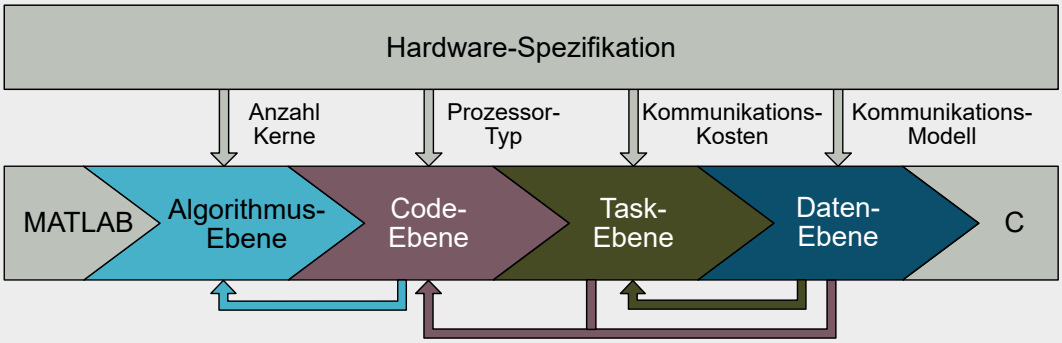


Oliver Wolf

Multigranulare Optimierung für heterogene Multicore-Systeme

Multigranulare Optimierung für heterogene Multicore-Systeme

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

Dipl.-Ing. Oliver Wolf (geb. Oey)

geboren in Konstanz

Tag der mündlichen Prüfung:
15.07.2024

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

Erklärung

Ich versichere wahrheitsgemäß, die Dissertation bis auf die dort angegebene Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Mannheim, den 23.05.2024

_____

Oliver Wolf

Zusammenfassung

In dieser Arbeit wird ein Verfahren vorgestellt, wie optimierter C-Code für eingebettete Mehrkernsysteme mit Hilfe einer ebenenübergreifenden Parallelisierung hinweg generiert werden kann. Ziel dabei ist es, die Algorithmenentwicklung von der Optimierung für das tatsächliche Zielsystem zu entkoppeln.

Zunächst werden die vier betrachteten Hierarchieebenen eingeführt und näher auf ihre Interaktionen eingegangen. Die Algorithmus-Ebene befasst sich mit der reinen Implementierung der Funktionalität und wie bekannte Eigenschaften von Algorithmen bereits frühzeitig ausgenutzt werden können. Auf der Code-Ebene wird die Darstellung des Algorithmus im Quellcode optimiert. Dazu kommen Code-Transformationen und andere Methoden zum Einsatz, um den Code anschließend besser analysieren und aufteilen zu können. Die Task-Ebene befasst sich mit der Verteilung der einzelnen Tasks auf die verfügbaren Ausführungseinheiten und die Daten-Ebene stellt schließlich die korrekte Synchronisation der Daten zwischen allen Kernen sicher.

Das Verfahren setzt auf die Entwicklung in der Sprache MATLAB® und die Erzeugung von C-Code für eingebettete Systeme. Im Anschluss wird näher auf diese Codegenerierung eingegangen, die notwendig ist, um die in MATLAB® vorliegenden Algorithmen in statisch analysierbaren C-Code zu konvertieren und nach den Optimierungsschritten den parallelen C-Code für die Zielplattform zu generieren. Fokus liegt auf der Herausforderung, wie aus einer Skriptsprache, bei der Datentypen und Variablengrößen dynamisch zur Laufzeit von einem Interpreter berechnet werden, effizienter C-Code generiert werden kann. Zusätzlich wird beschrieben, wie der korrekte Kontrollfluss einer Anwendung in einer parallelisierten Anwendung sichergestellt werden kann.

Im Folgenden wird die Realisierung der ebenenübergreifenden Parallelisierung genauer erläutert. Dazu wird zunächst die Spezifikation der Zielsysteme über eine Architekturbeschreibung eingeführt, um anschließend die wichtigsten Implementierungen je Ebene vorzustellen. Auf der Algorithmus-Ebene ist dies die Erkennung von bekannten Algorithmen und wie bekannte Eigenschaften ausgenutzt werden können. Die Parallelisierung auf Code-Ebene setzt den Fokus auf Code-Transformationen, die die intrinsische Parallelität der Anwendungen nutzbar machen. Auf der Task-Ebene wird auf die Verteilung von Tasks auf die verfügbaren Ausführungseinheiten eingegangen und welche Algorithmen für das Scheduling zum Einsatz kommen. Die Daten-Ebene befasst sich schließlich mit der Platzierung von Synchronisations- und Kommunikationsinstruktionen, um die korrekte und effiziente Ausführung der parallelisierten Anwendung sicherzustellen.

Schließlich wird das Verfahren mit drei Beispielanwendungen, zwei datenflusslastigen und einer kontrollflusslastigen Anwendung, evaluiert. Dazu werden die angewendeten Optimierungen auf den einzelnen Ebenen beschrieben, wie sie iterativ angewendet wer-

den und wie sich die Ebenen gegenseitig beeinflussen. Messungen auf verschiedenen Hardware-Plattformen runden die Ergebnisse ab.

Abschließend wird eine Zusammenfassung gegeben und mögliche zukünftige Erweiterungen beschrieben.

Abstract

This thesis presents an approach to generate optimized C code for embedded systems using an iterative multigranular approach across different levels of abstraction. The goal here is to decouple the algorithm development from the optimization for the actual target system.

First, the four hierarchical levels considered are introduced and their interaction is discussed in more detail. The algorithm level deals with the pure implementation of functionality and how known properties of algorithms can be exploited early on. At the code level, the representation of the algorithm in the source code is optimized. Code transformations and other methods are used for this purpose, in order to be able to analyze and split the application better afterwards. Finally, the task level deals with the distribution of the individual tasks to the available execution units and the data level ensures the correct synchronization of the data between all cores.

Next, the code generation steps necessary to convert the algorithms initially available in MATLAB® into statically analyzable C code and to generate the parallel C code for the target platforms after the optimization steps are discussed and solutions are presented. The challenges of generating efficient C code from a scripting language where data types and variable sizes are computed dynamically at runtime by an interpreter are discussed in more detail. In addition, it is discussed how the correct control flow of an application can be ensured in a parallelized application.

In the following, the realization of cross-level parallelization is described. For this purpose, the specification of the target systems is introduced via an architecture description, and then the most important implementations per level are presented. At the algorithm level, this is the detection of known algorithms and how known properties can be exploited. Parallelization at the code level focuses on code transformations that harness the intrinsic parallelism of applications. The task level deals with the distribution of tasks to the available execution units and which algorithms are used for scheduling. Finally, the data level deals with the placement of synchronization and communication instructions to ensure the correct and efficient execution of the parallelized application.

This approach is finally evaluated with three example applications, two data-flow-heavy and one control-flow-heavy. This is done by describing the optimizations applied at each level, how they are applied iteratively, and how the levels affect each other. Measurements on different hardware platforms complete the results.

Finally, a summary is given and possible future extensions are described.

Vorwort

Die vorliegende Arbeit begann während meiner Zeit als wissenschaftlicher Mitarbeiter am Institut für Technik der Informationsverarbeitung (ITIV) des Karlsruher Instituts für Technologie (KIT) und wurde während meiner Arbeit im mitgegründeten KIT-Spinoff emmtrix Technologies GmbH vollendet.

Mein erster Dank gilt meinem Doktorvater, Prof. Jürgen Becker. Ohne seine langjährige Unterstützung und wertvollen Tipps, die mir immer wieder die richtige Richtung gegeben haben, wäre diese Arbeit nicht möglich gewesen. Er hat mir stets den Freiraum gelassen, meine eigenen Ideen zu verfolgen, und mich dabei unterstützt, wo es nötig war.

Ein besonderer Dank gilt auch Dr. Timo Stripf, der durch unsere zahlreichen Diskussionen nicht nur viele wertvolle Ideen und Inhalte in diese Arbeit eingebracht hat, sondern mich auch stets inspiriert hat, neue Perspektiven zu betrachten.

Michael Rückauer möchte ich ebenfalls danken, der mir immer mit Rat und Tat zur Seite stand. Sein offenes Ohr und seine guten Ratschläge waren oft genau das, was ich brauchte, um Hindernisse zu überwinden und weiterzukommen.

Des Weiteren möchte ich Prof. Diana Göhringer und Prof. Michael Hübner danken, ohne deren Einfluss ich diese Arbeit wahrscheinlich nie durchgeführt hätte. Nach meinem Studium haben sie mich erst auf den Weg gebracht, eine Promotion anzustreben.

Ein besonderer Dank geht auch an Dr. Andreas Prell, der die Arbeit als einer der ersten gelesen hat und mit wertvollen Hinweisen die Qualität der Ausarbeitung einen guten Schritt nach vorne gebracht hat.

Der größte Dank gilt schließlich meiner Familie, die mir über all die Jahre den Rücken gestärkt hat. Vor allem möchte ich meine Frau Marlene hervorheben. Sie hat mir nicht nur den nötigen Antrieb gegeben, sondern auch die Geduld aufgebracht, meine Arbeitsphasen, sogar im Urlaub, zu ertragen. Ohne ihre Unterstützung und ihr Verständnis hätte ich diesen Weg nicht mehr erfolgreich beendet.

Mannheim, im Oktober 2024
Oliver Wolf

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation der Dissertation

Betrachtet man die Entwicklung von Prozessoren im Laufe der Zeit, kann beobachtet werden, dass sich die Integrationsdichte der Transistoren auf einem Chip etwa alle 24 Monate verdoppelt. Diese Beobachtung machte Gordon E. Moore bereits 1965 [1] und seine Prognose wurde als *Moore's Law* zum Leitsatz der Halbleitertechnologie. Wie in Abbildung 1.1 zu sehen ist, folgte die Entwicklung der Mikroprozessoren von 1970 bis 2021 diesem Trend.

Wurden ursprünglich Transistoren eingesetzt, um einen Prozessorkern immer leistungsfähiger und den Speicher immer größer zu machen, so geht der Trend bei eingebetteten Systemen immer mehr zu heterogenen Mehrkernsystemen. Diese bieten große Vorteile bei der Performanz sowie bei der Leistungsaufnahme, erfordern aber eine deutlich komplexere Programmierung. Die parallele Programmierung bringt dabei mehrere Herausforderungen mit sich:

- Das menschliche Denken ist grundsätzlich sequentiell, es folgt immer ein Gedanke auf den nächsten. Bei der Entwicklung von parallelen Anwendungen muss bereits im Vorfeld eine sinnvolle Aufteilung geschaffen werden, um alle Vorteile von Mehrkernsystemen nutzen zu können. Dies bedeutet einen erheblichen Mehraufwand, der bei sequentieller Entwicklung nicht anfällt und von der eigentlichen Programmierung des Algorithmus ablenkt.
- Bei der Verteilung auf mehrere Recheneinheiten müssen sowohl Daten- als auch Kontrollabhängigkeiten berücksichtigt werden. Dazu muss sichergestellt werden, dass durch Datentransfers jede Recheneinheit Zugriff auf alle benötigten Daten hat und dass es bei der Ausführung zu keinen Dead- oder Livelocks kommt, bei denen die eigentliche Berechnung nicht mehr zu Ende gebracht wird.
- Die Fehlersuche gestaltet sich bei Mehrkernsystemen ungleich komplexer, da durch die parallele Verarbeitung kein deterministisches Verhalten gewährleistet werden kann. Wird die Ausführung zu einem beliebigen Zeitpunkt unterbrochen, kann der aktuelle Zustand jeder Recheneinheit nicht eindeutig vorhergesagt werden.

Eine Lösung besteht nun darin, den Prozess der Parallelisierung zu vereinfachen und den Anwender durch Softwarelösungen zu unterstützen. Durch einen automatisierten Ansatz können viele Fehlerquellen umgangen und die Entwicklung deutlich beschleunigt werden. Damit der automatisch generierte Code auf der Zielplattform effizient ausgeführt werden kann, muss der Code jedoch bestimmte Anforderungen erfüllen und an die entsprechenden Gegebenheiten der Hardware angepasst werden.

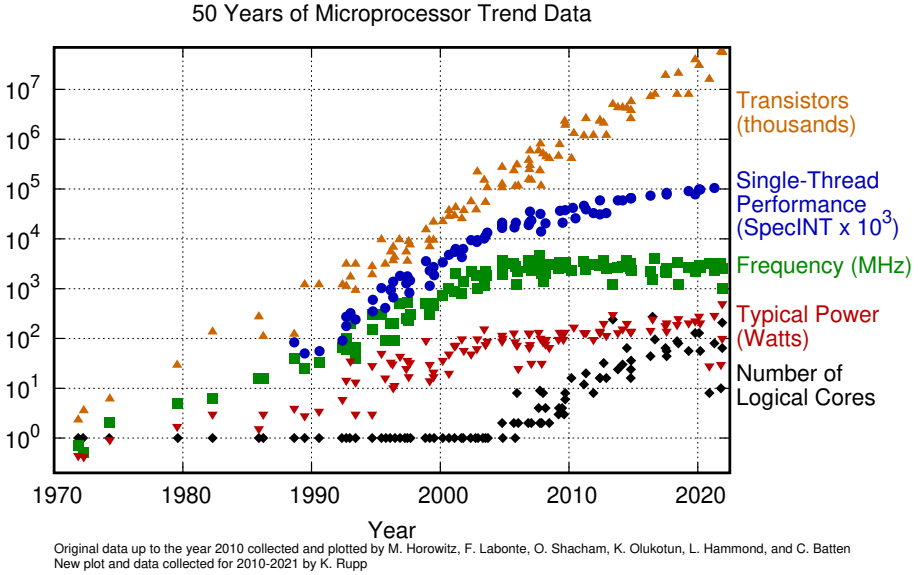


Abbildung 1.1.: Moore's Law anhand der Entwicklung der Transistoranzahl der letzten 50 Jahre [2]

1.2. Umfeld der Arbeit

Große Teile dieser Arbeit wurden im Rahmen des europäischen Projekts ALMA [3] durchgeführt. In diesem Projekt wurde eine Softwarelösung entwickelt, die Scilab-Code zunächst in sequentiellen C-Code transformiert, diesen dann mit grob- und feingranularen Ansätzen parallelisiert und schließlich parallelen C-Code für eine abstrakt beschriebene Zielarchitektur erzeugt. Die Ergebnisse wurden zur Gründung eines Spin-Offs genutzt und in der Firma emmtrix Technologies GmbH weiterentwickelt.

Im Rahmen dieser Arbeit wurden die verschiedenen Optimierungsschritte in Abstraktionsebenen eingeteilt und ein iteratives Vorgehen entwickelt, um von der rein abstrakten Algorithmendarstellung zum optimierten Quellcode für verschiedene Zielsysteme zu gelangen. Der Ansatz ist allgemein anwendbar, wird hier aber im Umfeld von eingebetteten Systemen betrachtet, die zusätzliche Anforderungen an Performanz, Reaktivität und Zuverlässigkeit stellen.

1.3. Aufbau der Dissertation

Diese Arbeit ist wie folgt gegliedert: Kapitel 2 beschreibt alle notwendigen Grundlagen dieser Arbeit bezüglich eingebetteter Systeme, der Darstellung von Algorithmen sowie Grundlagen zu Compilern, statischer Performanzabschätzung und Parallelisierung. Kapitel 3 definiert die Algorithmus-, Code-, Task- und Daten-Ebene und führt das Konzept der ebenenübergreifenden Parallelisierung ein. Kapitel 4 fasst alle notwendigen Codegenerierungsschritte zusammen, mit denen Algorithmen vom MATLAB®-Code über generischen C-Code bis hin zum parallelen C-Code für die Zielplattform übersetzt werden. Kapitel 5 stellt die Realisierung der einzelnen Ebenen sowie die ebenenübergreifenden Parallelisierung dar. In Kapitel 6 wird das Framework schließlich anhand von drei beispielhaften Algorithmen evaluiert, bevor die Arbeit in Kapitel 7 zusammengefasst und ein Ausblick gegeben wird.

2. Grundlagen

2.1. Eingebettete Systeme

Eingebettete Systeme sind gekennzeichnet durch ihre Spezialisierung auf bestimmte Aufgabengebiete und damit einhergehend einer Reduktion der verfügbaren Ressourcen auf das Nötigste. Klassische Aufgaben kommen aus den Gebieten der Überwachung, Steuerung oder Regelung. Als eingebettetes System wird dabei in der Regel ein Prozessor mit Speicher bezeichnet, der über Sensoren und/oder Aktoren mit der Außenwelt interagieren kann, wie es in Abb. 2.1 dargestellt ist. Gebräuchlich ist hierbei auch der Begriff des cyber-physischen Systems, das den Verbund zwischen der physischen Welt und der informationstechnischen Welt in den Vordergrund stellt. Oft arbeiten sie im Verborgenen in Geräten wie Kühlschränken, Waschmaschinen oder Automobilen. Ihre Verbreitung nimmt immer weiter zu und durch aktuelle Trends wie „Industrie 4.0“ oder das „Internet der Dinge“ wird ihre Entwicklung noch weiter angeheizt.

Die stetige Entwicklung sorgt dafür, dass eine immer höhere Performanz bei gleichzeitig niedrigerer Leistungsaufnahme erreicht werden kann, so dass sich neue Aufgabengebiete durch intelligentere Verarbeitung von Daten erschließen.

2.1.1. Anforderungen an die Softwareentwicklung bei eingebetteten Systemen

Eingebettete Systeme haben aufgrund ihrer typischen Anwendungsbereiche Einschränkungen und Anforderungen, mit denen die Systeme entworfen und betrieben werden müssen:

- Aufgrund der durch den Einsatzort vorgegebenen Abmessungen ergeben sich Einschränkungen hinsichtlich der Größe und damit der Anzahl der Komponenten. Prozessoren müssen eine möglichst kleine Fläche haben und Speicher ist nur begrenzt verfügbar.
- Die Stromversorgung ist oft begrenzt, so dass die Batterie oder der Akku möglichst lange halten muss. Aus diesem Grund muss die Hardware möglichst energieeffizient arbeiten und darf nur eine geringe Leistungsaufnahme haben.
- Bei eingebetteten Systemen kann es vorkommen, dass sie autonom, also ohne Zugriff auf das Internet oder ohne menschliches Eingreifen agieren müssen. Daher müssen sie robust und zuverlässig sein.
- Die Aufgaben, die eingebettete Systeme erfüllen sollen, werden technisch immer anspruchsvoller. Durch das Aufkommen des maschinellen Lernens gibt es viele An-

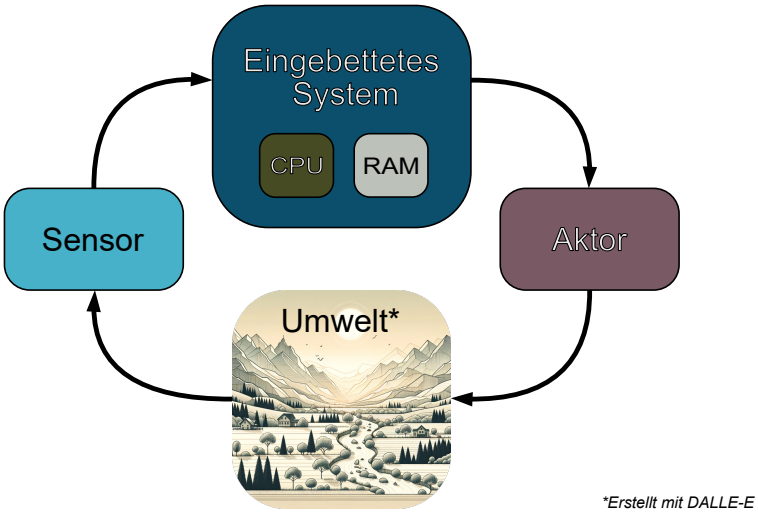


Abbildung 2.1.: Eingebettetes System

wendungsbereiche, in denen z.B. Muster in Sensordaten erkannt werden sollen, um dann verschiedene Aktionen auszulösen. Dies kann das Erkennen von Personen oder Hindernissen mit anschließenden Ausweichmanövern sein, kann aber auch auf Audio- oder andere Daten angewendet werden. Daraus ergeben sich Echtzeitanforderungen, da die entsprechende Reaktion immer innerhalb eines berechenbaren Zeitrahmens erfolgen muss. Neben dieser Latenz ergeben sich auch Anforderungen an den Durchsatz, damit alle eingehenden Sensordaten auch verarbeitet werden können.

Daraus ergeben sich folgende Anforderungen an die Softwareentwicklung für eingebettete Systeme:

- Hohe Performanz, damit die Aufgaben in der erwarteten Zeit erledigt werden können. Um Echtzeitanforderungen zu erfüllen, muss dazu die *maximale Ausführungszeit* (WCET, engl. *worst case execution time*) einer Anwendung auf der Zielhardware vorhersagbar sein. Aufgrund von Cache-Effekten, externen Einflüssen sowie komplexen Optimierungen der Compiler kann in der Regel keine exakte Zeit angegeben werden, sondern nur eine Abschätzung. Diese ist höher als die tatsächliche Ausführungszeit, wobei eine Vorhersage umso besser ist, je kleiner der abgeschätzte Zahlenwert ist [4], da dann die Differenz zur tatsächlichen WCET am geringsten ist. Auf die Berechnung der maximalen Ausführungszeit wird in dieser Arbeit nicht näher eingegangen, jedoch wird das Erreichen einer hohen Performanz als erstes Kriterium zum Erreichen der gewünschten Funktionalität herangezogen, um die Anforderungen an Latenz und Durchsatz zu erfüllen.
- Geringe Leistungsaufnahme zur Laufzeit: Die verfügbaren Ressourcen sollen möglichst effizient genutzt werden, um eine möglichst lange Laufzeit im Batteriebetrieb

zu gewährleisten. Auch wenn das Erreichen einer hohen Performance zunächst im Widerspruch zu einer effizienten Ausführung steht, können kürzere Berechnungszeiten dazu führen, dass das System insgesamt länger in energiesparenden Modi betrieben werden kann und somit die Laufzeit verlängert wird.

- Geringer Speicherverbrauch: Da große Speicher viel Platz benötigen, können in eingebetteten Systemen nur kleine Speicher verwendet werden. Um den verfügbaren Speicher besser zu nutzen, gibt es verschiedene Maßnahmen: Verwendung möglichst kleiner Datentypen, Reduzierung des Verwaltungsaufwands, Wiederverwendung von Speicherbereichen durch Freigeben von nicht mehr benötigtem Speicher und Entfernen von redundantem oder totem Code.
- Codequalität: Um die Anforderungen an Zuverlässigkeit und Robustheit zu erfüllen, muss sichergestellt werden, dass Fehler ausgeschlossen oder zumindest mögliche Fehlerquellen minimiert werden. Dabei ist zu beachten, dass parallele Systeme aufgrund ihrer Nebenläufigkeit prinzipbedingt fehleranfälliger sind als rein sequentielle Systeme. Automatisierung kann helfen, mögliche Programmierfehler von vornherein auszuschließen.
- Hardwareabstraktion: Um die Programmierung des Systems zu erleichtern, soll nur eine abstrakte Beschreibung der Eigenschaften der Zielhardware erforderlich sein. Ziel ist es, dass sich der Programmierer voll auf die Entwicklung des Algorithmus konzentrieren kann und Eigenheiten der Hardware automatisiert ausgenutzt werden können, ohne den Aufbau im Detail zu kennen.

2.1.2. Einsatz von Mehrkernprozessoren in eingebetteten Systemen

Um den Performanzanforderungen gerecht zu werden, findet auch im Bereich der eingebetteten Systeme ein Umschwung auf Mehrkernprozessoren statt. Diese Entwicklung konnte bereits in anderen Gebieten beobachtet werden und ist der Tatsache geschuldet, dass die technische Weiterentwicklung an physikalische Grenzen gestoßen ist. Wurde die Performanzsteigerung in der Vergangenheit noch maßgeblich von zwei Faktoren beeinflusst - der Strukturverkleinerung (und damit mehr Transistoren pro Fläche) und der Steigerung der Taktfrequenz -, so ist eine Steigerung des Taktes nicht mehr ohne Weiteres möglich. Die Entwicklung von Mehrkernprozessoren erlaubt es, bei gleichbleibender Leistungsaufnahme und gleicher Taktfrequenz die Rechenleistung zu vervielfachen.

Dies lässt sich anschaulich mit der α-Approximation aus [5] darstellen. Die dynamische Verlustleistung eines Prozessors ist in der Alpha-Darstellung:

$$P_{dyn} = \alpha C {U_S}^2 f$$

Dabei gilt:

- α ist der Aktivitätsfaktor, der mit einem Wert von 0 bis 1 angibt, wie häufig eine Schaltung aktiv ist. Eine 0 bedeutet dabei keine Aktivität und eine 1 die maximale Aktivität. Der tatsächliche Wert hängt unter anderem von der Art der Schaltung, der Anwendung sowie indirekt von der Taktfrequenz ab.

- P_{dyn} ist die dynamische Verlustleistung.
- C ist die Lastkapazität.
- U_S ist die Versorgungsspannung.
- f ist die Frequenz.

Zu beachten ist, dass auch die Frequenz von der Spannung abhängig ist:

$$f \propto \frac{(U_S - U_{th})^{\alpha}}{U_S}$$

In [6] wurde ein Single- und Dualcore ARM A9 bei gleicher Aufgabe verglichen. Der Singlecore Prozessor hatte dabei eine Frequenz von 1 GHz bei einer Spannung von 1,1 V. Der Dualcore Prozessor erreicht die gleiche Rechenleistung bei 550 MHz mit einer Spannung von 0,8 V. Dies erreicht er bei einer Leistungsaufnahme, die nur 60% des Singlecore Prozessors beträgt. Die Aufteilung auf mehrere Recheneinheiten benötigt immer einen größeren Aufwand und ist nicht immer ideal möglich. Beschränkt wird sie durch den sequentiellen Anteil im Programm, welcher nach Amdahls Gesetz [7] wie folgt dargestellt werden kann:

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Dabei ist $n \in \mathbb{N}$ die Anzahl an parallelen Ausführungseinheiten, $B \in [0, 1]$ der sequentielle Anteil am Algorithmus und $S(n)$ die theoretische Beschleunigung (engl. *speedup*), die durch eine parallele Ausführung erreicht werden kann. Programme haben immer einen rein sequentiellen Anteil, da Aufgaben wie Prozessinitialisierung oder Speicherallokation nur einmalig auf einem Prozessor ausgeführt werden müssen. Die tatsächlich erreichte Beschleunigung wird durch weitere Parameter, wie den Overhead für Synchronisierung und Kommunikation, noch weiter reduziert und ist in der Praxis noch kleiner.

2.1.2.1. Speicherarchitekturen

Abhängig von den Zugriffsmöglichkeiten der einzelnen Prozessorkerne auf den Speicher können verschiedene Speicherarchitekturen unterschieden werden.

Geteilter Speicher Von geteiltem Speicher spricht man, wenn der gesamte Adressbereich zwischen allen Prozessoren geteilt wird. Zugriffe auf gemeinsam genutzte Bereiche müssen arbitriert werden, damit es nicht zu ungewollten Veränderungen an Daten kommen kann. Da Daten für alle Kerne verfügbar sind, ist eine explizite Kommunikation nicht notwendig. Ein einfacher Aufbau ist in Abbildung 2.2 dargestellt.

Häufig kommen lokale Caches beim Zugriff auf den Hauptspeicher zum Einsatz, um die höheren Latenzen des Hauptspeichers zu kaschieren. Dabei ist es notwendig, dass sichergestellt wird, dass alle Kerne immer mit den korrekten Daten arbeiten. Die sogenannte Cache-Kohärenz kann dabei sowohl in Hardware als auch in Software erreicht werden.

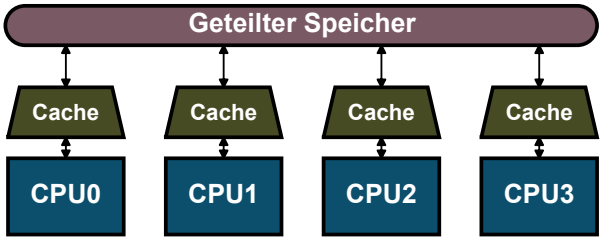


Abbildung 2.2.: Geteilter Speicher

Im Hinblick auf Manycore-Architekturen ist die Kohärenz zwar ein komplexes Aufgabengebiet, es kann aber gezeigt werden, dass das Problem nicht exponentiell mit der Anzahl der Kerne skaliert [8].

Die Verwendung von ausschließlich geteiltem Speicher ist aufgrund von sehr schlechter Skalierung der Latenzen nicht möglich, so dass die Skalierung durch den Overhead von zusätzlichen Caches oder Speichern sowie deren Verwaltung eingeschränkt wird.

