Plattformen mit Kachelvariante (a) bei aktivierter Interferenz-Analyse nur einen geringfügigen Unterschied[1]. Dies kann dadurch erklärt werden, dass die meisten Code-Segmente dort häufig auf die externen Speicher zugreifen müssen. Der Umfang an interferierenden Speicherzugriffen ist dadurch offenbar so hoch, dass die Interferenz-Kosten auch durch die Berücksichtigung der Synchronisationsstruktur nicht signifikant besser eingegrenzt werden können. Auf den Plattformen mit Kachelvariante (b) profitiert jedoch auch die *Hough320*-Anwendung deutlich von der Interferenz-Analyse, da in diesem Fall auch die TLM-Speicher von Interferenzeffekten betroffen sind.

Vergleicht man die Ergebnisse für die parallelen Programme mit den sequentiellen WCET-Werten, so führt die Parallelisierung bei aktiver Interferenz-Analyse in allen Fällen zu einer meist deutlichen Verbesserung der WCET. Insbesondere die rechenintensivere *Hough320*-Applikation profitiert stark von der Parallelisierung und erreicht auf der Plattformkonfiguration 2x2b/2 einen maximalen Speedup von $SP^w = 3{,}92$ auf 8 Kernen. Bei der FFT-Anwendung ist der Gewinn erwartungsgemäß weniger stark, da der Grad der Parallelität beim Zusammenfassen der FFT-Teilergebnisse schrittweise abnimmt. Ohne Interferenz-Analyse ist die Parallelisierung der FFT-Anwendung im Hinblick auf die WCET faktisch wirkungslos und in einigen Fällen sogar kontraproduktiv. Je nach Anwendung und Zielplattform ist die vorgestellte Interferenz-Analyse also zwingend erforderlich, um das Programm überhaupt gewinnbringend parallelisieren zu können. Gegenüber dem herkömmlichen Ansatz der zeitlichen Isolation mittels TDMA zeigt sich damit insgesamt ein klarer Vorteil für die Interferenz-Analyse.

Ein Vergleich zwischen den Plattformkonfigurationen zeigt außerdem, dass die Varianten mit mehr als einem externen Speicher – zumindest bei aktiver Interferenz-Analyse – einen deutlichen Vorteil haben. Besonders bei der FFT machen sich die Reduktion der Interferenzen durch die Verteilung der Speicherzugriffe auf mehrere externe Speicher sowie die entsprechend verkürzten Routen der Zugriffe im NoC deutlich bemerkbar. Die Plattform-Variante 2x2b/2 schneidet bei aktiver Interferenz-Analyse insgesamt am besten ab. Im Vergleich zur Variante 3x3a/3 fehlt dieser Konfiguration zwar ein Prozessorkern und ein externer Speicher, dies wird jedoch durch die Scratchpad-Speicher und die effizientere Kopplung der beiden Prozessorkerne innerhalb der Kacheln mehr als kompensiert. Letzteres kann u. a. durch die reduzierte Last im NoC erklärt werden, die aus der lokalen Kommunikation der beiden Kerne einer Kachel sowie dem Datenaustausch über den TLM bzw. den lokal angebundenen externen

[1] Kleinere Unterschiede können auch durch geringfügige Abweichungen im Schedule, bei der Code-Generierung oder der Neukompilierung des C-Codes entstehen.

Speicher resultiert. Dank der Interferenz-Analyse können die zusätzlich anfallenden Kosten durch Interferenzen bei TLM-Zugriffen dabei so weit eingegrenzt werden, dass sie diesen Vorteilen kaum entgegenstehen.

8.3. Ergebnisse zur SHM- und Kommunikationsoptimierung

In diesem Abschnitt soll mit einer weiteren Testreihe untersucht werden, welchen Einfluss die Verwendung gemeinsam genutzter Speicher (SHM) und die Optimierung der Synchronisationsstruktur im Rahmen der Compiler-Transformation aus Kapitel 4 haben. Der Parallelisierungsprozess wird dazu für die Plattformkonfiguration 2x2a/1 und mehrere Testapplikationen mit den folgenden vier Optimierungseinstellungen ausgeführt:

Ohne Optimierung: Bei der SHM Variablenauswahl werden alle Variablen der Menge $\mathcal{V}^p$ der privaten Variablen (vgl. Abschnitt 4.2.1.1) zugeteilt und die Synchronisationsoptimierung (vgl. Kapitel 6) wird übersprungen. Alle Kommunikations- und Synchronisationsoperationen bleiben damit an den vorläufigen Positionen, die bei der Datenpartitionierung festgelegt werden.

Sync: Die Synchronisationsoptimierung ist aktiv, während ausschließlich privat genutzte Variablen verwendet werden.

SHM: Die SHM Variablenauswahl wählt passende Variablen für die gemeinsame Nutzung aus, wohingegen die Synchronisationsoptimierung entfällt.

SHM & Sync: Es wird sowohl eine Auswahl von SHM-Variablen getroffen als auch die Synchronisationsoptimierung durchgeführt.

Für das TAWS werden außerdem erneut LEON3-Prozessoren ohne FPU angenommen. Abbildung 8.4 zeigt die WCET-Speedups SP^w im Vergleich zum sequentiellen Programm, die sich für die einzelnen Optimierungseinstellungen aus der statischen WCET-Analyse ergeben. Alle gezeigten Ergebnisse für dieselbe Testanwendung verwenden auch denselben Schedule (allerdings nicht denselben wie in Abbildung 8.3, weshalb die Ergebnisse nicht identisch sind). Die Speicherallokation nutzt die Kostenfunktion Y^{if}, während die Speicherzugriffszahlen für die abschließende WCET-Analyse aus der Binärdatei gewonnen werden (Methode 1 aus Abschnitt 8.1.3).

Für den k-Means Algorithmus und das TAWS entfallen die Varianten mit SHM-Nutzung, da die Variablen dieser Anwendungen auch bei aktiver SHM-Optimierung ausschließlich den privaten Speichern zugeordnet werden. Der externe Speicher wird aufgrund seiner höheren Zugriffszeit und der Tatsache, dass alle Daten der Anwendungen in den privaten TLMs Platz finden, in beiden

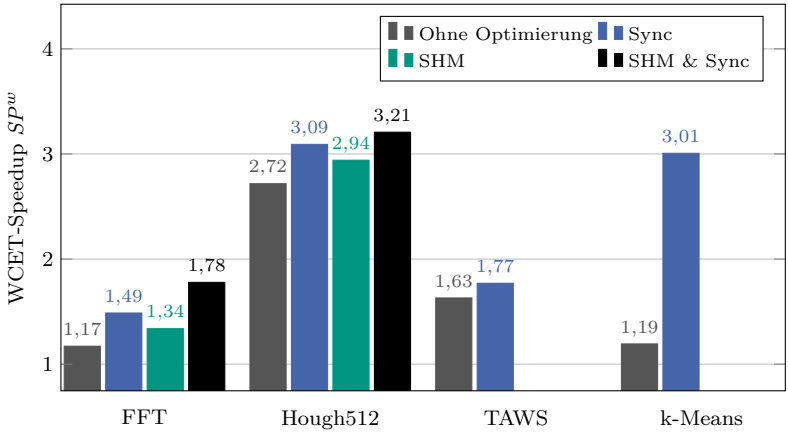


Abbildung 8.4.: WCET-Speedup für verschiedene Testapplikationen und Optimierungsszenarien für die Plattform 2x2a/1 (vgl. [RB20b])

Fällen nicht verwendet. Auch die mögliche Ersparnis bei den Kommunikationskosten durch die gemeinsame Nutzung des externen Speichers kann dies nicht kompensieren.

Die Ergebnisse für die beiden anderen Testanwendungen zeigen deutlich, dass der Einsatz gemeinsam genutzter Variablen den Speedup signifikant verbessert. Etwas stärker profitiert die FFT-Applikation von der Optimierung, da das schlechtere Verhältnis von Berechnungsaufwand zu Kommunikationsaufwand mit einem größeren Verbesserungspotential durch die gemeinsame Nutzung von Variablen einhergeht. Des Weiteren erweist sich die Methode zur Kommunikationsoptimierung aus Kapitel 6 für alle Testanwendungen als vorteilhaft. Vor allem beim k-Means Algorithmus ist die vorläufige Platzierung der Kommunikationsoperationen so ungünstig, dass die Berechnungen ohne diese Optimierung kaum noch parallel ausgeführt werden können. Bei der FFT liefert die Kommunikationsoptimierung ebenfalls einen deutlichen Beitrag zum Speedup, wobei sich hier besonders das Zusammenfassen von Synchronisationsoperationen als effektiv erweist. Durch die Optimierung kann die Anzahl der Synchronisationsoperationen dort von 78 auf 25 reduziert werden.

Insgesamt zeigen die Speedups in Abbildung 8.4 erneut, dass die WCET durch die Parallelisierung in allen Fällen deutlich gesenkt werden kann. Die in dieser Arbeit vorgestellte Optimierungsmethode für Kommunikation sowie die Möglichkeit,

neben Nachrichten-basierter Kommunikation auch gemeinsam genutzte Variablen zu verwenden, leisten einen signifikanten Beitrag zu diesen Ergebnissen.

8.4. Ergebnisse zur heterogenen Speicherverwaltung

In diesem Abschnitt soll das in Kapitel 7 vorgestellte Verfahren zur Speicherallokation unter Berücksichtigung von Interferenzeffekten experimentell evaluiert werden. Das Ziel besteht insbesondere darin, die ILP-Kostenfunktionen Y aus Gleichung (7.12), Y^{if} aus Gleichung (7.34) und Y^{crit} aus Gleichung (7.40) anhand der Problemgrößen und der erzielten WCET-Grenzen für die parallelen Programme zu vergleichen. Dabei werden alle Plattformkonfigurationen aus Tabelle 8.1 sowie die aufgrund ihres Speicherbedarfs am besten geeigneten Testanwendungen *FFT* und *Hough320* berücksichtigt. Um die Verfälschung der Ergebnisse durch unerkannte Speicheradressen bei der WCET-Analyse der Binärdatei zu vermeiden, wird die Anzahl $N^{mem}(cs_i, m_l)$ der Speicherzugriffe nach Methode 2 aus Abschnitt 8.1.3 anhand des Quellcodes ermittelt. Zur Realisierung der Kostenfunktion Y^{crit} werden die Werte $WCET^{comp}(cs_j)$ aus den *a priori* Ergebnissen einer sequentiellen WCET-Analyse geschätzt (vgl. Abschnitt 7.4.4).

8.4.1. Komplexität der ILP-Probleme

Mit der Komplexität der Anwendungen steigt typischerweise auch die Anzahl $|\mathcal{CS}|$ der Code-Segmente und damit die der Mengen $\mathcal{PS}$ stark an. In der Folge wächst auch die Anzahl der zusätzlichen Variablen und Nebenbedingungen des ILP-Modells mit Interferenz-Schätzung und damit letztlich die Komplexität des Optimierungsproblems. Da Algorithmen zur optimalen Lösung eines ILP-Problems i. A. NP-schwer sind, kann für große Probleme u. U. keine optimale Allokation mehr in annehmbarer Zeit gefunden werden. Um dem entgegenzuwirken, wird im Folgenden ein Verfahren verwendet, das die Interferenz-Kosten, falls nötig, nur für eine Auswahl von Ausführungskontexten $\mathcal{PS}$ in das ILP-Modell einbezieht. Dabei dient die Anzahl der Speicherzugriffe in den zugehörigen Code-Segmenten $cs_i \in \mathcal{PS}$ als Grundlage für die Entscheidung, ob die jeweiligen Interferenz-Kosten vernachlässigt werden oder nicht.

Die Auswahl der zu berücksichtigenden Mengen $\mathcal{PS}$ erfolgt nach dem folgenden Schema: Für jedes $\mathcal{PS}$ wird zunächst die maximale Anzahl von Speicherzugriffen $N^{pr}(\mathcal{PS}, \pi_k)$ pro beteiligtem Prozess π_k ermittelt. Jedem $\mathcal{PS}$ wird dann ein Wert

$$\Pi(\mathcal{PS}) \;:=\; \prod_{\pi_k \in \mathcal{P}} N^{pr}(\mathcal{PS}, \pi_k)$$

zugeordnet, der dem Produkt der Zugriffszahlen $N^{pr}(\mathcal{PS}, \pi_k)$ über alle Prozesse entspricht. Diese Größe wird daraufhin als Sortierkriterium verwendet, um eine absteigend nach $\Pi(\mathcal{PS})$ sortierte Liste aller $\mathcal{PS}$ zu erstellen. Dabei bildet $\Pi(\mathcal{PS})$ eine Metrik für die Kosten der potentiell kollidierenden Zugriffe im Ausführungskontext $\mathcal{PS}$. Das ILP-Modell wird dann aufgebaut, indem so lange Interferenz-Modelle für die Elemente $\mathcal{PS}$ am Anfang der Liste hinzugefügt werden, bis die Anzahl der dazu erforderlichen ILP-Variablen A_{sj}, Ψ_{sj} und $\overline{P}^{rt}(R_{kl}, \mathcal{PS})$ einen festgelegten Wert überschreitet. Der letztgenannte Grenzwert wird für jede Plattformkonfiguration und Testanwendung so gewählt, dass die Lösungszeit für das ILP-Problem im Regelfall unter einem Zeitlimit von 2000 s bleibt. Auf diese Weise werden Interferenz-Kosten vor allem dort vernachlässigt, wo ohnehin mit weniger Zugriffskonflikten zu rechnen ist, während sie in speicherintensiven Engpässen nach wie vor berücksichtigt werden. Dadurch entsteht in den weniger wichtigen Teilen des Programms zwar eine Ungenauigkeit, bei hinreichend großen Grenzwerten für die ILP-Variablenzahl halten sich die Unterschiede jedoch in Grenzen.

Die Lösungen der ILP-Probleme werden in der vorliegenden Evaluation mit Hilfe des *Coin-Or Branch-and-Cut* Lösers (siehe [16]) auf einem *Intel Xeon E5-2620 v3 Prozessor* berechnet. Dabei wird die MIP-Formulierung des Allokationsproblems verwendet, in der die Beschränkung auf ganzzahlige Werte bei den Variablen A_{sj} und $\overline{P}^{rt}(R_{kl}, \mathcal{PS})$ entfällt. Das Lösungsverfahren wird abgebrochen, falls bis zum Ablauf des genannten Zeitlimits von 2000 s keine optimale Lösung gefunden werden konnte. In diesem Fall wird die Code-Erzeugung mit der zu diesem Zeitpunkt vorhandenen, potentiell suboptimalen Lösung fortgesetzt.