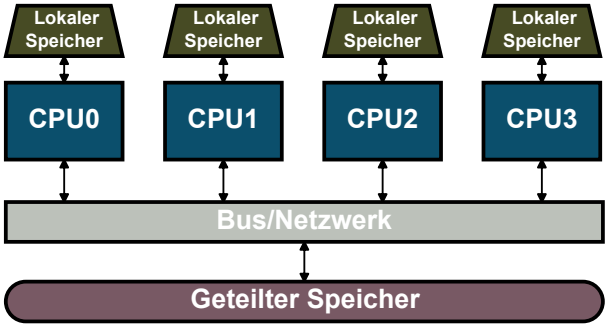


Abbildung 2.3.: NUMA-Speicherarchitektur

Nicht-gleichförmige Speicherzugriffe Bei den *nicht-gleichförmigen Speicherzugriffen* (NUMA, engl. *non-uniform memory access*) handelt es sich um eine Erweiterung von verteiltem Speicher um lokale Speicherbereiche. Diese können zu einem Kern oder einem Cluster gehören und können von anderen Kernen oder Clustern gelesen und mitunter auch geschrieben werden. Der Zugriff erfolgt über ein geteiltes Medium, beispielsweise einen Bus oder ein Netzwerk. Kennzeichnend für diese Art der Speicherverwaltung sind verschiedene Zugriffszeiten auf die unterschiedlichen Adressbereiche. So ist der Zugriff auf den lokalen Speicher recht schnell, Zugriffe auf Speicher von anderen Prozessoren oder den geteilten Hauptspeicher dauern deutlich länger. Dieser Ansatz erlaubt es, die Vorteile von unterschiedlichen Speichertechnologien auszunutzen und kann die Sicherstellung der Datenkohärenz vereinfachen, indem sichere Daten immer einem Prozessor

zugeordnet werden können. Ein Beispielaufbau ist in Abbildung 2.3 dargestellt. Jeder der vier CPU-Kerne hat seinen eigenen, lokalen Speicherbereich und kann über die Bus-/Netzwerk-Schnittstelle auf die Speicher der anderen Prozessoren sowie den geteilten Speicherbereich zugreifen.

Der Ansatz skaliert gut mit der Anzahl der Kerne, lediglich der Overhead durch die Verwaltung der Daten sowie die generelle Programmierbarkeit unter Beachtung des Speicheraufbaus kann Probleme bereiten.

Im Rahmen einer Studienarbeit wurde ein Algorithmus zur effizienten Nutzung von schnellem lokalem Speicher evaluiert. Eine Zusammenfassung des Verfahrens ist in Anhang A.1 dargestellt. Auch wenn die Ergebnisse vielversprechend sind, hat sich gezeigt, dass das Thema der Speicheroptimierung zu komplex ist, um es zusätzlich zur effizienten Nutzung der Ausführungseinheiten zu betrachten. Im Weiteren werden also keine speziellen Verfahren zur effizienteren Nutzung des vorhandenen Speichers betrachtet.

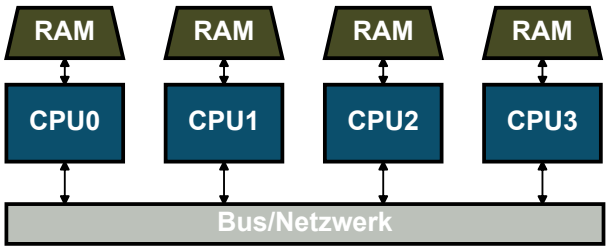


Abbildung 2.4.: Verteilter Speicher

Verteilter Speicher Von verteiltem Speicher spricht man, wenn jeder Kern ausschließlich seinen eigenen Speicherbereich hat. Dies führt zwei neue Probleme ein: Datenpartitionierung und Kommunikation. Da Daten nur im Speicher von einem Prozessor vorliegen, braucht jeder Prozessor entsprechende eigene Kopien mit denen er arbeiten kann. Änderungen an diesen Werten müssen durch explizite Kommunikation übermittelt werden. Die Skalierung ist besser als bei den anderen Strukturen, gerade bei Betrachtung von Manycore-Systemen. Dabei müssen aber einige Dinge beachtet werden: durch die Datenpartitionierung ist bekannt, welche Prozessoren auf welchen Daten arbeiten. Sämtliche gemeinsame Zugriffe müssen dementsprechend vorher geplant werden. Dies ist einfacher als bei einem Cache, dessen Verhalten nur schwer vorhersagbar ist. Ein beispielhafter Aufbau ist in Abbildung 2.4 dargestellt. Jeder der vier Kerne hat seinen eigenen RAM, auf den keiner der anderen Kerne direkt zugreifen kann. Auch gibt es keinen gemeinsamen Speicherbereich, den alle Kerne gleichzeitig verwenden können. Die Programmierung ist im Vergleich mit den anderen Architekturen aufwändiger, da sämtliche Synchronisation zwischen den Prozessoren explizit ausgeführt werden muss. Andere Architekturen können mit auf diese Weise geschriebenen Programmen ebenfalls angesteuert werden, wobei die explizite Kommunikation dann durch einfache Speicherzugriffe oder Synchronisierungsmöglichkeiten ersetzt werden können.

Ohne Beschränkung der Allgemeinheit wird im weiteren Verlauf dieser Arbeit von Systemen mit verteiltem Speicher ausgegangen. Jedes System mit geteiltem Speicher kann wie ein System mit verteiltem Speicher betrachtet werden, indem der vorhandene gemeinsame Speicher zwischen den Kernen verteilt wird und in der Software sämtliche Zugriffe auf Variablen über Funktionsaufrufe und Kopien von Variablen erfolgen. Caches werden als lokal für jeden Kern betrachtet und es wird angenommen, dass es keine Cache-Kohärenz gibt, so dass Daten in jedem Fall übertragen werden müssen. Systeme mit geteilten Caches oder ausgeklügelten Maßnahmen zur Erreichung von Cache-Kohärenz können andere Optimierungen erfordern und das Potenzial für eine höhere Leistung haben, werden aber im Folgenden nicht explizit betrachtet. Die maximal erreichbare Performanz wird zwar durch zusätzliche Kopieroperationen zwar reduziert, der flexiblere Umgang mit verschiedensten Zielsystemen ohne spezielle Betrachtung des Speichersystems wird im Rahmen dieser Arbeit jedoch höher gewichtet.

2.1.2.2. Abgrenzung zu High-Performance-Computing

Der Einsatz von vielen Recheneinheiten ist kein neues Konzept und wurde im Bereich der Supercomputer bereits zu Beginn der 90er Jahre des letzten Jahrhunderts eingeführt. Dabei wurden sogenannte *Rechnerverbunde* (HPC-Cluster, engl. *high-performance computing cluster*) entwickelt, bei denen mehrere Computer über Netzwerke verbunden wurden. Da dies den Kommunikationsoverhead zwischen verschiedenen Rechnern stark erhöht, haben sich zwei Arten an Workloads etabliert: *Embarrassingly Parallel* und eng gekoppelte Workloads.

Unter *Embarrassingly Parallel* versteht man die parallele Verarbeitung von Aufgaben, die unabhängig voneinander sind. Eine sehr hohe Auslastung wird dadurch erreicht, dass mehr Rechenaufgaben als tatsächliche Recheneinheiten/-cluster zur Verfügung stehen. Die Parallelisierung dieser Art von Workloads ist trivial, findet sich selten in dieser Form auf eingebetteten Systemen wieder und wird in dieser Arbeit nicht weiter betrachtet.

Unter eng gekoppelten Workloads versteht man eine große Berechnung, die dynamisch in kleinere Aufgaben unterteilt und parallel abgearbeitet wird. Damit einher geht ein großer Bedarf an Synchronisation zwischen den einzelnen Bereichen. Workloads dieser Art haben sehr viele dynamische Entscheidungen, die zur Laufzeit getroffen werden, so dass die Ausführungsdauer sowie der Bedarf an Rechenleistung und Speicher nur sehr schwer abschätzbar sind. Auch diese Art von Aufgaben werden in der Regel nicht auf eingebetteten Systemen ausgeführt und werden im Weiteren nicht weiter betrachtet.

Neben den typischen Anwendungen unterscheidet sich auch die Hardware grundlegend. HPCs sind vorrangig auf die erreichbare Performanz optimiert und können bei der Anzahl an Kernen, dem Speicher und der Leistungsaufnahme aus dem Vollen schöpfen. Stand Juni 2022 besitzt der erstplatzierte Rechencluster von www.top500.org, der Frontier - HPE Cray EX235a (https://www.top500.org/system/180047/) 8 730 112 Rechenkerne sowie eine Leistungsaufnahme von 21 100 kW. Hochperformante eingebettete Prozessoren wie der Kalray MPPA®Manycore (https://www.kalrayinc.com/products/mppa-technology) bestehen aus 80 Rechenkernen, von denen auch mehrere gleichzeitig in einem System oder einem Chip vereint werden können. Für den eingebetteten Bereich kommen typischerweise maximal rund 250 Kerne mit einer Leistungsaufnahme bis zu 15 W in Frage.

2.1.2.3. Zukünftige Entwicklungen

Der Einsatz von Multi- und Manycore-Architekturen wird in Zukunft noch weiter zunehmen. Zudem gibt es eine Entwicklung hin zu heterogenen Systemen. Hierbei kommen Beschleuniger für bestimmte, häufig genutzte Funktionen zum Einsatz. Bewährt haben sich dabei *programmierbare Grafikbeschleuniger* (general purpose graphic processing unit, engl. *GP-GPU*), Vektorprozessoren sowie Beschleuniger, die auf maschinelles Lernen abzielen. Diese Arten heterogener Systeme können auf verschiedene Weisen adressiert werden und bringen viele Herausforderungen bei der Aufteilung von Aufgaben zwischen der Hard- und der Software mit sich. Im Rahmen dieser Arbeit wird dabei nur ein kleiner Teilbereich betrachtet: die Programmierung von Beschleunigern in Sprachen, die sehr ähnlich zu C sind (wie OpenCL). Ziel ist es zu zeigen, dass die hier erarbeiteten Konzepte ebenfalls auf heterogenen Systemen zielführend eingesetzt werden können und sie nicht auf reine Mehrkernsysteme beschränkt sind.

2.2. Darstellung von Algorithmen

Es gibt viele Möglichkeiten, einen Algorithmus zur Lösung einer bestimmten Aufgabe darzustellen. Angefangen von sehr abstrakten Darstellungen wie z.B. mittels Pseudocode oder einer grafischen Modellierung wie Simulink oder Stateflow über mathematische Formen wie MATLAB®oder Mathematica bis hin zu Programmiersprachen. Ziel dieser Arbeit ist es, Code für Algorithmen an eingebettete Systeme anzupassen. Dazu sollen zunächst verschiedene Repräsentationsformen evaluiert und hinsichtlich ihres Aufwandes für die Portierung bis zur Ausführung auf dem System verglichen werden.

Im Rahmen dieser Arbeit wird für Algorithmen auf die gängige Definition von Rogers Jr. [9] zurückgegriffen: „*. . . an algorithm is a clerical (. . . deterministic . . .) procedure which can be applied to any of a certain class of symbolic inputs and which will eventually yield, for each such input, a corresponding symbolic output.*“ Sie können also mathematisch als eine Funktion aufgefasst werden, die auf Eingangsdaten angewendet wird, um bestimmte Ausgangsdaten zu erhalten. Entscheidend ist, dass das Verfahren deterministisch ist, d.h. bei jeder Ausführung immer gleich abläuft.

2.2.1. Abstrakte Darstellung

Gemäß des Satzes von Böhm-Jacopini [10] können Algorithmen mit Hilfe von drei Elementen beschrieben werden: Sequenzen von Anweisungen, Verzweigungen des Kontrollflusses und Schleifen, die Unterprogramme ausführen, bis eine Bedingung erfüllt ist. Für die reine Darstellung eines Algorithmus haben sich die folgenden drei Möglichkeiten bewährt:

Pseudocode: Von Pseudocode spricht man, wenn ein Algorithmus nicht mit Umgangssprache beschrieben wird, sondern man sich am Aufbau einer formalen Programmiersprache orientiert. Im Vordergrund steht dabei die Verwendung einer bekannten Struktur und nicht die Erstellung eines korrekten Programms in der Sprache. Gebräuchlich sind dabei `if` für Verzweigungen und `while` bzw. `for` für Schleifen. Dabei wird meist auf Klammern

verzichtet und stattdessen auf Schlüsselwörter wie `begin`, `then` oder `end` gesetzt, um den Anfang und das Ende von Verzweigungen oder Schleifen zu kennzeichnen. Häufig wird die Sprache C oder Pascal als Basis für den Pseudocode verwendet, dies ist aber nicht zwingend festgelegt. Da nicht alle Regeln der Programmiersprache beachtet werden, eignet sich der Code nicht dazu, den Algorithmus als Programm auf einem Computer auszuführen, sondern wird eingesetzt, um den Ablauf von Programmen anderen Menschen zu vermitteln.

Struktogramm: ein Struktogramm, auch Nassi-Schneidermann-Diagramm genannt, ist eine grafische Darstellung von Algorithmen, bei der die Struktur des Programms und damit insbesondere die Verschachtelungstiefe von Schleifen und Verzweigungen dargestellt wird. Sequenzen werden dabei durch übereinander liegende Blöcke dargestellt. Eine Verzweigung wird durch nebeneinander liegende Blöcke abgebildet. Bei Schleifen wird über die Dauer der Schleife ein Teil des Blocks vertikal verlängert, so dass alle Elemente, die innerhalb der Schleife liegen, nach rechts eingerückt sind. Eine Beispieldarstellung anhand einer Matrixmultiplikation ist in Abbildung 2.5 dargestellt. Das Diagramm kann nicht ohne weiteres in ein Programm überführt werden, da nicht klar festgelegt ist, auf welche Weise die einzelnen Elemente (z.B. Bedingungen) beschrieben werden müssen. Es wird daher nur als abstrakte Darstellung verwendet, um sich einen schnellen Eindruck über die Programmstruktur machen zu können.

Flussdiagramm: ein Flussdiagramm, auch als *Programmablaufplan* (PAP) oder Programmstrukturplan bekannt, ist ebenfalls eine grafische Darstellung von Algorithmen, setzt den Fokus aber mehr auf den Kontrollfluss. Die Reihenfolge wird explizit durch Pfeile angegeben, so dass eine Sequenz durch aufeinanderfolgende Blöcke dargestellt wird, die mit Pfeilen verbunden sind. Eine Verzweigung wird dargestellt, indem ein durch eine Raute dargestellter Bedingungsblock mehrere Ausgangspfeile hat. Eine Schleife wird schließlich dargestellt, indem ein Pfeil wieder zurück auf einen Bedingungsblock zeigt, der bereits vorher durchlaufen wurde. Die Grundelemente sind in Abbildung 2.6 dargestellt. Der Kontrollfluss wird von einem Startknoten hin zum Endknoten (grüne Ovale) dargestellt. Dazwischen können Operationen (blaue Rechtecke), Entscheidungen (orange Raute), Subprozesse (gelbe Rechtecke) und Datenblöcke (dunkelblaues Parallelogramm) verwendet werden. Auch diese Darstellung wird in der Regel nicht dazu verwendet, um den dargestellten Algorithmus auf einem Computer auszuführen, sondern nur, um den Fluss für Menschen einfacher darzustellen.

Sollen Algorithmen nun für Maschinen verständlich beschrieben werden, so existieren auch hier verschiedene Ansätze. Als grobe Unterteilung sollen im Folgenden zunächst Programmiersprachen als auch abstraktere bzw. grafische Darstellungen von Algorithmen vorgestellt werden. Eine Übersicht sortiert anhand der Abstraktion der Hardware und der Entfernung von der rein mathematischen Beschreibung ist in Abbildung 2.7 dargestellt.

2.2.2. Programmiersprachen

Programmiersprachen sind ein sehr umfassendes Thema und sollen deshalb im Folgenden in drei grobe Bereiche eingeteilt werden. Im Rahmen dieser Arbeit ist die Ausfüh-

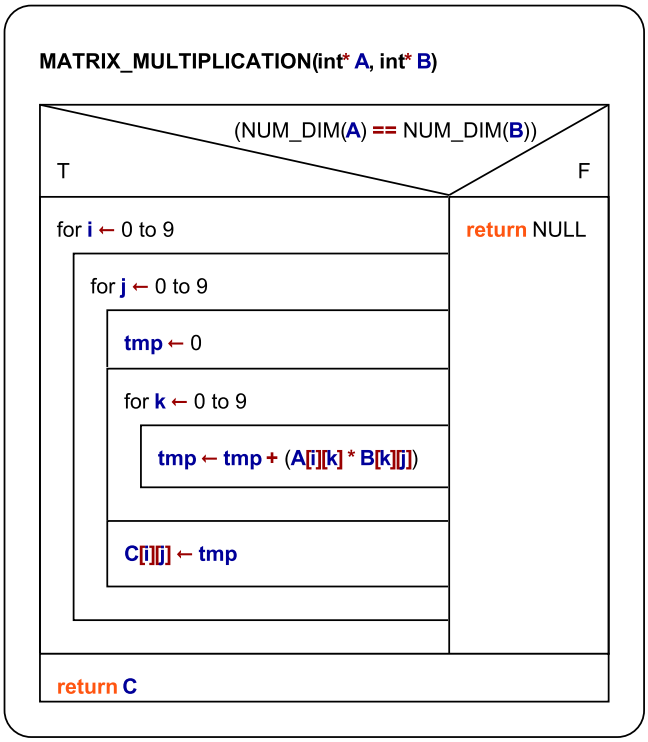


Abbildung 2.5.: Matrixmultiplikation als Struktogramm

rung auf eingebetteten Systemen relevant, weshalb die Einteilung zwischen klassischen Programmiersprachen und Skriptsprachen gemacht wird.

2.2.2.1. Klassische Programmiersprachen

Als klassische Programmiersprachen werden in dieser Arbeit alle Sprachen zusammengefasst, bei denen das geschriebene Programm von einem Compiler in Maschinencode übersetzt wird. Zu den verbreitetsten gehören dabei C/C++, Java und C#.

Es werden verschiedene Programmierparadigmen unterschieden, wobei moderne Programmiersprachen häufig mehr als ein Paradigma unterstützen.

Imperativ: Unter imperativer Programmierung versteht man die Programmierung über Anweisungen, die vom Computer nacheinander abgearbeitet werden. Sie ist das älteste Paradigma und wurde bereits bei den ersten Computern verwendet. Zur Strukturierung von Programmen wurde das Konzept von Funktionen eingeführt, die Codeteile in eigene Unterblöcke auslagert. Auf diese Weise kann bereits geschriebener Code wiederverwendet werden, indem die Funktion an anderer Stelle erneut aufgerufen wird. Dies ermög-

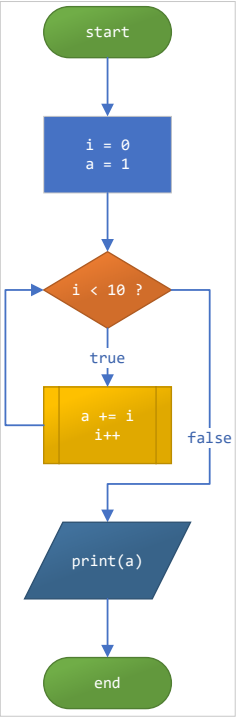


Abbildung 2.6.: Grundelemente eines Flussdiagramms

	Hardwarenah	Teilweise hardwarenah	Abstrahiert von Hardware
Mathematische Beschreibung			• Computer Algebrasystem (Mathematica, Maple) • Array-basierte Programmiersprachen (MATLAB®, Scilab)
Teilweise mathematische Beschreibung	Imperative Programmiersprachen (C, FORTRAN)	Objektorientierte Programmiersprachen (C++, JAVA)	Grafische Modellierung (Simulink, LabVIEW)
Abstrakte Darstellung	Assembler		Funktionale Programmiersprachen (Haskell)

Abbildung 2.7.: Darstellung von mathematischen Algorithmen

licht es zudem, bewährten Code in Bibliotheken auszulagern, die von anderen Programmierern eingesetzt werden können. Ein Beispiel ist die C-Standard-Bibliothek, welche verschiedene Funktionen zur Speicherverwaltung oder Datenumwandlung bereitstellt.

Objektorientiert: Bei der objektorientierten Programmierung wird das Konzept von Klassen angewendet, um Daten und Methoden in Objekten zu kapseln. Dies steigert die Wiederverwendbarkeit von Code noch weiter, da einfacher bereits existierende Klassen für verschiedene Aufgaben verwendet werden können. Zusätzlich ist es mit der sogenannten Vererbung möglich, Funktionalität von existierenden Klassen weiterzuverwenden und mit eigenen Anpassungen in neue Klassen zu bringen.

Funktional: Die funktionale Programmierung setzt auf mathematische Funktionen, um Programme darzustellen. Eine Funktion ist demnach eine Sammlung an Anweisungen, die auf die Eingangsdaten angewendet werden und die entsprechenden Ausgangsdaten erzeugt. Das Paradigma wurde aus akademischer Forschung heraus entwickelt und in Programmiersprachen überführt. Zu den bekannteren gehören Haskell und F#. Der funktionale Ansatz bietet durch seine Abstraktion Vorteile bei der Konzeption und Wartbarkeit von Programmen. Dies kommt aber zu Lasten einer schlechteren Performanz unter zeitkritischen Anwendungen.

Wie bereits in Abbildung 2.7 dargestellt, ist die imperative Programmierung noch sehr hardwarenah, da im einfachsten Fall die tatsächliche Ausführung auf einem Prozessor der sequentiellen Abarbeitung von Anweisungen entspricht. Dies ist nicht mehr der Fall, sobald parallele Architekturen oder Prozessoren mit Ausführungspipelines, die das umsortieren von Anweisungen ermöglichen, zum Einsatz kommen. Die Programmierung ist noch sehr nah an der rein mathematischen Beschreibung des Algorithmus, der im Allgemeinen auch als sequentielle Anweisungen vorliegt.

Bei der objektorientierten Programmierung wird in der Regel bereits weniger hardwarenah programmiert, da ansonsten einzelne Klassen für verschiedene Architekturen optimiert werden müssten, was jedoch der allgemeinen Nutzung widerspricht. Üblicherweise wird auf abstrakterer Ebene mit Klassen und ihren Daten sowie Methoden programmiert. Dies sorgt zudem dafür, dass man sich auch von der mathematischen Beschreibung entfernt, da Objekte in der Regel komplexer als einfache mathematische Konstrukte wie Matrizen oder Mengen sind.

Die funktionale Programmierung abstrahiert noch weiter und sorgt dafür, dass sich der Aufbau und die Syntax der Programme von der tatsächlichen Ausführung auf der Hardware unterscheidet. Dies erleichtert die Entwicklung von Algorithmen, sorgt aber dafür, dass die Ausführung auf der Hardware in der Regel nicht so performant ist, da ein größerer Overhead zur Unterstützung der Funktionen benötigt wird. Um eine gute Ausführungsgeschwindigkeit zu erreichen, sind häufig einige zusätzliche Schritte und tatsächliches Profiling notwendig [11]. Zu den bekanntesten funktionalen Sprachen gehören Haskell [12] und Scheme [13].

2.2.2.2. Skriptsprachen

Im Rahmen dieser Arbeit wird unter Skriptsprache eine Programmiersprache verstanden, die zur Ausführung einen Interpreter benötigt. Im Gegensatz zu den klassischen

Programmiersprachen werden die Skripte vor der Ausführung nicht in Maschinencode übersetzt, sondern ein Interpreter analysiert den Quellcode zur Laufzeit und führt den entsprechenden Code aus. Dies bietet große Vorteile bei der Entwicklung, da hardwarerelevante Teile wie Speicherverbrauch oder Initialisierungscode nicht in der Sprache dargestellt werden müssen, sondern dynamisch vom Interpreter durchgeführt werden. Diese Dynamik sorgt dafür, dass mehr Speicher benötigt wird und in der Regel nicht die gleiche Performanz erreicht wird, wie mit direkter Programmierung. Da der Interpreter zudem auch zusätzlich zum eigentlichen Programm auf dem Zielsystem ausgeführt werden muss, sind Skriptsprachen keine gute Wahl, um Programme zu entwickeln, die auf Hardware mit begrenzten Ressourcen ausgeführt werden sollen. Die bekanntesten Vertreter sind JavaScript [14], Python [15] und Perl [16].

2.2.2.3. Arraybasierte Programmiersprachen

Unter dem Begriff *Arraybasierte Programmiersprachen* werden in dieser Arbeit die Sprachen der Entwicklungsumgebungen MATLAB®[17] und Scilab [18] zusammengefasst. Sie wurden entwickelt, um numerische Berechnungen mit wenigen Zeilen an Code beschreiben zu können. Es sind Skriptsprachen, können aber auch den Programmiersprachen der vierten Generation [19] zugeordnet werden, die durch ihre Spezialisierung auf bestimmte Anwendungsgebiete und möglichst effizienten Quellcode gekennzeichnet werden.

MATLAB® wird von MathWorks® vertrieben und bildet zusammen mit seiner modellbasierten Entwicklungsumgebung Simulink den Quasi-Standard bei der modellgestützten Entwicklung von Algorithmen, die auf numerischen Verfahren beruhen. Es ist weit verbreitet in der Algorithmenentwicklung, der Simulation und in der Forschung.

Scilab ist eine quelloffene Software, die ebenfalls darauf abzielt, numerische Berechnungen zu vereinfachen. Es wird seit 2012 von Scilab Enterprises weiterentwickelt und nutzt eine Syntax, die sehr ähnlich wie die von MATLAB® ist. Im Folgenden werden grundlegende Unterschiede im Vergleich zu klassischen Programmiersprachen wie C betrachtet, die die Basis für die Codegenerierung in Abschnitt 4.3.1 bilden.

Ein wichtiger Aspekt bei der Entwicklung dieser Sprachen war, dass die Syntax möglichst nah an der tatsächlichen mathematischen Beschreibung des Problems angelehnt ist. Aus dieser Anforderung heraus ergab sich der generelle Aufbau: es sind imperative Sprachen, bei denen Befehle nacheinander abgearbeitet werden. Datentypen müssen nicht explizit spezifiziert werden, sondern werden von einer Laufzeitumgebung interpretiert. Da dies einen großen Overhead generiert, werden häufig verwendete Funktionen mittels Just-in-time-Kompilierung effizient bereitgestellt. Dazu wird auf eine effiziente Berechnung als C-Code zurückgegriffen. Häufig werden die Sprachen von Physikern und Mathematikern verwendet, die als Experten auf ihren jeweiligen Gebieten nur wenig Kenntnis von der Hardware haben, auf denen die Algorithmen später ausgeführt werden. Für die Performanz-Optimierung auf der Zielhardware stehen nur wenige Möglichkeiten zur Verfügung, da dies nicht im Fokus der Sprachen liegt.

Datentypen

Sowohl in MATLAB® als auch in Scilab müssen Datentypen nicht explizit angegeben werden. Der standardmäßig verwendete Datentyp ist eine Fließkommazahl mit doppelter Genauigkeit (double), die 64 Bit an Speicher benötigt. Es werden aber auch andere, aus gängigen Programmiersprachen bekannte Datentypen unterstützt. So gibt es neben den Fließkommazahlen mit einfacher Genauigkeit noch ganze Zahlen (Integer) mit 8, 16, 32 und 64 Bit und jeweils mit und ohne Vorzeichen. Diese Zahlendarstellungen müssen jedoch explizit angegeben werden und können mit Hilfe von Funktionen ineinander überführt werden. Zudem werden neben den reellen Zahlen $\mathbb{R}$ noch die imaginären Zahlen unterstützt.

Zum Einsatz kommt zudem eine dynamische Typisierung, bei der der Datentyp durch Zuweisung einer Variablen zur Laufzeit angepasst wird. Wird beispielsweise einer Variablen, die eine ganze Zahl mit 8 Bit darstellt, eine Fließkommazahl zugewiesen, so wird der Typ der Variablen ebenfalls zu einer Fließkommazahl. Auf diese Weise können Zuweisungen, die bei einer statischen Typprüfung für Fehler sorgen würden, bei einer dynamischen Typprüfung dennoch durchgeführt werden.

Die aus anderen Programmiersprachen mächtigen Zeiger (Pointer), die auf bestimmte Adressen zeigen können, existieren weder in MATLAB® noch in Scilab. Aufgrund der Zielsetzung der Sprachen für Anwender mit mathematischem Hintergrund wären Zeiger, deren Benutzung technisches Verständnis der Architektur benötigt, nicht hilfreich und würden das Erlernen der Sprachen erschweren.

Vektoren und Matrizen

Alle Variablen sind standardmäßig Matrizen, im einfachsten Fall entspricht ein skalarer Wert einer 1x1-Matrix. Vektoren sind eine Sonderform einer Matrix, bei der nur eine Dimension genutzt wird. Ähnlich wie der Datentyp ist dabei die Größe und die Form einer Matrix nicht fix und kann zur Laufzeit verändert werden. Sobald auf ein Element einer Matrix zugegriffen wird, das außerhalb der aktuellen Größe liegt, wird die Matrix dynamisch vergrößert, um den Zugriff zu ermöglichen. Ein Zugriff kann dabei ein einfacher Lesezugriff über einen Index sein, aber auch eine mathematische Operation oder Funktion, bei der die Matrix mit Matrizen anderer Größe verrechnet wird. Alle neuen Elemente einer Matrix werden dabei mit Nullen aufgefüllt.

Funktionen

Funktionen können, wie in gängigen Programmiersprachen üblich, verwendet werden, um wiederkehrende Aufgaben zu kapseln und das Wiederholen von Code zu verhindern. Funktionsparameter werden dabei zwischen Eingangs- und Ausgangsparametern unterschieden. Eingangsparameter werden als Wertparameter (engl. *call by value*) übergeben; es wird also eine Kopie der Variablen erzeugt und Zuweisungen innerhalb der Funktion wirken sich nicht auf das Original aus. Funktionen haben keinen eigenen Datentyp und können einen Rückgabewert nicht direkt erzeugen. Alle Ergebnisse müssen über die

Ausgangsparameter nach außen gegeben werden oder über das Schlüsselwort *return* an Variablen übergeben werden, die dann auch außerhalb der Funktion zur Verfügung stehen.

2.2.2.4. Hardwarespezifische Sprachen

Neben den klassischen Prozessoren existieren eine Vielzahl an anderen Hardware-Architekturen, die spezieller für eine Aufgabe optimiert sind. So können digitale Signalprozessoren (DSPs) deutlich effizienter große Datenmengen verarbeiten, indem beispielsweise die vielgenutzte multiply-accumulate Instruktion in einem einzelnen Taktschritt verarbeitet werden kann. Grafikkarten (GPUs) sind für die Verarbeitung von großen Matrizen bzw. Bildern optimiert. Zur effizienten Programmierung dieser Hardware wird in der Regel auf eigene Programmierkonstrukte gesetzt, die als Erweiterung für etablierte Sprachen entwickelt werden. So sind die bei der Programmierung von GPUs verbreiteten Sprachen CUDA [20] und OpenCL [21] Erweiterungen der Sprache C. Dies birgt zwar den Vorteil, dass keine ganz neue Sprache erlernt werden muss, hat aber immer noch einen deutlichen Mehraufwand, da neben der Sprache auch Kenntnisse über den Aufbau der Hardware insbesondere der Gruppierung der einzelnen Ausführungseinheiten und der Speicher für die Programmierung benötigt werden.

2.2.3. Hardwarebeschreibungssprachen

Werden Programmiersprachen dazu verwendet, um Mikroprozessoren zu programmieren, so werden Hardwarebeschreibungssprachen dazu verwendet, die Funktionalität von Hardware zu beschreiben. Sie arbeiten auf einer gänzlich anderen Abstraktionsebene und beschreiben Logik mit Hilfe von Signalen und Speicherelementen. Die Sprachen werden hauptsächlich für zwei Aufgaben verwendet, die jeweils andere Elemente der Sprachen verwenden. Die wichtigste Aufgabe ist die Hardware-Synthese, bei der die einzelnen Teile des Codes in Hardware-Elemente übersetzt werden, beispielsweise in Standardzellen für eine *anwendungsspezifische integrierte Schaltung* (ASIC, engl. *application-specific integrated circuit*) oder in eine Netzliste für einen *Field Programmable Gate Array* (FPGA). Die zweite Aufgabe ist der Test der entwickelten Schaltungen. Dazu stehen weitere Elemente in den Sprachen zur Verfügung, die beispielsweise Wartezeiten darstellen, sich aber nicht auf der Hardware realisieren lassen.

Die Entwicklung in einer Hardwarebeschreibungssprache erfordert viele Grundkenntnisse über das Verhalten von elektrischen Schaltungen und Logik, erlaubt es jedoch, Algorithmen optimal für eine bestimmte Performanz zu optimieren. Die bekanntesten Sprachen sind VHDL [22] und (System) Verilog [23].

2.2.4. Grafische Darstellung

Neben der rein textuellen Beschreibung und der abstrakten grafischen Darstellung existieren noch verschiedenste Ansätze, um Algorithmen grafisch darzustellen. Bekannte kommerzielle Lösungen sind Mathworks® Simulink®/Stateflow® und LabVIEW von Natio-

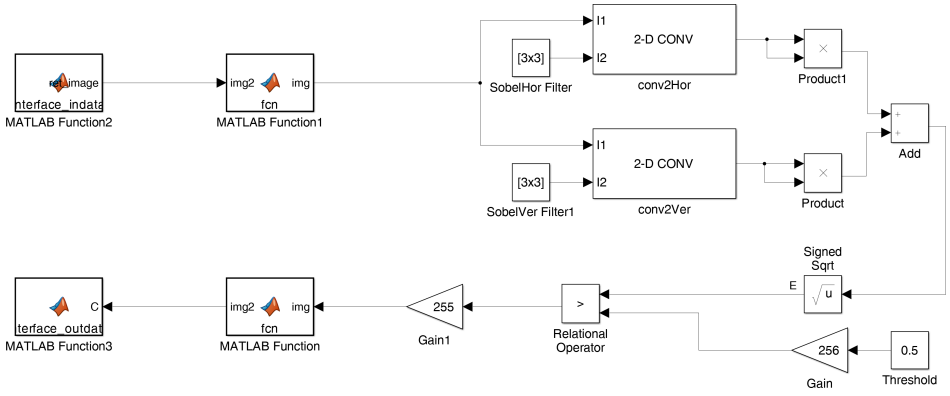


Abbildung 2.8.: Sobel-Filter in Simulink

nal Instruments™. Sie erlauben eine einfache Darstellung von Algorithmen, so lange Standardfunktionen verwendet werden. Ein Beispiel ist in Abbildung 2.8 dargestellt. Der gezeigte Sobel-Filter wird in der Bildverarbeitung verwendet, um die Kanten mit großem Kontrast im Bild hervorzuheben. Ein Beispiel ist in Abbildung 2.9 dargestellt. Der Algorithmus besteht hauptsächlich aus einer horizontalen und einer vertikalen Faltung (engl. *convolution*), welche direkt als „2-D CONV"-Block verwendet werden können. Andere Blöcke stellen einfache mathematische Operationen auf Matrizen (Produkt, Addition) oder in der MATLAB®-Scriptsprache realisierte Blöcke dar. Auf diese Weise kann sehr einfach eine Verarbeitungskette von einem Anfang (hier: MATLAB Function2) bis zu einem Ende (hier: MATLAB Function3) dargestellt werden. Neben dieser reinen Datenverarbeitung, eignet sich Simulink auch sehr gut, um Kontroll-Algorithmen zu entwickeln, bei denen abhängig von verschiedenen Eingangswerten bestimmte Ausgangssignale ausgegeben werden.

Durch die Beschreibung von ganzen Systemen, können Algorithmen sehr einfach mit einem Modell der Umgebung simuliert werden. Das elektrische System, auf dem der Algorithmus ausgeführt werden soll, ist dann nur eine Teilkomponente des Ganzen. Mit diesem Hintergrund ist ersichtlich, dass eine hardwarenahe Beschreibung des Algorithmus nicht im Fokus der grafischen Darstellung liegt. Um ausführbaren Code für die Zielhardware zu erzeugen, liefern die Hersteller bereits Werkzeuge mit, um automatisiert C/C++-Code zu erzeugen. Im Vordergrund steht dabei eine funktional korrekte Ausführung auf der Zielplattform mit nur wenigen Anpassungen, die die Performanz beeinflussen.

2.3. Compiler

In diesem Abschnitt werden einige Grundlagen über *Compiler* eingeführt, soweit sie für diese Arbeit von Bedeutung sind. Dazu wird zunächst der allgemeine Aufbau eines Compilers beschrieben und welche Funktionen er üblicherweise erfüllen muss. Anschließend

(a) Originalbild (b) Verarbeitetes Bild

Abbildung 2.9.: Ein Bild vor und nach der Verarbeitung mit einem Sobel-Filter

werden verschiedene Zwischendarstellungen eingeführt, die im Rahmen dieser Arbeit zum Einsatz kommen.

2.3.1. Aufbau von Compilern

Die Hauptaufgabe eines Compilers ist es, ein Programm, das in einer Programmiersprache vorliegt, in eine andere Sprache zu übersetzen. Der gebräuchlichste Einsatz ist es, eine manuell in einer Hochsprache wie C geschriebene Anwendung in Maschinencode zu übersetzen, der dann von einem Prozessor ausgeführt werden kann. Der erzeugte Code muss jedoch nicht zwingend von einem Prozessor ausführbar sein. Die Übersetzung in eine andere oder sogar die gleiche Programmiersprache kann ebenfalls von einem Compiler durchgeführt werden. Dieser wird dann als Source-to-Source-Compiler bezeichnet.

Ein Compiler wird in zwei grobe Bereiche eingeteilt: das Frontend und das Backend.

Wie der Name schon andeutet, kümmert sich das *Frontend* zu Beginn um die Verarbeitung des Eingangsquellcodes. Dazu wird der Code zunächst eingelesen, auf korrekte Syntax überprüft und in eine besser zu verarbeitende Zwischendarstellung (siehe Kapitel 2.3.2) überführt. Anschließend wird der Code semantisch analysiert, um sicherzustellen, dass das Programm formal korrekt beschrieben wurde. Nun können erste Optimierungen durchgeführt werden, die sich hauptsächlich auf die Struktur des Programms

beziehen. So kann beispielsweise nicht erreichbarer oder redundanter Code entfernt werden, der keinen Einfluss auf die Ausführung des Programms hat.

Die Hauptaufgaben des *Backends* sind weitere Optimierungen und schließlich die Erzeugung des Zielcodes. Dabei lässt sich der Übergang vom Frontend zum Backend mitunter nicht klar festlegen. Optimierungen im Backend sind üblicherweise komplexer als die im Frontend und zielen darauf ab, die Performanz oder den Speicherverbrauch bei der Ausführung zu verbessern. Teilweise können sie dabei auch Auswirkungen auf das Verhalten der Anwendung haben, indem beispielsweise die Berechnung von nicht weiter genutzten Ergebnissen eingespart wird oder durch Transformationen Schleifen hinzugefügt oder entfernt werden. Abschließend findet im Backend die Codegenerierung statt. Dabei wird die interne Zwischendarstellung in den Zielcode übersetzt, der damit für die weitere Verarbeitung oder die Ausführung auf der Hardware bereit ist.

2.3.2. Zwischendarstellungen

Für die automatische Verarbeitung wird das Programm innerhalb eines Compilers in verschiedenen Zwischendarstellungen repräsentiert, die sich jeweils für bestimmte Optimierungen besser eignen.

2.3.2.1. Vereinfachter Abstrakter Syntaxbaum

Ein *abstrakter Syntaxbaum* (AST, engl. *abstract syntax tree*) ist eine abstrahierte Darstellung eines Programmablaufs. Implizite Darstellungen werden entfernt und Ausdrücke, die sich auf unterschiedliche Art darstellen lassen, werden vereinheitlicht. Quellcode 2.1 ist als Beispiel eines AST in Abbildung 2.10 dargestellt.

```
1     for (i = 0; i < 64; ++i) {
2       A[i][j] = (i == j);
3     }
```

Quellcode 2.1: C-Beispielcode für AST

Jedes Konstrukt im Quellcode wird durch einen Knoten im AST dargestellt: Blöcke wie hier der For-Block mit der Initialisierung, der Bedingung, der Iteration und dem Body sowie die Variablen A , i und j mit ihren Zugriffen und den arithmetischen Operationen wie dem *kleiner als* sowie dem Vergleich. Zu beachten ist hier, dass der AST nicht den Kontrollfluss abbildet und deshalb auch bei Schleifen keine zirkulären Verbindungen zwischen den Knoten einfügt.

Innerhalb dieser Arbeit werden Programmabläufe häufig mit einer vereinfachten Darstellung, wie sie in Abbildung 2.11 zu sehen ist, dargestellt. Sie beschränkt sich auf die reine Darstellung der Blöcke und hat keine weiteren Knoten für Variablen oder Operationen.

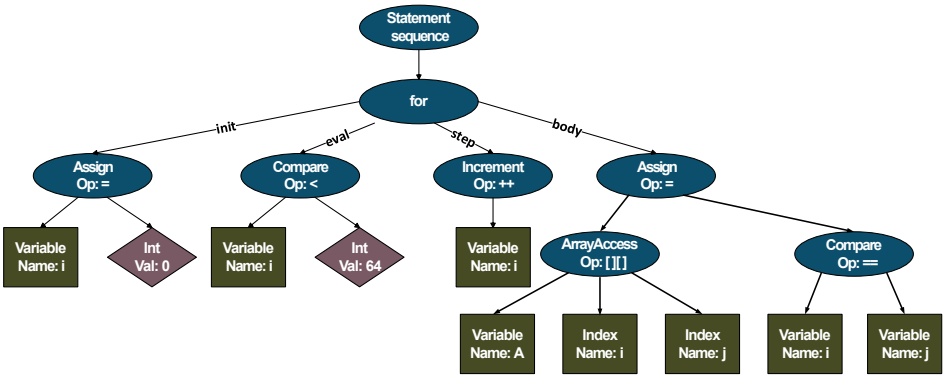


Abbildung 2.10.: Beispiel für einen abstrakten Syntaxbaum (AST)

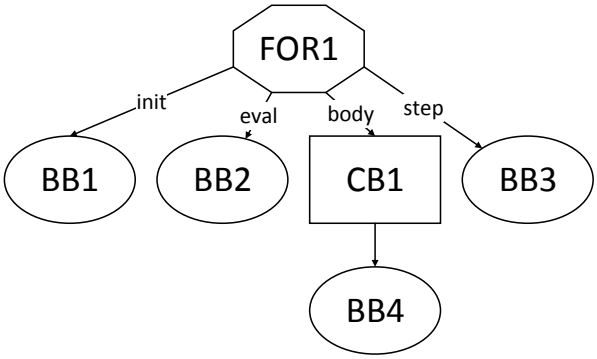


Abbildung 2.11.: Beispiel für einen vereinfachten abstrakten Syntaxbaum (AST)

Sie kommt immer dann zum Einsatz, wenn rein die Struktur des Programms und nicht die tatsächlichen Instruktionen oder Variablen im Vordergrund stehen.

Die Hauptelemente sind *Basisblöcke* (BB), *Kompositblöcke* (CB, engl. *composite blocks*) und Kontrollstrukturen wie If-Blöcke oder Schleifen. Als Basisblöcke werden Blöcke bezeichnet, die nur Ausdrücke enthalten, intern aber keine Sprünge oder andere Strukturen, die den Kontrollfluss verändern. Lediglich der letzte Ausdruck darf ein Sprung sein. Die Ausdrücke werden sequentiell abgearbeitet. Kompositblöcke bezeichnen Container für andere Blöcke. Sie können dabei weitere Kompositblöcke, aber auch Basisblöcke oder Schleifen beinhalten. If-Blöcke werden aus mehreren Blöcken zusammengesetzt. Die Bedingung wird in einem Basisblock überprüft, der *then* und der *else* Zweig wird abhängig von der Komplexität als Basisblock oder Kompositblock dargestellt. Eine *while*-Schleife besteht aus einem Basisblock für die Bedingung und einem Basisblock oder Kompositblock für den Rumpf der Schleife. Eine *for*-Schleife schließlich besteht aus drei Basisblö-

cken für die Initialisierung, die Evaluation und die Schrittweite sowie aus einem Basis- oder Kompositblock für den Rumpf.

2.3.2.2. Kontroll-Datenflussgraphen

Kontrollflussgraphen (CFG, engl. *control flow graph*) werden verwendet, um alle Pfade, die bei der Ausführung eines Programms durchlaufen werden können, darzustellen. Die Knoten stellen Basisblöcke dar, gerichtete Kanten zeigen die Reihenfolge sowie bedingte Verzweigungen des Kontrollflusses. Schleifen und andere Kontrollstrukturen werden nicht als explizite Strukturen dargestellt, sondern implizit über die Abhängigkeiten zwischen Basisblöcken. Ein Beispiel ist in Abbildung 2.12 für den Quellcode 2.2 dargestellt.

```
1     for (i = 0; i < max; i++) {
2       if (a += i < 3)
3         b = a;
4     }
5     c = a;
```

Quellcode 2.2: C-Beispielcode für CFG

Die For-Schleife FOR1 wird durch BB5, BB6, BB8, BB9 und BB10 repräsentiert. Bedingungen werden beispielsweise in BB6 und BB8 überprüft und führen zu einer Verzweigung des Kontrollflusses. Andere gängige Darstellungen sehen ein Zusammenziehen von Basisblöcken vor, wenn die Anzahl der Eingangskanten sowie der Ausgangskanten nicht größer als eins ist. Dies könnte im Beispiel mit BB7 und BB11 durchgeführt werden.

Weiterhin kann für Basisblöcke im CFG eine Dominanzrelation definiert werden. Ein Basisblock M dominiert einen Basisblock N, wenn M auf allen Pfaden vom Anfang zu N durchlaufen werden muss. Dies kann ausgenutzt werden, um den Vergleich von verschiedenen Positionen im Kontrollfluss zu optimieren.

Bei der grafischen Darstellung mittels eines *Kontroll-Datenflussgraph* (CDFG, engl. *control-dataflow graph*) wird ein CFG um existierende Datenabhängigkeiten erweitert. Datenabhängigkeiten beziehen sich auf Lese- und Schreibzugriffe bei Variablen. Der Block, in dem die Variable geschrieben wird, muss zeitlich vor dem Block, in dem die Variable gelesen wird, ausgeführt werden. Ein Beispiel für einen kombinierten CDFG für Quellcode 2.3 kann Abbildung 2.13 entnommen werden. Graue Pfeile stehen dabei für unbedingte Kontrollabhängigkeiten, rote für bedingte und blaue für Datenabhängigkeiten.

2.3.2.3. Hierarchische Task-Graphen

Task-Graphen werden verwendet, um eine Anwendung aufgeteilt in verschiedene Aufgaben (engl. *tasks*) darzustellen. Tasks werden als Knoten dargestellt, Abhängigkeiten als

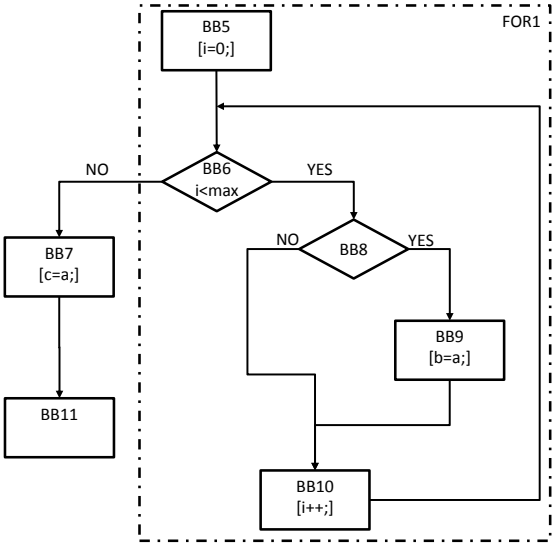


Abbildung 2.12.: Einfacher Kontrollfluss-Graph

```
1
2       int x, y, i;
3
4       for (i = 0; i < 20; i = i + 1) {
5         x = i * i;
6         y = i;
7       }
```

Quellcode 2.3: C-Beispielcode für CDFG

gerichtete Verbindungen. Ausgehend von einem CDFG wird in einem Task eine Menge an Basisblöcken versammelt, die hintereinander ausgeführt werden müssen. Durch die höhere Abstraktion eignen sich Tasks besser zur Aufteilung auf mehrere Prozessoren, da eine zu feine Aufteilung den Bedarf an Kommunikation sehr schnell steigert.

Problematisch wird die Verarbeitung von Schleifen im Graph. Bilden Tasks einen Ring, wird dadurch die Zuweisung von Prozessoren erschwert. Eine Lösung des Problems ergibt sich durch die Verwendung von *hierarchischen Task-Graphen* (HTG, engl. *hierarchical task graphs*) [24]. Hierbei wird bei Auftreten einer Schleife eine neue Hierarchieebene erzeugt und die Schleife darin platziert. Dies sorgt dafür, dass Zyklen nur innerhalb einer Ebene existieren, auf einer höheren Hierarchieebene aber nicht mehr ersichtlich sind. Abhängigkeiten bestehen in dieser Darstellung immer nur zwischen Tasks auf der gleichen Ebene. Ein Beispiel ist in Abbildung 2.14 dargestellt.

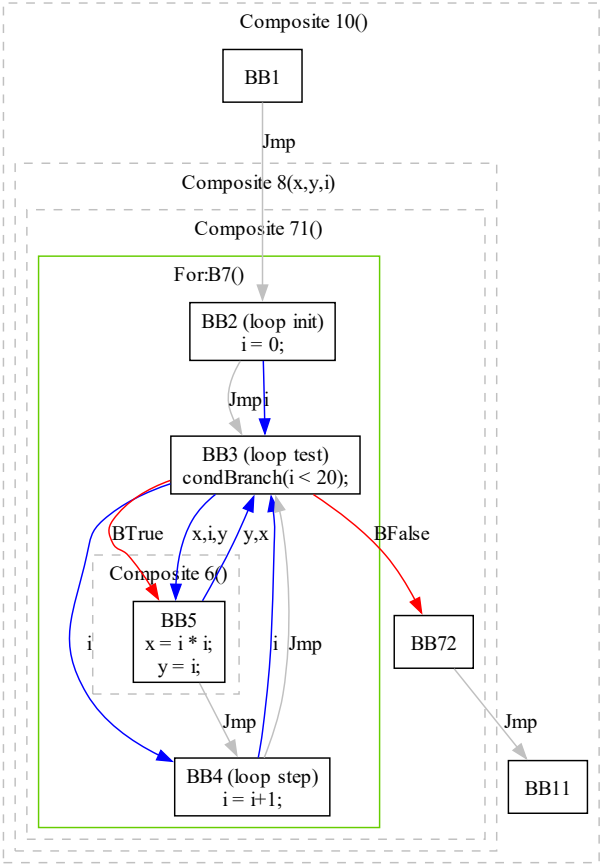


Abbildung 2.13.: Einfacher Kontroll-Datenfluss-Graph

Man unterscheidet zwei Arten von Tasks:

- *Leaf Tasks* bilden die kleinste Einheit der Tasks und sind mit Basisblöcken gleichzusetzen. Sie dürfen nur Instruktionen, aber keine weiteren Tasks beinhalten. In Abbildung 2.14 sind dies T1, T2 und T4 sowie T3.0 bis T3.5.
- *Hierarchische Tasks* sind Container, die verschiedene andere Tasks beinhalten können. Über sie werden die einzelnen Hierarchieebenen unterschieden; alles innerhalb eines hierarchischen Tasks befindet sich eine Ebene tiefer. Da Abhängigkeiten nur auf der gleichen Ebene existieren dürfen, werden innerhalb eines hierarchischen Tasks noch zusätzliche Leaf Tasks als Eingangs- und Ausgangsknoten hinzugefügt. Von bzw. zu ihnen führen alle Abhängigkeiten, die auf der höheren Ebene direkt auf den hierarchischen Task zeigen. In Abbildung 2.14 ist T3 ein hierarchischer Task, der

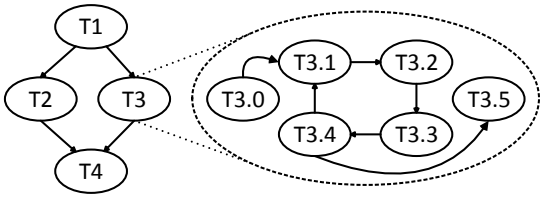


Abbildung 2.14.: Beispiel für einen hierarchischen Task-Graph

die Leaf Tasks T3.0 bis T3.5 beinhaltet. T3.0 ist dabei der Eingangstask und T3.5 der Ausgangstask von T3.

2.3.2.4. Static-Single-Assignment-Darstellung

Unter der *Static-Single-Assignment-Darstellung* (SSA, engl. *static single assignment form*) [25] versteht man eine Darstellung, bei der jede Variable genau ein Mal geschrieben wird. Für jeden weiteren Schreibvorgang wird eine neue Version der Variablen erzeugt und mit einer aufsteigenden, eindeutigen Nummer versehen. Aus Variable *a* wird somit *a*_1, *a*_2. . . . Die SSA-Darstellung vereinfacht einige Analysen deutlich. So kann beispielsweise toter Code, der für die Ausführung nicht benötigt und aufgerufen wird, einfach erkannt werden, da kein Lesezugriff auf eine bestimmte Version einer Variablen erfolgt. Das gleiche Beispiel wie in 2.3.2.2 kann in Abbildung 2.15 in der SSA-Darstellung betrachtet werden. Man sieht sehr gut, dass bei Beginn eines Durchgangs einer Schleife sichergestellt werden muss, in welcher Version die Variablen *i*, *x* und *y* verwendet werden müssen.

Kann bei der statischen Analyse nicht eindeutig bestimmt werden, welche Version einer Variablen gelesen wird, wird eine sogenannte Φ-Funktion eingeführt. Sie stellt eine Funktion dar, die alle Versionen, die eine Variable annehmen kann, als Eingangsparameter entgegennimmt und immer die passende Version zurückliefert. Dieser Fall tritt in der Regel bei Verzweigungen des Kontrollflusses durch Schleifen oder bedingte Sprünge auf. Ein Beispiel kann in Abbildung 2.15 bei Basisblock 3, dem Evaluationsblock der for-Schleife, betrachtet werden. Die Φ-Funktionen sind hier für die Variablen *x*, *y* und *i* notwendig, da beim ersten Schleifenaufruf Werte von vor der Schleife bzw. der Initialisierung verwendet werden und bei jedem weiteren Aufruf die Werte aus der Schleife bzw. dem Inkrementblock.

Eine Variante der SSA-Darstellung wird *Concurrent Static Single Assignment* (CSSA) Darstellung genannt. Sie wurde in [26] eingeführt und betrachtet explizit parallele Anwendungen mit überlappender Semantik und Synchronisation. Dazu wird neben der Φ-Funktion noch eine Ψ- und eine Π-Funktion eingeführt. Eine Ψ-Funktion wird verwendet, um zusammenlaufende Definitionen einer Variablen über mehrere Prozessoren hinweg zusammenzuführen. Sie stellt somit eine Φ-Funktion dar, die auf verschiedene Prozessoren achtet. Π-Funktionen kommen bei geteilten Variablen zum Einsatz und markieren, dass eine Definition auf einem anderen Prozessor erfolgt sein könnte. Die CSSA-Darstellung wird in dieser Arbeit nicht weiter verwendet, da die Ψ-Funktion mit Φ-Funktionen darge-

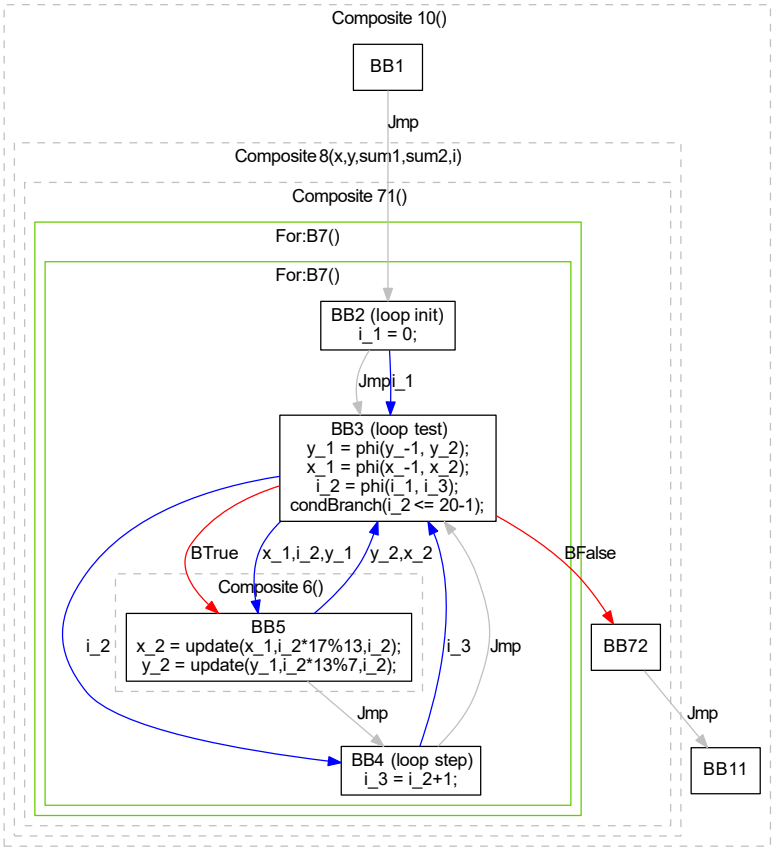


Abbildung 2.15.: Kontroll-Datenfluss-Graph in SSA-Darstellung

stellt werden kann und globale Variablen bei der dargestellten Parallelisierung vermieden wurden.

2.3.3. Parallelisierende Compiler

Bei *parallelisierenden Compilern* wird neben den eigentlichen Aufgaben eines Compilers eine automatisierte Parallelisierung der Anwendung durchgeführt. Ziel hierbei ist es, das Programm möglichst gut an die Zielplattform anzupassen und parallele Recheneinheiten auszunutzen. Es existieren verschiedenste Verfahren, um Parallelität zu finden und auszunutzen. Beispielhaft soll in diesem Abschnitt die Werkzeugkette, die im EU-Projekt ALMA entwickelt wurde, dargestellt werden.

2.3.3.1. Die ALMA-Werkzeugkette als parallelisierender Compiler

Ein grober Überblick über die Werkzeugkette, die im Rahmen des EU-Projekts ALMA entwickelt wurde, kann Abbildung 2.16 entnommen werden. Die gesamte Kette kann als Compiler von den Sprachen Scilab/MATLAB® hin zu parallelem C-Code verstanden werden. Dabei wird der Code für eine Architektur optimiert, die mit einer *Architekturbeschreibungssprache* (ADL, engl. *architecture description language*) beschrieben wird. Der parallelisierende Compiler besteht aus den einzelnen Schritten *feingranulare Parallelisierungsextraktion* (engl. *fine-grained parallelization extraction*), *grobgranulare Parallelisierungsextraktion* (engl. *coarse-grained parallelization extraction*) und der *parallelen Codegenerierung* (engl. *parallel code generation*).