Einige Statistiken zu den erzeugten ILP-Optimierungsproblemen für die einzelnen Plattformkonfigurationen und Zielfunktionen sind für die FFT-Anwendung in Tabelle 8.4 und für Hough320 in Tabelle 8.5 aufgelistet. Darin bezeichnet T^l die Lösungszeit, V^{ilp} die Anzahl der ILP/MIP-Variablen und C^{ilp} die Anzahl der Nebenbedingungen. Die Spalten mit der Bezeichnung „% d. $\mathcal{PS}$“ geben außerdem an, für welchen Prozentsatz der Mengen $\mathcal{PS}$ die Interfernz-Kosten im ILP-Modell berücksichtigt wurden.

Während die Optimierungsprobleme ohne Interferenzmodell größtenteils in weniger als 1 s gelöst werden konnten, steigen die Lösungszeiten für Y^{if} und Y^{crit} deutlich an. Die Berücksichtigung von Interferenz-Kosten in allen Hardware-Komponenten der Plattform bringt also eine signifikant höhere Problemkomplexität mit sich. Bis auf den Fall der FFT mit Zielfunktion Y^{if} und Plattformkonfiguration 2x2a/1 konnten jedoch für alle Szenarien optimale Lösungen innerhalb des Zeitlimits gefunden werden. Es zeigt sich auch, dass die Lösungszeiten bei der FFT-Anwendung deutlich höher sind als bei Hough320, obwohl die ILP-Probleme in letzterem Fall deutlich mehr Variablen und Nebenbedingungen

FFT								
	Variante Y^{crit}				Variante Y^{if}			
Plattform	T^l [s]	V^{ilp}	C^{ilp}	% d. $\mathcal{PS}$	T^l [s]	V^{ilp}	C^{ilp}	% d. $\mathcal{PS}$
2x2a/1	1469,0	1706	1403	45 %	2039,2	1349	974	30 %
2x2a/2	584,1	2876	3178	26 %	118,9	2049	1220	13 %
2x2b/1	572,4	2731	2727	30 %	6,6	1804	1079	10 %
2x2b/2	466,8	3243	2788	19 %	63,6	3039	2509	19 %
3x3a/1	100,8	2329	2846	42 %	1,4	1543	1160	21 %
3x3a/3	4,0	2861	2743	16 %	1553,3	2646	2448	16 %

FFT			
	Variante Y		
Plattform	T^l [s]	V^{ilp}	C^{ilp}
2x2a/1	0,3	1015	640
2x2a/2	0,9	1549	720
2x2b/1	0,4	1468	743
2x2b/2	0,9	1959	801
3x3a/1	0,4	1041	658
3x3a/3	4,0	1587	745

Tabelle 8.4.: Statistiken zu den ILP-Problemen für die FFT-Anwendung mit der Anzahl der Variablen V^{ilp}, der Nebenbedingungen C^{ilp} und den Lösungszeiten T^l (vgl. [RB20a])

enthalten. Eine höhere Anzahl an Variablen/Nebenbedingungen hat also nicht notwendigerweise längere Lösungszeiten zur Folge.

Durch die Variation des Anteils der Mengen $\mathcal{PS}$, für die die Interfezenz-Kosten berücksichtigt werden, kann der Kompromiss zwischen der Genauigkeit des ILP-Modells und den benötigten Lösungszeiten prinzipiell flexibel eingestellt werden. Da auch die statische WCET-Analyse bei komplexeren Anwendungen einige Zeit in Anspruch nehmen kann, sind etwas höhere Lösungszeiten meist akzeptabel, insbesondere wenn die Interferenz-Kosten im erzeugten parallelen Programm dadurch signifikant reduziert werden können.

8.4.2. Vergleich der Optimierungsvarianten

Für die Lösungen der in den Tabellen 8.4 und 8.5 aufgeführten Optimierungsprobleme werden im Folgenden die durch Interferenz verursachten Verzögerungen sowie die Auswirkungen auf die letztendlichen WCET-Grenzen der parallelen Programme untersucht. Die entsprechenden Resultate für die FFT sind in Abbildung 8.5 und die für die Hought320-Anwendung in Abbildung 8.6 dargestellt. Für die Experimente mit derselben Plattformvariante kommt dabei derselbe

Hough320								
	Variante Y^{crit}				Variante Y^{if}			
Plattform	T^l [s]	V^{ilp}	C^{ilp}	% d. $\mathcal{PS}$	T^l [s]	V^{ilp}	C^{ilp}	% d. $\mathcal{PS}$
2x2a/1	3,2	2781	4973	100 %	0,5	2642	4782	100 %
2x2a/2	51,3	3868	6592	42 %	3,5	3742	6423	42 %
2x2b/1	21,6	4864	8109	43 %	1,2	3957	6641	15 %
2x2b/2	471,9	5145	8310	8 %	1,7	4992	8098	8 %
3x3a/1	823,2	3881	6653	63 %	23,1	3111	5057	29 %
3x3a/3	30,7	5182	8904	27 %	69,6	4996	8651	27 %

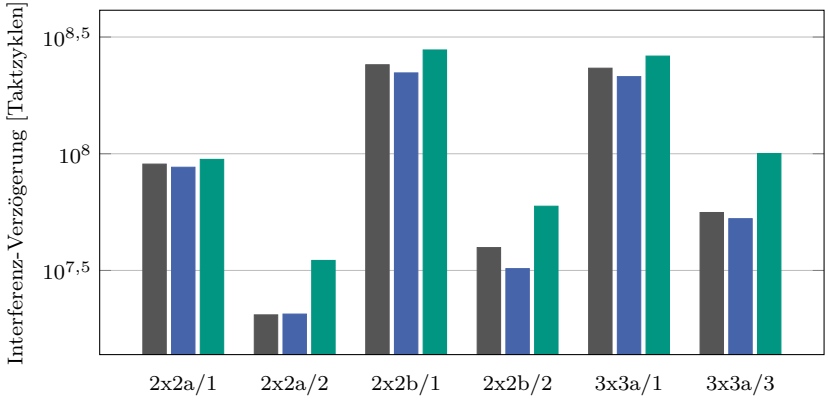
Hough320			
	Variante Y		
Plattform	T^l [s]	V^{ilp}	C^{ilp}
2x2a/1	0,2	1753	2901
2x2a/2	0,3	2626	4200
2x2b/1	0,3	3623	6307
2x2b/2	0,7	4492	7598
3x3a/1	0,2	2607	4553
3x3a/3	0,5	3913	6672

Tabelle 8.5.: Statistiken zu den ILP-Problemen für die Hough320-Anwendung mit der Anzahl der Variablen V^{ilp}, der Nebenbedingungen C^{ilp} und den Lösungszeiten T^l (vgl. [RB20a])

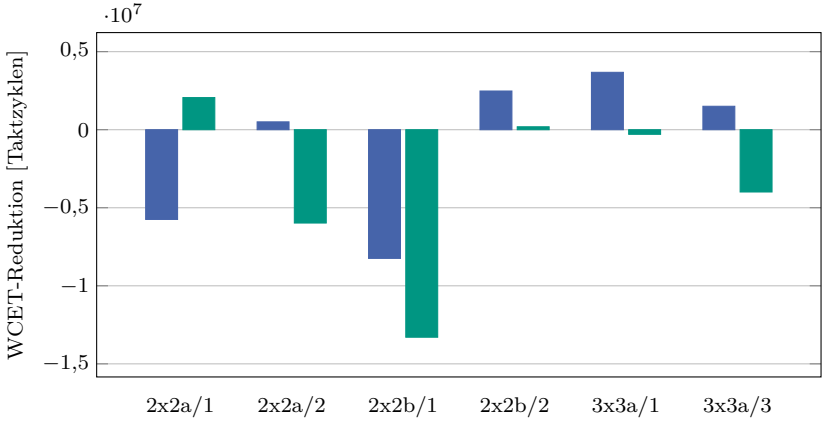
Schedule zum Einsatz, während bei der Speicherallokation verschiedene Lösungen mit je einer der Kostenfunktionen Y, Y^{if} und Y^{crit} berechnet und in einem parallelen Programm umgesetzt werden. Die Auswahl gemeinsam genutzter Variablen ist dabei nicht aktiv, um zu verhindern, dass die Freiheitsgrade im Allokationsproblem durch die Vorentscheidung über SHM-Variablen einschränkt werden.

Die Balkendiagramme in den Abbildungen 8.5a und 8.6a zeigen die Summen aller durch Interferenz verursachten Verzögerungen $IF(cs_i)$ (vgl. Abschnitt 3.3.3.3) für alle Speicherzugriffe im gesamten Programm (jeweils logarithmisch aufgetragen). Die Problemvariante Y^{if} zeigt in allen Testszenarien eine deutliche Reduktion der Interferenz-Kosten im Vergleich zu Y. Für die Hough320-Anwendung zeigt sich eine besonders starke Verbesserung bei den Plattformkonfigurationen 2x2a/1 und 3x3a/3, wo eine Reduktion um 49 % bzw. 41 % erreicht wird.

Verglichen mit den anderen Varianten führen die Lösungen des ILP-Problems mit Y^{crit} in den meisten Fällen zu den höchsten Interferenz-Verzögerungen. Dies ist insofern nicht verwunderlich, als Interferenz-Kosten für alle Speicherzugriffe abseits des kritischen Pfades nicht in die Gesamtkosten Y^{crit} eingehen. Im Gegensatz zu Y und Y^{if}, in denen alle Zugriffe das gleiche Gewicht haben, toleriert

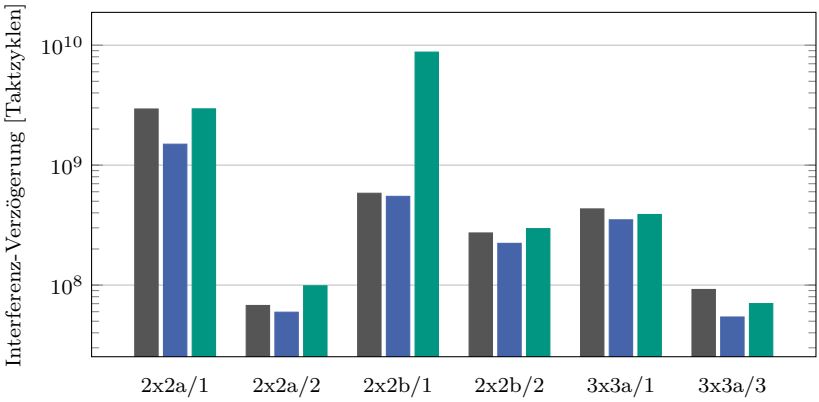


(a) Summe der Interferenz-Verzögerungen über alle Code-Segmente (logarithmisch)

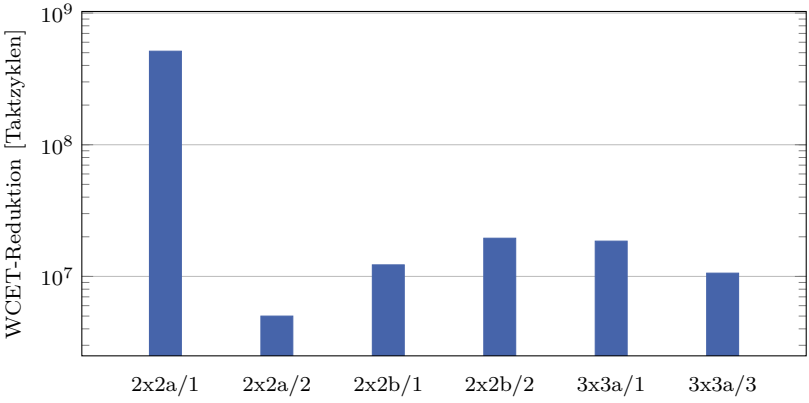


(b) WCET-Reduktion im Vergleich zur Optimierungsvariante Y

Abbildung 8.5.: Vergleich der erzeugten Speicherallokationen in den Optimierungsvarianten Y (■■), Y^{if} (■■) und Y^{crit} (■■) für die FFT-Anwendung auf verschiedenen Zielplattformen (vgl. [RB20a])



(a) Summe der Interferenz-Verzögerungen über alle Code-Segmente (logarithmisch)



(b) WCET-Reduktion im Vergleich zur Optimierungsvariante Y (logarithmisch)

Abbildung 8.6.: Vergleich der erzeugten Speicherallokationen in den Optimierungsvarianten Y (■), Y^{if} (■) und Y^{crit} (■) für die Hough320-Anwendung auf verschiedenen Zielplattformen (vgl. [RB20a])

die Variante Y^{crit} daher auch höhere Interferenz-Kosten, solange der geschätzte kritische Pfad nicht beeinflusst wird. Die in den Abbildungen dargestellte Summe aller Interferenz-Verzögerungen geht hingegen nur für Y^{if} direkt in die Gesamtkosten ein. Erwartungsgemäß liefert diese Kostenfunktion deshalb auch in allen Fällen die niedrigsten Interferenz-Verzögerungen.

Der Vergleich der verschiedenen Plattform-Varianten zeigt, dass die Interferenz-Verzögerungen (stärker noch als die WCET in Abbildung 8.3) signifikant sinken, wenn die Plattform mit mehr als einem externen Speicher ausgestattet ist. Die Plattform 2x2a/2 hat, im Vergleich zu anderen Konfigurationen mit mehreren externen Speichern, weniger Kerne und damit auch weniger Potential für kollidierende Zugriffe. Aus diesem Grund sind dort auch die insgesamt niedrigsten Interferenz-Verzögerungen zu beobachten. Kein klarer Trend zeigt sich dagegen beim Vergleich der Interferenz-Kosten für die Plattformen 2x2b/1 und 3x3a/1 bzw. 2x2b/2 und 3x3a/3. Bei der FFT-Anwendung ist die Kachelvariante (b) im Vorteil, während bei Hough320 das größere 3x3 NoC mit geringeren Interferenz-Kosten einhergeht. Dies lässt sich dadurch erklären, dass die FFT-Anwendung eine größere Zahl von vergleichsweise kleinen Variablen enthält, sodass die Scratchpad-Speicher im Kachel-Typ (b) effizienter genutzt werden können.

Die Diagramme zu den Interferenz-Verzögerungen werden mit den Abbildungen 8.5b und 8.6b um eine Darstellung der WCET-Reduktion bei Verwendung der Kostenfunktionen Y^{if} und Y^{crit} im Vergleich zur Kostenfunktion Y ergänzt. Die Balken stehen dort jeweils für die Differenz zwischen den durch statische Analysen ermittelten WCET-Werten für Y und den entsprechenden WCET-Werten für Y^{if} bzw. Y^{crit}. Grundsätzlich wäre hier zu erwarten, dass Y^{crit} die stärkste Reduktion der WCET bewirkt, da eine Schätzung ebendieser WCET hier als Kostenfunktion dient. Für die FFT und Y^{crit} weisen die tatsächlichen Ergebnisse in Abbildung 8.5b jedoch vorwiegend eine erhöhte (d. h. eine negative Reduktion) oder bestenfalls marginal verbesserte WCET auf. Für die Hough320-Anwendung ergeben sich ähnliche Resultate, die aber zugunsten einer logarithmischen Darstellung für Y^{if} in Abbildung 8.6b nicht dargestellt sind. Eine mögliche Erklärung für das schlechte Abschneiden von Y^{crit} ist die Verwendung von relativ ungenauen *a priori* Werten für $WCET^{comp}(cs_j)$ bei der Schätzung des kritischen Pfads. Durch diverse Veränderungen des Programmcodes beim Durchlaufen der S2S-Transformation und des C-Compilers kann es zu deutlichen Abweichungen dieser *a priori* Werte von der letztendlichen WCET im parallelen Programm kommen. Hierbei können diverse Details, wie beispielsweise die Nutzung der Caches und Prozessor-Pipelines, eine Rolle spielen. Diese Aspekte lassen sich im Vorfeld kaum vorhersagen und können bereits bei kleinen Modifikationen im Quellcode signifikante Änderungen am Zeitverhalten bewirken.

Im Gegensatz zu Y^{crit} wird die WCET durch die Speicherallokation mit Y^{if} in den meisten Fällen deutlich reduziert. Für die FFT auf den Plattformen 2x2a/1 und 2x2b/1 konnte jedoch keine Verbesserung erzielt werden. In ersterem Fall kann das darauf zurückgeführt werden, dass die ILP-Optimierung dort das Zeitlimit überschreitet und die Code-Erzeugung deshalb mit einer suboptimalen Lösung fortgesetzt wird. In zweiterem Fall werden mit 10 % nur für einen verhältnismäßig kleinen Teil der Mengen $\mathcal{PS}$ die Kosten für Interferenzen im ILP-Modell berücksichtigt. Hinzu kommt, dass es mit lediglich einem externen Speicher in der Konfiguration 2x2b/1 für die Allokation größerer Variablen praktisch keine alternativen Speicher gibt, die zur weiteren Senkung der Interferenz-Kosten genutzt werden könnten. Im Fall der Hough320-Anwendung führt die Interferenz-optimierte Speicherallokation hingegen in allen Fällen zu signifikant verbesserten WCET-Werten. Dabei sticht insbesondere das Ergebnis für die Plattformkonfiguration 2x2a/1 mit einer um eine Größenordnung höheren Reduktion hervor. Dieses Szenario ist das einzige, in dem Interferenzeffekte für 100 % der Mengen $\mathcal{PS}$ im ILP-Modell berücksichtigt wurden. Die deutlich reduzierten WCET-Ergebnisse dürften nicht zuletzt auch darauf zurückzuführen sein.

Insgesamt zeigen die Ergebnisse für die verschiedenen Strategien zur Speicherallokation, dass die Kostenfunktion Y^{if} in den meisten der evaluierten Szenarien sowohl bei den Interferenz-Verzögerungen als auch der WCET des Gesamtprogramms zu den besten Ergebnissen führt. Die Modellierung von Interferenzeffekten bei der Speicherallokation erweist sich vor allem dann als effektiv, wenn 100 % der Mengen $\mathcal{PS}$ einbezogen werden. Besonders bei größeren Mehrkernprozessoren kann letzteres allerdings zu erheblich höheren Lösungszeiten für das ILP-Problem führen. Zumindest für die größeren Plattformkonfigurationen und die verwendete MIP-Bibliothek sind dann bereits die Grenzen der Skalierbarkeit bei annehmbaren Lösungszeiten erreicht. Dies zeigt sich z. B. an der Lösungszeit von 1553,3 s für die FFT und Plattform-Variante 3x3a/3, bei der Interferenz-Kosten für lediglich 16 % der Code-Segmente berücksichtigt wurden. Mit der Möglichkeit, diesen Prozentsatz flexibel anzupassen, lässt sich der Ansatz aber grundsätzlich auch auf größere Plattformen anwenden, indem ein entsprechender Kompromiss zwischen Lösungszeit und Qualität der Speicherallokation eingestellt wird. Für die Kostenfunktion Y konnte in allen Fällen eine optimale Lösung innerhalb weniger Sekunden gefunden werden, sodass zumindest das Interferenz-freie Basismodell (mit Interferenzmodellen für 0 % der Code-Segmente) noch weiter nach oben skaliert werden kann. Eine zusätzliche Verbesserung der Lösungszeit lässt sich ggf. auch durch die Verwendung effizienterer ILP-Löser oder durch heuristische Verfahren zur Berechnung näherungsweise optimaler Lösungen erreichen.

Auch wenn das Zielkriterium der WCET durch die Schätzung des kritischen Pfades im ILP-Modell direkt modelliert werden kann, erweist sich die Verwendung der zugehörigen Kostenfunktion Y^{crit} bei der experimentellen Evaluation nicht als vorteilhaft. Die zugrundeliegenden *a priori* Schätzungen der WCET-Beiträge einzelner Programmteile sind offenbar nicht hinreichend präzise um den tatsächlichen kritischen Pfad korrekt abzuschätzen. Dieses Problem besteht grundsätzlich auch in einigen der in Abschnitt 3.2.1 genannten Allokationsmethoden für Scratchpad-Speicher. In diesem Zusammenhang sieht die Lösung von Wan et al. in [114] z. B. vor, nicht nur einen kritischen Pfad, sondern die k längsten Pfade des CFG zu berücksichtigen. Ein ähnliches Vorgehen könnte auch für das vorgestellte Optimierungsmodell zu besseren Ergebnissen führen. Bei zukünftigen Erweiterungen in diese Richtungen bliebe dann zu untersuchen, wie diese Methode im Vergleich zu der bisherigen Problemvariante mit Y^{if} als Kostenfunktion abschneidet.

8.5. Fazit

Die vorgestellten Ergebnisse zeigen, dass automatische Parallelisierung für harte Echtzeit-Software bei Verwendung einer geeigneten Interferenz-Analyse zu einer deutlichen Verbesserung der WCET im Vergleich zum sequentiellen Fall beitragen kann. Auch Applikationen wie das TAWS, die eher kontrollflusslastig sind und einen niedrigen Bedarf an Rechenleistung aufweisen, konnten auf einer Plattformkonfiguration mit vier Kernen, sowohl hinsichtlich der WCET als auch der gemessenen Ausführungszeit, von der Parallelisierung profitieren. Für die FFT- und Hough320-Testanwendungen konnte ferner gezeigt werden, dass die WCET durch die Analyse von Interferenzeffekten mit Hilfe des Plattformmodells aus Kapitel 5 deutlich präziser eingegrenzt werden kann als unter der Annahme von zeitlicher Isolation. Die Ergebnisse demonstrieren damit die erfolgreiche Anwendung der Compiler-Transformation und des Programmiermodells aus Kapitel 4.

Durch die Evaluation mit sechs verschiedenen Varianten der Zielplattform konnte die Plattformunabhängigkeit des Ansatzes zur automatischen Parallelisierung demonstriert werden. Ferner zeigen die Ergebnisse für die verschiedenen Speicherhierarchien, dass Architekturen mit mehreren verteilten Speichern tendenziell bessere WCET-Ergebnisse erzielen. Sowohl die Verwendung von Scratchpad-Speichern als auch der Einsatz von mehr als einem externen Speicher erweisen sich erwartungsgemäß als geeignet, die WCET paralleler Programme in Verbindung mit der Methode zur Speicherallokation aus Kapitel 7 signifikant zu verbessern.

Sowohl für die Optimierung von Kommunikation & Synchronisation als auch die Möglichkeit zur Verwendung gemeinsam genutzter Variablen als Ergänzung zur Nachrichten-basierten Kommunikation konnte eine Verbesserung der WCET gezeigt werden. Bei den letztgenannten gemeinsam genutzten Variablen gilt dies zumindest für alle Testanwendungen, deren Speicherbedarf die Kapazitäten der TLMs in der Zielplattform übersteigt. Die optimierte Kommunikationsplatzierung trägt dagegen in allen getesteten Szenarien zu einer Verbesserung der WCET-Grenzwerte für die parallelisierten Anwendungen bei.

Weniger eindeutig sind die Ergebnisse bei der Speicherallokation, wo die Schätzung des kritischen Pfades im Mittel nicht zu einer Verbesserung im Vergleich zu den übrigen Kostenfunktionen führt. Ungeachtet dessen erweist es sich als vorteilhaft, Interferenzeffekte in den Optimierungsmodellen zur Speicherallokation einzubeziehen. Das dabei verwendete ILP-Modell bildet die Interferenz-Kosten nach dem Plattformmodell in Kapitel 5 bis auf kleinere Näherungen vollständig ab und kann zur Bestimmung einer optimalen Allokation (in Bezug auf die verwendete Zielfunktion) genutzt werden. Die Lösungszeiten solcher ILP-Probleme steigen mit zunehmender Modell-Komplexität typischerweise überproportional stark an, sodass die Skalierbarkeit für die Suche nach einer beweisbar optimalen Lösungen begrenzt ist. Insbesondere bei größeren Zielplattformen und Testapplikationen kann dem entgegengewirkt werden, indem Interferenzeffekte nur für ausgewählte Programmteile einbezogen werden. In der vorgestellten Evaluation konnte mit dieser Methode für die meisten Fälle eine optimale Lösung des vereinfachten Optimierungsproblems gefunden werden.

Insgesamt konnte mit den vorgestellten Ergebnissen der erfolgreiche Einsatz der im Rahmen dieser Arbeit entstandenen Methoden und Software-Werkzeuge demonstriert werden. Ein quantitativer Vergleich mit dem Stand der Technik gestaltet sich jedoch als schwierig, da kaum vergleichbare Lösungen für die Parallelisierung allgemeiner C-Programme und die Interferenz-Analyse auf Basis der Synchronisationsstruktur existieren. Insbesondere ist auch der Vergleich mit Ansätzen, die auf die Minimierung der durchschnittlichen Ausführungszeit abzielen (vgl. Abschnitt 3.3.1), aufgrund der unterschiedlichen Optimierungsziele wenig sinnvoll.

Im Hinblick auf die Parallelisierung allgemeiner Programme (siehe Abschnitt 3.3.2.1) wurden für die teilautomatisierten Ansätze auf Basis von Entwurfsmustern [33; 113; 103] WCET-Speedups für verschiedene Testszenarien publiziert. So erreichen Frieb et al. in [33] für eine Anwendung aus dem Bereich der Kransteuerung einen WCET-Speedup von ca. 2,4 auf vier Kernen. Auf acht Prozessorkerne konnte diese Anwendung jedoch nicht mehr erfolgreich skaliert werden. Weiterhin wurde von Ungerer et al. in [113] ein WCET-Speedup von

bis zu 2,6 auf acht Kernen für einen Algorithmus zur 3D-Pfadplanung veröffentlicht. Durch die Anwendung der Entwurfsmuster-basierten Parallelisierung auf AUTOSAR-Anwendungen konnten Stegmeier et al. in [103] anhand einer Management-Software für Dieselmotoren außerdem einen Speedup von bis zu 3,5 auf vier Kernen zeigen. Die Autoren verwenden jedoch WCET-Approximationen auf Basis von Messungen, die sich nicht direkt mit den Ergebnissen einer statischen WCET-Analyse vergleichen lassen.

Unter den Ansätzen zur Parallelisierung azyklischer Graphen in Abschnitt 3.3.2.1 ist die Arbeit von Didier et al. [24] am ehesten mit dem ARGO-Ansatz und den in dieser Arbeit vorgestellten Beiträgen vergleichbar. Auf vier Kernen erreichen die Autoren hier zumindest bei einer der evaluierten Applikationen einen Speedup von etwas mehr als 3,0. Details zu den Testanwendungen werden in der Arbeit jedoch aus Vertraulichkeitsgründen nicht genannt, sodass eine genauere Einschätzung der Ergebnisse kaum möglich ist.

Aufgrund der verschiedenartigen Testanwendungen, Zielplattformen und Herangehensweisen bei Parallelisierung und WCET-Analyse ist ein aussagekräftiger Vergleich der experimentellen Ergebnisse aus den genannten existierenden Arbeiten mit dem hier vorgestellten Ansatz kaum möglich. Daher sei an dieser Stelle lediglich erwähnt, dass sich der in dieser Arbeit erreichte WCET-Speedup von 3,21 auf vier Prozessorkernen (für die Hough512-Applikation) durchaus in ähnlichen Größenordnungen wie die besten Ergebnisse der existierenden Arbeiten bewegt. Im Gegensatz zu den letztgenannten, ermöglichen die Beiträge dieser Arbeit aber auch die Parallelisierung iterativer Algorithmen, wie z. B. des k-Means Algorithmus, in Verbindung mit einem hohen Automatisierungsgrad.

Kapitel 9.

Schlussfolgerung und Ausblick

Dieses Kapitel fasst die Ergebnisse der vorliegenden Arbeit und die daraus resultierenden Schlussfolgerungen zusammen. Im Anschluss folgt ein Ausblick auf mögliche weiterführende Arbeiten und zukünftige Erweiterungen der Beiträge dieser Arbeit.

9.1. Zusammenfassung und Schlussfolgerungen

Im Rahmen dieser Arbeit wurden wesentliche Beiträge zu einer Werkzeugkette für die automatische Parallelisierung harter Echtzeitanwendungen vorgestellt, prototypisch implementiert und evaluiert. Durch ein spezialisiertes Programmiermodell können die darauf aufbauenden parallelen Programme einer MHP-Analyse unterzogen werden, wodurch sich Interferenz-Kosten anhand der ohnehin nötigen Kommunikation und Synchronisation effizient eingrenzen lassen. In der vorgestellten experimentellen Evaluation konnten anhand zweier Testanwendungen und verschiedener Konfigurationen der Zielplattform signifikante Vorteile dieses Vorgehens gegenüber einem Vergleichsfall mit einer pessimistischeren Interferenz-Schätzung nachgewiesen werden. Aufgrund seiner Verwendung als Zieldarstellung für die automatisierte Code-Generierung ist das Programmiermodell vorwiegend auf gute statische Vorhersagbarkeit und weniger auf Hardware-Abstraktion oder den Komfort bei der manuellen Programmierung ausgelegt. In Kombination mit der automatischen Parallelisierung kann dies zu weniger pessimistischen WCET-Grenzen beitragen, während der Endanwender aufgrund der Entwurfsautomatisierung keine weiteren Nachteile in Kauf nehmen muss.