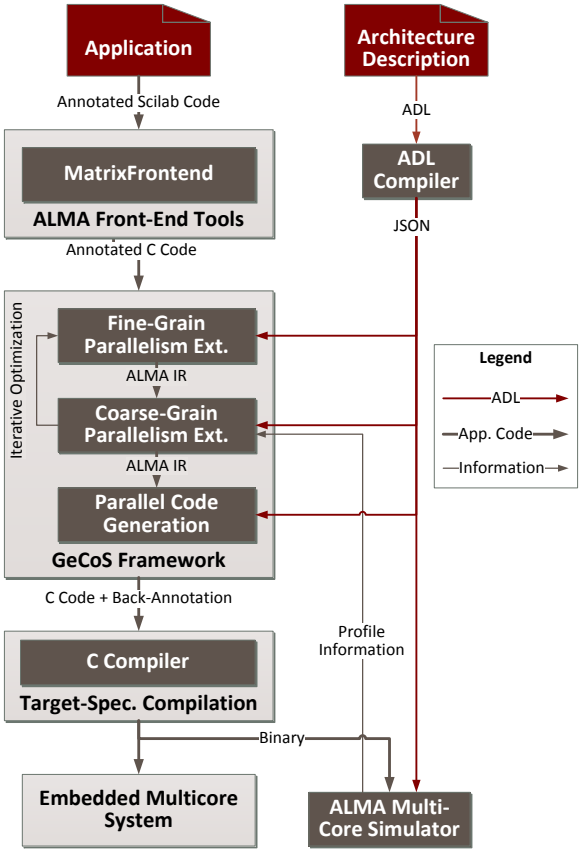


Abbildung 2.16.: Übersicht über die ALMA-Werkzeugkette

MatrixFrontend

Das MatrixFrontend dient dazu, den arraybasierten Scilab/MATLAB®-Code in statischen C-Code zu transformieren. Dabei ist wichtig, dass der erzeugte Code effizient auch auf eingebetteten Systemen ausgeführt werden kann und dass er sich gut für die spätere Parallelisierung analysieren lässt. Das Konzept der Transformation wird in Abschnitt 4.3.1 beschrieben, die tatsächliche Realisierung in Abschnitt 4.3.3.

GeCoS Framework

Das *GeCoS* (Generic Compiler Suite) *framework* [27] ist ein Source-to-Source Compiler, der in ALMA für die Transformation von sequentiellem C-Code hin zu parallelem C-Code verwendet wurde. Es wurde von der CAIRN Gruppe der Universität Rennes I entwickelt und setzt auf das *Eclipse Modeling Framework* (EMF) der integrierten Entwicklungsumgebung Eclipse [28]. GeCoS nutzt dabei vollständig den durch das EMF vorgegebenen modellbasierten Ansatz, um Programme mit Hilfe der GeCoS-Zwischendarstellung darzustellen. Es bietet JAVA-Klassen für alle gängigen Objekte wie Blöcke, Instruktionen und Symbole sowie vorgefertigte Entwurfsmuster wie Besucher, abstrakte Fabriken und Adapter.

GeCoS arbeitet in einzelnen Schritten, die als Transformation oder Durchgang (engl. pass) bezeichnet werden. Dabei ist es modular aufgebaut, so dass das Framework leicht um neue Klassen und Transformationen erweitert werden kann. Alle folgenden Parallelisierungsschritte und die Codegenerierung wurden innerhalb von GeCoS realisiert.

Die interne Zwischendarstellung ist ein Hybrid aus mehreren Darstellungen und umfasst einen AST sowie einen CDFG und kann um Strukturen eines HTG angereichert werden. Die Grundelemente sind Symbole, Instruktionen und Blöcke, sowie weitere Kernelemente, die hauptsächlich Container für die anderen Elemente bilden.

Symbole entsprechen Variablen in C, existieren aber auch in einer SSA-Darstellung als Definitionen und Verwendungsstellen. Unter Instruktionen werden alle Operationen und Funktionen zusammengefasst, die auf die Symbole angewendet werden können. Blöcke werden schließlich hinsichtlich ihres Einsatzzwecks unterschieden. Basisblöcke sind Container für Instruktionen und beinhalten keine Kontrollstrukturen. Diese werden durch eigene Blöcke wie If-Blöcke oder For-Blöcke dargestellt. Sie setzen sich aus mehreren Basisblöcken zusammen. So besitzt ein For-Block beispielsweise Blöcke für die Initialisierung von Variablen, für die Bedingung und für die Schrittweite. Zusammengesetzte Kompositblöcke sind schließlich Container für andere Blöcke und werden im C-Code über geschweifte Klammern definiert. Für eine HTG-Darstellung existieren des Weiteren Blatt-Taskknoten (engl. *leaf task nodes*) sowie hierarchische Taskknoten (engl. *hierarchical task nodes*) als abgeleitete Klassen von Basisblöcken und zusammengesetzten Blöcken.

Kontrollabhängigkeiten können sowohl implizit als auch explizit angegeben werden. Implizit wird eine Ausführungsreihenfolge durch die Anordnung von Blöcken innerhalb der GeCoS-IR angegeben. Diese können auch explizit über Eingangs- und Ausgangskanten dargestellt werden, welche Basisblöcke miteinander verbinden. Die Kanten können automatisch durch eine existierende Transformation hinzugefügt werden. Datenabhän-

gigkeiten werden auf zwei unterschiedliche Arten dargestellt. So haben zum einen SSA-Symbole Referenzen auf ihre Verwendungsstellen (bei Definitionen) oder ihre Definitionen (bei Verwendungsstellen). Zum anderen werden Datenkanten zwischen Basisblöcke eingefügt, bei denen eine Datenabhängigkeit aufgrund einer Schreib-Lese-Beziehung einer Variablen existiert.

Die GeCoS-Zwischendarstellung kann in einer *XML Metadata Interchange* (XMI)-Datei, einem Standardformat für den Datenaustausch, das auf *Extensible Markup Language* (XML) [29] aufsetzt, ausgegeben werden.

Feingranulare Parallelisierung

Die *feingranulare Parallelisierung* zielt auf die Ausnutzung von Parallelität auf Datenebene ab. Dabei kommen Instruktionen zum Einsatz, die gleichzeitig auf mehrere Daten angewendet werden – *single instruction, multiple data* (SIMD). Nehmen wir als Beispiel eine 32 Bit Architektur. Bei einer einfachen Addition werden zwei 32 Bit Integer Werte addiert und liefern einen 32 Bit Wert als Ergebnis zurück. Mit einem SIMD-Befehl ist es nun möglich, anstelle von einer 32 Bit Zahl zwei 16 Bit (*short*) Zahlen oder vier 8 Bit (*char*) Zahlen als Daten zu verwenden. Anstelle von einer Addition werden so zwei bzw. vier ausgeführt. Da die gleiche Recheneinheit verwendet wird, ändert sich nichts an der Dauer, aber es werden mehrere Berechnungen mit kleineren Zahlen parallel ausgeführt.

Um SIMD-Befehle effizient ausnutzen zu können, ist es sinnvoll, möglichst kleine Datentypen zu verwenden. Dies sind bei gängigen 32 Bit Architekturen 8 Bit *char*-Werte und 16 Bit *short*-Werte. Da der Basistyp bei arraybasierten Programmiersprachen 64 Bit *double*-Werte mit Gleitkommadarstellung sind, ist eine Umwandlung nicht ohne weiteres möglich. Innerhalb der ALMA-Werkzeugkette wird deshalb eine Umwandlung der Gleitkommazahlen nach Festkommazahlen durchgeführt.

Bei der Gleitkommadarstellung wird eine Zahl x aus Mantisse m und Exponent e zusammengesetzt: $x = m \cdot 10^e$. Diese in der Wissenschaft gebräuchliche Darstellung kann sehr große Zahlenbereiche abbilden. Durch explizite Hardware kann die Berechnung mit Gleitkommazahlen entscheidend beschleunigt werden, da ansonsten eine Emulation mit ganzen Zahlen in Software durchgeführt werden muss.

Festkommazahlen teilen sich auf in einen Anteil n vor und einen Anteil k nach dem Komma, so dass in der Summe $n + k$ Bit verwendet werden, um eine Zahl darzustellen. Der Wertebereich ist kleiner als bei einer Gleitkommazahl mit der gleichen Anzahl an Bit, dafür werden einzelne Zahlen mit einer höheren Genauigkeit dargestellt. Es ist keine spezielle Hardwareunterstützung notwendig, um mit Festkommazahlen rechnen zu können. Recheneinheiten für ganze Zahlen können verwendet werden, wenn sichergestellt wurde, dass das Komma bei allen Zahlen an der gleichen Position ist.

Bei der automatischen Umwandlung von Gleitkommazahlen hin zu Festkommazahlen geht durch die Eingrenzung des Wertebereichs Genauigkeit verloren. Der entstehende Fehler kann berechnet werden und gilt als Maß für die Abweichung der erzeugten Ergebnisse im Vergleich zur Fließkommadarstellung. Diese Abschätzung wird nun verwendet, um alle Variablen mit möglichst kleinen Datentypen darzustellen. Anschließend kön-

nen unabhängige Berechnungen gesucht werden und durch SIMD-Befehle zu weniger Instruktionen zusammengefasst werden.

Die feingranulare Parallelisierung mit ihrer Festkommadarstellung und der Ausnutzung von SIMD-Befehlen wird im weiteren Verlauf dieser Arbeit nicht weiter betrachtet.

Grobgranulare Parallelisierung

Die *grobgranulare Parallelisierung* zielt auf die Ausnutzung von *Parallelität auf Task-Ebene* (TLP, engl. *task-level parallelism*) ab. Ziel ist es, dass mehrere Prozessoren parallel an Tasks arbeiten und Abhängigkeiten durch Synchronisation und Datenaustausch gehandhabt werden.

In ALMA wird dabei auf eine Darstellung mit hierarchischen Task-Graphen gesetzt. Beginnend mit der untersten Hierarchieebene wird eine effiziente Verteilung von Tasks auf verfügbare Prozessoren gesucht, so dass die Ausführungszeit minimiert wird. Das Problem wird dabei als Optimierungsproblem modelliert, bei dem unterschiedliche Lösungen über die Ausführungszeit bewertet werden. Die Lösung erfolgt entweder über Heuristiken wie Simulated Annealing [30], die mit relativ kurzer Laufzeit gute, aber nicht unbedingt die besten Lösungen liefern oder über ganzzahlige lineare Optimierung (engl. Integer linear programming), welche mit längerer Laufzeit optimale Lösungen findet.

Als Eingangsdaten für die Algorithmen werden Profiling-Informationen verwendet, welche durch Ausführung auf einem Simulator ermittelt werden. Dazu wird zunächst der sequentielle Programmcode mit Profiling-Informationen versehen, die Start und Ende von Tasks mit einer eindeutigen Identifikationsnummer markieren. Der Simulator misst die Anzahl der Taktzyklen, die zwischen zwei zusammengehörenden Profiling-Instruktionen vergangen ist. Auf diese Weise kann für jeden hierarchischen und Leaf Task die Dauer in der Anzahl an Taktzyklen angegeben werden. Anhand dieser Informationen wird eine Parallelisierung vorgeschlagen, wobei der Aufwand für Datentransfers abhängig von der Zielplattform zunächst nur abgeschätzt wird. In einer weiteren Simulation werden nun nach demselben Prinzip Taktzyklen für die Dauer der Kommunikation inklusive ihrer Wartezeiten ermittelt. Diese Erkenntnisse werden anschließend verwendet, um iterativ eine bessere Parallelisierung auf die verschiedenen Prozessoren zu erreichen.

Neben der eigentlichen Zuordnung von Tasks auf die Recheneinheiten, wird während der grobgranularen Parallelisierung auch das Scheduling (engl. für Zeitplanerstellung) durchgeführt. Soweit es die Abhängigkeiten erlauben wird die Reihenfolge von Tasks so verändert, dass eine bessere Auslastung der Prozessoren erreicht wird.

Die bei der grobgranularen Parallelisierung zur Zwischendarstellung hinzugefügten Informationen über zugewiesene Prozessoren und das Scheduling der einzelnen Tasks bilden eine wichtige Grundlage für die parallele Codegenerierung, um automatisch parallelen C-Code zu erzeugen.

Parallele Codegenerierung

Ziel der *parallelen Codegenerierung* ist die Erzeugung von C-Code, der an den Compiler der Zielplattform übergeben werden kann. Innerhalb von ALMA wurde die Codegenerierung noch um weitere Themen erweitert, so dass in dieser Transformation auch das Einfügen von Kommunikationsinstruktionen, die Verwaltung von Speicherhierarchien und die Entwicklung einer *Programmierschnittstelle* (API, engl. *application programming interface*) zur Abstraktion von Hardwareeigenschaften bearbeitet wird.

2.4. Performanzabschätzung

Für viele leistungsrelevante Entscheidungen ist es wichtig, ein Modell für die Ausführungszeit einzelner Blöcke und Instruktionen zu haben. Dabei werden unterschiedliche Anforderungen an die Genauigkeit des Modells gestellt. Generell kann gesagt werden, dass für grobe Entscheidungen auf Task- oder Funktionsebene eine grobe Abschätzung ausreicht. Genauere Werte werden benötigt, wenn es um tatsächliche Zeiten geht. Als Beispiel seien hier heterogene Systeme genannt, bei denen die absoluten Zeiten für die Berechnung auf dem Prozessor mit den Zeiten für die Ausführung auf einer Grafikkarte verglichen werden sollen.

2.4.1. Ermittlung von Laufzeiten

Im Folgenden werden verschiedene Möglichkeiten vorgestellt, um Performanzwerte mit unterschiedlicher Genauigkeit zu ermitteln. Die Methoden sind dabei nach steigender Genauigkeit sortiert.

2.4.1.1. Statische Codeanalyse

Unter statischer Analyse versteht man die automatische Analyse des Quell- oder Objektcodes ohne diesen auszuführen. Häufig wird sie durchgeführt, um sicherheitsrelevante Aspekte zu überprüfen [31] oder um die Codequalität durch frühe Erkennung von Fehlern zu verbessern [32].

Statische Analyse des Quellcodes

Laufzeiten können bereits mit Hilfe einer statischen Analyse des Quellcodes ermittelt werden. Dies bietet in der Regel zeitliche Vorteile, da der Code nicht erst kompiliert und ausgeführt werden muss. Stattdessen wird zunächst für jede Instruktion eine Zyklusdauer ermittelt. Dies kann aus dem Datenblatt erfolgen oder mittels speziell instrumentierten Codes, aus dem die Zyklendauer für einzelne Instruktionen herausgerechnet werden können. Anschließend kann die Dauer für Blöcke aus der Summe der beinhalteten Instruktionen bestimmt werden.

Eine einfache Abschätzung wurde bereits im letzten Abschnitt mit Quellcode 2.8 aufgezeigt: die innerste Operation wurde mit zwei Taktzyklen abgeschätzt. Da kein weiterer Code hinzugefügt werden muss, werden die Zeiten von Kontrollstrukturen auf höheren Ebenen nicht beeinflusst. Nimmt man jetzt für das Initialisieren der Variablen, den Vergleich und der Erhöhung jener jeweils einen Taktzyklus an, kann eine gesamte Dauer von 420 Zyklen geschätzt werden. Mit den Zugriffen auf die Variablen wurde aber bereits ein großes Problem angesprochen: die Abhängigkeit der Performanz vom Compiler und dem zur Verfügung stehenden Cache.

Die Auswirkungen des Cache abzuschätzen ist komplex und kann eigentlich nur durch ein genaues Modell abgebildet werden. Da dies potenziell bei jeder Instruktion benötigt wird, führt das die statische Analyse ad absurdum, wenn ein Teil der Hardware quasi simuliert werden muss. Ignoriert man das Speicherverhalten, entfernt man sich zwar vom tatsächlichen Verhalten auf der Hardware, kann die ermittelten Zyklen aber relativ zueinander vergleichen, um beispielsweise lang andauernde Aufgaben zu erkennen.

Der Einfluss des Compilers kann sogar eine noch größere Rolle spielen. Im konkreten Beispiel kann ein Compiler die Schleifen teilweise oder ganz entrollen. Wird dies gemacht, so entfallen die Erhöhungen der Zählvariablen i und j sowie deren Vergleich mit 10. Dies entspricht im Beispiel bis zu 220 Zyklen weniger. Möglich ist sogar, dass die gesamte Berechnung von x, die nur von konstanten Werten abhängt, bereits vom Compiler durchgeführt wird und auf die Anweisung $x = 900$ reduziert wird. Man sieht, dass die tatsächliche Ausführung anschließend auf 1 oder 200 Zyklen reduziert worden sein kann im Vergleich zu den geschätzten 420 Zyklen der statischen Analyse.

Statische Analyse der Compiler-Zwischendarstellung

Wie im vorigen Abschnitt verdeutlicht, kann der Compiler einen sehr großen Einfluss auf die Performanz des tatsächlich ausgeführten Programms haben. Für genauere Werte kann es deshalb sinnvoll sein, direkt auf die Zwischendarstellung des Compilers nach den Optimierungen zuzugreifen, um die Performanzabschätzung durchzuführen. Dies wurde in [33] für den LLVM-Compiler [34] gemacht. Die Ergebnisse können deutlich genauer sein und das Verhalten bei Ausführung auf der Hardware besser abschätzen. Der Einfluss der Caches und anderer Speicher auf dem Zielsystem kann jedoch auch mit dieser Methode nicht genau bestimmt werden.

2.4.1.2. Simulation

Simulationen von Prozessorsystemen können auf den verschiedensten Abstraktionsebenen durchgeführt werden.

Am weitesten entfernt von der Hardware sind Instruktionsset-Simulatoren, die kompilierten Code mit dem gleichen Verhalten ausführen können wie das System. Dabei wird in der Regel aber nicht zyklenakurat simuliert, so dass Zeiten nur im Verhältnis zueinander verglichen werden können. Durch vereinfachte Speichersysteme kann auch der Einfluss von Caches nicht ermittelt werden. Diese Art der Simulation eignet sich besonders dafür, die korrekte Funktionalität von Code zu überprüfen. Häufig kommt dabei SystemC zum

Einsatz, was eine Erweiterung der Sprache C++ um Elemente zur Hardwaremodellierung, wie Signale und Zeit, ist. SystemC bietet somit alle Vorteile von C++ und erlaubt es, Simulatoren zu entwickeln, die um vielfaches schneller sein können als die tatsächliche Hardware.

Mittels zyklenakkurater Simulation können ganze Systeme simuliert und Zeiten für den zuvor kompilierten Code ermittelt werden. Durch genauere Speichermodelle kann auch der Einfluss von Caches und anderer Peripherie gemessen werden. Je nach Simulator und simulierter Plattform kann die Performanz von der tatsächlichen Hardware erreicht werden oder um Faktoren langsamer sein. Die Modellierung kann auch hier mit SystemC erfolgen.

Für die genausten Simulationen ist es notwendig, das gesamte System auf Register-Transfer-Ebene zu beschreiben. Dazu eignen sich Hardwarebeschreibungssprachen wie VHDL und Verilog. Aufgrund ihrer Komplexität dauern Simulationen um mehrere Größenordnungen länger als die tatsächliche Ausführung auf der Hardware.

2.4.1.3. Ausführung auf der Hardware

Profiling

Ein gängiger Ansatz ist es, mit Hilfe von Funktionen den Quellcode zu instrumentieren, so dass die Laufzeit von jedem Task und jeder Kommunikation gemessen werden kann. Dazu werden im Quellcode Befehle eingefügt, die den Start und das Ende eines zu messenden Abschnitts darstellen. Die dazwischen gemessene Zeit kann nun als Wert für die Ausführung des Blocks verwendet werden. Zu beachten ist dabei, dass die hinzugefügten Instruktionen selber Einfluss auf die tatsächliche Ausführung haben und somit die Zeiten verfälschen können.

Tracing

Das Hardware-Tracing ist die genauste Möglichkeit, um Zeiten für die unterstützten Plattformen zu ermitteln. Dabei wird zusätzlich Hardware in den Systemen oder Prozessoren verbaut, die es erlaubt, Zeiten zu messen, ohne die tatsächliche Ausführungszeit zu beeinflussen. Dies liefert sehr genaue Werte für die Architektur, bei der alle Compileroptimierungen mit betrachtet werden können. Dieses Vorgehen ist die bevorzugte Methode zur Ermittlung von Zeiten auf der Hardware, muss aber von der Plattform explizit unterstützt werden. Weitere Details können in der Implementierung für ARM-Prozessoren – *ARM CoreSight Architecture* [35] – entnommen werden.

2.4.2. Anforderungen an die Performanzabschätzung

Im Rahmen dieser Arbeit werden Performanzwerte auf zwei unterschiedlichen Abstraktionsebenen benötigt.

Zum einen wird eine Abschätzung auf Task-Ebene benötigt, um Parallelisierungs- und andere grobgranulare Entscheidungen treffen zu können. Dabei ist die wichtigste Aufgabe, dass lange Tasks erkannt werden können. Als Faustformel gilt bei der Ausführung von Software, in Anlehnung an das Pareto-Prinzip [36], dass 80% der Zeit in 20% des Codes verbracht wird. Die Erfahrung zeigt, dass für viele Programme sogar die Verteilung 90 zu 10 angenommen werden kann. Auch wenn es keine festen Regeln sind, ist das Prinzip sehr verbreitet in der Optimierung von Software. Durch die Konzentration auf die entscheidenden 10-20% des Codes kann die Performanz schneller gesteigert werden, als wenn Optimierungen auf die anderen Teile des Codes angewendet werden. Das Prinzip kann auch verwendet werden, um das Programm iterativ noch weiter zu optimieren, indem es erneut auf das optimierte Programm angewendet wird. Zur Identifikation der entscheidenden Codestellen ist es nötig, die Tasks in Relation zueinander vergleichen zu können. Absolute Zeiten sind also nicht notwendig, um den entsprechenden Code erkennen zu können.

Des Weiteren wird eine Möglichkeit gebraucht, den tatsächlichen Vorteil der verschiedenen Optimierungen und Transformationen zu ermitteln und auch die Ausführungen auf verschiedener Hardware vergleichen zu können. Hier sind genauere Werte notwendig, da Vergleiche nur über die tatsächlichen Zeiten gemacht werden können. Aufgrund der verschiedenen Komponenten, seien es Speicher oder Hardware-Beschleuniger, haben die Zeiten keine gemeinsame Basis und ein rein relativer Vergleich wird sehr ungenau. Bei Optimierungen kommt noch hinzu, dass die Modellierung sehr komplex werden kann und das Zusammenspiel mit anderen Optimierungen nicht immer ersichtlich ist. Optimierungen hinsichtlich verbesserter Speicherzugriffe und Cache-Nutzung erfordern zum Nachweis ebenfalls sehr genaue Werte, um den Einfluss auf die gesamte Laufzeit angeben zu können.

2.4.3. Realisierung der Performanzabschätzung

Um die Anforderungen zu erfüllen, wurden zwei Möglichkeiten realisiert, um Zeiten zu ermitteln. Für die relativen Abschätzungen wurde eine statische Analyse des Quellcodes gewählt. Die Genauigkeit ist ausreichend, um die Tasks zu ermitteln, in denen die meiste Zeit benötigt wird. Sollte der Bedarf nach einer höheren Genauigkeit aufkommen, sind Erweiterungen denkbar, die auf die Analyse des Assembler-Codes oder der Zwischendarstellung eines Compilers, beispielsweise LLVM, setzen.

Um sich nicht auf eine Zielarchitektur festzulegen, wurde für die Ermittlung von genaueren Werten auf Profiling mit Hilfe von instrumentiertem Quellcode gesetzt. Das eigentlich überlegene Tracing über dedizierte Hardware ist nicht für jede Plattform verfügbar und somit nicht allgemein benutzbar. Um die Flexibilität der Zielplattform beizubehalten ist das Profiling besser geeignet, da auf der Plattform nur eine Möglichkeit benötigt wird um Zeiten zu messen.

2.4.3.1. Realisierung der statischen Analyse

Die statische Analyse ist in drei Schritte eingeteilt. Als Vorbereitung für die Unterstützung einer neuen Plattform, werden Zeiten für die einzelnen Operationen gemessen. Anschließend wird der zu testende Code instrumentiert, um die Ausführungshäufigkeiten der einzelnen Blöcke zu ermitteln. Erst im dritten und letzten Schritt findet die Analyse des eigentlichen Quellcodes statt.

Messung der Ausführungszeiten von Operationen

Die kleinste ausführbare Einheit im Quellcode sind Operationen. Diese, von einem Prozessor ausgeführt, lassen sich in die folgenden Kategorien einteilen:

1. Lesen und Schreiben von Daten aus dem Hauptspeicher.
2. Einfache arithmetische Operationen wie Addition oder Subtraktion mit ganzen Zahlen.
3. Erweiterte arithmetische Operationen wie Multiplikation von ganzen Zahlen.
4. Einfache logische Operationen wie AND oder OR.
5. Befehle zur Beeinflussung des Kontrollflusses wie beispielsweise Sprünge oder If-Bedingungen.
6. Komplexere arithmetische Operationen wie die Berechnung von Fließkomma-Addition oder Multiplikation.

Für eine Abschätzung über die Dauer von Tasks werden die Ausführungszeiten der einzelnen Operationen benötigt. Da später nur ein relativer Vergleich stattfinden soll, können Zeiten in Anzahl an Taktzyklen ausgedrückt werden. Die Ausführungszeiten der einzelnen Operationen haben jedoch in der Regel sehr große Unterschiede bei unterschiedlichen Architekturen:

1. Der Zugriff auf den Hauptspeicher wird vor allem durch den generellen Aufbau des Systems beeinflusst. Dazu gehört die Position des Speicherkontrollers, direkt in der CPU oder als Teil des Chipsatzes, die Verbindung zwischen Speicher und Prozessor aber auch die Geschwindigkeit des Speichers.
2. Arithmetische Operationen sind die Operationen, die in vielen Algorithmen den Großteil der Berechnungen ausmachen. Aus diesem Grund haben moderne Prozessoren häufig mehrere parallele Ausführungseinheiten für die arithmetischen Berechnungen. Dadurch kann die Ausführungszeit auf Bruchteile eines Taktschritts gesenkt werden.
3. Bei Operationen wie der Multiplikation von ganzen Zahlen ist der entscheidende Faktor, ob eine Hardware-Unterstützung existiert oder nicht. Ohne eine Unterstützung kann eine Multiplikation auch über Additionen berechnet werden, was jedoch um den Faktor 30 oder mehr länger dauern kann. Mit dedizierter Hardware kann die Berechnung auf bis zu einen Taktzyklus gesenkt werden. Üblicherweise wird eine Implementierung mit einer Pipeline realisiert, die nach einer Verzögerung von 3 bis 5 Taktzyklen in jedem Schritt ein neues Ergebnis ausgeben kann.

4. Gängige arithmetische Einheiten unterstützen sowohl einfache Berechnungen mit ganzen Zahlen als auch logische Operationen. Aus diesem Grund ist die Ausführungszeit häufig identisch.
5. Die Dauer von Sprungbefehlen liegt neben der tatsächlichen Hardware und der Adressberechnung auch an der Art des Befehls. So müssen bei einem Funktionsaufruf alle aktuellen Werte auf dem Stack zwischengespeichert werden, um nach dem Aufruf wieder zur Verfügung zu stehen.
6. Auch bei den komplexen arithmetischen Operationen ist der entscheidende Faktor die Existenz einer Hardware-Komponente. Auch komplexe Fließkomma-Operationen können über Software auf einfache Operationen abgebildet werden, die Dauer wird dann jedoch um Größenordnungen schlechter. In Hardware kann die Ausführung wieder auf wenige Taktzyklen heruntergebrochen werden.

Aufgrund dieser Unterscheidungen können die Zeiten für Operationen von beliebigen Prozessoren nicht einfach aus bekannten Daten heraus abgeschätzt werden. Zeiten müssen also durch Ausführung auf der Hardware ermittelt werden.

Um realistische Zahlen zu ermitteln, muss bei der Entwicklung des Testprogramms der Compiler berücksichtigt werden. Üblicherweise werden Programme mit Optimierungen kompiliert, gängig ist ein Level $O2$ oder $O3$, wobei höhere Zahlen eine aggressivere Optimierung bedeuten und $O0$ eine Kompilierung ohne Optimierungen darstellt. Der Compiler wird mit Optimierungen immer versuchen, die Rechenzeit zu minimieren und die gewünschten Ergebnisse mit möglichst wenigen oder einfachen Instruktionen zu berechnen. Eine wichtige Optimierung ist hierbei die Stärkenreduktion (engl. *strength reduction*). Sie hat zum Ziel, eine teure Instruktion, die für die Ausführung mehrere Taktzyklen benötigt, durch eine einfachere Instruktion zu ersetzen. Als Beispiel kann eine Multiplikation mit 4 durch eine Bitverschiebung um 2 realisiert werden. Eine andere Optimierung führt Berechnungen mit Konstanten bereits während der Kompilierung durch. So wird bei einem Ausdruck $i = 23 * 34$ keine Berechnung auf der Hardware durchgeführt, sondern der Wert 782 direkt der Variablen i zugewiesen. Aber auch dies ist abhängig von der weiteren Verwendung und unter Umständen wird Variable i gar nicht für die weitere Berechnung benötigt und gleich ganz heraus optimiert. Der Code des Testprogramms muss also so geschrieben sein, dass die Operationen, die gemessen werden sollen, auf jeden Fall ausgeführt werden und nicht durch einfache Optimierungen durch andere Operationen ersetzt werden können. Ein Beispiel einer Funktion mit 100 Multiplikationen ist in Quellcode 2.4 dargestellt. Um eine Optimierung von Konstanten zu verhindern, werden die verwendeten Variablen in Zeilen 2 bis 6 mit Zufallszahlen initialisiert. Die Funktion wird als C++ Template-Funktion realisiert, so dass der Datentyp mittels des Template-Parameters T über verschiedene Aufrufe hinweg verändert werden kann. Die Anzahl an Schleifeniteration wird über einen Funktionsparameter (Zeile 1) angegeben und kann somit auch nicht zur Optimierung der For-Schleife (ab Zeile 7) eingesetzt werden. Die eigentlichen Multiplikationen werden im Code explizit 100 mal ausgeschrieben. Beispielhaft sind in Zeile 8 und 9 die ersten 10 Multiplikationen dargestellt. Die verkettete Berechnung mit 5 Variablen sorgt dafür, dass jede einzelne Multiplikation auch wirklich durchgeführt werden muss, um das Endergebnis in Variable c zu berechnen.

```
1 static uint32_t profileMul(uint32_t iterations) {
```

```
2     T a = rand();
3     T b = rand();
4     T c = rand();
5     T d = rand();
6     T e = rand();
7     for (uint32_t i = 0; i < iterations; ++i) {
8       d = a*b; e = b*c; a = c*d; b = d*e; c = e*a;
9       d = a*b; e = b*c; a = c*d; b = d*e; c = e*a;
10      // more manual repetitions
11    }
12    return (uint32_t)c;
13  }
```

Quellcode 2.4: Testfunktion mit 100 Multiplikationen

Die Ausführung von multiplen Operationen in einer Schleife hat dabei mehrere Vorteile: zum einen sollen hier Instruktionen gemessen werden, die einzeln nur wenige Taktzyklen benötigen. Da eine einzelne Messung abhängig von der Software und dem Zugriff auf Zähler oder vergleichbare Hardware-Einheiten aber in der Regel deutlich mehr Zeit benötigt, ist die Messung zu ungenau um die Dauer von einzelnen Instruktionen zu messen. Der Mittelwert über alle ausgeführten Instruktionen erhöht hier die Genauigkeit drastisch. Da das Programm zudem mit *O*2 kompiliert wurde, können durch die vielen Ausführungen noch die Einflüsse von anderen Faktoren gemittelt dazu genommen werden. Da moderne Prozessoren über Instruktions-Pipelines verfügen und eine Multiplikation im C-Code aus einzelnen Schritten wie `instruction fetch`, `decode`, `load` und `store` besteht, wird durch diese durchschnittliche gemessene Dauer eine typische Ausführungszeit einer Instruktion ermittelt. So kann sich beispielsweise auch eine Dauer von 0,5 Taktzyklen für eine Addition ergeben, wenn die Pipeline zwei Additionen gleichzeitig ausführen kann.