Nicht nur das Programmiermodell, sondern auch das im Rahmen dieser Arbeit entwickelte Modell für echtzeitfähige Mehrkernprozessoren bildet eine wesentliche Voraussetzung für die präzise und plattformunabhängige Interferenz-Analyse. Außerdem können mit dem generischen Plattformmodell und der zugehörigen

ADL-Spezifikation alle nötigen Informationen für die automatische Parallelisierung strukturiert dargestellt werden. Auf diese Weise wird es möglich, neben Interferenz-Analysen auch ganze Werkzeugketten zur Entwurfsautomatisierung für parallele Echtzeit-Software plattformunabhängig aufzubauen.

Neben dem Programmier- und Plattformmodell befasst sich der größte Teil der Beiträge dieser Arbeit mit der Generierung paralleler Programme im Zuge der automatischen Software-Parallelisierung. In diesem Zusammenhang wurde eine mehrstufige S2S-Transformation, die auf statische Vorhersagbarkeit und die Unterstützung des vorgestellten Programmiermodells ausgelegt ist, entwickelt und prototypisch umgesetzt. Diese Transformation stellt Eigenschaften wie die Abwesenheit von *Deadlocks* oder die Nachverfolgbarkeit von Schleifengrenzen sicher und ermöglicht ein hybrides Kommunikationsmodell aus gemeinsam genutzten Speichern und Nachrichten-basierter Kommunikation. Sie ist so aufgebaut, dass jeder Zwischenschritt eine funktional korrekte Zwischendarstellungen erzeugt und das Phase-Ordering Problem durch die mehrstufigen Entscheidungsschritte abgemildert wird. Bei der Speicherallokation wird Letzteres z. B. durch die Aufteilung in eine frühe Vorentscheidung über gemeinsam genutzte bzw. private Variablen und die erst später getroffenen endgültigen Allokationsentscheidungen erreicht.

Darüber hinaus wurden im Rahmen der S2S-Transformation neue Lösungsmethoden für die Optimierungsprobleme der Kommunikationsoptimierung und der Interferenz-optimierten Speicherallokation vorgeschlagen. Bei ersteren konnte die optimale Platzierung einer oder mehrerer Kommunikations- und Synchronisationsoperationen innerhalb eines CFG auf das bekannte Standard-Problem des minimalen s-t-Schnitts (vgl. Abschnitt 2.3.6.2) zurückgeführt werden. Darauf aufbauend können redundante Operationen mit Hilfe eines Clustering-Verfahrens zusammengefasst werden, um unnötigen Kommunikationsaufwand einzusparen. Für das Optimierungsproblem der Speicherallokation wurde eine neue ILP-Formulierung vorgeschlagen, die die Bestimmung von optimalen Lösungen mit Standard-Lösungsverfahren ermöglicht und dabei – im Gegensatz zu herkömmlichen Methoden – auch Interferenzeffekte berücksichtigt. In der experimentellen Evaluation wurde für beide Optimierungen gezeigt, dass die statisch bestimmte WCET des parallelen Programms von den vorgestellten Methoden signifikant profitieren kann. Lediglich die Schätzung des kritischen Pfades bei der Speicherallokation erweist sich mit den verwendeten *a priori* Informationen als zu ungenau, um die Ergebnisse verbessern zu können.

Im Hinblick auf die eingangs definierten Ziele (vgl. Abschnitt 1.2) konnten im Rahmen dieser Arbeit adäquate Lösungen und deutliche Fortschritte in allen angeführten Themenschwerpunkten erreicht werden:

1. Die Erzeugung paralleler Programme aus sequentiellen Echtzeitprogrammen und die statische Analysierbarkeit des Ausgabecodes konnte erfolgreich demonstriert werden. Der Entwicklungsaufwand ist dabei vergleichbar mit dem Fall der herkömmlichen Software-Entwicklung für Einzelkernprozessoren.
2. Die in dieser Arbeit vorgestellten Methoden, Modelle und Compiler-Werkzeuge unterstützen auch iterative Programme und beschränken sich damit nicht auf Anwendungen, die durch azyklische Graphen dargestellt werden können. In Verbindung mit einem existierenden HTG-basierten Scheduling-Verfahren konnte dies in Kapitel 8 erfolgreich demonstriert werden.
3. Die vorgestellten Optimierungsverfahren konnten erfolgreich auf verschiedene Testanwendungen und Zielplattformen angewendet werden. Mit wenigen Ausnahmen, wie der Schätzung des kritischen Pfads bei der Speicherallokation, konnten die WCET-Grenzen der parallelisierten Programme dadurch signifikant verbessert werden.
4. Unter Verwendung des vorgestellten Plattformmodells und der zugehörigen erweiterten ADL konnten sowohl die Beiträge dieser Arbeit als auch die in [RKB+19] und Abschnitt 3.3.3 beschriebene ARGO-Werkzeugkette plattformunabhängig aufgebaut werden.
5. Mit dem parallelen Programmiermodell, das die effiziente Berechnung von MHP-Informationen erlaubt, wurde in dieser Arbeit eine wesentliche Voraussetzung für die statische Interferenz-Analyse unter Verwendung der Synchronisationsstruktur geschaffen. Zudem stellen die in Kapitel 5 beschriebenen Modelle für Speicherzugriffszeiten in Mehrkernprozessoren einen entscheidenden Baustein für die Realisierung einer präzisen und plattformunabhängigen Interferenz-Analyse bereit. Dadurch wird es möglich, Interferenzeffekte anhand der Synchronisationsstruktur des Programms effektiv einzugrenzen, ohne dabei zeitliche Isolation voraussetzen zu müssen.

Betrachtet man diese Arbeit vor dem Hintergrund des vorherigen Standes der Technik aus Kapitel 3, so kann sie zusammen mit den ARGO-Werkzeugen (vgl. Abschnitt 3.3.3) wesentlich zur Schließung der eingangs erwähnten Lücke bei Ansätzen zur hochautomatisierten Parallelisierung allgemeiner Echtzeitprogramme beitragen. Anders als in bisherigen Arbeiten sind dabei weder umfangreiche manuelle Arbeiten bei der Parallelisierung der Software noch die Annahme von zeitlicher Isolation oder azyklischen Anwendungsmodellen erforderlich. Weiterhin wurde die Bandbreite an unterstützten Zielplattformen durch die Verwendung eines generischen Plattformmodells bei der automatischen Parallelisierung von

Echtzeit-Software deutlich erweitert. Im Bereich der WCET-optimierten Speicherverwaltung wurde außerdem erstmals ein Interferenz-Modell bei der Optimierung der Speicherallokation in heterogenen Speicherhierarchien verwendet. Durch die Schaffung der Voraussetzungen für eine Interferenz-Analyse auf Basis der Synchronisationsstruktur eröffnet diese Arbeit zudem die Möglichkeit auf Zeit-basierte Interferenz-Analysen, die z. B. auf *zeitgesteuerte Schedules* zurückgreifen, zu verzichten. Dies kann insbesondere beim Aufbau gemischt-kritischer Systeme von Vorteil sein (vgl. Abschnitt 3.1.1).

In ihrer Gesamtheit bieten die Beiträge dieser Arbeit damit ein umfassendes Fundament zum Aufbau komplexer Werkzeugketten zur Automatisierung des Entwurfs von paralleler Software unter harten Echtzeitanforderungen. Durch die erfolgreiche Integration dieser Beiträge in die ARGO-Werkzeugkette konnte bereits eine Umfassende und plattformunabhängige Lösung demonstriert werden. Diese lässt sich durch weitere vorgeschaltete Schritte nicht nur für Eingabeprogramme in der Programmiersprache C, sondern z. B. auch für den *modellbasierten* Entwurf paralleler Echtzeit-Software einsetzen (siehe z. B. [RKB+19]). Im Kontext solcher ganzheitlichen Ansätze können die Ergebnisse dieser Arbeit wesentlich dazu beitragen, die effiziente und kostengünstige Nutzung von Mehrkernprozessoren in Zukunft auch im Bereich harter Echtzeitsysteme voranzutreiben.

9.2. Ausblick

Während in dieser Arbeit schon einige Konzepte und prototypische Lösungen im Umfeld der Parallelisierung harter Echtzeitanwendungen vorgestellt wurden, gibt es an verschiedenen Stellen noch Potential für weiterführende Arbeiten. So legen die vorgestellten experimentellen Ergebnisse z. B. bei der Speicherallokation zwei direkte Erweiterungen des vorgestellten Ansatzes nahe:

1. Die Verbesserung der Vorhersage des kritischen Pfades im Optimierungsmodell.
2. Heuristische Lösungsverfahren für das Problem der Interferenz-optimierten Speicherallokation.

In ersterem Fall könnte eine mögliche Lösung darin bestehen, die in der ARGO-Werkzeugkette vorgesehene iterative Optimierungssteuerung (vgl. Abschnitt 3.3.3.3) zu nutzen. Dadurch können die Ergebnisse der Mehrkern-WCET-Analyse in einer zweiten Iteration des Optimierungsschritts zur Speicherallokation berücksichtigt werden, um eine präzisere Vorhersage des kritischen Pfades zu berechnen. Aufgrund der Lösungszeiten bei der Suche nach einer optimalen Lösung des ILP-Problems sind in einem solchen iterativen Ansatz jedoch eher

heuristische Lösungsverfahren sinnvoll. Eine alternative Lösungsmöglichkeit bietet der bereits in der Literatur verwendete Ansatz, die k längsten Pfade anstelle eines einzelnen kritischen Pfades bei der Optimierung zu berücksichtigen.

Zukünftige Erweiterungen zur heuristischen Lösung des Problems der Speicherallokation in Gegenwart von Interferenzeffekten könnten z. B. auf einer Reihe von bekannten Meta-Heuristiken aufbauen. Die in dieser Arbeit entwickelte formale Beschreibung als ILP-Problem bietet dabei den Vorteil, dass die Ergebnisse der Heuristiken mit den durch ILP gewonnenen optimalen Lösungen verglichen werden können. Auf diese Weise lässt sich bei der Evaluation einer Heuristik genau quantifizieren, wie nahe deren Lösung an das tatsächliche Optimum herankommt. Darüber hinaus gibt es auch heuristische Ansätze, die direkt auf der ILP-Formulierung eines Optimierungsproblems aufbauen und eine approximative Lösung bestimmen können. Ein solcher Ansatz wurde z. B. in [21] zur Lösung von ILP-basierten Scheduling-Problemen mit Hilfe von evolutionären Algorithmen verfolgt.

Auch bei der Kommunikationsoptimierung gibt es Möglichkeiten für zukünftige Erweiterungen. Der in dieser Arbeit vorgestellte Ansatz verfolgt bisher das Ziel, die Anzahl der Kommunikations- und Synchronisationsoperationen so weit wie möglich zu reduzieren. Dass dies grundsätzlich sinnvoll ist, zeigt sich an dem deutlich verringerten Kommunikationsaufwand in der Evaluation aus Kapitel 8. Da Synchronisationsoperationen jedoch auch zur Eingrenzung von Interferenzeffekten genutzt werden, können die Interferenz-Kosten durch das gezielte Hinzufügen redundanter Synchronisation ggf. noch weiter gesenkt werden. Zur Auswahl geeigneter Positionen für die entsprechenden Synchronisationsoperationen bietet sich der in dieser Arbeit vorgestellte Ansatz auf Basis des minimalen Kantenschnitts an. Da solche redundanten Operationen für die funktionale Korrektheit des Programms nicht benötigt werden, ist für deren Positionierung lediglich die Abwägung zwischen den Synchronisationskosten und den zu erwartenden Interferenzen ausschlaggebend.

Ausgehend von der bisherigen prototypischen Umsetzung der vorgestellten Konzepte gilt es abschließend noch die Frage nach einem möglichen Transfer der Forschungsergebnisse in die Anwendung zu erörtern. Gerade bei sicherheitskritischen Hardware-/Software-Systemen sind oft einige Hürden zu nehmen, bevor neue Technologien wie Mehrkernprozessoren oder die automatische Parallelisierung in der Anwendung etabliert werden können. In diesem Zusammenhang sind insbesondere auch die Anforderungen für die Qualifizierung von Software-Werkzeugen nach den einschlägigen Sicherheitsstandards wie DO-330 (siehe z. B. [118]) in der Luftfahrt zu nennen. Vor diesem Hintergrund stellen die Ergebnisse dieser Arbeit in Verbindung mit einer Reihe von anderen Vorarbeiten zunächst nur eine erste Machbarkeitsstudie dar. Der Transfer der Konzepte in

die tatsächliche Anwendung erfordert dagegen – sowohl in der Forschung als auch der kommerziellen Produktentwicklung – noch erhebliche weiterführende Arbeiten. Die Umsetzung entsprechender parallelisierender Compiler für harte Echtzeit-Software in einem kommerziellen Produkt ist also noch keineswegs abgeschlossen.

Anhang A.

Ergänzende Beweise und Herleitungen

A.1. Herleitung der durchschnittlichen Nutzung von Arbiter-Eingängen

Im Folgenden soll die Beziehung

$$\hat{n}_j^{aktiv}(c, \mathcal{PS}) \quad = \quad \frac{1}{a_j} \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right)$$

aus Gleichung (5.6) im Detail hergeleitet werden. Betrachtet man die in Abbildung 5.5 bzw. Abschnitt 5.3.1 dargelegte Situation für den zu untersuchenden Arbiter-Eingang j, so gibt es für $a_{n^{max}-1}$ der a_j Dateneinheiten im ungünstigsten Fall $n^{aktiv} = n^{max}$ gleichzeitig aktive Eingänge. Analog zu dem in Abschnitt 5.3.1 angeführten Beispiel gilt im nächsten Schritt für $a_{n^{max}-2} - a_{n^{max}-1}$ der Dateneinheiten $n^{aktiv} = n^{max} - 1$ (Zumindest solange das Datenaufkommen an den Arbiter-Eingängen unterhalb von a_j bleibt). Führt man dieses Schema fort und bildet den Durchschnitt $\hat{n}_j^{aktiv}(c, \mathcal{PS})$ über die Werte n^{aktiv} für alle a_j Dateneinheiten, dann folgt:

$$\hat{n}_j^{aktiv}(c, \mathcal{PS}) \; = \; \frac{1}{a_j} \cdot \left(n^{max} \cdot a_{n^{max}-1} + \sum_{s=j+1}^{n^{max}-1} s \cdot (a_{s-1} \; - \; a_s) \right) . \qquad \text{(A.1)}$$

Die Mittelwertbildung startet dabei erst bei $s = j + 1$, da die Arbiter-Eingänge $s \leq j$ aufgrund der Reihenfolge der Indizes ein gleiches oder größeres Datenaufkommen als j haben und damit stets Interferenzen verursachen. Die Anteile für $s \leq j$ gehen deshalb als Konstante ein und müssen in der Summe nicht gesondert betrachtet werden. Außerdem wird der Summand $n^{max} \cdot a_{n^{max}-1}$ für die Dateneinheiten mit der höchsten Zahl aktiver Arbiter-Eingänge separat aufaddiert.

Durch das Umstellen der Summe in Gleichung (A.1), lässt sich die Beziehung wie folgt umformen:

$$\hat{n}_j^{aktiv}(c, \mathcal{PS}) \;=\; \frac{1}{a_j} \cdot \left(n^{max} \cdot a_{n^{max}-1} + \sum_{s=j+1}^{n^{max}-1} s \cdot a_{s-1} - \sum_{s=j+1}^{n^{max}-1} s \cdot a_s \right) .$$

Durch Indexverschiebung in der ersten Summe ergibt sich daraus:

$$\begin{aligned}
&\hat{n}_j^{aktiv}(c, \mathcal{PS}) \\
&\quad= \frac{1}{a_j} \cdot \left(n^{max} \cdot a_{n^{max}-1} + \sum_{s=j}^{n^{max}-2} (s+1) \cdot a_s - \sum_{s=j+1}^{n^{max}-1} s \cdot a_s \right) \\
&\quad= \frac{1}{a_j} \cdot \left(\sum_{s=j}^{n^{max}-1} (s+1) \cdot a_s - \sum_{s=j+1}^{n^{max}-1} s \cdot a_s \right) \\
&\quad= \frac{1}{a_j} \cdot \left(\sum_{s=j}^{n^{max}-1} s \cdot a_s + \sum_{s=j}^{n^{max}-1} a_s - \sum_{s=j+1}^{n^{max}-1} s \cdot a_s \right) \\
&\quad= \frac{1}{a_j} \cdot \left(j \cdot a_j + \sum_{s=j+1}^{n^{max}-1} s \cdot a_s + \sum_{s=j}^{n^{max}-1} a_s - \sum_{s=j+1}^{n^{max}-1} s \cdot a_s \right) .
\end{aligned}$$

Nach der Umformung heben sich zwei der Summen weg und es folgt die gesuchte Beziehung aus Gleichung (5.6):

$$\hat{n}_j^{aktiv}(c, \mathcal{PS}) \quad = \quad \frac{1}{a_j} \cdot \left(j \cdot a_j + \sum_{s=j}^{n^{max}-1} a_s \right) .$$

A.2. Beweis für die ILP-Realisierung des min-Operators

Im Folgenden soll gezeigt werden, dass aus den Nebenbedingungen

$$A_{sj} \geq a_s - \epsilon \cdot \Psi_{sj} \quad \text{und} \quad A_{sj} \geq a_j + \epsilon \cdot \Psi_{sj} - \epsilon$$

aus Gleichung (7.26) für die optimale Lösung eines ILP-basierten Minimierungsproblems

$$A_{sj} = \min(a_s, a_j)$$

folgt. Als Grundannahme wird dabei vorausgesetzt, dass die ILP-Zielfunktion mit sinkenden Werten von A_{sj} streng monoton abfällt und dass Ψ_{sj} ausschließlich in die beiden genannten Nebenbedingungen eingeht. Außerdem wird $\epsilon \geq a_s \geq 0$ und $\epsilon \geq a_j \geq 0$ gefordert. Auf das ILP-Problem zur Speicherallokation aus Abschnitt 7.4.3 treffen all diese Voraussetzungen zu.

Beweis. Eine optimale Lösung des ILP-Problems muss auch den Wert von A_{sj} minimieren, da ein kleinerer Variablenwert durch die Grundannahme auch zu einem niedrigeren Wert der Zielfunktion führt. Der Wert der Variablen Ψ_{sj} kann vom Lösungsverfahren außerdem ohne Zusatzkosten frei gewählt werden, da die Variable nur in den obigen Nebenbedingungen verwendet wird. Für $\Psi_{sj} = 1$ wird die erste der beiden Nebenbedingungen auf der linken Seite mit dem hohen negativen Wert $-\epsilon$ beaufschlagt, sodass die Nebenbedingung für alle $A_{sj} > 0$ immer erfüllt ist. Bei $\Psi_{sj} = 0$ ist nicht die erste, sondern die zweite Nebenbedingung von diesem Effekt betroffen und wird de-facto deaktiviert. Durch die Festlegung von Ψ_{sj} kann der Lösungsalgorithmus also zwischen den Nebenbedingungen $A_{sj} \geq a_s$ oder $A_{sj} \geq a_j$ frei wählen. Da A_{sj} in keinem der beiden Fälle gleichzeitig kleiner als a_s und a_j sein kann, ist der minimal erreichbare Wert durch $A_{sj} = \min(a_s, a_j)$ gegeben. Unter den genannten Vorbedingungen muss A_{sj}, unabhängig von a_s und a_j, in der optimalen Lösung des ILP-Problems also diesen minimalen Wert annehmen. ■

Verzeichnisse

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Symbolverzeichnis

$aPPIR = (\mathcal{CS}, E^{cfp}, E^{sync})$	Azyklischer PPIR Graph
A_{sj}	ILP-Variable für das Minimum von a_s und a_j
$\mathcal{B}^{-}(s)$	Menge von CFG-Knoten, die *vor* dem Synchronisationsknoten $s \in \mathcal{S}$ ausgeführt werden müssen
$\mathcal{B}^{+}(s)$	Menge von CFG-Knoten, die *nach* dem Synchronisationsknoten $s \in \mathcal{S}$ ausgeführt werden müssen
C	Menge aller Komponenten im Plattform-Komponenten-Graph
$CFG = (V, E^{cf}, TYP)$	Kontrollflussgraph
$CG = (V^{cg}, E^{cg})$	Plattform-Komponenten-Graph
cs_i	Ein Code-Segment in einem azyklischen PPIR Graphen
$\mathcal{CS}$	Menge von Code-Segmenten in einem azyklischen PPIR Graphen
δ_p	ILP-Entscheidungsvariable, die angibt, ob eine Variable v_p in einen größeren Gültigkeitsbereich verschoben wird
$D^{np}\big(R_{kl}, \mathcal{PS}\big)$	Verzögerungszeit für das Versenden eines „Non-Posted"-Zugriffs über R_{kl} im Kontext $\mathcal{PS}$.
$\overline{D}(cs_i)$	Summe der Verzögerungszeiten aller Speicherzugriffe in einem Code-Segment cs_i
$D^{p}\big(R_{kl}, \mathcal{PS}\big)$	Verzögerungszeit für das Versenden eines „Posted Write"-Zugriffs über R_{kl} im Kontext $\mathcal{PS}$.
DS	Datendurchsatz
E^{cf}	Menge von Kontrollflusskanten
E^{sync}	Menge von Synchronisationskanten
$FIFO_{k,l} = (\pi_k, \pi_l, \nu)$	FIFO-Kanal zwischen den Prozessen π_k und π_l
$G^{st}(M) = \big(V^{st}, E^{st}, TYP\big)$	Subgraph zur Synchronisationsplatzierung für eine Menge M von Synchronisationsknoten

$G^{stack} = \left(V^{stack}, E^{stack}\right)$	Stack-Graph für ein C-Programm
$\mathcal{I}$	Vektor aus 0/1 Entscheidungsvariablen, der eine bestimmte Speicherallokation festlegt
$IF(cs_i)$	Durch Interferenzen verursachte Verzögerungszeit bei der Ausführung eines Code-Segments cs_i
I_{pl}	Eine 0/1 Entscheidungsvariable im Vektor $\mathcal{I}$, die angibt, ob eine Variable $v_p \in \mathcal{V}$ dem Speicher $m_l \in \mathcal{C}$ zugewiesen ist
J	Kombinierter Lösungsgraph für die Optimierung der Synchronisationsknoten $s \in \mathcal{S}$
$\mathfrak{J}(c, R_{kl}, \mathcal{PS})$	Arbiter-Eingang an Komponente c, der einer Route R_{kl} im Kontext $\mathcal{PS}$ zugeordnet ist
$K\big(\, POS(s) \,\big)$	Gesamtkosten einer Lösung $POS(s)$ des Platzierungsproblems für Synchronisation
L	Latenz
$\lambda(v_p)$	Lebensspanne einer Variablen v_p
$L_j^{\overline{eff}}(c, n^{aktiv})$	Effektive Latenz für den j-ten Eingang einer Komponente c
$\overline{L}_j(c, \mathcal{PS})$	Akkumulierte Gesamtlatenz für alle Zugriffe über den Arbiter-Eingang j der Komponente c im Ausführungskontext $\mathcal{PS}$
$\overline{L}^{rt}(R_{kl}, \mathcal{PS})$	Akkumulierte Gesamtlatenz für alle Zugriffe über die Route R_{kl} im Ausführungskontext $\mathcal{PS}$
$L^{rt}\big(R_{kl}, \mathcal{PS}\big)$	Latenz einer Route R_{kl} im Kontext $\mathcal{PS}$
m_l	Eine Speicherkomponente aus der Menge $\mathcal{M}$
$\lvert m_l \rvert$	Speicherkapazität der Speicherkomponente m_l
$\mathcal{M} \subset \mathcal{C}$	Menge aller Speicherkomponenten in einem Plattform-Komponenten-Graph CG
$\mu : V \mapsto \mathcal{P}$	Funktion zur Zuweisung von Task-Knoten zu einem parallelen Prozess $\pi_k \in \mathcal{P}$
$\mu^{bb} : V \mapsto \mathcal{P}$	Funktion zur Zuweisung von Basisblöcken zu einem parallelen Prozess $\pi_k \in \mathcal{P}$
$\mu^{cs} : CS \mapsto \mathcal{C}$	Funktion zur Zuweisung der Code-Segmente $cs_i \in CS$ zu den Prozessorkomponenten $c_k \in \mathcal{C}$
$\mu^{v} : \mathcal{V} \mapsto \mathcal{M}$	Funktion zur Zuweisung der Variablen $v \in \mathcal{V}$ des Programms zu Speicherkomponenten $m_l \in \mathcal{M}$

n^{aktiv}	Anzahl der aktiven Eingänge einer Arbitrierungseinheit
$n^{cf}(e)$	Ausführungshäufigkeiten einer Kontrollflusskanten e
n^{np+}	Anzahl der Dateneinheiten, einschließlich Protokollinformationen, die für eine Zugriffs-Anfrage vom Typ *non-posted* anfallen
n^{np-}	Anzahl der Dateneinheiten, einschließlich Protokollinformationen, die für eine Antwort auf eine Zugriffs-Anfrage anfallen
n^{p+}	Anzahl der für einen *posted Write* anfallenden Dateneinheiten, einschließlich Protokollinformationen
$N^{mem}(cs_i, m_l)$	Maximalzahl der Speicherzugriffe auf den Speicher m_l im Code-Segment cs_i
$N^{mem,np}(cs_i, m_l)$	Anzahl der Zugriffe vom Typ *non-posted* innerhalb des Code-Segments cs_i auf den Speicher m_l
$N^{mem,p}(cs_i, m_l)$	Anzahl der *posted Writes* innerhalb des Code-Segments cs_i auf den Speicher m_l
$\overline{N}(R_{kl}, \mathcal{PS})$	Die maximale Anzahl der Dateneinheiten, die eine Route R_{kl} im Kontext $\mathcal{PS}$ durchlaufen können
$n^{\pi}(v_p)$	Anzahl der parallelen Prozesse $\pi_k \in \mathcal{P}$, die auf eine Variable $v_p \in \mathcal{V}$ zugreifen
$N^{var}(cs_i, v_p)$	Die maximale Anzahl von Zugriffen auf eine Variable $v_p \in \mathcal{V}$ innerhalb des Code-Segments cs_i
$\mathcal{O}$	Landau-Symbol / $\mathcal{O}$-Notation
P	Pitch
$\mathcal{P}$	Endliche Menge von Prozessen
$P_j^{eff}(c, n^{aktiv})$	Effektiver Pitch für den j-ten Eingang einer Komponente c
π_k	Prozess
$POS(s) \subseteq \mathcal{VP}(s)$	Lösung des Platzierungsproblems für einen Synchronisationsknoten $s \in \mathcal{S}$
$\mathrm{Pot}(X)$	Potenzmenge von X

$\overline{P}_j(c, \mathcal{PS})$	Akkumulierter Pitch für alle Zugriffe über den Arbiter-Eingang j der Komponente c im Ausführungskontext $\mathcal{PS}$
$\overline{P}^{rt}(R_{kl}, \mathcal{PS})$	Akkumulierter Pitch für alle Zugriffe über die Route R_{kl} im Ausführungskontext $\mathcal{PS}$
$P^{rt}\big(R_{kl}, \mathcal{PS}\big)$	Pitch einer Route R_{kl} im Kontext $\mathcal{PS}$
$\mathcal{PS}(cs_i)$	Die Menge aller Code-Segmente, die während der Ausführungszeit von cs_i auf den verfügbaren Kernen ausgeführt werden können
Ψ_{sj}	ILP-Hilfsvariable zur Bestimmung des Minimums von a_s und a_j
R_{kl}	Route im Plattform-Komponenten-Graphen CG
$\mathfrak{S}$	Maximale Stack-Größe in einem C-Programm
$\mathcal{S}$	Menge von Synchronisationsknoten $\mathcal{S} \subset V$ in einem Programmgraphen
SP	Speedup
T	Ausführungszeit eines Programms
$TG = (V, E, w)$	Task-Graph
$\Theta(v_p)$	Die Menge aller Variablen v_q, deren Gültigkeitsbereiche mit dem von v_p in Konflikt stehen
$\theta(v_p, v_q)$	Indikatorfunktion, die angibt, ob die Gültigkeitsbereiche zweier Variablen v_p und v_q miteinander in Konflikt stehen
t	Zeitpunkt
TN_i	Task-Knoten
$TYP(bb_i)$	Typ des Knotens bb_i in einem Programmgraphen
v_p	Variable in einem C-Programm
$\lvert v_p \rvert$	Der von einer Variablen v_p belegte Speicherplatz
$\mathcal{V}$	Menge aller Variablen in einem C-Programm
$\mathcal{V}^p$	Teilmenge der C-Variablen, die jeweils von nur einem einzigen Prozess genutzt werden dürfen
$\mathcal{V}^s$	Teilmenge der C-Variablen, die von mehreren Prozessen gemeinsam genutzt werden können
$\mathcal{VP}(s)$	Teilmenge der Kanten eines CFG, die als neue Positionen für den Synchronisationsknoten $s \in \mathcal{S}$ in Frage kommen
$w(e)$	Kantengewicht für die Kante e eines Graphen