Werden die so ermittelten Werte für Instruktionen zur Abschätzung von Ausführungszeiten von Tasks oder ganzen Funktionen und Programmen verwendet, so muss beachtet werden, dass alle weiteren Optimierungen, die sich aus dem größeren Kontext aller Kontrollstrukturen ergibt, nicht mitbetrachtet werden können. Die Zahlen können also nur in Relation zueinander verwendet werden, um die Größenordnung des Speedups der parallelen Anwendung abschätzen zu können.

Ermittlung der Anzahl an Aufrufen

Bei der statischen Code-Analyse zur Ermittlung der Aufrufhäufigkeiten wird der Quellcode statisch analysiert, das bedeutet, ohne dass das Programm kompiliert und ausgeführt wird. Nur für Schleifen, deren Kopf mit konstanten Ausdrücken dargestellt werden und die keine Änderungen an der Schleifenvariablen im Rumpf beinhalten, kann die Anzahl an Iterationen bestimmt werden. Als *einfache* Schleife werden For-Schleifen bezeichnet, für die gilt:

- Es existiert eine ganzzahlige Schleifenvariable, die im Kopf der Schleife verwendet wird und im Rumpf ausschließlich gelesen wird.

- Die Schleifenvariable wird vor der ersten Iteration auf einen festen Wert gesetzt. Dies kann im Initialisierungsblock der Schleife geschehen oder bereits davor.
- Im Evaluationsblock wird die Schleifenvariable mit einem konstanten Wert verglichen. Dies umfasst <, <=, >, >= und ==.
- Im Schrittblock wird die Schleifenvariable mit einer konstanten Variablen und einer der arithmetischen Operatoren +=, -= oder *= verändert.

Für solche einfachen Schleifen kann die Anzahl an Iterationen, die Schrittweite sowie der minimale und maximale Wert der Schleifenvariablen bestimmt werden.

Neben der Zeit pro Operation wird die Ausführungshäufigkeit der einzelnen Tasks benötigt. Für einfache Schleifen der Art „`i = 0; i < 100; ++i;`“ ist die Ermittlung trivial und kann direkt aus der Bedingung entnommen werden. Häufig gibt es aber Fälle, die von (Eingangs-) Daten abhängen. Gerade bei If-Blöcken oder While-Schleifen würde für eine statische Analyse sehr viel Wissen über den aktuellen Wert von Variablen benötigt, so dass die Analyse sehr schnell komplex wird oder gar keinen Wert ermitteln kann. Aus diesem Grund wurde stattdessen auf das Profiling mit Hilfe von instrumentiertem Code gesetzt. Eine beispielhafte Messung ist in Quellcode 2.5 dargestellt. Zunächst wird ein globales Array `__PROFILE` mit 64 Bit vorzeichenlosen Integer-Werten erzeugt (Zeile 1). Der Wert `NUM` ist bei der Code-Generierung bekannt und gibt die Anzahl an instrumentierten Stellen im Code an. Dabei werden die folgenden Positionen betrachtet:

- Start einer Funktion (Zeile 10)
- Beginn eines Schleifen-Body (Zeile 19)
- Der `then`- und der `else`-Zweig eines If-Blocks (Zeile 12 und 24)

Jede Position hat eine eindeutige Identifikationsnummer `ID`, deren Position im Array `PROFILE` bei jedem Aufruf um 1 erhöht wird. Am Ende des Programms wird das Array in einer Datei ausgegeben (Zeile 31).

```
1  uint64_t __PROFILE[NUM];
2
3  void write_profiling_data(uint64_t* Profile, int Count, const char*
       FileName) {
4    FILE* fp = fopen(FileName, "wb");
5    fwrite(Profile, sizeof(*Profile), Count, fp);
6    fclose(fp);
7  }
8
9  int function() {
10   __PROFILE[0] += 1ul;
11   if (/*condition*/)) {
12     __PROFILE[3] += 1ul;
13     if (x1_data > x2_data) {
14       __PROFILE[1] += 1ul;
15       tx_data = x1_data;
16       y2_data = ty_data;
17     }
18     for (i1 = 0; i1 < (d_data + 1); i1++) {
19       __PROFILE[2] += 1ul;
```

```
20          x_data = x1_data + i1;
21          y_data = y1_data + chain1_data;
22        }
23      } else {
24        __PROFILE[6] += 1ul;
25        if (y1_data > y2_data) {
26          __PROFILE[4] += 1ul;
27          tx_data = x1_data;
28          y2_data = ty_data;
29        }
30      }
31      write_profiling_data(__PROFILE, NUM, "function.profdat");
32  }
```

Quellcode 2.5: Instrumentierung des C-Quellcodes

Der so instrumentierte Code kann anschließend auf dem Entwicklungsrechner ausgeführt werden. Eine Ausführung auf dem Zielsystem ist nicht notwendig, da die Anzahl an Aufrufen auf beiden Systemen bei gleichen Eingangsdaten identisch sein muss.

Die ermittelten Werte können nun zusammen mit der statischen Abschätzung der Ausführungszeiten der Codeteile zur Abschätzung der Ausführungszeit der Anwendung verwendet werden. Im hier beschriebenen Workflow können mit Hilfe der Daten Pragmas im generierten C-Code eingefügt werden, die die absolute und die relative Anzahl an Aufrufen darstellen. Beispiele können Quellcode 2.6 entnommen werden. Die erste Zahl hinter dem Pragma `PERFINFO` stellt die absolute Anzahl an Aufrufen des aktuellen Code-Abschnitts dar. Die zweite Zahl steht für die relative Anzahl an Aufrufen im Vergleich zum übergeordneten Block. Eine Verteilung von 16 und 4 wie in Zeile 1 sagt aus, dass der Abschnitt bei der Programmausführung insgesamt 16-mal ausgeführt wird. Dabei findet eine Ausführung bei jeder Ausführung des übergeordneten Blocks statt. In Zeile 4 ist ein Beispiel dargestellt, bei der der aktuelle Block bei jeder Ausführung des Elternblocks 32-mal ausgeführt wird. Dies wird durch Schleifen im Code erzeugt. Bei If-Bedingungen schließlich gibt es den Fall, dass die relative Anzahl an Aufrufen kleiner als 1 sein kann. Wie in Zeile 7 dargestellt, wird der `then`-Block des If-Blocks nur in 33 % der Fälle ausgeführt, der `else`-Block hingegen in 67 % der Fälle.

```
1   #pragma PERFINFO 16 4
2
3   for (i = 0; i < 32; i++) {
4   #pragma PERFINFO 512 32
5
6   if(/*condition*/) {
7   #pragma PERFINFO 171 0.33
8   } else {
9   #pragma PERFINFO 342 0.67
10  }
```

Quellcode 2.6: Absolute und relative Anzahl an Aufrufen

Da die Anzahl an Aufrufen durch die tatsächliche Ausführung ermittelt werden, hängen die Werte von den gewählten Eingangsdaten ab. Eine einmalige Ausführung mit festen Daten liefert in diesem Fall nur unzureichende Daten und muss für andere Daten wiederholt werden.

Analyse des Quellcodes

Mit den gemessenen Zeiten für einzelne Operationen sowie der ermittelten Anzahl an Aufrufen, kann nun der Quellcode analysiert werden. Dazu wurde in GeCoS (siehe Abschnitt 2.3.3.1) eine Transformation gemäß des Visitor Patterns entwickelt. Beispielhafte Code-Abschnitte sind in Quellcode 2.7 dargestellt. Der Visitor besucht alle Objekte, die den Code repräsentieren mit einer Tiefensuche, die beginnend bei Funktionen hin zu Blöcken (Zeile 1) bis hinunter zu Instruktionen (Zeile 8) geht, bevor die nächsten Instruktionen besucht werden. Vor dem herabsteigen wird der aktuelle Kontext gespeichert (Zeile 16), der den aktuellen Zählerstand sowie die aktuelle Tiefe enthält. Dies wird wiederholt, bis ein Basisblock mit Instruktionen erreicht wird. Die Ausführungszeit des Blocks kann nun ermittelt werden, indem für jede Instruktion innerhalb des Blocks die zuvor gemessene Zeit genommen wird und mit der ermittelten Anzahl an Aufrufen des Blocks multipliziert wird. Beim Wiederaufsteigen aus dem Block heraus kann nun der Kontext auf den zuvor gespeicherten Kontext addiert werden (Zeile 26) und ergibt somit die Ausführungszeit des Elternblocks. Alle so ermittelten Zeiten werden als Zusatzinformationen (Annotationen) direkt an die entsprechenden Blöcke gehängt (Zeile 31).

```
1  public void visitIfBlock(IfBlock i) {
2    Context lc = visitBefore(i);
3    super.visitIfBlock(i);
4    annotateBlock(i);
5    visitAfter(i, lc);
6  }
7
8  public void visitGenericInstruction(GenericInstruction n) {
9      Context lc = visitBefore(n);
10
11     c.counter += cycleProvider.getCycleCount(n);
12     super.visitGenericInstruction(n);
13     visitAfter(n, lc);
14 }
15
16 public Context visitBefore(EObject ast) {
17   Context lc = new Context();
18
19   lc.counter = c.counter;
20   c.depth += 1;
21   c.counter = 0;
22
23   return lc;
24 }
25
26 public void visitAfter(EObject ast, Context lc) {
```

```
27      c.depth -= 1;
28      c.counter += lc.counter;
29    }
30
31    private void annotateBlock(Block b) {
32      double execCount = numExec.getNumberOfExecutions(b).absolute;
33      double execCountBody = numExec.getNumberOfExecutions(b).relative;
34
35      Annotation.addTotalTimesExecutedCount(b, execCount);
36      Annotation.addTimesExecutedByParentLoop(b, execCountBody);
37
38      Annotation.addTotalEstimatedCount(b, c.counter * execCount);
39      Annotation.addEstimatedCount(b, c.counter);
40    }
```

Quellcode 2.7: Relevante Methoden der Performanzabschätzung

Wie bereits in Abschnitt 2.4.1.1 dargestellt, stimmen die so ermittelten Werte nicht zwingend mit der tatsächlichen Ausführung auf der Hardware überein, können aber verwendet werden, um länger andauernde Code-Abschnitte zu identifizieren indem die Zeiten in Relation zueinander verwendet werden.

2.4.3.2. Realisierung des Profiling

Um das tatsächliche zeitliche Verhalten der Anwendung auf der Zielhardware zu ermitteln, wurde neben der statischen Codeanalyse noch ein Profiling mit instrumentiertem Code auf der Zielhardware eingesetzt. Dazu werden an passenden Stellen im Quellcode Instruktionen eingefügt, die den Start und das Ende eines Codeabschnitts markieren und einen entsprechenden Zähler starten bzw. stoppen. Wichtig hierbei ist, dass sichergestellt werden kann, dass der erzeugte Code auch in der erwarteten Reihenfolge ausgeführt wird. Bestehen keine Abhängigkeiten zwischen den Anweisungen eines Tasks, so kann der Compiler die Anweisungen auch in einer anderen Reihenfolge bearbeiten. Aus diesem Grund eignen sich für das Profiling am besten Funktionen, die mit Komponenten außerhalb des Prozessors interagieren. Beispiele sind Schreibzugriffe auf explizite Speicheradressen oder Ausgangsports. C-Compiler können hier nicht sicher bestimmen, ob die Reihenfolge verändert werden darf oder nicht. Deshalb wird sichergestellt, dass die ursprüngliche Reihenfolge zumindest in Relation zu diesen Schreibzugriffen beibehalten wird. Start und Ende eines Tasks können auf diese Weise eindeutig festgelegt werden. Werden die Anweisungen dazwischen vertauscht, wird dennoch die Zeit für alle Anweisungen gemessen.

Eine Zeitermittlung auf diese Weise muss anschließend von externen Komponenten unterstützt werden. Dies kann beispielsweise ein Simulationsframework sein, das auf die erzeugten Signale reagiert und entsprechende Zeiten akkumuliert. Wichtig ist, dass der Simulator mindestens zyklenapproximativ ist, d.h. dass zumindest ein zeitliches Modell für die Instruktionen existiert und nicht nur rein auf funktionaler Ebene simuliert wird.

Der größte Nachteil dieser Methode ergibt sich aus der Laufzeit der hinzugefügten Instruktionen. Um ungewollte Optimierung des Compilers zu verhindern, müssen tatsäch-

liche Instruktionen verwendet werden, die bei der späteren Ausführung auf der Hardware oder der Simulation selbst Zeit benötigen. Dabei kann schon eine Dauer von einem Zyklus einen beträchtlichen Einfluss auf die gesamte Ausführung haben. Betrachtet man beispielsweise eine häufig auftretende, zweifach verschachtelte For-Schleife, bei der in der innersten Schleife eine simple Operation durchgeführt wird, wie sie in Quellcode 2.8 dargestellt wird. Die Operation besteht aus zwei Additionen, für die der Einfachheit halber jeweils eine Zyklendauer von eins angenommen wird. Dazu kommen Zugriffe auf die Variablen x, i und j, welche abhängig von Cache und Compileroptimierungen jedoch vernachlässigt werden können, da die Werte in der Regel in den Registern des Prozessors behalten werden können. Für die innere Schleife kann deshalb eine Dauer von etwa zwei Zyklen geschätzt werden, wenn kein Profiling durchgeführt wird. Durch Hinzunahme der neuen Instruktionen verdoppelt sich die Dauer auf vier Zyklen, die aber nicht direkt bei der Messung der Additionen auftreten, sondern erst bei der Messung der beiden For-Schleifen. Die Dauer der inneren wird um 20 Zyklen erhöht, die der äußeren sogar um 200 Zyklen. Ein weiterer Nachteil besteht darin, dass die hinzugefügten Instruktionen bewusst so gewählt werden, dass der Compiler die Reihenfolge der Instruktionen nicht verändern kann. Diese Einschränkung sorgt dafür, dass die Ausführung des instrumentieren Codes länger dauert, als die Zeit, die allein für die Ausführung der neuen Befehle benötigt wird.

Um die Nachteile auszugleichen und den Messfehler so gering wie möglich zu halten, kann die Anzahl der tatsächlich gemessenen Zeiten reduziert werden. So ist es in der Regel nicht notwendig, Zeiten für jede Iteration einer innersten Schleife zu ermitteln. Wird stattdessen nur die Dauer der gesamten Schleife oder sogar der direkt darüber liegenden Schleife gemessen, kann der Overhead der Messung deutlich reduziert werden und Compileroptimierungen können besser auf den Code angewendet werden. Für das Beispiel in Quellcode 2.8 würde es also ausreichen, die Zeiten außerhalb von beiden For-Schleifen zu ermitteln. Zur Vereinfachung können die in Abschnitt 2.4.3.1 ermittelten Zeiten als Referenz verwendet werden und alle Blöcke, die einen zuvor gesetzten Schwellwert nicht überschreiten, können vom Profiling ausgeschlossen werden. Es werden also nur noch Zeiten von Tasks ermittelt, die eine relevante Dauer vorweisen können. Damit reichen sie zugleich aus, um die Parallelisierung auf Task-Ebene zu evaluieren, da eine feinere Aufteilung auch nicht betrachtet wurde.

```
1  for (i = 0; i < 10; i++) {
2    for (j = 0; j < 64; ++j) {
3      //start_profiling
4      x += i + j;
5      //end_profiling
6    }
7  }
```

Quellcode 2.8: Zweifach verschachtelte For-Schleife

2.4.4. Zusammenfassung der Performanzabschätzung

Für die Performanzwerte in den nächsten Schritten wurden zwei Möglichkeiten zur Ermittlung der Werte implementiert.

Mit Hilfe einer einfachen statischen Analyse des Quellcodes können Tasks identifiziert werden, die besonders viel Zeit bei der Ausführung benötigen. Unterstützt wird der Ansatz durch Profiling auf der Hardware. Dabei werden zum einen Ausführungshäufigkeiten mit Beispieldaten ermittelt. Zum anderen werden für jede Hardware die Anzahl an Taktzyklen gemessen, die für eine Operation benötigt wird.

Profiling mittels instrumentiertem Quellcode wird verwendet, um tatsächliche Zeiten auf der Hardware zu ermitteln. Dabei wird auf atomare Instruktionen gesetzt, um sicherzustellen, dass die Ausführung tatsächlich in der gewünschten Reihenfolge durchgeführt wird und nicht durch den Compiler verändert wird.

2.5. Parallele Anwendungen

Parallele Anwendungen im Rahmen dieser Arbeit sind Programme, die explizit für die Ausführung auf mehreren Kernen optimiert wurden. Das bedeutet, dass sowohl die Aufteilung auf parallele Stränge als auch deren Synchronisation durch explizite Anweisungen im Quellcode geregelt wird und nicht notwendigerweise durch ein Betriebssystem erfolgen muss.

Die allgemeinere Form paralleler Anwendungen wird als Multi-Tasking-Programmierung bezeichnet. Dabei wird eine einzelne Anwendung in Tasks zerlegt, die zwar alle miteinander in Beziehung stehen, aber nicht notwendigerweise nacheinander ausgeführt werden müssen. Ein Beispiel ist eine Anwendung, die auf Daten von einem Sensor warten muss. Ist sie rein sequentiell programmiert, muss an einer bestimmten Stelle im Quellcode auf die eintreffenden Daten gewartet werden. Dies kann nur aktiv durch sogenanntes *Polling* geschehen, bei dem der Prozessor aktiv Daten aus einem (Zwischen-)Speicher liest, bis die neuen Werte eintreffen. In diesem Fall kann der Prozessor nichts anderes tun, bis alle Daten vorhanden sind. Werden an dieser Stelle parallele Tasks verwendet, kann der Empfang der Daten ausgelagert werden. Das Betriebssystem auf diesem einen Kern kann nun dafür sorgen, dass regelmäßig zwischen den einzelnen Tasks der Anwendung gewechselt wird. Auf diese Weise kann der Prozessor noch andere Aufgaben erledigen, obwohl die Daten noch nicht vollständig vorliegen. Da es in dieser Arbeit jedoch um die Beschleunigung der Anwendungsleistung durch Parallelisierung auf mehrere Ausführungseinheiten und damit um eine bessere Ausnutzung der vorhandenen Hardware-Ressourcen geht, wird das reine Multi-Tasking auf einem einzelnen Kern nicht weiter betrachtet.

2.5.1. Grundlagen und Begriffserklärungen

2.5.1.1. Aufteilung von Programmen

Mapping und Scheduling Unter Mapping oder auch auf Deutsch „Zuweisung“ versteht man die Auswahl eines bestimmten Prozessors oder -kerns des Rechensystems zur Abarbeitung eines Tasks. Dabei werden dynamisches und statisches Mapping unterschieden. Dynamisch findet zur Laufzeit statt und wird in der Regel von einem Betriebssystem erledigt. Dabei wird ein Task auf einem verfügbaren Kern ausgeführt, kann aber durch andere parallele Vorgänge vom Betriebssystem verdrängt und zu einem späteren Zeitpunkt auf einem anderen Kern wieder fortgesetzt werden. Das Mapping kann sich zur Laufzeit ändern und ist nicht fix über die gesamte Ausführungsdauer. Statisches Mapping erfolgt zur Compile-Zeit und weist Kerne bereits fix über die gesamte Ausführung zu. Damit verringert sich der Overhead, der durch das Stoppen und die Wiederaufnahme auf einem anderen Kern erzeugt wird. Dafür können Kerne nicht dynamisch mit anderen Tasks belegt werden, während sie nichts anderes zu tun haben.

Das Mapping wird üblicherweise immer zusammen mit dem Scheduling, auch Ablaufplanung genannt, durchgeführt. Dabei werden die Abhängigkeiten zwischen den einzelnen Tasks analysiert und eine optimierte Reihenfolge für die Ausführung bestimmt. Von einem Betriebssystem kann diese Reihenfolge verwendet werden, um dynamisch neue Aufgaben auf verfügbaren Kernen zu starten. Das statische Scheduling kann eingesetzt werden, um die Ausführung auf den einzelnen Kernen hinsichtlich Auslastung und Reduktion von Wartezeiten zu optimieren.

Prozess Ein Prozess ist eine Instanz von einem Computerprogramm, das gerade ausgeführt wird. Programme können aus mehreren Prozessen bestehen. Jeder Prozess hat seinen eigenen Speicher und kann nicht direkt auf Daten eines anderen Prozesses zugreifen. Dazu werden spezielle Funktionen zur Interprozess-Kommunikation benötigt, die in der Regel von einem Betriebssystem oder einem Standard (z. B. POSIX [37]) vorgegeben wird.

Task Ein Task oder zu Deutsch Aufgabe ist eine Arbeitseinheit und Teil einer ganzen Anwendung. Die Bezeichnung ist nicht auf die Verwendung in parallelen Anwendungen beschränkt und wird im Folgenden auch für Teile von sequentiellen Algorithmen verwendet.

Thread Unter einem Thread oder auf Deutsch auch Ausführungsstrang versteht man parallele Teile einer Anwendung, die in der Regel von einem Prozess aus gestartet werden. Sie teilen alle Ressourcen und haben einen gemeinsamen Adressraum und können somit auf Daten von anderen Threads zugreifen. Jeder Thread wird zu einem Zeitpunkt nur von einem einzelnen Kern ausgeführt, dieser kann aber zur Laufzeit durch ein Betriebssystem wechseln. Auf diese Weise können sich mehrere Threads zur gleichen Zeit mehrere Prozessorkerne teilen.

2.5.1.2. Synchronisationsmechanismen

Im Folgenden werden gängige Konzepte zur Synchronisierung von verschiedenen Kernen dargestellt.

Semaphor Die Synchronisierung mittels eines Semaphors basiert darauf, dass nur der Kern, der einen Token hat, auf die Ressource zugreifen kann. Dazu fragt jeder Kern vor Benutzung nach einem Token. Bekommt er einen, darf er auf die Ressource zugreifen und muss nach dem Zugriff den Token zurückgegeben. Eine übliche Implementierung setzt auf einen Zähler, der mit der maximalen Anzahl an gleichzeitigen Zugriffen initialisiert wird. Hat ein Kern einen Token, so wird der Zähler verringert, gibt er ihn zurück, so wird der Zähler wieder erhöht. Steht der Zähler auf 0, müssen alle weiteren Kerne warten, bis ein Token zurückgegeben wird. Die Phasen werden auch als „Reservieren“ und „Freigeben“ bezeichnet. Wichtig bei der Realisierung ist der Einsatz von atomaren Instruktionen, so dass Reservieren oder Freigeben immer abgeschlossen ist, bevor eine andere Aktion abgearbeitet werden kann.

Mutex Ein Mutex ist eine Spezialform eines Semaphors, bei dem nur ein einzelner gleichzeitiger Zugriff erlaubt ist (**mut**ual **ex**clusive).

Barriere Mit einer Barriere kann im Code eine Stelle markiert werden, an dem alle parallelen Stränge aufeinander warten müssen. Erst wenn alle die Barriere erreicht haben, wird mit der Verarbeitung fortgesetzt. Da im worst-case alle bis auf einen Kern warten, sollten Barrieren nur eingesetzt werden, wenn zu erwarten ist, dass sie etwa zeitgleich von allen Kernen erreicht wird. Im Idealfall führen alle Kerne die gleichen Berechnungen auf unterschiedlichen Daten aus, so dass die Ausführungszeiten nahezu identisch sind.

Fence Unter Fence (Zaun) oder auch Speicherbarriere versteht man eine Instruktion im Quellcode, die dem Compiler verbietet, Instruktionen über den Fence hinweg zu verschieben. Übliche Anwendungsfälle sind Seiteneffekte von Instruktionen vor und nach dem Fence oder Datenabhängigkeiten, die erfordern, dass die Instruktionen in einer bestimmten Reihenfolge ausgeführt werden müssen. In diesem Sinn ist ein Fence eigentlich kein Synchronisationsmechanismus bei der Parallelisierung, nichtsdestotrotz spielt er auch bei der Parallelisierung eine Rolle. Da Fences üblicherweise eingesetzt werden, wenn die entsprechenden Instruktionen Seiteneffekte haben, müssen diese korrekt bei der parallelen Ausführung beachtet werden.

2.5.2. Herausforderungen und Risiken

Bei der Parallelisierung gibt es Herausforderungen, die bei der rein sequentiellen Programmierung nicht auftreten können. Bei Nichtbeachtung oder fehlerhafter Handhabung können diese Fälle aufgrund ihres nichtdeterministischen Verhaltens zu fehlerhaften Programmen oder zu völlig unerwarteten Performanzproblemen führen. Im Folgenden werden die am häufigsten auftretenden Fälle beispielhaft dargestellt, eine umfassendere Auflistung kann [38] entnommen werden.

2.5.2.1. Race Conditions

Die so genannten „Race Conditions“, zu Deutsch „Wettlauf-Situationen“ treten auf, wenn das Ergebnis einer Berechnung oder Operation vom Zeitpunkt der Fertigstellung einer anderen Operation abhängt. Die hier vorrangig beschriebenen Daten-Races wurden erstmals in [39] erwähnt. Der Begriff „Race Condition“ wurde ursprünglich im Zusammenhang von integrierten Schaltkreisen verwendet [40], wurde aber bereits in [41] bei der Entwicklung von parallelen Anwendungen aufgegriffen. Der folgende Text basiert auf der eigenen Zusammenfassung in [42]. Häufig entstehen sie im Zusammenspiel mit gemeinsam genutzten Ressourcen wie Speicher oder Schnittstellen. Als Beispiel kann die Operation aus Tabelle 2.1 genommen werden. Die angedachte Operation ist das Erhöhen einer gemeinsam genutzten Variablen von zwei Kernen aus. Heute übliche Prozessoren werden in der Regel mit Pipelining realisiert, so dass eine Operation in mehrere Teilschritte aufgeteilt wird, von denen nur jeweils eine in einem Taktschritt ausgeführt wird. Da ein Prozessor pro Taktschritt mehrere dieser Teilschritte parallel ausführen kann, erhöht sich der Durchsatz an Instruktionen und damit die Performanz des Prozessors. Im gewählten Beispiel führt jeder Prozessor die drei Einzelschritte `lesen`, `addieren` und `schreiben` aus. Hat die Variable zu Beginn den Wert 0, wird sie von Kern 1 auf 1 erhöht und anschließend von Kern 2 auf 2 erhöht. Bei paralleler Ausführung kann es nun passieren, dass die Befehle der beiden Kerne überlappend ausgeführt werden. Kern 2 liest beispielsweise nur leicht versetzt nach dem ersten Kern die gemeinsame Variable. Da Kern 1 den Wert noch nicht erhöht hat, liest auch Kern 2 den Anfangswert von 0. Anschließend erhöhen beide Kerne die Variable um 1 und schreiben ihr jeweiliges Ergebnis zurück. Da beide Kerne nichts von der Änderung des jeweils anderen Kerns mitbekommen haben, wird der Wert am Ende nur um eins erhöht und das falsche Ergebnis in die Variable geschrieben.

Auch wenn der Fall auf den ersten Blick sehr trivial erscheint, wird er in ähnlicher Form häufig als einfache Möglichkeit zur Synchronisation von Zugriffen auf gemeinsam genutzte Ressourcen verwendet. So kann eine naive Synchronisation implementiert werden, indem eine globale Variable definiert wird, auf die alle Kerne zugreifen können. Zur Synchronisation soll ein Kern darauf warten, dass die Variable einen bestimmten Wert annimmt. Gesetzt wird der Wert von einem oder mehreren anderen Kernen. Wird dabei nicht beachtet, dass die Operation aus mehreren Einzelschritten besteht und deshalb über mehrere Taktzyklen hinweg abgearbeitet wird, kann sie durch eine parallele Verarbeitung eines anderen Kerns verfälscht werden. Das Verhalten der Anwendung wird dadurch unvorhersehbar. Bei mehreren Ausführungen kann es von geringsten kleinen zeitlichen Schwankungen, die beispielsweise durch das Betriebssystem erzeugt werden, abhängen, ob das Programm korrekt arbeitet oder nicht. Aufgrund der deutlich aufwendigeren Reproduzierbarkeit des Fehlers und der damit einhergehenden zeitraubenden Fehlersuche, wird eine nachträgliche Fehlerbehebung sehr kostspielig.