$w^{joint}(e^{joint})$	Kantengewicht einer Kante e^{joint} des kombinierten Lösungsgraphen J
$WCET^{comp}(cs_i)$	Anteil der WCET eines Code-Segments cs_i, der auf reine Berechnungen ohne Speicherzugriffszeiten entfällt
$WCET^{crit}$	Die WCET des kritischen Pfades in der aPPIR eines parallelen Programms
$WCET^{sync}(e, \pi_k)$	WCET bis zum Erreichen der Kontrollflusskante e in Prozess π_k im Bezug auf den CFG nach der Datenpartitionierung
$\mathcal{W}(q, \zeta_i)$	Die Menge aller Pfade in einem Stack-Gaphen G^{stack}, die in dessen Quelle q beginnen und in der Senke ζ_i enden.
$Y(\mathcal{I})$	Zielfunktion für die Speicherallokation ohne Berücksichtigung von Interferenz-Effekten
$Y^{crit}(\mathcal{I})$	Zielfunktion für die Speicherallokation, die die WCET des kritischen Pfades beschreibt
$Y^{if}(\mathcal{I})$	Zielfunktion für die Speicherallokation unter Berücksichtigung von Interferenz-Effekten

Abkürzungsverzeichnis

AADL	Architecture Analysis & Design Language
ADL	Architekturbeschreibungssprache (englisch *Architecture Description Language*)
AHB	Advanced High-performance Bus
aPPIR	Azyklischer PPIR Graph
ARGO	ARGO-Projekt (Abkürzung für „WCET-Aware Parallelization of Model-Based Applications for Heterogeneous Parallel Systems“)
AST	Abstrakter Syntaxbaum (english *Abstract Syntax Tree*)
AUTOSAR	AUTomotive Open System ARchitecture
CB	Crossbar
CFG	Kontrollflussgraph (englisch *Control-Flow Graph*)
CPS	Cyber-Physikalisches System
CPU	Central Processing Unit
DFS	Depth-First Search (englisch für *Tiefensuche*)
DMA	Direct Memory Access (englisch für *Speicherdirektzugriff*)
DRAM	Dynamic Random-Access Memory
EAST-ADL	Electronics Architecture and Software Technology - Architecture Description Language
FFT	Fast Fourier Transform
FIFO	„First In – First Out“ Prinzip
FPGA	Field Programmable Gate Array

FPU	Floating Point Unit
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
GS-Channel	Guaranteed Service Channel
HEFT-LA	Heterogeneous Earliest Finish Time mit Look Ahead
HTG	Hierarchischer Task-Graph (englisch *Hierarchical Task Graph*)
ILP	Ganzzahlige lineare Optimierung (englisch *Integer Linear Programming*)
IoT	Internet of Things
IR	Zwischendarstellung (englisch *Intermediate Representation*)
ISA	Instruction Set Architecture
KPN	Kahn Process Network
LRU	„Least Recently Used“ (Cache-Ersetzungsstrategie)
MC	Memory Controller
MHP	„May-Happen-in-Parallel“
MIP	Gemischt-ganzzahlige lineare Optimierung (englisch *Mixed-Integer linear Programming*)
MoC	Model-of-Computation
MPI	Message Passing Interface
MPSoC	Multi-processor System on Chip (englisch für *Ein-Chip-System* mit mehreren Prozessoren)
NA	Netzwerk Adapter
NoC	Network-on-Chip
NUMA	Non-Uniform Memory Access

PN	Prozessnetzwerke (englisch *Process Networks*)
PPG	Paralleler Programmgraph (englisch *Parallel Program Graph*)
PPIR	Parallele Programm IR
RAM	Random-Access Memory
RAW	Read-after-Write
RR	Rundlaufverfahren (englisch *Round Robin*)
RTOS	Echtzeitbetriebssystem (englisch *Real-Time Operating System*)
S2S	Quellcode-zu-Quellcode-Compiler (englisch *Source-to-Source Compiler*)
SESE	„Single-Entry Single-Exit“
SHIM	Software-Hardware Interface for Multi-Many-Core
SHM	Shared Memory
SMP	Symmetric Multiprocessor (englisch für *Symmetrischer Mehrkernprozessor*)
SoC	System on Chip
SPARC	Scalable Processor ARChitecture
SPM	Scratchpad Memory
SRAM	Static Random-Access Memory
SSA	Static Single Assignment
TAWS	Terrain Awareness and Warning System
TDMA	Zeitmultiplexverfahren (englisch *Time Division Multiple Access*)
TLM	Tile-Local Memory
UML	Unified Modeling Language
WAR	Write-after-Read
WAW	Write-after-Write
WCAT	Worst Case Access Times
WCET	Worst Case Execution Time

XML	Extensible Markup Language
XSD	XML Schema Definition

Literatur- und Quellennachweise

[1] S. V. Adve und K. Gharachorloo, „Shared memory consistency models: a tutorial“, *Computer*, Jg. 29, Nr. 12, S. 66–76, Dez. 1996. DOI: 10.1109/2.546611.

[2] S. Altmeyer, R. I. Davis, L. Indrusiak, C. Maiza, V. Nelis und J. Reineke, „A Generic and Compositional Framework for Multicore Response Time Analysis“, in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, Ser. RTNS '15, Lille, France: ACM, 2015, S. 129–138, ISBN: 978-1-4503-3591-1. DOI: 10.1145/2834848.2834862. Adresse: http://doi.acm.org/10.1145/2834848.2834862.

[3] A. H. Ashouri, G. Palermo, J. Cavazos und C. Silvano, „The Phase-Ordering Problem: A Complete Sequence Prediction Approach“, in *Automatic Tuning of Compilers Using Machine Learning*. Cham: Springer International Publishing, 2018, Kap. 5, S. 85–113, ISBN: 978-3-319-71489-9. DOI: 10.1007/978-3-319-71489-9_5. Adresse: https://doi.org/10.1007/978-3-319-71489-9_5.

[4] O. Avissar, R. Barua und D. Stewart, „Heterogeneous Memory Management for Embedded Systems“, in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Ser. CASES '01, Atlanta, Georgia, USA: Association for Computing Machinery, 2001, S. 34–43, ISBN: 1581133995. DOI: 10.1145/502217.502223. Adresse: https://doi.org/10.1145/502217.502223.

[5] R. Baert, E. Brockmeyer, S. Wuytack und T. J. Ashby, „Exploring Parallelizations of Applications for MPSoC Platforms Using MPA“, in *Proceedings of the Conference on Design, Automation and Test in Europe*, Ser. DATE '09, Nice, France: European Design und Automation Association, 2009, S. 1148–1153, ISBN: 978-3-9810801-5-5. Adresse: http://dl.acm.org/citation.cfm?id=1874620.1874898.

[6] C. Ballabriga, H. Cassé, C. Rochange und P. Sainrat, „OTAWA: An Open Toolbox for Adaptive WCET Analysis“, in *Software Technologies for Embedded and Ubiquitous Systems*, S. L. Min, R. Pettit, P. Puschner und T. Ungerer, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 35–46, ISBN: 978-3-642-16256-5.

[7] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis und T. Nolte, „Contention-free execution of automotive applications on a clustered many-core platform“, in *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*, IEEE, 2016, S. 14–24. DOI: 10.1109/ECRTS.2016.14.

[8] U. Bondhugula, A. Hartono, J. Ramanujam und P. Sadayappan, „A Practical Automatic Polyhedral Parallelizer and Locality Optimizer“, *SIGPLAN Not.*, Jg. 43, Nr. 6, S. 101–113, Juni 2008, ISSN: 0362-1340. DOI: 10.1145/1379022.1375595. Adresse: http://doi.acm.org/10.1145/1379022.1375595.

[9] P. Brucker, *Scheduling Algorithms*. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-69516-5.

[10] V. Brühl, „Künstliche Intelligenz, Maschinelles Lernen und Big Data-Grundlagen, Marktpotenziale und wirtschaftspolitische Relevanz“, *WiSt-Wirtschaftswissenschaftliches Studium*, Jg. 48, Nr. 11, S. 34–41, 2019.

[11] A. Burns und R. Davis, „Mixed criticality systems – a review“, *Department of Computer Science, University of York, Tech. Rep*, S. 1–81, 2019.

[12] J. Canny, „A Computational Approach to Edge Detection“, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Jg. PAMI-8, Nr. 6, S. 679–698, Nov. 1986, ISSN: 1939-3539. DOI: 10.1109/TPAMI.1986.4767851.

[13] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan und J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[14] P. Clauss, „Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs“, in *Proceedings of the 10th international conference on Supercomputing*, 1996, S. 278–285.

[15] Cobham plc, *LEON3 Produkt-Webseite*, 2020. Adresse: `https://www.gaisler.com/index.php/products/processors/leon3` (besucht am 28. 02. 2020).

[16] COIN-OR Foundation, *Coin-Or Branch-and-Cut Solver*, 2019. Adresse: `https://projects.coin-or.org/Cbc` (besucht am 22. 10. 2019).

[17] D. Cordes, O. Neugebauer, M. Engel und P. Marwedel, „Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs“, in *2013 42nd International Conference on Parallel Processing*, Okt. 2013, S. 950–959. DOI: 10.1109/ICPP.2013.113.

[18] D. Cordes, P. Marwedel und A. Mallik, „Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming“, in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ACM, 2010, S. 267–276. DOI: 10.1145/1878961.1879009.

[19] R. Davis und A. Burns, „A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems“, *University of York, Department of Computer Science, techreport YCS-2009-443*, 2009.

[20] R. I. Davis und L. Cucu-Grosjean, „A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems“, *Leibniz Transactions on Embedded Systems*, Jg. 6, Nr. 1, S. 03–1, 2019.

[21] K. Deb und K. Pal, „Efficiently Solving: A Large-Scale Integer Linear Program Using a Customized Genetic Algorithm“, in *Genetic and Evolutionary Computation – GECCO 2004*, K. Deb, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 1054–1065, ISBN: 978-3-540-24854-5.

[22] V. Debruyne, F. Simonot-Lion und Y. Trinquet, „EAST-ADL — An Architecture Description Language“, in *Architecture Description Languages*, P. Dissaux, M. Filali-Amine, P. Michel und F. Vernadat, Hrsg., Boston, MA: Springer US, 2005, S. 181–195, ISBN: 978-0-387-24590-4.

[23] J. F. Deverge und I. Puaut, „WCET-Directed Dynamic Scratchpad Memory Allocation of Data“, in *Proc. 19th Euromicro Conf. Real-Time Systems (ECRTS'07)*, Juli 2007, S. 179–190. DOI: 10.1109/ECRTS.2007.37.

[24] K. Didier, D. Potop-Butucaru, G. Iooss, A. Cohen, J. Souyris, P. Baufreton und A. Graillat, „Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware“, *ACM Trans. Archit. Code Optim.*, Jg. 16, Nr. 3, 24:1–24:27, Juli 2019, ISSN: 1544-3566. DOI: 10.1145/3328799. Adresse: http://doi.acm.org/10.1145/3328799.

[25] R. O. Duda und P. E. Hart, „Use of the Hough Transformation to Detect Lines and Curves in Pictures“, *Commun. ACM*, Jg. 15, Nr. 1, S. 11–15, Jan. 1972, ISSN: 0001-0782. DOI: 10.1145/361237.361242. Adresse: https://doi.org/10.1145/361237.361242.

[26] ESI Group, *Scilab.org Webseite*, 2019. Adresse: https://www.scilab.org/ (besucht am 15. 08. 2019).

[27] H. Falk und P. Lokuciejewski, „A compiler framework for the reduction of worst-case execution times“, *Real-Time Systems*, Jg. 46, Nr. 2, S. 251–300, Okt. 2010, ISSN: 1573-1383. DOI: 10.1007/s11241-010-9101-x. Adresse: https://doi.org/10.1007/s11241-010-9101-x.

[28] P. H. Feiler, D. P. Gluch und J. J. Hudak, „The architecture analysis & design language (AADL): An introduction“, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Techn. Ber., 2006.

[29] C. Ferdinand und R. Heckmann, „aiT: Worst-case execution time prediction by static program analysis“, *Building the Information Society*, S. 377–383, 2004. DOI: 10.1007/978-1-4020-8157-6_29.

[30] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski und O. Sentieys, „GeCoS: A framework for prototyping custom hardware design flows“, in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2013, S. 100–105. DOI: 10.1109/SCAM.2013.6648190.

[31] F. Franchetti, Y. Voronenko und M. Püschel, „FFT program generation for shared memory: SMP and multicore“, in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, 2006, S. 115.

[32] J. Freitag, S. Uhrig und T. Ungerer, „Virtual Timing Isolation for Mixed-Criticality Systems“, in *30th Euromicro Conference on Real-Time*

Systems (ECRTS 2018), S. Altmeyer, Hrsg., Ser. Leibniz International Proceedings in Informatics (LIPIcs), Bd. 106, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 13:1–13:23, ISBN: 978-3-95977-075-0. DOI: 10.4230/LIPIcs.ECRTS.2018.13. Adresse: http://drops.dagstuhl.de/opus/volltexte/2018/8990.

[33] M. Frieb, R. Jahr, H. Ozaktas, A. Hugl, H. Regler und T. Ungerer, „A Parallelization Approach for Hard Real-Time Systems and Its Application on Two Industrial Programs“, *International Journal of Parallel Programming*, Jg. 44, Nr. 6, S. 1296–1336, Dez. 2016, ISSN: 1573-7640. DOI: 10.1007/s10766-016-0432-7. Adresse: https://doi.org/10.1007/s10766-016-0432-7.

[34] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa und K. Lange, „AUTOSAR – A Worldwide Standard is on the Road“, in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Bd. 62, 2009, S. 5.

[35] M. Girkar und C. D. Polychronopoulos, „The hierarchical task graph as a universal intermediate representation“, *International Journal of Parallel Programming*, Jg. 22, Nr. 5, S. 519–551, Okt. 1994. DOI: 10.1007/bf02577777.

[36] A. Goknil, J. Suryadevara, M.-A. Peraldi-Frati und F. Mallet, „Analysis Support for TADL2 Timing Constraints on EAST-ADL Models“, in *Software Architecture*, K. Drira, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 89–105, ISBN: 978-3-642-39031-9.

[37] K. Goossens und A. Hansson, „The aethereal network on chip after ten years: Goals, evolution, lessons, and future“, in *Design Automation Conference*, Juni 2010, S. 306–311. DOI: 10.1145/1837274.1837353.

[38] G. Goulas, C. Valouxis, P. Alefragis, N. S. Voros, C. Gogos, O. Oey, T. Stripf, T. Bruckschloegl, J. Becker, A. El Moussawi, M. Naullet und T. Yuki, „Coarse-Grain Optimization and Code Generation for Embedded Multicore Systems“, in *2013 Euromicro Conference on Digital System Design*, Sep. 2013, S. 379–386. DOI: 10.1109/DSD.2013.48.