In der Praxis stehen verschiedene Möglichkeiten bereit, um das Auftreten von Race Conditions zu verhindern. Um eine zuverlässige Synchronisation von Kernen über den gemeinsamen Speicher hinweg zu gewährleisten, ist der Einsatz von atomaren Operationen notwendig. Diese stellen eine Menge von Einzelbefehlen dar, die immer zusammen ausgeführt werden und als Ganzes erfolgreich beendet werden müssen. Auf diese Weise kann sichergestellt werden, dass kein gleichzeitiger Zugriff auf eine gemeinsame Variable erfolgen kann und ein anderer Kern warten muss, bis die Variable wieder zugreifbar ist.

Allgemein können Race Conditions verhindert werden indem die Kerne im Verlauf des Programms an sinnvollen Stellen synchronisiert werden.

Takt	Kern 1	Kern 2	Wert
1	Lesen		0
2	Addieren		0
3	Schreiben		1
4		Lesen	1
5		Addieren	1
6		Schreiben	2

(a) Sequentieller Zugriff

Takt	Kern 1	Kern 2	Wert
1	Lesen		0
2	Addieren	Lesen	0
3	Schreiben	Addieren	1
4		Schreiben	1

(b) Überlappender Zugriff

Tabelle 2.1.: Simple Lese-/Schreiboperation mit parallelem Zugriff

2.5.2.2. Deadlocks

Der folgende Text ist eine Erweiterung der Zusammenfassung aus [43].

Ein Deadlock, auf Deutsch auch ‚Verklemmung' genannt, beschreibt einen Programmzustand, in dem einer oder mehrere Prozesse auf Bedingungen warten, die niemals auftreten können. In diesem Zustand kann das Programm nicht weiter abgearbeitet werden. Deadlocks können beispielsweise entstehen, wenn auf gemeinsam genutzte Ressourcen in unterschiedlicher Reihenfolge zugegriffen wird.

Ein Beispiel ist in Abbildung 2.17 dargestellt. Neben den beiden Kernen 1 und 2 sind zwei gemeinsam genutzte Ressourcen 1 und 2 vorhanden. Der Zugriff auf die Ressourcen wird über einen Mutex geregelt: jeder Kern muss zunächst den exklusiven Zugriff anfordern. Erst wenn dieser gewährt wurde, kann die Ressource vom entsprechenden Kern verwendet werden. Ist die Ressource in Benutzung, müssen andere Anfragen warten, bis der Zugriff gewährt wird. Nach Verwendung müssen Kerne mitteilen, wenn der Zugriff wieder freigegeben wird. Beide Kerne wollen eine Operation durchführen, für die sie jeweils beide Ressourcen benötigen. Im ersten Schritt fordern die beiden Kerne exklusiven Zugriff auf eine der beiden Ressourcen an: Kern 1 möchte Ressource 1 verwenden, Kern 2 Ressource 2. Der exklusive Zugriff wird gewährt, da es in beiden Fällen keine konkurrierenden Anfragen gibt. Im zweiten Schritt fordern beide Kerne zusätzlich die jeweils andere Ressource an, ohne die bereits gewährten Ressourcen freizugeben. Da nur exklusiver Zugriff erlaubt ist, müssen sowohl Kern 1 als auch Kern 2 auf die gegenseitige Freigabe der Ressourcen warten. Da beide Ressourcen belegt sind und vom Programmablauf her auch nicht wieder freigegeben werden, kommt es zu einem Deadlock und die Abarbeitung des Programms wird blockiert.

Ähnlich wie bei den ‚Race Conditions' können Änderungen am Timing der einzelnen Anfragen zu einem anderen Verhalten führen. Im beschriebenen Beispiel wäre dann eine Situation denkbar, bei der Kern 1 Zugriff auf beide Ressourcen bekommt, bevor Kern 2 exklusiven Zugriff auf Ressource 2 erhält. Der Deadlock tritt demnach nicht immer auf und man hat ein unbestimmbares Verhalten, das die Fehlerfindung erschwert.

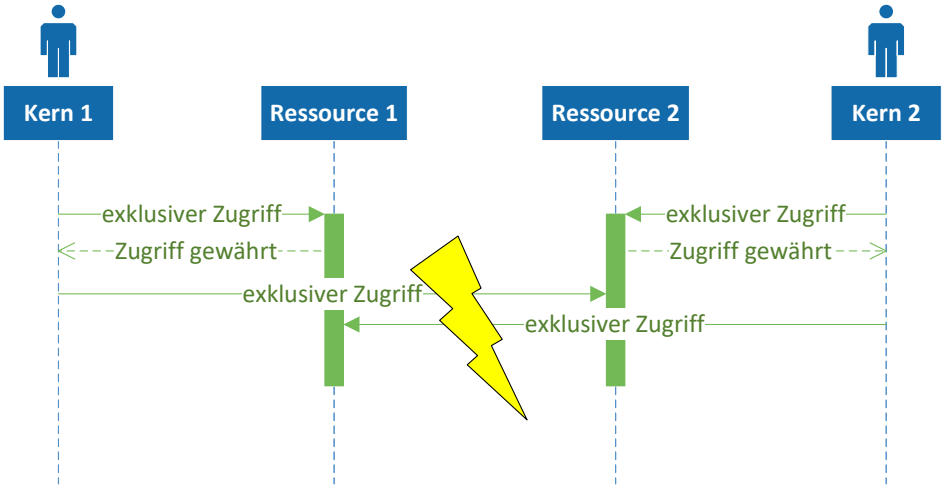


Abbildung 2.17.: Deadlock bei konkurrierendem Zugriff

Es existieren einige bewährte Methoden, um das Auftreten von Deadlocks zu verhindern:

- Vermeidung von exklusiven Zugriffen: Anstatt einem Prozessor den alleinigen Zugriff auf die Ressource zu geben, kann ein Puffer oder eine Warteschlange verwendet werden, um den Zugriff auf die Ressource nacheinander abzuarbeiten. Sind weitere Synchronisationen notwendig, weil beispielsweise zwei Ressourcen gleichzeitig verwendet werden müssen, so muss dies auf einer anderen Abstraktionsebene geregelt werden, beispielsweise durch Einsatz eines Managers.
- Vorausschauendes Scheduling der Zugriffe: Anstatt die Zugriffe auf die einzelnen Ressourcen nacheinander anzufordern, können alle benötigten zeitgleich angefordert werden. Nur, wenn alle verfügbar sind, wird der Zugriff gewährt. Auf diese Weise wird verhindert, dass mehrere überlappende Anfragen wie im Beispiel entstehen können.
- Priorisierung von Zugriffen: Allen Anfragen wird eine Priorität zugewiesen, die beispielsweise fix pro Kern vergeben wird. Anfragen werden dann immer nur nach der höchsten Priorität vergeben. Zugriffe mit niedriger Priorität werden entsprechend verzögert oder sogar entzogen. Bei der Realisierung sollte darauf geachtet werden, dass auch Zugriffe mit niedriger Priorität bedient werden, um die korrekte Funktionalität der Anwendung zu gewährleisten.
- Fixe Zugriffsreihenfolge: Um die Probleme des Beispiels zu umgehen, kann festgelegt werden, dass immer in der gleichen Reihenfolge auf die Ressourcen zugegriffen werden muss. Im Beispiel kann dies bedeuten, dass immer erst Ressource 1 angefordert werden muss, wenn gleichzeitiger Zugriff auf Ressource 1 und 2 benötigt wird.

2.5.2.3. Performanz

Die Performanz von parallelen Anwendungen hängt von vielen verschiedenen Faktoren ab. Im Folgenden werden alle Punkte aufgelistet, die sich im Vergleich zu sequentiellen Fällen anders verhalten.

Synchronisationsoverhead Ein großer Unterschied bei der Abschätzung paralleler Anwendungen im Vergleich zu sequentiellen entsteht durch die notwendige Synchronisation der einzelnen Kerne. Der tatsächliche Zeitaufwand für eine einzelne Synchronisation hängt dabei sehr stark von der Realisierung ab. Bei Systemen mit gemeinsamem Speicher reicht es aus, den Zugriff auf eine Variable zu gewähren. Ein einfaches Signal bzw. eine kurze Nachricht reicht hierfür aus und kostet die Kerne nicht viel Zeit. Bei Systemen mit verteiltem Speicher, in denen jeder Kern eigene Kopien von Variablen besitzt, müssen Daten erst von einem Speicher in einen anderen kopiert werden. Wird das von den Kernen selbst erledigt, sind diese für die Dauer des Kopiervorgangs beschäftigt. Effizienter ist der Einsatz von einer Speicherdirektzugriffs-Einheit (*Direct Memory Access* (DMA) *Unit*), die die Kopieroperation über den Bus auf Befehl selbstständig erledigen kann.

Der größere Aufwand entsteht jedoch durch die reinen Wartezeiten, die bei Kernen entstehen, die auf Daten von anderen Kernen warten müssen. Sind bei einer parallelen Anwendung nicht alle Kerne gleichmäßig ausgelastet, so gibt es immer Kerne, die auf Daten von anderen Kernen warten. Eine Verteilung kann in der Regel nie eine optimale Lastverteilung für alle Kerne über die gesamte Ausführung garantieren. Als häufigstes Beispiel kann die Aufteilung von Daten betrachtet werden: sollen mehrere Kerne parallel auf unterschiedlichen Teilen einer größeren Datenmenge arbeiten, so muss dies erst von einem einzelnen Kern aus gestartet werden. Dieser muss entweder den anderen Kernen synchronisiert mitteilen, auf welchen Daten sie arbeiten dürfen oder die Daten sogar aufteilen und entsprechend kopieren. In beiden Fällen warten alle anderen beteiligten Kerne, bis sie mit der Arbeit überhaupt anfangen können.

Die Betrachtung des Overheads bei der Abschätzung der Performanz von parallelen Anwendungen sollte also nicht unterschätzt werden und kann unter Umständen auch dafür sorgen, dass Anwendungen gar nicht erst sinnvoll parallelisiert werden können.

Gemeinsam genutzte Ressourcen Bei der parallelen Ausführung gibt es sehr häufig den Fall, dass Ressourcen gleichzeitig von mehreren Kernen verwendet werden. Häufigstes Beispiel ist die gemeinsame Verwendung des Speichers oder der Zugriff auf Bussysteme. Dies kann sich auch auf den eben beschriebenen Synchronisationsoverhead auswirken, da die Kommunikation zwischen zwei Kernen auch die Kommunikation zwischen anderen Kernen beeinflussen kann. Um die Verwendung dieser Ressourcen dennoch zu ermöglichen, findet in der Regel eine Arbitrierung statt. Im einfachsten 'Round Robin'-Verfahren wird reihum zwischen den Kernen gewechselt, komplexere Ansätze können auch mit Priorisierungen arbeiten. Allen Verfahren ist gemein, dass sie für eine Verzögerung beim Zugriff auf die Ressource sorgen, die bei der Abschätzung mit einbezogen werden muss.

Caches Wird der Hauptspeicher häufig von mehreren Kernen verwendet, so gilt das nur selten für Caches. Die schnellen Zwischenspeicher werden üblicherweise in verschiedene Level eingeteilt. Je niedriger das Level, desto schneller die Performanz und desto weniger Kerne teilen sich den Cache. So wird Level 1 Cache in der Regel nur von einem Kern verwendet, Level 2 meistens ebenso und erst ab Level 3 teilen sich Kerne den Cache. Diese Aufteilung sorgt jetzt dafür, dass bei einer Parallelisierung die Menge an verfügbarem Level 1 Cache vergrößert wird und allein dadurch ein Speedup erreicht werden kann. Hierdurch kann theoretisch ein super-linearer Speedup erreicht werden, der größer ist als die reine Summe der Rechenleistung. In der Praxis kann dieser Effekt aber nur selten beobachtet werden, da ein super-linearer Effekt erst entstehen kann, wenn die wichtigsten Daten der Anwendung nicht in den Cache von einem Kern passen, aber in den von zweien.

Betriebsmodi Um die Effizienz von Multicore-Prozessoren zu steigern, werden verschiedene Ansätze zur Optimierung der Leistungsaufnahme eingesetzt. Unter reaktiven Maßnahmen versteht man die Anpassung der Betriebsmodi zur Laufzeit. Sind Kerne nicht ausgelastet, können sie mit geringerer Taktfrequenz und Versorgungsspannung betrieben oder sogar ganz abgeschaltet werden. Das Umschalten zwischen verschiedenen Modi kann jedoch zu einer Verzögerung (Lag) führen, während der andere Kerne warten müssen (siehe [44]). Dieser Overhead kann nur durch detaillierte Hardwaremodelle abgebildet werden.

3. Konzeption der ebenenübergreifenden Parallelisierung

Unter Parallelisierung versteht man üblicherweise zwei verschiedene Ansätze: die feingranulare und die grobgranulare Parallelisierung. Die feingranulare Parallelisierung konzentriert sich auf die Ausnutzung der Datenparallelität, indem beispielsweise statt einer Addition mit zwei 32-Bit-Zahlen vier Additionen mit 8-Bit-Zahlen durchgeführt werden. Diese Art der Parallelisierung kann einen großen Einfluss auf die Leistung haben, geht aber über den Rahmen dieser Arbeit hinaus und wird im Folgenden nicht betrachtet. Unter grobgranularer Parallelisierung versteht man die Ausnutzung der parallelen Recheneinheiten eines Prozessors oder Systems. Üblicherweise geschieht dies auf der Ebene von Tasks, die ausführbare Teile der gesamten Anwendung darstellen. Ziel der grobgranularen Parallelisierung ist es nun, die Tasks den verschiedenen Recheneinheiten über die Zeit zuzuweisen. Diese Arbeit konzentriert sich auf diese Art der Parallelisierung, geht aber über das Konzept der grobgranularen Parallelisierung hinaus.

Der neue Ansatz in dieser Arbeit ist die Betrachtung der semi-automatisierten Parallelisierung eines abstrakt beschriebenen Algorithmus über mehrere Abstraktionsebenen hinweg. Die alleinige Konzentration auf die Parallelisierung von Tasks deckt nur einen Teil des Potentials zur Performanzsteigerung ab. Im Folgenden werden vier verschiedene Abstraktionsebenen eingeführt, auf denen die Parallelisierung bzw. eine Optimierung des Parallelisierungpotentials stattfinden kann. Anschließend wird das Gesamtkonzept der ebenenübergreifenden Parallelisierung vorgestellt. Die betrachteten Ebenen sind in Abbildung 3.1 dargestellt.

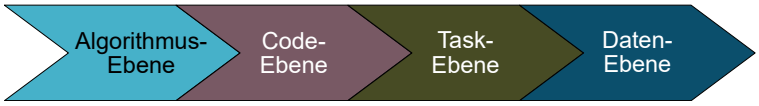


Abbildung 3.1.: Ebenen der Parallelisierung

3.1. Parallelisierungsebenen

3.1.1. Konzeption der Algorithmus-Ebene

Die Algorithmus-Ebene bezieht sich in dieser Arbeit auf die abstrakte Darstellung eines Algorithmus, der unabhängig von der tatsächlichen Hardware-Plattform, auf der er aus-

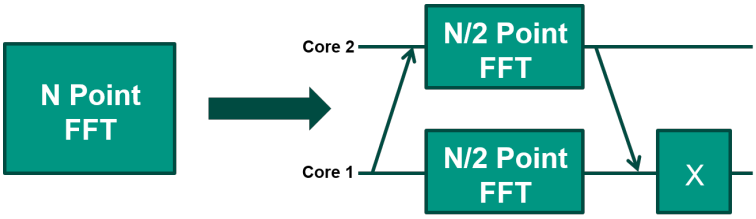


Abbildung 3.2.: Parallelisierung der FFT

geführt werden soll, entwickelt wurde. Die Auswahl und die Implementierung des Algorithmus hat dabei einen dramatischen Einfluss auf die Performanz.

Als Beispiel sollen hier Sortieralgorithmen verwendet werden: Es existieren viele verschiedene Algorithmen, die sich hinsichtlich Laufzeit, Speicherbedarf und der Stabilität unterscheiden. Stabil ist ein Sortieralgorithmus, der nach der Ausführung immer das gleiche Ergebnis liefert. Bei instabilen können Elemente mit der gleichen Wertigkeit vertauscht sein. Die Auswahl eines bestimmten Algorithmus bestimmt das Potential für die Parallelisierung sehr stark. Ein „*Binary Tree Sort*“ baut zunächst einen binären Suchbaum auf und traversiert diesen anschließend, um die Elemente in richtiger Reihenfolge auszugeben. Der Aufbau des Baumes basiert auf dem Einfügen von Elementen in eine Datenstruktur und lässt sich nur begrenzt parallelisieren. Das Traversieren ist ebenfalls nur eingeschränkt parallelisierbar. Andere Algorithmen wie *Quicksort* oder *Hyperquicksort* setzen auf Untermengen der tatsächlichen Elemente, welche dann unabhängig sortiert werden können. Sie eignen sich deshalb besser für die parallele Ausführung. Eine Umwandlung von einem Algorithmus hin zu einem anderen ist nicht ohne weiteres möglich und kann nicht automatisiert durch Analyse des ursprünglichen Algorithmus erreicht werden.

Andere Algorithmen wie die Fast Fourier Transformation (FFT) haben Eigenschaften auf der Algorithmus-Ebene, die für die Parallelisierung ausgenutzt werden können. Beispielsweise kann eine FFT mit 1024 Punkten durch 2 FFTs mit jeweils 512 Punkten und einen zusätzlichen Berechnungsschritt, der die Ergebnisse sammelt, ersetzt werden. Dies ist beispielhaft in Abbildung 3.2 dargestellt. Verallgemeinert kann eine n-Punkte FFT durch m FFTs mit n/m Punkten ersetzt werden. Auch diese Art der Ersetzung kann nicht vollständig automatisiert werden, sondern kann nur auf bereits bekannte Algorithmen angewendet werden. Abhängig von der Programmiersprache können Hardware-Eigenschaften bereits auf dieser Ebene zur Optimierung betrachtet werden. Denkbar sind bereits Implementierungen, die aus mehreren unabhängigen Schritten bestehen, die bereits gut parallel ausgeführt werden können oder Implementierungen, die bereits auf die Menge an genutztem Speicher hin optimiert wurden. In dieser Arbeit sollen beide Eigenschaften „parallele Implementierung“ und „intrinsische Parallelität“ betrachtet werden.

3.1.1.1. Anforderungen an die Optimierungen auf Algorithmus-Ebene

A1 Die zuverlässige Erkennung von Algorithmen muss sichergestellt werden. Dabei darf kein Algorithmus fehlerhaft klassifiziert werden, so dass keine unter Umstän-

den falschen Optimierungsschritte angeboten werden. Nichterkennung von optimierbaren Algorithmen ist zulässig, da die korrekte Funktionalität dennoch sichergestellt ist.

A2 Funktionale Änderungen, die durch andere Implementierung entstehen können, müssen dokumentiert werden. Dies gilt vor allem für Fälle, in denen Ergebnisse aufgrund von Genauigkeitsunterschieden von der ursprünglichen Realisierung abweichen.

A3 Optimierungen, die abhängig von der gewählten Zielplattform gesetzt werden müssen, wie beispielsweise eine Vorverteilung auf verschiedene Ausführungseinheiten, müssen als parametrisierbare Optionen bereitgestellt werden, um die Anforderungen von späteren Ebenen bedienen zu können. Mögliche Auswirkungen auf den Speicherbedarf, die Leistungsaufnahme oder Optimierungen für die Anzahl an Kernen sollen für die späteren Ebenen bereitgestellt werden.

A4 Alle durchgeführten Optimierungen oder Realisierungsalternativen sind unabhängig von der gewählten Zielarchitektur.

3.1.2. Konzeption der Code-Ebene

Die Code-Ebene bezieht sich auf die tatsächliche Repräsentation des Algorithmus als Quellcode. Wie einzelne Berechnungen dargestellt werden gibt vor, wie viele (unabhängige) Tasks generiert werden können und bestimmt somit, wie gut die Parallelisierung arbeiten kann. Code-Transformationen verändern zwar den Code der Anwendung, aber nichts am Ergebnis der Berechnungen. Transformationen können genutzt werden, um die intrinsische Parallelität, die beispielsweise bei der Verarbeitung von größeren Datenmengen existiert, für die Parallelisierung verwendbar zu machen. Die wichtigsten Transformationen zur Optimierung der Anwendung für parallele Recheneinheiten:

Loop-Unrolling (Schleifen entrollen)
Ziel des Entrollens einer Schleife ist es, dass in einem Iterationsschritt der Schleife mehrere Schritte des nicht-optimierten Codes ausgeführt werden. Zum Beispiel bedeutet ein Faktor von 4, dass in einer Iteration der neuen Schleife 4 Iterationen der ursprünglichen Schleife ausgeführt werden. Dies sorgt üblicherweise für eine Verringerung des Schleifen-Overheads zu Lasten der tatsächlichen Code-Größe. Um für die Parallelisierung nützlich zu sein, muss der Code nach der Transformation besser analysierbar sein, indem die Abhängigkeiten zwischen den einzelnen Iterationen nicht mehr existieren.

Loop-Fission (Schleifenspaltung)
Gibt es im Body einer Schleife Ausdrücke, die unabhängig voneinander ausgeführt werden können, ist es möglich, anstelle einer Schleife zwei zu verwenden, die die gleichen Iterationsräume haben. Da beide Schleifen keine Abhängigkeiten untereinander haben, können sie anschließend parallel ausgeführt werden.

Loop-Tiling
Loop-Tiling kann auf Schleifen angewendet werden, die über verschachtelte For-Schleifen auf mehrdimensionale Arrays zugreifen. Ziel des *Loop-Tilings* ist es, den

Datenzugriff so zu verändern, dass die ursprüngliche Schleife aufgebrochen werden kann und jede neue Schleife nur auf einen Teil der Daten zugreift.

Permutation
Unter *Permutation* versteht man das Vertauschen von inneren und äußeren Schleifen, sowie Anpassung der Indizes, um das korrekte Verhalten der Anwendung sicherzustellen. Die Vertauschung sorgt dafür, dass der Zugriff auf die Daten, gerade wenn sie in einem Array gespeichert sind, verändert werden. Diese andere Darstellung der Datenabhängigkeiten kann dafür sorgen, dass beispielsweise *Loop-Fission* oder *Loop-Tiling* angewendet werden kann und somit mehr (unabhängige) Tasks für eine spätere Parallelisierung generiert werden können. Geprägt wurde der Begriff erstmals in [45].

Schleifenverzerrung (Loop-Skewing)
Ziel dieser Transformation ist es, bei verschachtelten For-Schleifen Abhängigkeiten zwischen einzelnen Iterationen der inneren Schleife in Abhängigkeiten der äußeren Schleife umzuwandeln. Auf diese Weise kann es möglich sein, dass weitere Transformationen auf der äußeren Ebene angewendet werden können, um die Anzahl der tatsächlichen Tasks zu erhöhen.

Loop-Splitting
Beim *Loop-Splitting* wird eine Schleife in mehrere Schleifen aufgeteilt, die zwar den gleichen Body haben, aber auf unterschiedlichen Bereichen der Indizes arbeiten. Ein Sonderfall dieser Transformation ist das so genannte *Loop-Peeling*, bei dem die erste Iteration vor die tatsächliche Schleife gezogen wird. Ist diese erste Iteration aufgrund von Initialisierungen oder anderen Sonderfällen komplexer als die restlichen Iterationen, so kann dieser Rest häufig besser optimiert werden, wenn der komplexe Fall nicht mehr Teil der Schleife ist.

Alle hier erwähnten Transformationen beziehen sich auf Schleifen und haben als Ziel, die Anzahl der unabhängigen Tasks zu erhöhen. Der Fokus liegt in dieser Arbeit auf den Schleifen-Transformationen, da sie das größte Potential zur Verbesserung der Parallelität haben.

Neben den Schleifen und deren Anzahl an Iterationen spielen auf dieser Ebene noch andere Eigenschaften eine Rolle. Zunächst wären das die Datenzugriffe bzw. die -lokalität, die großen Einfluss auf die sequentielle Ausführung unter Berücksichtigung der Cache- und Speicherhierarchien haben. Daneben ist zudem wichtig, wie groß die tatsächliche Anzahl an einzelnen Codeblöcken ist und wie viele Instruktionen sich darin befinden. Schließlich ist zur Durchführung der einzelnen Transformationen noch die statische Analysierbarkeit des Codes wichtig. Der gesamte Daten- sowie Kontrollfluss muss analysierbar sein, um den Code zuverlässig verändern zu können.

3.1.2.1. Anforderungen an die Optimierungen auf Code-Ebene

C1 Code-Transformationen dürfen keine funktionalen Änderungen am Code durchführen. Alle Ergebnisse müssen identisch mit dem ursprünglichen Code sein.

$$Ergebnis_{transformiert} = Ergebnis \tag{3.1}$$

C2 Transformationen müssen sicherstellen, dass der Code nach Anwendung immer noch korrekt funktioniert und keine neuen Fehler hinzugekommen sind.

C3 Werden Transformationen eingesetzt, um den Code besser an die Verarbeitung von späteren Ebenen anzupassen, so müssen etwaige Optionen parametrisiert werden, um eine spätere Anpassung zu ermöglichen.

$$trafo = f(algo, par_1, \ldots, par_n) \tag{3.2}$$

3.1.3. Konzeption der Task-Ebene

Die Task-Ebene bezieht sich in dieser Arbeit auf die Ebene, in der Tasks einzelne Ausführungseinheiten der Zielplattform zugewiesen werden. Die Anzahl der Tasks wird durch den Algorithmus und die Transformationen auf der Code-Ebene vorgegeben. Bei der Parallelisierung wird sowohl das Mapping als auch das Scheduling durchgeführt. Das bedeutet, dass ausgewählt wird, auf welchem Kern oder Prozessor ein Task ausgeführt wird und in welcher Reihenfolge die Tasks dort ausgeführt werden. Dabei müssen alle Abhängigkeiten zwischen den Tasks mit betrachtet werden, die hauptsächlich durch den Datenfluss, aber zum Teil auch durch den Kontrollfluss vorgegeben werden. Diese sorgen abhängig von der jeweiligen Zielarchitektur für einen Synchronisations- oder Kommunikationsoverhead, der bei der Parallelisierung beachtet werden muss. Neben den reinen Datenabhängigkeiten müssen zudem noch Gegenabhängigkeiten und Ausgabeabhängigkeiten betrachtet werden. Unter Gegenabhängigkeiten versteht man Abhängigkeiten, bei denen eine Variable nach dem Lesen von einer anderen Operation neu gesetzt wird. Die Veränderung der Reihenfolge dieser Operation sorgt für falsche Ergebnisse bei der Ausführung. Werden Variablen in komplexeren Anwendungen häufiger geschrieben und auch wieder gelesen, kann eine paarweise Vertauschung der Operationen möglich sein. Gleiches gilt für die so genannten Ausgabeabhängigkeiten, bei denen eine Variable von unterschiedlichen Anweisungen geschrieben wird.

Wichtig für die Parallelisierungsentscheidungen ist außerdem die Ausführungszeit der einzelnen Tasks im Verhältnis zueinander und im Vergleich zum Overhead, der bei der Synchronisation entsteht. Werden diese nicht korrekt betrachtet, kann es zu einer zu feinteiligen Parallelisierung kommen, bei der der Overhead der Kommunikation größer ist als der Zugewinn durch die parallele Ausführung und das Programm im Endeffekt langsamer ausgeführt wird als in der sequentiellen Realisierung.

Bei der Verteilung von Tasks auf die verfügbaren Kerne ist noch wichtig, dass Tasks selten isoliert betrachtet werden können. Häufig existieren Datenabhängigkeiten zwischen Tasks, so dass die Zuweisung eines Tasks Auswirkungen auf andere Tasks hat, die erst später ausgeführt werden können. Ein Beispiel ist in Abbildung 3.3 dargestellt. Task 1 und Task 2 sind unabhängig voneinander und könnten parallel ausgeführt werden. Task 3 und Task 4 hängen direkt von Task 1 bzw. Task 2 ab und sollten bei paralleler Ausführung von Task 1 und Task 2 ebenfalls auf den gewählten Kernen ausgeführt werden. Task 5 ist nun aber von Task 3 und Task 4 abhängig. Bei paralleler Ausführung von Task 1 und Task 2 führt dies nun zwingend zu einem Synchronisationsoverhead, egal, auf welchem Kern Task 5 ausgeführt wird. Aus diesem Grund ist es wichtig, Tasks nicht isoliert zu

betrachten und in diesem Beispiel Task 1 und Task 2 wirklich nur parallel auszuführen, wenn der Gewinn größer ist als der spätere Synchronisierungsoverhead.

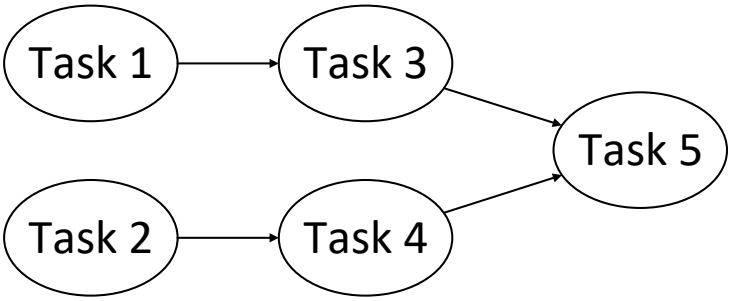


Abbildung 3.3.: Abhängigkeiten zwischen Tasks

3.1.3.1. Anforderungen an die Parallelisierung auf Task-Ebene

T1 Die Parallelisierung auf Task-Ebene zielt darauf ab, die Laufzeit des Algorithmus auf der Zielhardware durch parallele Ausführung zu minimieren. Scheduling-Algorithmen können auch für andere Aspekte eingesetzt werden, beispielsweise wie Reduktion der Leistungsaufnahme oder des Speicherbedarfs, aber im Rahmen dieser Arbeit wird nur die Laufzeitverkürzung betrachtet.

$$T_{parallel} = max(T_1, \ldots, T_n) < T_{sequentiell} \tag{3.3}$$

T2 Wie bereits auf der Code-Ebene in C1, dürfen bei Optimierungen auf Task-Ebene keine funktionalen Änderungen am Algorithmus durchgeführt werden.

$$Ergebnis_{parallelisiert} = Ergebnis \tag{3.4}$$

T3 Als Ergebnis nach der Parallelisierung auf Task-Ebene müssen alle Tasks, bzw. Codeteile des Algorithmus eindeutig den Ausführungseinheiten der gewählten Architektur zugewiesen werden. Dabei darf die Ausführungsreihenfolge unter Berücksichtigung der Daten- und Kontrollabhängigkeiten verändert werden.

$$\forall Tasks \exists Kernzuweisung \tag{3.5}$$

Eine Ausführungsreihenfolge R ist gültig, wenn für alle Anweisungen i und j in R gilt:

a) Wenn i eine Datenabhängigkeit zu j hat, dann wird i vor j in R ausgeführt.

b) Wenn i eine Kontrollabhängigkeit zu j hat, dann wird i vor j in R ausgeführt, falls es erforderlich ist, um das Programm korrekt auszuführen.

3.1.4. Konzeption der Daten-Ebene

Die Daten-Ebene bezieht sich in dieser Arbeit auf die Ebene, die für die Synchronisation und den Datenaustausch zwischen den Kernen verantwortlich ist. Ohne Beschränkung der Allgemeinheit wird im Rahmen der Arbeit dabei von Hardware-Modellen mit verteiltem Speicher ausgegangen. Jedes System mit geteiltem Speicher kann auch wie ein System mit verteiltem Speicher betrachtet und behandelt werden, indem der vorhandene Speicher fest den Kernen zugeordnet wird. Das bedeutet, dass jedes Mal, wenn ein Kern auf eine Variable von einem anderen Kern zugreifen will, eine explizite Kommunikation stattfinden muss. Dadurch geht Performanz verloren, da vor jedem Zugriff eine Kopie der Variablen im Speicher erzeugt werden muss und auch der Speicherbedarf der Anwendung steigt. Die spezielle Unterstützung von Systemen mit geteiltem Speicher ist aber sehr aufwendig und würde den Rahmen dieser Arbeit sprengen. Um das Konzept der ebenenübergreifenden Optimierung zu zeigen ist sie zudem nicht notwendig. Sollte der Trend zu immer mehr Recheneinheiten noch weiter anhalten, werden zukünftige Systeme mit verteilten und hierarchisch angeordneten Speichern arbeiten müssen. Beim Transfer von Daten spielt neben dem Speicherbedarf auf den beteiligten Kernen noch der Overhead je Transfer und die Art der Kommunikation eine Rolle. Für den Overhead ist hauptsächlich die Software-Realisierung der Kommunikation relevant. So benötigt eine komplexe Implementierung wie MPI [46] viel Zeit, um verschiedenste Flags der unterschiedlichen Modi zu setzen und zu überprüfen. Bei der Art der Kommunikation spielt eine große Rolle, wie lange der sendende und auch der empfangende Prozessor mit dem Transfer der Daten beschäftigt sind. Bei synchronen Implementierungen sind beide Kerne für die Dauer des Transfers blockiert, bei einer rein asynchronen Übertragung können die Kerne noch andere Aufgaben erledigen. Es existieren zudem Zwischenformen, bei denen Daten in Zwischenspeichern gehalten werden, um die beteiligten Kerne zu entkoppeln.

3.1.4.1. Anforderungen an die Parallelisierung auf Daten-Ebene

D1 Die Parallelisierung auf Daten-Ebene muss die korrekte parallele Ausführung der Anwendung sicherstellen. Dazu müssen alle Daten- und Kontrollabhängigkeiten, die durch die Parallelisierung auf Task-Ebene über Kerngrenzen hinweg gehen, korrekt beachtet werden, indem entsprechende Synchronisationsbefehle eingefügt werden. Wichtig dabei ist zudem die Vermeidung von Deadlocks und Race Conditions.

D2 Der notwendige Kommunikations- und Synchronisationsoverhead soll minimiert werden. Dies umfasst sowohl die effiziente, hardwarenahe Umsetzung der Kommunikations-API unter Berücksichtigung von etwaigen Betriebssystemen als auch eine allgemeine Optimierung zur Minimierung von Synchronisationspunkten.

$$Overhead = f_{min}(Overhead_{API}, Anzahl_{sync}) \tag{3.6}$$

D3 Für die Kommunikation wird ein asynchrones Modell mit einem Datenpuffer verwendet. Die Übertragung der Daten, die in den Puffer passen, erfolgt rein asynchron, d.h. der sendende Prozessor kann nach dem Senden der Daten mit der Verarbeitung fortfahren, unabhängig davon, ob der empfangende Prozessor die Daten

bereits gelesen hat. Wenn die Daten nicht vollständig in den Puffer passen, muss der sendende Prozessor warten, bis der empfangende Prozessor die Daten wieder aus dem Puffer gelesen hat.

3.1.5. Wichtigste Eigenschaften je Ebene

Zusammenfassend werden im Folgenden die für die jeweiligen Ebnenen wichtigsten Eigenschaften des Algorithmus, der Zielplattform und eingesetzter Bibliotheken sowie einer typischen Fragestellung aufgelistet.

- Algorithmus-Ebene:
  - Implementierung: Besteht der Algorithmus aus vielen (MATLAB®-)Funktionsaufrufen oder werden eher grundlegende Sprachelemente verwendet?
  - Rekursion: Werden Funktionen rekursiv aufgerufen?
  - Bekannter Algorithmus: Werden bekannte Algorithmen verwendet, die unter Umständen bereits Teil einer Bibliothek sind?
  - Nähe zur Hardware: Wurde der Algorithmus bereits mit einer bestimmten Hardware im Sinn entwickelt, um bereits effiziente Datentypen einzusetzen oder unabhängige Programmteile zu definieren?
  - Bekannte Eigenheiten: Existieren bekannte Eigenheiten des Algorithmus, die für eine bestimmte Optimierung ausgenutzt werden können?
- Code-Ebene:
  - Schleifen: Existieren Schleifen im Code und wie gut lassen sich die Anzahl an Iterationen und interne Abhängigkeiten bestimmen?
  - Datenzugriffe/-lokalität: Mit welchen Zugriffsmustern werden auf die Daten eines Arrays zugegriffen?
  - Anzahl Instruktionen: Wie viele Instruktionen befinden sich in einem Task der Anwendung?
  - Statische Analysierbarkeit: Kann der Daten- und Kontrollfluss innerhalb des Quellcodes eindeutig bestimmt werden?
  - Anzahl Code-Blöcke: Wie viele Code-Blöcke und damit später Tasks sind im Programm vorhanden und mit welchen Optionen kann die Anzahl verändert werden?
  - Anzahl Iterationen: Sind die Anzahl an Iterationen von Schleifen im Programm (statisch) bestimmbar?
- Task-Ebene:
  - Anzahl unabhängiger Tasks: Wie viele der Tasks sind unabhängig voneinander und können potentiell parallel ausgeführt werden, um die Laufzeit zu minimieren?

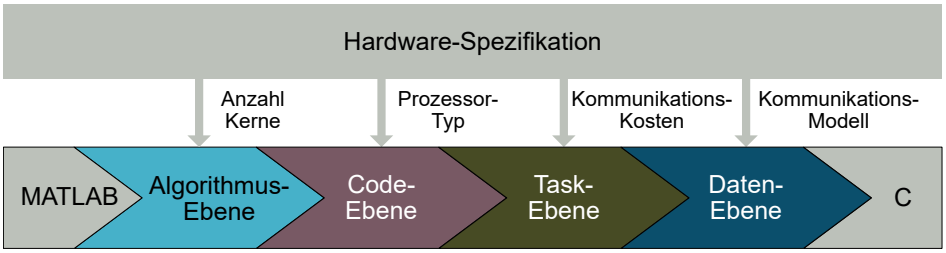


Abbildung 3.4.: Übersicht der Ebenen zusammen mit den Hardware-Informationen

  - Cluster/Ketten von Tasks: Existieren Gruppen an Tasks, die immer auf dem gleichen Kern ausgeführt werden sollten, um den Kommunikationsoverhead zu minimieren?
  - Ausführungsreihenfolge: In welcher Reihenfolge müssen die einzelnen Tasks ausgeführt werden, um alle Abhängigkeiten zu erfüllen und die Laufzeit zu minimieren?
    * Datenabhängigkeiten: Welche Tasks lesen die Daten, die vom aktuellen Task geschrieben werden?
    * Kontrollabhängigkeiten: Welche Bedingung muss erfüllt sein, damit ein bestimmter Task überhaupt ausgeführt wird?
    * Ausgabeabhängigkeiten: Haben Funktionsaufrufe oder Schreibvorgänge auf Variablen noch Seiteneffekte wie die Ausgabe auf einer Konsole, so dass sie zwingend in der gleichen Reihenfolge ausgeführt werden müssen, um gleiche Ergebnisse zu generieren?
    * Gegenabhängigkeiten: Ist eine Anweisung von einem vorigen Wert einer Variablen abhängig?
- Daten-Ebene:
  - Speicherbedarf: Wie viel Speicher müssen für Variablen und ihre Kopien auf den einzelnen Kernen vorhanden sein?
  - Wartezeiten: Wie lange muss ein empfangender Kern auf Daten warten?
  - Overhead je Transfer: Wie hoch sind die Kosten, um eine gewisse Datenmenge an einen anderen Kern zu senden?
  - Art der Kommunikation: Findet die Kommunikation synchron oder asynchron statt?

In Abbildung 3.4 werden die Ebenen in ihrer Reihenfolge vom MATLAB®-Code hin zum parallelen C-Code und ihre Nutzung der wichtigsten Hardware-Eigenschaften dargestellt.

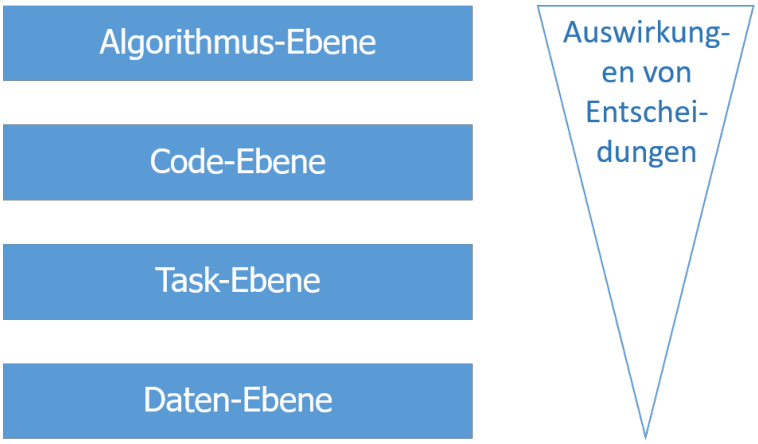


Abbildung 3.5.: Auswirkungen der einzelnen Ebenen

3.2. Automatierungsmöglichkeiten der einzelnen Ebenen

Entscheidungen auf den einzelnen Ebenen haben unterschiedliche Auswirkungen auf die Performanz der finalen Anwendung. Wie in Abbildung 3.5 dargestellt, haben Entscheidungen auf der Algorithmus-Ebene den größten Einfluss. Eine Implementierung, die bereits für die Ausführung auf mehreren Kernen entwickelt wurde, hat große Auswirkungen auf die tatsächlich erreichbare Performanz. Entscheidungen auf der Code-Ebene können ebenfalls einen großen Einfluss haben. Können beispielsweise Schleifen in einzelne, kleinere Schleifen zerlegt werden, erhöht das das Potential für eine effizientere Ausführung drastisch. Die Task-Ebene entscheidet anschließend, auf welcher Ausführungseinheit und in welcher Reihenfolge die einzelnen Tasks ausgeführt werden. Die Auswirkungen werden als niedriger eingestuft, da die Entscheidungsmöglichkeiten sehr durch die Anzahl der Tasks beeinflusst werden, welche durch die beiden oberen Ebenen vorgegeben wird. Der Einfluss der Daten-Ebene ist schließlich nur noch gering hinsichtlich einer Verbesserung der Performanz. Ineffizienz führt allerdings dazu, dass die Anwendung deutlich langsamer ausgeführt wird, wenn beispielsweise ein Datentransfer in jeder Iteration einer Schleife ausgeführt werden muss. Da die Parallelisierung auf Task-Ebene erfolgt, die Optimierung der Synchronisierung und der Daten aber auf dem AST, kann sich die Performanz im Vergleich zum Modell des Schedulers noch leicht verschlechtern. Im Idealfall sollte aber der abgeschätzte Zeitpunkt und die abgeschätzte Dauer erreicht werden.

In Abbildung 3.6 ist farblich dargestellt, wie gut sich die einzelnen Ebenen automatisieren lassen:

Algorithmus-Ebene
: Im Allgemeinen lassen sich Optimierungen auf der Algorithmus-Ebene nur schwer automatisieren. Eine alternative Implementierung für einen beliebigen Code automatisch zu generieren ist derzeit nicht möglich, da dafür in der Regel viel Verständnis und Erfahrung benötigt wird, um sie zu realisieren. Nach derzeitigem Stand

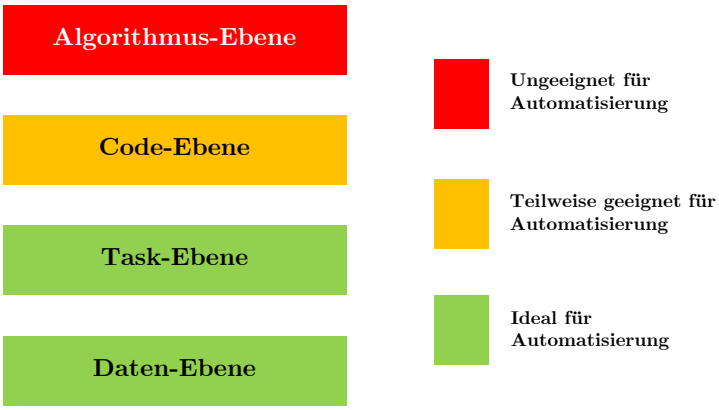


Abbildung 3.6.: Automatisierungsmöglichkeiten

sind auch moderne Machine-Learning-Ansätze noch nicht dazu in der Lage. Ähnliches gilt für die Ausnutzung von Eigenschaften von bestimmten Algorithmen. Ein pragmatischer Ansatz mit Hilfe einer Datenbank erlaubt es jedoch, zumindest für bekannte Algorithmen verschiedene Realisierungs-Alternativen zur Auswahl vorzugeben, die dann hinsichtlich der Parallelisierung für die jeweilige Zielplattform optimiert wurden.

Code-Ebene

Das Anwenden einzelner Transformationen auf den Quellcode kann sehr gut automatisiert werden. Die Analyse und das anschließende Umformen kann sehr gut auf den entsprechenden Zwischendarstellungen angewendet werden. Problematisch ist aber, dass auf der Code-Ebene Code-Transformationen wie *Loop-Fission* ausgeführt werden können, die aus einer Schleife mehrere unabhängige generieren können. Eine automatische Auswahl erfordert nun komplexe Optimierungsverfahren, da ansonsten der Entwurfsraum sehr schnell zu nicht mehr beherrschbaren Größen anwächst. Bei einer zweifach verschachtelten For-Schleife sorgen beispielsweise die sechs hier vorgestellten Transformationen gemäß

$$x = \sum_{k=1}^{n} \left(\frac{n!}{(n-k)!} \right)$$

für 1236 verschiedene Möglichkeiten. Zwei Schleifen dieser Art sorgen bereits für 1527696 Möglichkeiten. Simples Ausprobieren aller Möglichkeiten ist keine praktikable Option, so dass eine intelligente Auswahl von erfolgsversprechenden Transformationen und ihrer Reihenfolge benötigt wird. Dieses Thema wird in dieser Arbeit aufgrund des Umfangs jedoch nicht weiter betrachtet und Code-Transformationen werden manuell ausgewählt und automatisch auf passenden Code-Abschnitten ausgeführt.

Task-Ebene
Mapping und Scheduling ist ein bekanntes Problem, für das zahlreiche verschiedene Algorithmen existieren. Als Optimierungsproblem können verschiedene Ansätze wie *Integer Linear Programming* oder Heuristiken eingesetzt werden, die das Problem möglichst effizient optimieren. Der Trade-Off besteht dabei zwischen der Dauer für die Parallelisierung sowie der Güte der jeweiligen Lösung. Für die Güte existieren dabei verschiedene Kriterien, beispielsweise die gesamte Ausführungszeit des Programms (Makespan), die absolute Leistungsaufnahme oder der Speicherverbrauch. Die Parallelisierung auf der Task-Ebene lässt sich aufgrund der existierenden Verfahren und den verschiedenen Optimierungszielen sehr gut automatisieren, wobei der Algorithmus gut an die genaue Fragestellung angepasst werden kann.

Daten-Ebene
Optimierungen auf der Daten-Ebene können sehr gut automatisiert werden. Die tatsächliche Platzierung der Anweisungen für die Synchronisierung und den Datenaustausch erfolgt an der bestmöglichen Stelle im Programm. Auch hierbei können verschiedene Kriterien als Referenz verwendet werden. Gebräuchlich sind dabei die Performanz im Sinne der Laufzeit, bzw. der Reduktion von Wartezeiten, sowie der Speicherbedarf von Variablen.

3.3. Ebenenübergreifende Konzepte

Die Ebenen können zwar unabhängig voneinander, hintereinander ausgeführt werden, die besonderen Vorteile des Ansatzes kommen jedoch erst durch eine Kombination der Ebenen zustande:

Algorithmus-Ebene
Die Auswahl der tatsächlichen Realisierung eines Algorithmus hat großen Einfluss auf alle weiteren Ebenen. Sie gibt vor, an welchen Stellen Code-Transformationen überhaupt ausgeführt werden können, was die Basis-Anzahl an Elementen für die Task-Ebene ist und wie viel Daten überhaupt auf die einzelnen Variablen abfallen. Das bedeutet aber auch, dass Vorbereitungen, die das Potential für die Parallelisierung bzw. Optimierung auf einer späteren Ebene erhöhen, aber nicht genutzt werden (können), rückwirkend wieder verhindert werden sollten, um keinen unnötigen Overhead zu generieren. Mögliche Gründe dafür sind:

- Die Generierung einer hohen Zahl an unabhängigen Tasks tief in den Kontrollstrukturen des Algorithmus kann für die Parallelisierung nicht genutzt werden, da alle Kerne bereits mit gröberen Strukturen ausgelastet sind. Hier reduzieren weniger Tasks den Overhead und sind zu bevorzugen.
- Der Algorithmus arbeitet auf großen Datenstrukturen, die nicht sinnvoll aufgeteilt werden können. Dies äußert sich in einem sehr hohen Kommunikationsaufwand, da häufig große Datenmengen kommuniziert werden müssen. Sinnvoller kann hier eine Realisierung sein, die bereits auf Teilstücken arbeitet.

- Der Algorithmus wurde mit rekursiven Funktionen anstelle von Schleifen realisiert. Dies schränkt die Möglichkeiten der hier eingesetzten Code-Transformationen ein und verhindert eine sinnvolle Parallelisierung.

Sinnvolle Optimierungen auf Algorithmus-Ebene sind:

- Erhöhung der Anzahl an Tasks auf hohen Abstraktionsebenen des Algorithmus.
- Arbeiten auf kleineren Datenmengen.
- Nutzung von vielen Schleifen, die höheres Optimierungspotential bieten.

Code-Ebene
Die Transformationen auf der Code-Ebene sind sehr auf den Input aus der Algorithmus-Ebene angewiesen. Werden hier nur wenige Schleifen eingesetzt, kann auf dieser Ebene nicht viel optimiert werden. Die Granularität der Transformationen sorgt dafür, dass an dieser Stelle sehr gut unabhängige Tasks für die Parallelisierung auf Task-Ebene generiert werden können. Die Abstraktionsebene sollte aber auch hier in Zusammenarbeit ermittelt werden, da ansonsten viele unbenutzbare Tasks generiert werden können. Sämtliche Transformationen, die den Zugriff auf die Daten verändern können, haben zudem das Potential, den Overhead auf der Daten-Ebene zu reduzieren.

Task-Ebene
Nach den vorbereitenden Optimierungen auf den vorherigen Ebenen, findet der Hauptteil der Parallelisierung auf dieser Ebene statt. Als reiner Optimierungsalgorithmus auf einer Graphen-Darstellung werden hier alle Abhängigkeiten beachtet, die für die Aufteilung auf die einzelnen Kerne entscheidend sind. Dies sorgt zwar für die korrekte Funktionalität, mitunter aber zu einer falschen Abschätzung der tatsächlichen Performanz. Da die Platzierung und Optimierung der Kommunikation erst später und anhand von einer anderen Darstellung durchgeführt werden, kann die Synchronisation und die Vervielfältigung der Daten noch nicht vollständig betrachtet und nur als Hinweise an die nächste Ebene übergeben werden.

Daten-Ebene
Auf der Daten-Ebene werden alle Entscheidungen, die vorher getroffen wurden im C-Code umgesetzt. Dies garantiert dabei die korrekte Ausführung des Algorithmus, indem alle Datenabhängigkeiten korrekt im Kontrollfluss aufgelöst werden. Dies verhindert das Auftreten von fehlerhafter Parallelisierung durch die Vermeidung von Race Conditions und Deadlocks. Zur statischen Evaluation des parallelen Codes kann eine parallele Performanzabschätzung (siehe Abschnitt 6.1.2.1) eingesetzt werden.

Zum Informationsaustausch zwischen den verschiedenen Ebenen soll eine einfache Textdatei zum Einsatz kommen. Relevante Informationen können auf jeder Ebene hinzugefügt und aktualisiert werden. Optimierungen, die eigentlich vorher angewendet werden, können die Informationen bei einer erneuten Ausführung mit einbeziehen. Spätere Optimierungen können bereits auf Analyseergebnisse und die Änderungen an der Darstellung zurückgreifen. Wichtig ist, dass die Datei sowohl maschinenlesbar als auch manuell

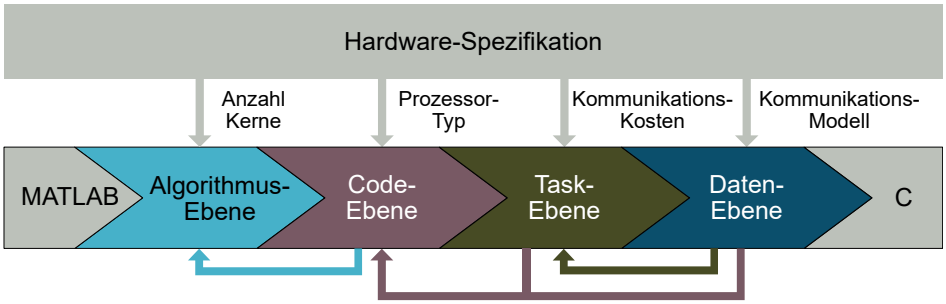


Abbildung 3.7.: Interaktionen zwischen den Ebenen

editierbar ist, um eine Mischung aus automatisierter und manueller Exploration zu erlauben.

Abbildung 3.7 veranschaulicht typische Interaktionsmöglichkeiten zwischen den Ebenen: Die Code-Ebene kann von der Algorithmus-Ebene verschiedene Realisierungen des Algorithmus anfordern, die für eine unterschiedliche Anzahl von Kernen vorbereitet sind. Die Task-Ebene und die Daten-Ebene können von der Code-Ebene Code-Transformationen anfordern, um die Anzahl der möglichen Tasks und die Anzahl der notwendigen Kommunikation anzupassen. Schließlich kann das Feedback der Datenebene dazu verwendet werden, die Parallelisierung auf der Task-Ebene zu optimieren.

3.4. Konzeption des Frameworks

Wie in Abschnitt 3.2 gezeigt, eignen sich die vier Ebenen unterschiedlich gut für eine automatisierte Anwendung. Hinzu kommt, dass die Algorithmus- und Code-Ebene, die den größten Einfluss auf die spätere Performanz haben, sich am schwierigsten automatisieren lassen. Ziel dieser Arbeit ist die Untersuchung, in wie weit eine Optimierung über die verschiedenen Ebenen hinweg bis hin zu einer effizienten Ausführung auf der Hardware realisiert werden kann. Dazu ist es notwendig, ein Framework zur Hand zu haben, mit dem Optimierungen auf den Ebenen möglichst (zeit-) effizient ausgeführt und evaluiert werden können. Dazu werden die Ebenen im Folgenden noch mal aus dieser Sichtweise analysiert:

Algorithmus-Ebene
: Da eine automatisierte Nutzung für allgemeine Algorithmen nicht möglich ist, muss hier auf einen bibliotheksbasierten Ansatz zurückgegriffen werden. Für bekannte Funktionen oder Algorithmen können verschiedene Realisierungen hinterlegt werden, die für verschiedene Anwendungsfälle optimiert werden. Eine automatische Erkennung von unterstützten Funktionen anhand des Quellcodes liegt außerhalb der Reichweite dieser Arbeit und wird nicht weiter betrachtet. Stattdessen erfolgt die Erkennung rein namensbasiert, um bei Bedarf verschiedene Realisierungen über ein Skript oder eine grafische Oberfläche auswählen zu können. Auf die gleiche Art

und Weise werden auch bekannte Eigenschaften von Algorithmen unterstützt, indem mögliche Parameter nach außen kommuniziert werden.

Code-Ebene
Um den Aufwand in vertretbarem Rahmen zu halten, soll auf eine voll-automatisierte Lösung, die verschiedene Transformations-Kombinationen anwendet und evaluiert verzichtet werden. Stattdessen soll das Framework ermöglichen, dass manuell einzelne Transformationen ausgewählt werden und anschließend automatisch angewendet werden. Um Transformationen auszuwählen, sollen diese entweder direkt im Quellcode mit Hilfe von Pragmas an den Code-Blöcken oder in einer grafischen Aufbereitung markiert werden können. Eine automatisierte Analyse kann dabei verwendet werden, um die Anwendbarkeit der Transformation vorher zu überprüfen.

Task- und Daten-Ebene
Da sich diese beiden Ebenen sehr gut automatisieren lassen, werden hier nur Möglichkeiten gebraucht, Parameter bei den Optimierungseinstellungen zu verändern. Diese sollten abhängig von der gewählten Zielplattform gesetzt werden können, um Unterschiede in der Kernanzahl oder der Kommunikationsinfrastruktur mit einzubeziehen.

Zusammenfassend muss das Framework die folgende Funktionalität bzw. Komponenten beinhalten:

- Eine Bibliothek mit unterschiedlichen Implementierungen bekannter Algorithmen. Diese sollten sich bereits bezüglich ihrer generellen Eignung für eine parallele Ausführung unterscheiden. Die Auswahl der Implementierung kann manuell erfolgen, um das Wissen über die spätere Zielplattform bereits an dieser Stelle einfließen zu lassen.
- Die Möglichkeit, vorausgewählte Code-Transformationen an bestimmten Stellen im Quellcode und in bestimmter Reihenfolge anzuwenden.
- Einen automatischen Scheduler, der einzelne Tasks möglichst effizient auf die verfügbaren Kerne der Zielplattform verteilt und dabei den Overhead für die Kommunikation mit einbezieht.
- Generierung der Kommunikation und Synchronisation in einem extra Schritt, um die Optimierung auf der Daten-Ebene zu realisieren.
- Automatische Generierung von C-Code, der an den Compiler der Zielarchitektur übergeben werden kann. Dies ist notwendig, um den Algorithmus anschließend auf der Zielplattform evaluieren zu können.

Als Basis kommt emmtrix Parallel Studio [47] (ePS) zum Einsatz. Es ist über Plugins für die Programmierumgebung Eclipse [28] realisiert und kann einfach mit weiteren Funktionen erweitert werden. In Kombination mit emmtrix Code Generator [48] (eCG) bildet die Toolchain damit die gesamte hier relevante Verarbeitungskette ab: Algorithmen können in abstrakter Darstellung als MATLAB®-Code oder Simulink-Modell eingelesen werden und werden nach der Verarbeitung als angepasster, paralleler C-Code ausgegeben. Die klare Trennung der Ebenen kann in der Praxis nicht so genau umgesetzt werden, so dass die vier Ebenen wie folgt abgebildet werden:

Algorithmus-Ebene

Entscheidungen und Optimierungen auf der Algorithmus-Ebene werden bereits im eCG durchgeführt. Das bedeutet, dass bereits der generierte, generische C-Code diese Optimierungen beinhaltet. Auswahlmöglichkeiten können dabei als Pragma-Direktiven im Quellcode eingefügt werden, damit die Entscheidungen auch erst später mit Hilfe der grafischen Darstellung durchgeführt werden können. Die Bibliothek der Realisierungs-Alternativen kann dabei sowohl als MATLAB®-Datei vorliegen, die dann wiederum als Basis für den generierten C-Code dient oder direkt als optimierter C-Code, der an den entsprechenden Stellen aufgerufen wird. Ein weiterer wichtiger Aspekt dieser Ebene, die gerade im Zusammenspiel der Tools zum Tragen kommt, ist ein optimierter Einsatz von Schleifen und die Darstellung der Datentypen, um den Algorithmus noch genau ausführen zu können. Die Details zur Konzeptionierung und der Realisierung dieser Thematik sind in Abschnitt 4.3 dargestellt.