[39] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. Jacobs und K. Langendoen, *Modern Compiler Design*. Springer New York, 2012. DOI: 10.1007/978-1-4614-4699-6.

[40] J. Heißwolf, „A Scalable and Adaptive Network on Chip for Many-Core Architectures“, Diss., Karlsruher Institut für Technologie (KIT), 2014. Adresse: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000045305.

[41] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari und S. Kobbe, „Invasive manycore architectures“, in *17th Asia and South Pacific Design Automation Conference*, Jan. 2012, S. 193–200. DOI: 10.1109/ASPDAC.2012.6164944.

[42] J. L. Hennessy und D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[43] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson und D. M. Johnson, „RTOS Support for Multicore Mixed-Criticality Systems“, in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, Apr. 2012, S. 197–208. DOI: 10.1109/RTAS.2012.24.

[44] P. V. Hough, *Method and means for recognizing complex patterns*, US Patent 3,069,654, Dez. 1962.

[45] R. Hwang, M. Gen und H. Katayama, „A comparison of multiprocessor task scheduling algorithms with communication costs“, *Computers & Operations Research*, Jg. 35, Nr. 3, S. 976–993, 2008, Part Special Issue: New Trends in Locational Analysis, ISSN: 0305-0548. DOI: 10.1016/j.cor.2006.05.013. Adresse: http://www.sciencedirect.com/science/article/pii/S0305054806001432.

[46] „ISO/IEC 9899:1999 Information technology — Programming languages — C“, International Organization for Standardization (ISO), Geneva, CH, Standard, Dez. 1999.

[47] „ISO/IEC 9899:2011 Information technology — Programming languages — C“, International Organization for Standardization (ISO), Geneva, CH, Standard, Dez. 2011.

[48] R. Jahr, M. Gerdes und T. Ungerer, „On Efficient and Effective Model-based Parallelization of Hard Real-Time Applications.“, in *Model-Based Development of Embedded Systems (MBEES)*, Citeseer, 2013, S. 50–59.

[49] J. Jalle, J. Abella, E. Quiñones, L. Fossati, M. Zulianello und F. J. Cazorla, „AHRB: A high-performance time-composable AMBA AHB bus“, in *Proc. IEEE 19th Real-Time and Embedded Technology and Applications Symp. (RTAS)*, Apr. 2014, S. 225–236. DOI: 10.1109/RTAS.2014.6926005.

[50] M. M. Kafshdooz und A. Ejlali, „Dynamic Shared SPM Reuse for Real-Time Multicore Embedded Systems“, *ACM Trans. Archit. Code Optim.*, Jg. 12, Nr. 2, 12:1–12:25, Mai 2015, ISSN: 1544-3566. DOI: 10.1145/2738051. Adresse: http://doi.acm.org/10.1145/2738051.

[51] G. Kahn, „The semantics of a simple language for parallel programming“, *In Information Processing*, Jg. 74, S. 471–475, 1974.

[52] E.-Y. Kang und P.-Y. Schobbens, „Schedulability Analysis Support for Automotive Systems: From Requirement to Implementation“, in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, Ser. SAC '14, Gyeongju, Republic of Korea: ACM, 2014, S. 1080–1085, ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2554929. Adresse: http://doi.acm.org/10.1145/2554850.2554929.

[53] D. Kästner, M. Schlickling, M. Pister, C. Cullmann, G. Gebhard, R. Heckmann und C. Ferdinand, „Meeting Real-Time Requirements with Multi-core Processors“, in *Computer Safety, Reliability, and Security*, F. Ortmeier und P. Daniel, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 117–131, ISBN: 978-3-642-33675-1. DOI: 10.1007/978-3-642-33675-1_10.

[54] S. Kehr, M. Panić, E. Quiñones, B. Böddeker, J. B. Sandoval, J. Abella, F. J. Cazorla und G. Schäfer, „Supertask: Maximizing runnable-level parallelism in AUTOSAR applications“, in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, März 2016, S. 25–30.

[55] T. Kelter, „WCET analysis and optimization for multi-core real-time systems“, Diss., Technische Universität Dortmund, 2015. DOI: 10.17877/DE290R-7209.

[56] T. Kelter und P. Marwedel, „Parallelism analysis: Precise WCET values for complex multi-core systems“, *Science of Computer Programming*, Jg. 133, S. 175–193, 2017, Formal Techniques for Safety-Critical Systems (FTSCS 2014), ISSN: 0167-6423. DOI: 10.1016/j.scico.2016.01.007.

Adresse: http://www.sciencedirect.com/science/article/pii/S0167642316000277.

[57] S. Kempf, R. Veldema und M. Philippsen, „Is There Hope for Automatic Parallelization of Legacy Industry Automation Applications?“, *PARS*, Nr. 28, No. 1, S. 80–89, 2011. DOI: 10.1007/BF03341987.

[58] A. E. Kiasari, A. Jantsch und Z. Lu, „Mathematical Formalisms for Performance Evaluation of Networks-on-Chip“, *ACM Comput. Surv.*, Jg. 45, Nr. 3, Juli 2013, ISSN: 0360-0300. DOI: 10.1145/2480741.2480755. Adresse: https://doi.org/10.1145/2480741.2480755.

[59] Y. Kim, D. Broman, J. Cai und A. Shrivastaval, „WCET-aware dynamic code management on scratchpads for Software-Managed Multicores“, in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014, S. 179–188. DOI: 10.1109/RTAS.2014.6926001.

[60] P. Kohli und A. Chadha, „Enabling Pedestrian Safety Using Computer Vision Techniques: A Case Study of the 2018 Uber Inc. Self-driving Car Crash“, in *Advances in Information and Communication*, K. Arai und R. Bhatia, Hrsg., Cham: Springer International Publishing, 2020, S. 261–279, ISBN: 978-3-030-12388-8.

[61] B. Korte und J. Vygen, *Combinatorial Optimization*. Springer Berlin Heidelberg, 2018. DOI: 10.1007/978-3-662-56039-6.

[62] A. Kröhling, „Digitalisierung – Technik für eine nachhaltige Gesellschaft?“, in *CSR und Digitalisierung: Der digitale Wandel als Chance und Herausforderung für Wirtschaft und Gesellschaft*, A. Hildebrandt und W. Landhäußer, Hrsg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, Kap. 2, S. 23–49, ISBN: 978-3-662-53202-7. DOI: 10.1007/978-3-662-53202-7_2. Adresse: https://doi.org/10.1007/978-3-662-53202-7_2.

[63] S. Kumar, A. Jantsch, J. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja und A. Hemani, „A network on chip architecture and design methodology“, in *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, Apr. 2002, S. 117–124. DOI: 10.1109/ISVLSI.2002.1016885.

[64] V. Kumar, *Introduction to parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[65] T. Kuroda, „CMOS design challenges to power wall“, in *Digest of Papers. Microprocesses and Nanotechnology 2001. 2001 International Microprocesses and Nanotechnology Conference (IEEE Cat. No.01EX468)*, Okt. 2001, S. 6–7. DOI: 10.1109/IMNC.2001.984030.

[66] E. A. Lee, „Cyber Physical Systems: Design Challenges“, in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, Mai 2008, S. 363–369. DOI: 10.1109/ISORC.2008.25.

[67] S. Lee, F. Mallet und R. d. Simone, „Dealing with AADL End-to-End Flow Latency with UML MARTE“, in *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, März 2008, S. 228–233. DOI: 10.1109/ICECCS.2008.14.

[68] T. Lefeuvre, I. Fassi, C. Cullmann, G. Gebhard, E. K. Kasnakli, I. Puaut und S. Derrien, „Using polyhedral techniques to tighten WCET estimates of optimized code: A case study with array contraction“, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, S. 925–930. DOI: 10.23919/DATE.2018.8342142.

[69] R. Leupers, M. A. Aguilar, J. F. Eusse, J. Castrillon und W. Sheng, „MAPS: A Software Development Environment for Embedded Multicore Applications“, *Handbook of Hardware/Software Codesign*, S. 917–949, 2017. DOI: 10.1007/978-94-017-7267-9_2.

[70] X. Li, Y. Liang, T. Mitra und A. Roychoudhury, „Chronos: A timing analyzer for embedded software“, *Science of Computer Programming*, Jg. 69, Nr. 1, S. 56–67, 2007, Special issue on Experimental Software and Toolkits, ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.01.014. Adresse: http://www.sciencedirect.com/science/article/pii/S0167642307001633.

[71] Y. Liu und W. Zhang, „Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors“, *Journal of Computing Science and Engineering*, Jg. 9, Nr. 2, S. 51–72, 2015.

[72] 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 Comput. Surv.*, Jg. 52, Nr. 3, Juni 2019, ISSN: 0360-0300. DOI: 10.1145/3323212. Adresse: https://doi.org/10.1145/3323212.

[73] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini und A. Marongiu, „Combining PREM Compilation and ILP Scheduling for High-performance and Predictable MPSoC Execution“, in *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, Ser. PMAM'18, Vienna, Austria: ACM, 2018, S. 11–20, ISBN: 978-1-4503-5645-9. DOI: 10.1145/3178442.3178444. Adresse: http://doi.acm.org/10.1145/3178442.3178444.

[74] Y. Moy, E. Ledinot, H. Delseny, V. Wiels und B. Monate, „Testing or formal verification: Do-178c alternatives and industrial experience“, *IEEE software*, Jg. 30, Nr. 3, S. 50–57, 2013.

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

[76] D. Mueller und U. Durak, „Enhanced Functions for a Parallel Multicore Ground Proximity Warning System“, in *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, IEEE, 2018, S. 1–7. DOI: 10.1109/DASC.2018.8569446.

[77] B. Nichols, D. Buttlar, J. Farrell und J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

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

[79] D. Oehlert, A. Luppold und H. Falk, „Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems“, in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, M. Bertogna, Hrsg., Ser. Leibniz International Proceedings in Informatics (LIPIcs), Bd. 76,

Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 1:1–1:22, ISBN: 978-3-95977-037-8. DOI: 10.4230/LIPIcs.ECRTS.2017.1. Adresse: http://drops.dagstuhl.de/opus/volltexte/2017/7160.

[80] D. Padua, Hrsg., *Encyclopedia of Parallel Computing*. Springer US, 2011. DOI: 10.1007/978-0-387-09766-4.

[81] C. Pagetti, J. Forget, H. Falk, D. Oehlert und A. Luppold, „Automated Generation of Time-predictable Executables on Multicore“, in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, Ser. RTNS '18, Chasseneuil-du-Poitou, France: ACM, 2018, S. 104–113, ISBN: 978-1-4503-6463-8. DOI: 10.1145/3273905.3273907. Adresse: http://doi.acm.org/10.1145/3273905.3273907.

[82] M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella und F. J. Cazorla, „RunPar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores“, in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Okt. 2014, S. 1–10. DOI: 10.1145/2656075.2656096.

[83] S. Pasricha und N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, ISBN: 0123705215.

[84] S. Pellegrini, T. Hoefler und T. Fahringer, „Exact Dependence Analysis for Increased Communication Overlap“, in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner und J. J. Dongarra, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 89–99, ISBN: 978-3-642-33518-1.

[85] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo und L. Sha, „Handling Mixed-criticality in SoC-based Real-time Embedded Systems“, in *Proceedings of the Seventh ACM International Conference on Embedded Software*, Ser. EMSOFT '09, Grenoble, France: ACM, 2009, S. 235–244, ISBN: 978-1-60558-627-4. DOI: 10.1145/1629335.1629367. Adresse: http://doi.acm.org/10.1145/1629335.1629367.

[86] J. Peña, J. Lozano und P. Larrañaga, „An empirical comparison of four initialization methods for the K-Means algorithm“, *Pattern Recognition Letters*, Jg. 20, Nr. 10, S. 1027–1040, 1999, ISSN: 0167-8655. DOI: 10.

1016/S0167-8655(99)00069-0. Adresse: http://www.sciencedirect.com/science/article/pii/S0167865599000690.

[87] Q. Perret, P. Maurère, É. Noulard, C. Pagetti, P. Sainrat und B. Triquet, „Mapping Hard Real-time Applications on Many-core Processors“, in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, Ser. RTNS '16, Brest, France: ACM, 2016, S. 235–244, ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997496. Adresse: http://doi.acm.org/10.1145/2997465.2997496.

[88] M. B. Petersen, A. V. Riber, S. T. Andersen und M. Schoeberl, „Time-Predictable Distributed Shared Memory for Multi-Core Processors“, in *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, Okt. 2018, S. 1–7. DOI: 10.1109/NORCHIP.2018.8573463.

[89] D. Pilaud, N. Halbwachs und J. Plaice, „LUSTRE: A declarative language for programming synchronous systems“, in *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, Bd. 178, 1987, S. 188.

[90] C. Pitter und M. Schoeberl, „A Real-time Java Chip-multiprocessor“, *ACM Trans. Embed. Comput. Syst.*, Jg. 10, Nr. 1, 9:1–9:34, Aug. 2010, ISSN: 1539-9087. DOI: 10.1145/1814539.1814548. Adresse: http://doi.acm.org/10.1145/1814539.1814548.

[91] W. Puffitsch, R. B. Sørensen und M. Schoeberl, „Time-Division Multiplexing vs Network Calculus: A Comparison“, in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, Ser. RTNS '15, Lille, France: Association for Computing Machinery, 2015, S. 289–296, ISBN: 9781450335911. DOI: 10.1145/2834848.2834868. Adresse: https://doi.org/10.1145/2834848.2834868.

[92] G. Ramalingam, „Context-sensitive Synchronization-sensitive Analysis is Undecidable“, *ACM Trans. Program. Lang. Syst.*, Jg. 22, Nr. 2, S. 416–430, März 2000, ISSN: 0164-0925. DOI: 10.1145/349214.349241. Adresse: http://doi.acm.org/10.1145/349214.349241.

[93] H. Rihani, M. Moy, C. Maiza, R. I. Davis und S. Altmeyer, „Response Time Analysis of Synchronous Data Flow Programs on a Many-Core

Processor“, in *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, Ser. RTNS '16, Brest, France: ACM, 2016, S. 67–76, ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997472. Adresse: http://doi.acm.org/10.1145/2997465.2997472.

[94] B. Rouxel, S. Derrien und I. Puaut, „Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures“, *ACM Trans. Embed. Comput. Syst.*, Jg. 16, Nr. 5s, 164:1–164:20, Sep. 2017, ISSN: 1539-9087. DOI: 10.1145/3126496. Adresse: http://doi.acm.org/10.1145/3126496.

[95] S. Rubini, P. Dissaux und F. Singhoff, „Modeling Shared-Memory Multiprocessor Systems with AADL“, in *ACVI 2014–Architecture Centric Virtual Integration Workshop Proceedings*, 2014, S. 49.

[96] J. Rumbaugh, I. Jacobson und G. Booch, *The unified modeling language reference manual*. Pearson Higher Education, 2004.

[97] V. Sarkar und B. Simons, „Parallel Program Graphs and their classification“, in *Languages and Compilers for Parallel Computing*, Springer Berlin Heidelberg, 1994, S. 633–655. DOI: 10.1007/3-540-57659-2_36.

[98] S. E. Schaeffer, „Graph clustering“, *Computer Science Review*, Jg. 1, Nr. 1, S. 27–64, 2007, ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2007.05.001. Adresse: http://www.sciencedirect.com/science/article/pii/S1574013707000020.

[99] T. Schildhauer, D. G. Adlmaier-Herbst, J. Hofmann, H. Krcmar, W. Hünnekens, D. Michelis, P. F. Stephan, A. Termer und H. Voss, „Schlüsselfaktoren der Digitalisierung – Entwicklungen auf dem Weg in die digitale Zukunft“, in *Digitalisierung und Kommunikation : Konsequenzen der digitalen Transformation für die Wirtschaftskommunikation*, M. Stumpf, Hrsg. Wiesbaden: Springer Fachmedien Wiesbaden, Apr. 2019, Kap. 1, S. 13–34, ISBN: 978-3-658-26113-9. DOI: 10.1007/978-3-658-26113-9_2. Adresse: https://doi.org/10.1007/978-3-658-26113-9_2.

[100] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki et al., „T-CREST: Time-predictable multi-core architecture for embedded systems“, *Journal of Systems Architecture*, Jg. 61, Nr. 9, S. 449–471, 2015, ISSN: 1383-7621. DOI:

10.1016/j.sysarc.2015.04.002. Adresse: http://www.sciencedirect.com/science/article/pii/S1383762115000193.

[101] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra und D. Walker, *MPI – the Complete Reference: The MPI core*. MIT press, 1998, Bd. 1.

[102] R. M. Stallman et al., *Using and porting the GNU compiler collection*. Free Software Foundation, 1999, Bd. 86.

[103] A. Stegmeier, S. Kehr, D. George, C. Bradatsch, M. Panic, B. Bödekker und T. Ungerer, „Evaluation of fine-grained parallelism in AUTOSAR applications“, in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Juli 2017, S. 121–128. DOI: 10.1109/SAMOS.2017.8344619.

[104] D. Steinberg, F. Budinsky, E. Merks und M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[105] T. Stripf, O. Oey, T. Bruckschloegl, J. Becker, G. Rauwerda, K. Sunesen, G. Goulas, P. Alefragis, N. S. Voros, S. Derrien, O. Sentieys, N. Kavvadias, G. Dimitroulakos, K. Masselos et al., „Compiling Scilab to high performance embedded multicore systems“, *Microprocessors and Microsystems*, Jg. 37, Nr. 8, Part C, S. 1033–1049, 2013, Special Issue on European Projects in Embedded System Design: EPESD2012, ISSN: 0141-9331. DOI: 10.1016/j.micpro.2013.07.004. Adresse: http://www.sciencedirect.com/science/article/pii/S014193311300094X.

[106] V. Suhendra, T. Mitra, A. Roychoudhury und Ting Chen, „WCET centric data allocation to scratchpad memory“, in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, Dez. 2005, 10 pp.–232. DOI: 10.1109/RTSS.2005.45.

[107] H. Sutter, „The free lunch is over: A fundamental turn toward concurrency in software“, *Dr. Dobb's journal*, Jg. 30, Nr. 3, S. 202–210, 2005.

[108] S. Taha, A. Radermache, S. Gerard und J. Dekeyser, „An Open Framework for Detailed Hardware Modeling“, in *2007 International Symposium on Industrial Embedded Systems*, 2007, S. 118–125.

[109] R. N. Taylor, „Complexity of analyzing the synchronization structure of concurrent programs“, *Acta Informatica*, Jg. 19, Nr. 1, S. 57–84,

Apr. 1983, ISSN: 1432-0525. DOI: 10.1007/BF00263928. Adresse: https://doi.org/10.1007/BF00263928.

[110] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat und G. Snelting, „Invasive computing: An overview“, in *Multiprocessor System-on-Chip*, Springer, 2011, S. 241–268.

[111] The Multicore Association, *Software-Hardware Interface for Multi-Many-Core (SHIM) Specification V1.00 Final*, 2015. Adresse: http://www.multicore-association.org/workgroup/shim.php (besucht am 23. 08. 2017).

[112] K. Triantafyllidis, E. Bondarev und P. H. N. de With, „ProMARTES: Performance Analysis Method and Toolkit for Real-Time Systems“, in *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2013*, M.-M. Louërat und T. Maehne, Hrsg., Cham: Springer International Publishing, 2015, S. 281–302, ISBN: 978-3-319-06317-1. DOI: 10.1007/978-3-319-06317-1_15. Adresse: https://doi.org/10.1007/978-3-319-06317-1_15.

[113] T. Ungerer, C. Bradatsch, M. Frieb, F. Kluge, Joerg, Mische, A. Stegmeier, R. Jahr, M. Gerdes, P. Zaykov, L. Matusova, Z. J. J. Li, Z. Petrov, B. Böddeker et al., „Experiences and Results of Parallelisation of Industrial Hard Real-time Applications for the parMERASA Multi-core“, in *3rd Workshop on High-performance and Real-time Embedded Systems (HiRES 2015), Amsterdam, the Netherlands*, 2015.

[114] Q. Wan, H. Wu und J. Xue, „WCET-aware Data Selection and Allocation for Scratchpad Memory“, *SIGPLAN Not.*, Jg. 47, Nr. 5, S. 41–50, Juni 2012, ISSN: 0362-1340. DOI: 10.1145/2345141.2248425. Adresse: http://doi.acm.org/10.1145/2345141.2248425.

[115] M. H. Wiggers, M. J. G. Bekooij und G. J. M. Smit, „Modelling Run-Time Arbitration by Latency-Rate Servers in Dataflow Graphs“, in *Proceedingsof the 10th International Workshop on Software & Compilers for Embedded Systems*, Ser. SCOPES '07, Nice, France: Association for Computing Machinery, 2007, S. 11–22, ISBN: 9781450378345. DOI: 10.1145/1269843.1269846. Adresse: https://doi.org/10.1145/1269843.1269846.

[116] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat und P. Stenström, „The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools“, *ACM Trans. Embed. Comput. Syst.*, Jg. 7, Nr. 3, 36:1–36:53, Mai 2008, ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. Adresse: http://doi.acm.org/10.1145/1347375.1347389.

[117] World Wide Web Consortium, *Extensible Markup Language (XML) 1.1 (Second Edition) - W3C Recommendation 16 August 2006*, 2006. Adresse: https://www.w3.org/TR/2006/REC-xml11-20060816/ (besucht am 21. 09. 2020).

[118] W. K. Youn, S. B. Hong, K. R. Oh und O. S. Ahn, „Software certification of safety-critical avionic systems: DO-178C and its impacts“, *IEEE Aerospace and Electronic Systems Magazine*, Jg. 30, Nr. 4, S. 4–13, Apr. 2015, ISSN: 1557-959X. DOI: 10.1109/MAES.2014.140109.

Betreute studentische Arbeiten

[Hie17] L. Hielscher, „Konzept für Shared-Memory Unterstützung in einem Compiler-Framework für echtzeitfähige Multi-Core Systeme“, Master-Arbeit, Karlsruher Institut für Technologie (KIT), 2017.

[Nug17] L. M. A. Nugroho, „Konzept und Implementierung eines Frameworks zur effizienten Hardware in der Loop Co-Simulation von SystemC-Modellen und FPGA-basierten Hardwarekomponenten“, Master-Arbeit, Karlsruher Institut für Technologie (KIT), 2017.

[Pau17] S. Paul, „Computation Offoading in Edge Computing for Internet of Things: A Case Study of Epileptic Seizure Prediction“, Master-Arbeit, Karlsruher Institut für Technologie (KIT), 2017.

[Spe17] S. Speiser, „Bewertung von Ansätzen zur Compiler-Optimierung von SystemC basierten Hardware-Simulationen“, Bachelor-Arbeit, Karlsruher Institut für Technologie (KIT), 2017.

[Rie18] V. Rietz, „Scratchpad-Speicher-Optimierung in einem Compiler-Framework für Echtzeitanwendungen“, Bachelor-Arbeit, Karlsruher Institut für Technologie (KIT), 2018.

[Her19] T. Herzog, „Compiler-Optimierungen zur effizienten Ansteuerung von DMA-Einheiten in echtzeitfähigen Multi-Core Systemen“, Bachelor-Arbeit, Karlsruher Institut für Technologie (KIT), 2019.

Eigene Veröffentlichungen

[RB20a] S. Reder und J. Becker, „Interference-Aware Memory Allocation for Real-Time Multi-Core Systems“, in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, S. 148–159. DOI: 10.1109/RTAS48715.2020.00-10. Adresse: https://doi.org/10.1109/RTAS48715.2020.00-10.

[RB20b] S. Reder und J. Becker, „WCET-aware Code Generation and Communication Optimization for Parallelizing Compilers“, in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, S. 210–215. DOI: 10.23919/DATE48585.2020.9116400. Adresse: https://doi.org/10.23919/DATE48585.2020.9116400.

[RKB+19] S. Reder, F. Kempf, H. Bucher, J. Becker, P. Alefragis, N. Voros, S. Skalistis, S. Derrien, I. Puaut, O. Oey, T. Stripf, C. Ferdinand, C. David, P. Ulbig et al., „Worst-Case Execution-Time-Aware Parallelization of Model-Based Avionics Applications“, *Journal of Aerospace Information Systems*, Jg. 16, Nr. 11, S. 521–533, 2019. DOI: 10.2514/1.I010749. Adresse: https://doi.org/10.2514/1.I010749.

[RMB+18] S. Reder, L. Masing, H. Bucher, T. ter Braak, T. Stripf und J. Becker, „A WCET-aware parallel programming model for predictability enhanced multi-core architectures“, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, März 2018, S. 943–948. DOI: 10.23919/DATE.2018.8342145.

[RRB+15] S. Reder, C. Roth, H. Bucher, O. Sander und J. Becker, „Adaptive algorithm and tool flow for accelerating SystemC on many-core architectures“, *Microprocessors and microsystems*, Jg. 39, Nr. 8, S. 1063–1075, 2015, ISSN: 0141-9331, 1872-9436. DOI: 10.1016/j.micpro.2015.06.001.

[RtBM18] S. Reder, T. ter Braak und L. Masing, „ARGO Deliverable 2.6: WCET-aware multi-core programming model (Rev 2)“, The ARGO project consortium, EU Project Deliverable, 2018. Adresse: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5be7d8f6a&appId=PPGMS.

[ATK+18] P. Alefragis, G. Theodoridis, M. Katsimpris, C. Valouxis, C. Gogos, G. Goulas, N. Voros, S. Reder, K. Kasnakli, M. Bednara, D. Müller, U. Durak und J. Becker, „Mapping and Scheduling Hard Real Time Applications on Multicore Systems : The ARGO Approach“, in *Applied Reconfigurable Computing - Architectures, Tools, and Applications, Proceedings of the 14th International Symposium, ARC 2018, Santorini, Greece, 2nd - 4th May 2018. Ed.: C. Antonopoulos*, Ser. Lecture Notes in Computer Science, Bd. 10824, Springer, Cham, 2018, S. 700–711, ISBN: 978-3-319-78889-0. DOI: 10.1007/978-3-319-78890-6_56.

[DPA+17] S. Derrien, I. Puaut, P. Alefragis, M. Bednara, H. Bucher, C. David, Y. Debray, U. Durak, I. Fassi, C. Ferdinand, D. Hardy, A. Kritikakou, G. Rauwerda, S. Reder et al., „WCET-aware parallelization of model-based applications for multi-cores: The ARGO approach“, in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, März 2017, S. 286–289. DOI: 10.23919/DATE.2017.7927000.

[RHD+16] S. Reder, R. Heckmann, S. Derrien, T. ter Braak, L. Masing und G. Keramidas, „ARGO Deliverable 4.1: Specification of a WCET-aware abstract architecture description“, The ARGO project consortium, EU Project Deliverable, 2016. Adresse: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5afa7145e&appId=PPGMS.

[RRB+14] C. Roth, S. Reder, H. Bucher, O. Sander und J. Becker, „Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures“, in *17th Euromicro Conference on Digital System Design, DSD 2014; Verona; Italy; 27 August 2014 through 29 August 2014*, IEEE, Piscataway (NJ), 2014, S. 137–145, ISBN: 978-1-4799-5793-4. DOI: 10.1109/DSD.2014.62.

[RBR+13a] C. Roth, H. Bucher, S. Reder, F. Buciuman, O. Sander und J. Becker, „A SystemC modeling and simulation methodology for fast and accurate parallel MPSoC simulation“, in *26th Symposium on Integrated Circuits and Systems Design (SBCCI'13), Curitiba, Brazil, September 2-6, 2013*, IEEE, Piscataway (NJ), 2013, S. 1–6, ISBN: 978-1-47-991130-1. DOI: 10.1109/SBCCI.2013.6644853.

[RBR+13b] C. Roth, H. Bucher, S. Reder, O. Sander und J. Becker, „Improving parallel MPSoC simulation performance by exploiting dynamic routing delay prediction“, in *8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC'13), Darmstadt, July 10-12, 2013*, IEEE, Piscataway (NJ), 2013, S. 1–8, ISBN: 978-1-4673-6180-4. DOI: 10.1109/ReCoSoC.2013.6581524.

[RRS+12] C. Roth, S. Reder, O. Sander, M. Huebner und J. Becker, „A Framework for exploration of parallel SystemC simulation on the single-chip cloud computer“, in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques, Desenzano del Garda, Italy, March 19-23, 2012. Ed.: G. Riley*, ICST, Brussels, 2012, S. 202–207, ISBN: 978-1-4503-1510-4. DOI: 10.4108/icst.simutools.2012.247751.

[RRE+12] C. Roth, S. Reder, G. Erdogan, O. Sander, G. Almeida, H. Bucher und J. Becker, „Asynchronous Parallel MPSoC Simulation on the Single-chip Cloud Computer“, in *2012 International Symposium on System-on-Chip (SoC'12), Tampere, Finland, October 10-12, 2012*, IEEE, Piscataway (NJ), 2012, 8 S. ISBN: 978-1-4673-2895-1.

Index