Code-Ebene

Transformationen auf Code-Ebene können im Framework an zwei verschiedenen Stellen angewendet werden: zum einen ist es möglich, Transformationen direkt im eCG durchzuführen und damit bereits optimierteren C-Code zu generieren. Dazu können einzelne Elemente direkt im MATLAB®-Code mit Pragmas versehen werden, so dass eine Transformation direkt vor der Erzeugung des C-Codes angewendet werden kann. Bei Änderungen an den Einstellungen muss jedoch jedes Mal direkt der eCG erneut aufgerufen werden, was die Dauer jedes Iterationsschritts verlängern kann. Zum anderen können Transformationen erst später auf dem wieder eingelesenen C-Code angewendet werden. Dies kann auch auf beliebigen Eingangs-C-Code angewendet werden und ist somit flexibler in der Handhabung. Zudem erlaubt es eine einfachere Evaluation, wenn mehrere Transformationen nacheinander ausgeführt werden. Mehr Details zu diesem Thema sind in Abschnitt 5.3 dargestellt.

Task-Ebene

Optimierungen auf der Task-Ebene können direkt auf den „Mapping and Scheduling“-Schritt von ePS abgebildet werden. Als Eingabe wird direkt die transformierte Darstellung des Algorithmus verwendet, als Ausgabe eine eindeutige Zuordnung der Tasks zu den verfügbaren Kernen der Zielplattform. Mehr Details zu dieser Thematik sind in Abschnitt 5.4 dargestellt.

Daten-Ebene

Optimierungen auf der Daten-Ebene werden als Teilschritt der Generierung des parallelen C-Codes durchgeführt. Dazu werden die Kernzuweisungen aus der vorherigen Ebene angewendet und wenn notwendig die passenden Synchronisations- und Kommunikationsinstruktionen an optimalen Stellen im Quellcode platziert. Mehr Details zu dieser Thematik sind in Abschnitt 5.5 dargestellt.

3.5. Definition der Zielhardware

Die Zielhardware, auf der der Algorithmus später ausgeführt werden soll und für die die optimierte Parallelisierung durchgeführt wird, wird im Rahmen dieser Arbeit durch folgende Unterscheidungsmerkmale definiert:

Befehlssatz der Prozessoren: Die Zielhardware darf aus heterogenen Prozessorverbünden bestehen, bei denen die einzelnen Kerne unterschiedliche Befehlssätze verwenden können. Dies erfordert die Unterstützung verschiedener Compiler sowie die Generierung des passenden C-Codes. Damit hat dies den größten Einfluss auf die Parallelisierung auf Code-Ebene, aber auch eine sinnvolle Wahl des Algorithmus auf der ersten Ebene kann einen großen Einfluss haben.

Anzahl an Kernen: Die Anzahl an Kernen ist eine wichtige Eigenschaft, die relevant für alle Ebenen ist: auf der Algorithmus-Ebene sollte eine Realisierung gewählt werden, die auf die Anzahl an Kernen optimiert wurde. Code-Transformationen sollten die Anzahl an Tasks für die Task-Ebene abhängig von der Kernanzahl optimieren und schließlich benötigt die Parallelisierung auf Daten-Ebene die Information, um die Kommunikation und die Variablen-Duplikation entsprechend durchzuführen.

Taktfrequenz: Die Taktrate der Kerne oder Prozessoren ist hauptsächlich auf der Task-Ebene relevant, um die abgeschätzten Zeiten grafisch darzustellen und anhand von ihnen die einzelnen Aufgaben auf die Kerne zu verteilen.

Datenaustausch zwischen den Kernen: Für die Optimierung auf der Task- und der Daten-Ebene ist die Art der Verbindung zwischen den Kernen und ihr Durchsatz bzw. ihre Latenz wichtig. Dabei spielt zudem die Software-Realisierung eine Rolle, um den Overhead berechnen zu können. Relevante Unterscheidungsmerkmale sind der Einsatz von gemeinsam genutzten Speicherbereichen, eingesetzte Bussysteme oder Netzwerke und gegebenenfalls der Einfluss der verwendeten Betriebssysteme.

Speicherhierarchien: Der Aufbau des Speichersubsystems mit der Menge des Speichers, dem Einsatz von Caches sowie etwaiger lokaler Speicher ist ebenfalls auf allen Ebenen relevant. Auf der Algorithmus-Ebene kann eine bereits eine speicher-effiziente Realisierung ausgewählt werden. Transformationen auf der Code-Ebene können die Verwendung des verfügbaren Speichers optimieren. Die Task-Ebene muss beachten, wie viel Speicher die Menge an Tasks pro Kern benötigt. Abschließend kann die Verteilung der Daten je Kern und zwischen den Kernen noch auf der Daten-Ebene optimiert werden.

3.6. Stand der Technik

Parallele Software-Entwicklung ist kein neues Thema und nahm ihre Anfänge bereits in den 1960er Jahren mit den ersten parallelen Supercomputern. Dennoch gibt es Stand heute immer noch viele verschiedene Themen, bei denen die effizientere Programmierung von heterogenen Mehrkernsystemen im Vordergrund steht.

Ein Thema, das bereits seit Beginn beschäftigt, ist die vollständig automatisierte Parallelisierung in einem Compiler oder einem Werkzeug. Bereits 1995 wurde in [49] von Ian

Foster ein Verfahren vorgeschlagen, das als Standard in der Literatur gilt. Es setzt auf vier Schritte: Partitionierung, Kommunikation, Ansammlung (engl. *agglomeration*) und Mapping der Problemgrößen auf die verfügbaren Kerne. Alle vier Schritte können auf den drei hinteren Ebenen dieser Arbeit wiedergefunden werden. Durch Hinzunahme der Algorithmus-Ebene für spezifischere Implementierungen sowie Codetransformationen kann hier eine bessere Ausnutzung der Zielhardware erreicht werden. In [50] und [51] wird ein Ansatz des Forschungstools AutoPar-Clava vorgestellt. Die Grundidee ist, dass C-Quellcode automatisch Schleifen analysiert und auf mögliche parallele Ausführung untersucht. Die Spezifikation dieser Schleifen erfolgt über eine zusätzliche Textdatei, so dass über eine Mustererkennung nach passenden Schleifen gesucht werden kann. Die Parallelität wird anschließend über OpenMP-Direktiven im Quellcode annotiert. Ein ähnlicher Ansatz, der ebenfalls in C Code mit OpenMP mündet wird in [52] vorgestellt. Noch stärker auf die Analyse von Schleifen konzentriert sind Projekte wie Pluto [53] und Par4all [54]. Diese setzen auf das Polytopmodell (siehe Abschnitt 4.2) und ermöglichen Code-Transformationen, um Schleifen effizienter parallel auszuführen. Ein neueres Framework namens Daedalus [55] setzt ebenfalls auf Codetransformationen auf Basis des Polytopmodells, um Algorithmen für Mehrkernprozessoren zu parallelisieren. Ziel des Frameworks ist die gleichzeitige Entwicklung von Hard- und Software, um ein möglichst effizientes System zu entwickeln. Ziel des hier vorgestellten Ansatz ist die Ausnutzung existierender Hardware, wobei auch hier wieder durch Optimierungen auf Algorithmus-Ebene weiteres Potential greifbar gemacht werden kann.

Automatisierte Parallelisierung, die sich vorwiegend auf die Parallelisierung von Schleifen konzentriert, kann sehr gute Ergebnisse für Anwendungen liefern, die durch Schleifen dominiert werden. Andere Codeteile wie Funktionsaufrufe oder unabhängige Instruktionen, die parallel ausgeführt werden könnten, werden nicht betrachtet.

Ein komplett anderer Ansatz ist die spekulative Parallelisierung, auch bekannt als *Thread Level Speculation* (TLS) , wie sie beispielsweise in [56] vorgestellt wird. Das Grundkonzept setzt darauf, dass Anwendungen rein sequentiell programmiert werden und der eingesetzte Compiler T4 (***t**rees for **t**iny **t**imestamped **t**asks*) die Anwendung in Tasks aufteilt, die wahrscheinlich parallel ausgeführt werden können. Abhängigkeiten zwischen den Tasks werden erst zur Laufzeit erkannt, so dass es vorkommen kann, dass die Ausführung von Tasks abgebrochen und Teile erneut berechnet werden müssen. Für eine effiziente Umsetzung wird eine Hardware-Unterstützung benötigt, um diesen Overhead möglichst klein zu halten. Eine mögliche Hardware-Plattform ist Swarm[57]. Das Konzept skaliert zwar auf mehrere Dutzend Kerne und erreicht bei passenden Anwendungen auch eine sehr gute Auslastung aller Kerne mit passenden Performanzgewinnen, ist aber aufgrund der Anforderungen an die Hardware und damit einhergehenden komplexeren Prozessoren nicht für den Einsatz in eingebetteten Systemen mit bereits existierender Hardware geeignet.

Ein weiterer großer Bereich ist die Parallelisierung über die Erkennung von bekannten, parallelisierbaren Mustern. In [58] wird in der Big Data Domäne auf die Erkennung von Cluster-Algorithmen gesetzt. Diese werden für verschiedene Anwendungen eingesetzt und zeichnen sich durch Charakteristiken aus, die sich auch über verschiedene Implementierungen wiedererkennen lassen. Der Ansatz definiert Kernels, die in der domänenspezifischen Sprache Dwarf umgesetzt sind, um bekannte Elemente aufzufinden. Für die-

se können anschließend optimierte Implementierungen eingesetzt werden, die auf Schleifenebene optimiert wurden. Der parallele Code wird schließlich mittels MPI umgesetzt und bildet eine Master-Worker Konfiguration. Laut den Autoren ist der Ansatz auch auf andere Domänen anwendbar, da er aber mit Dwarf auf eine *domänenspezifische Sprache* (domain specific language, engl. *DSL*) setzt, ist der Portierungsaufwand recht groß. Dennoch bildet der Ansatz eine Alternative für eine Optimierung auf Algorithmus-Ebene.

In [59] wird dargestellt, wie strukturierte, parallele Programmier-Methodologien Einzug in die vorherrschenden Programmiermodelle gefunden haben. Der Ansatz ist, dass Parallelität in Programmen nur über strukturierte parallele Kompositionen erfolgen darf. Zu den bekanntesten gehören *map*, *pipeline*, *farm* und *MapReduce*. Sie stellen dar, wie Daten vor der Berechnung verteilt, parallel ausgeführt und wie die Ergebnisse wieder gesammelt werden. Sie wurden aus dem Ansatz von Algorithmus-Skeletten entwickelt. Darunter wird eine Zweiteilung verstanden, bei der der Anwendungsentwickler das parallele Verhalten rein über die verfügbaren Skelette abbilden darf und der Systemprogrammierer optimierte Implementierungen für das eingesetzte System bereitstellen muss. Daraus haben sich parallele Entwurfsmuster entwickelt, die beispielsweise in OpenMP für Systeme mit geteiltem Speicher, CUDA und OpenCL für GP-GPUs und MPI für verteilte Cluster umgesetzt wurden und damit den de facto Standard für die parallele Entwicklung bilden. Dies erklärt, weshalb diese Programmiermodelle in fast allen Veröffentlichungen eingesetzt werden und auch keine parallele Programmiersprache derzeit beliebt genug ist, etwas daran zu ändern.

In [60] werden die parallelen Programmiermodelle OpenMP, OpenCL, CUDA und OpenACC hinsichtlich Produktivität, Performanz und Leistungsaufnahme verglichen. Die Produktivität wird dabei über ein eigens entwickeltes Tool ermittelt, indem der Quotient aus Codezeilen des parallelen Codes und der gesamten Codezeilen gebildet wird. Über eine Konfigurationsdatei wird definiert, welche Sprachkonstrukte im Quellcode spezifisch für das zu untersuchende Programmiermodell sind, um anschließend ihren Anteil am Quellcode zu bestimmen. Der Vergleich der erreichten Performanz wird über die Ausführungszeit der Programme ermittelt. Als Hardware wurden Intel Xeon Prozessoren zusammen mit Intel Xeon Phis sowie Nvidia GeForce GTX Titan X als Grafikbeschleuniger eingesetzt. Auch wenn diese nicht in eingebetteten Systemen zum Einsatz kommen, so lassen sich die Ergebnisse über die Programmiermodelle dennoch auf andere CPUs und GPUs hinsichtlich Produktivität und Performanz übertragen. Die wichtigsten Ergebnisse sind wie folgt:

- OpenACC benötigt zwar um Faktor 6,7 weniger Codezeilen als OpenCL, erreicht dessen Performanz aber nur selten und ist in der Regel deutlich langsamer. Da in dieser Arbeit der Quellcode zu größten Teilen automatisiert generiert wird und die Performanz damit wichtiger ist, wird OpenACC nicht weiter betrachtet.
- OpenCL benötigt für eine ähnliche Performanz wie CUDA rund doppelt so viele Codezeilen.
- Die Performanz von OpenCL und CUDA hängt von der tatsächlichen Anwendung ab, mal ist das eine schneller, mal das andere.
- OpenMP benötigt nur rund ein Drittel des Aufwands von OpenCL und CUDA und kann somit effizienter vom Programmierer eingesetzt werden. Dafür ist die erziel-

te Performanz mit GPU-Beschleunigung bei den ausgewählten Testfällen deutlich größer als bei reiner Ausführung auf der CPU.

- Der Overhead durch Einsatz der Programmiermodelle hängt sehr stark vom Entwickler ab. Als Richtwerte ergaben sich 5 % für OpenMP, 10 % für OpenCL und 7 % für CUDA.

Die Arbeit fasst sehr gut die Parallelisierung unter Einsatz von Programmiermodellen zusammen. Im Kontext dieser Arbeit konzentriert sich diese Art der Parallelisierung auf die Code- und die Daten-Ebene: es werden hauptsächlich Schleifen betrachtet und die Kommunikation auf Daten-Ebene ist durch die Programmiermodelle möglichst effizient umgesetzt. Abstraktere Betrachtungen auf der Algorithmus- oder Task-Ebene spielen hier keine große Rolle.

Was in den vorherigen Veröffentlichungen beispielhaft dargestellt wurde, ist in [61] noch mal allgemein über die Programmierung von Multi- und Manycore-Systemen zusammengefasst. Neben den Grundlagen werden mehr Details zu den verschiedenen Programmieransätzen vorgestellt, sowie Frameworks gezeigt, die diese Konzepte umsetzen.

Das Forschungsprojekt P-SOCRATES, dessen Ergebnisse in [62] dargestellt sind, zeigt auf, wie Konzepte aus dem HPC-Bereich im eingebetteten Umfeld eingesetzt werden können. Traditionell wurden die beiden Bereiche sehr getrennt voneinander betrachtet: beim HPC ist das Ziel, die Ausführungszeit der typischen Aufgaben zu minimieren und die maximale Performanz aus der verfügbaren Hardware zu holen. Auf seltene Fälle wird nicht hin optimiert, so dass sie zeitlich nicht abgeschätzt werden können. Eingebettete Systeme haben aufgrund ihrer Einsatzorte wie die Automotive-, Avionik- oder Automatisierungs-Branchen harte Echtzeitanforderungen, da ihre Antwortzeiten immer innerhalb der Zeitbudgets liegen müssen. Die Einführung von Multicore-Prozessoren sowie deren Weiterentwicklung hin zu Manycore- oder heterogenen Systemen mit Beschleunigern bringen nun aber die Herausforderungen der Parallelisierung ebenfalls in den eingebetteten Bereich. Das im Projekt entwickelte Framework setzt dabei auf bewährte Elemente wie OpenMP zur Daten- und Kontrollflussanalyse sowie der Codegenerierung, Task-Scheduling sowie die Anpassungen an ein Echtzeitbetriebssystem. Der Ansatz setzt dabei auf keine vergleichbaren Optimierungen auf Algorithmus-Ebene und hat aufgrund der OpenMP-Basis auch nur wenig Einfluss auf die Daten-Ebene.

4. Codegenerierung

4.1. Einordnung im Workflow

Der Quellcode wird an verschiedenen Stellen innerhalb des Frameworks in anderen Code übersetzt. In einem ersten Schritt wird der Quellcode in Form eines MATLAB®- oder Scilab-Skripts in generischen C-Code umgewandelt. Dieser ist speziell darauf optimiert, in einem späteren Schritt automatisiert analysiert und parallelisiert zu werden. Er wird nicht für eine bestimmte Zielplattform optimiert, sondern allgemein hinsichtlich Performanz und Speicherverbrauch unter Beibehaltung der Funktionalität und Genauigkeit des ursprünglichen Programms. Diese Optimierung ist speziell auf den Einsatz auf eingebetteten Systemen ausgerichtet und der Code könnte bereits mit Hilfe eines Compilers für die Zielplattform ohne Parallelisierung auf dieser ausgeführt werden. Im Hinblick auf die Parallelisierungsebenen können in diesem Codegenerierungsschritt bereits die Algorithmus- und die Code-Ebene berücksichtigt werden.

Nach der Parallelisierung auf Task-Ebene wird in einem abschließenden Schritt die Parallelisierung auf Daten-Ebene durchgeführt und paralleler C-Code generiert, der für die ausgewählte Zielplattform optimiert wurde. Der Verlauf der verschiedenen generierten Quellcodes ist in Abbildung 4.1 dargestellt.

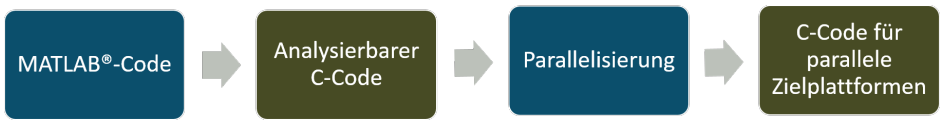


Abbildung 4.1.: Verlauf der Codegenerierung

Abschnitt 4.3 geht näher auf die Generierung des analysierbaren C-Codes und die dabei erfolgenden Optimierungsschritte ein. In Abschnitt 4.4 wird schließlich die Generierung des parallelen C-Codes beschrieben, der für die gewählte Zielplattform optimiert wurde.

4.2. Stand der Technik

Ein typischer Entwicklungszyklus im Bereich eingebetteter Systeme setzt auf die Entwicklung von Algorithmen in array-basierten Programmiersprachen und hier im speziellen der Sprache MATLAB®. Nachdem die Funktionalität der Algorithmen innerhalb von MATLAB® verifiziert wurden, werden sie auf die eingebetteten Systeme überführt. Dies wird in der Regel über die Sprache C ermöglicht, die sich als Standardsprache für die Programmierung eingebetteter Systeme etabliert hat. Neben der manuellen Neuentwicklung

gibt es eine Vielzahl an automatischen Codegeneratoren, sowohl kommerzielle als auch aus der Forschung, die den MATLAB®-Code nach C konvertieren.

Das bekannteste Tool ist der Embedded Coder [63] direkt von Mathworks. Er läuft weitgehend automatisch und akzeptiert nur wenige Nutzereingaben. Erstellt wird C/C++-Code mit Unterstützung für verschiedene Standards wie AUTOSAR oder ASAP2, sowie Rückverfolgbarkeitsberichte und eine Dokumentation von Codeschnittstellen, um die Qualifizierung des Codes gemäß DO-178, IEC 61508 oder ISO 26262 zu ermöglichen.

Aufgrund von spezielleren Anforderungen ist die automatische Generierung von C-Code aus MATLAB®-Code ein beliebtes Thema in der Forschung. In der Anfangszeit war die Umwandlung von MATLAB®-Code hin zu anderen Programmiersprachen ein wichtiges Thema, um eine höhere Ausführungsgeschwindigkeit zu erreichen. Beispielhaft sei hier die Arbeit [64] erwähnt, in der MATLAB®-Code nach Fortran 90 übersetzt wurde. Damit konnten die Ausführungszeiten in Einzelfällen um den Faktor 1000 beschleunigt werden.

Forschung in der neueren Vergangenheit zielt eher darauf ab, Unzulänglichkeiten am Embedded Coder durch neue Lösungen zu beheben. Ein flexiblerer Ansatz aus dem Forschungsbereich ist der MATISSE MATLAB® Compiler [65], der über eine aspektorientierte Programmiersprache namens LARA [66] gesteuert werden kann. LARA bietet den Vorteil, dass die Konfiguration getrennt vom eigentlichen Compiler erfolgen kann und somit einfach für verschiedene Fälle angepasst werden kann. Auf diese Weise lässt sich effizienter C- als auch OpenCL-Code [67] erzeugen, der speziell an die Zielarchitekturen angepasst ist.

Die Codegenerierung für spezielle Architekturen wird in vielen Arbeiten angewendet. In [68] werden Simulationen von Gehirnmodellen durch ein Framework für die Ausführung auf GPUs optimiert. MATLAB® wird bereits für die Entwicklung dieser Algorithmen eingesetzt, aber für die effizienteste Ausführung auf speziell entwickelter Hardware wird sehr viel Wissen für die Programmierung benötigt. Da die Anpassung dabei auch noch so speziell ist, ist die Simulation auch nicht mehr portabel und muss für alle Architekturen neu erstellt werden. GPUs bieten neben der benötigten Performanz auch noch klarere Programmierschnittstellen, die sehr gut mit automatischer Codegenerierung bedient werden können. Dabei können Hardware-Eigenschaften wie die Anzahl an Streaming-Prozessoren, die maximale Anzahl an Threads je Block sowie die Nutzung der Register und der lokalen Speicher mit betrachtet werden. Auf diese Art und Weise können neuronale Simulationen sehr effizient auf GPUs ausgeführt werden.

Ein vergleichbarer Ansatz wurde in [69] verfolgt, um modellprädiktive Steuerungen auf FPGAs zu bringen. Ziel war es, eine einfach zu benutzende Toolbox zu haben, die Rapid Prototyping auf eingebetteten Systemen, FPGAs und heterogenen Systemen ermöglicht. Sie nutzen die Modellierung und das Wissen über die Algorithmen aus, um mittels High Level Synthese Direktiven Codeteile/Schleifen zu markieren, die auf der Hardware parallel oder in Pipelines abgearbeitet werden können.

In [70] wird die Optimierung von Code für spezielle Architekturen auf die Ausnutzung von Festkommadatentypen betrachtet. Um Filter mit *unendlicher Impulsantwort* (IIR, engl. *infinite impulse response*) auf Hardware zu bringen, müssen Randbedingungen bezüglich der Quantisierung und Einschränkungen bei der Genauigkeit betrachtet werden. Dies eignet sich sehr gut für eine automatische Generierung, wobei viele Codegeneratoren

aber Probleme bezüglich der Komplexität des generierten Codes besitzen. Durch den Fokus auf spezielle Hardware sowie Unterstützung bestimmter Algorithmen, kann sich die Optimierung auf die hier auftretenden Probleme konzentrieren: Analyse von Pol- und Nullstellen, Zerlegung in Basiselemente wie Multiplikation, Addition und Verzögerung, Berechnung der notwendigen Matrizen und der Impulsantwort. Der hieraus generierte C-Code wird hinsichtlich Laufzeit und Speicherbedarf für das Zielsystem optimiert und kann mittels Konfigurationsparametern noch weiter verfeinert werden.

Die Codegenerierung für spezielle Algorithmen wird in [71] und [72] für Optimierungsalgorithmen ausgenutzt. Dabei setzt ersteres auf konvexe Probleme, bei denen der Lösungsraum aus konvexen Funktionen besteht. Durch diese Beschränkung kann sehr effizienter Code, der ohne weitere Bibliotheken und mit nur wenig Speicher auskommt, generiert werden. Das Vorgehen besteht dabei aus zwei Schritten: zunächst wird ein Solver für das Problem erstellt, der anschließend in effizienten C-Code übersetzt wird. [72] konzentriert sich auf das Lösen von nicht-konvexen Optimierungsproblemen. Diese sind komplexer, da sie potentiell viele lokale Minima und eine große Varianz bei den Lösungen besitzen und deshalb mindestens NP schwer sind. Auch sie setzen auf die Möglichkeit, das Optimierungsproblem in MATLAB® (oder Python) zu spezifizieren, um anschließend einen Solver zu erstellen und für diesen effizienten C- (oder Rust-) Code zu generieren. Beiden Lösungen ist gemein, dass sie zwar effizienten C-Code generieren, der auch für eingebettete Systeme geeignet ist, dabei aber keine speziellen Anpassungen an die tatsächliche Hardware vornehmen.

In [73] wurde schließlich evaluiert, wie sich ein modellbasierter Entwicklungs-Workflow mit automatischer Codegenerierung für einen Anwendungsfall aus der Automotive-Industrie eignet. Dabei setzen sie auf MATLAB®, Simulink und Stateflow zur Modellierung der Anwendung und den Embedded Coder zur Codegenerierung. Die automatische Konfiguration erfolgt über den integrierten Code Advisor, so dass keine Optimierung vom Anwender erfolgte. Die Qualität des automatischen Codes wurde mit einer rein manuellen Implementierung verglichen. Im Ergebnis ist der generierte Code im Allgemeinen nicht schlechter, bringt aber ein paar Nachteile mit sich. So ist die Datengröße in der Regel höher, da viele Checks zur Überprüfung der Sicherheit eingefügt werden, die manuell aber als redundant und unnötig erachtet werden. Insgesamt ist der Code zwischen 25-50 Prozent größer. Bei den Laufzeiten gibt es zwar einzelne größere Unterschiede, aber in Summe ist der generierte Code schneller. Die Studie zeigt also deutlich die Vorteile der automatischen Codegenerierung, aber auch einen gravierenden Nachteil: ein komplett automatischer Workflow kann Optimierungen, die im speziellen Fall durchgeführt werden können, nicht mit betrachten.

Die Übersetzung von MATLAB®-Code in andere Sprachen bringt einige Herausforderungen mit sich, die bereits in [74] betrachtet wurden. Dem speziellen Thema der Typinferenz haben die Autoren in [75] noch eine weitere Veröffentlichung gewidmet.

Alle hier vorgestellten Arbeiten haben als Ziel, den generierten C-Code direkt auf dem eingebetteten System einsetzen zu können. Die in Abschnitt 4.3 dargestellten Anforderungen zur Generierung von analysierbarem Code, der anschließend noch weiterverarbeitet werden soll, werden nur teilweise erfüllt.

Aufgrund der Komplexität bei der Programmierung von Mehrkernarchitekturen existieren bereits viele Arbeiten, die sich mit Konzepten zur einfacheren Programmierung befassen. Die Lösungen lassen sich dabei grob in drei Bereiche einteilen:

- *Entwicklung neuer Sprachen und Eingabemethoden*: Mit Sprachen, die speziell für die Programmierung von parallelen Architekturen entworfen wurden, lässt sich die unterliegende Hardware effizient programmieren. Für den Programmierer ist das Erlernen mit großem Aufwand verbunden, da er neben der neuen Syntax noch mit den Konzepten paralleler Programmierung vertraut sein muss. Dafür kann auf diese Weise performanter Code erzeugt werden, der mit kleinen Einschränkungen auch auf anderen Architekturen ausgeführt werden kann. Bekannte Sprachen aus dem Bereich sind X10 [76], Go [77], Erlang [78] oder Julia [79]. Die Sprachen können anhand ihres vorherrschenden Programmierparadigmas kategorisiert werden, dazu gehören Aktorenmodelle, koordinierende Sprachen, Datenflussprogrammierung, verteiltes Rechnen, ereignisgetriebene Programmierung, funktionales Programmieren oder Multi-Threading. Betrachtet man jedoch die Verbreitung von Programmiersprachen [80], so zeigt sich, dass sich keine Sprache, die auf die Programmierung paralleler Systeme spezialisiert ist, wirklich durchsetzen kann und dass gängige, allgemeinere Sprachen wie Java oder C/C++ immer noch deutlich beliebter sind. Es zeichnet sich auch nicht ab, dass sich eine bestimmte Sprache aus diesem Bereich in naher Zukunft durchsetzen würde.
- *Entwicklung von Programmiermodellen für gängige Programmiersprachen*: Durch Programmierstandards wie *Message Passing Interface* (MPI) [46] kann Parallelität in gebräuchlichen Sprachen wie C oder Fortran dargestellt werden. Bei geschickter Programmierung können die parallelen Verarbeitungseinheiten der Hardware ausgenutzt werden. Die Probleme dieses Ansatzes liegen neben dem Erlernen der parallelen Konzepte noch in einem relativ großen Overhead der einzelnen Befehle des Programmierstandards. Dieser ergibt sich dadurch, dass die Standards nicht auf eine spezielle Architektur hin optimiert werden, sondern möglichst breit eine Vielzahl an unterschiedlichen Hardwarerealisierungen unterstützen können sollen. Angefangen mit eingebetteten Systemen werden über Workstations bis hin zu HPC-Clustern sehr unterschiedliche Plattformen bedient, bei denen sehr große Unterschiede zwischen der Performanz und den zur Verfügung stehenden Ressourcen liegen. Die bekanntesten Vertreter sind MPI, OpenCL, CUDA und OpenMP.
- *Automatische Parallelisierung*: Die Grundidee hierbei ist, das nötige Wissen über eine effiziente, parallele Programmierung in einem Softwarewerkzeug unterzubringen. Auf diese Weise kann sich der Programmierer rein auf seinen Algorithmus konzentrieren und wird nicht durch andere Randbedingungen abgelenkt. Zum Einsatz kommen dabei gängige Programmiersprachen wie C/C++, die ursprünglich nicht für den Einsatz auf Mehrkernarchitekturen entwickelt wurden. Um die Parallelität darzustellen, kann auf existierende Programmiermodelle für die Architekturen zurückgegriffen werden.

Diese Arbeit beschäftigt sich mit der automatischen Parallelisierung von Anwendungen, so dass im Folgenden der aktuelle Stand der Technik auf diesem Gebiet noch genauer dargestellt werden soll.

Ein beliebter Ansatz ist die Verwendung des *Polytopmodells* (engl. *polyhedral model*). Mit Hilfe des Modells werden Schleifen, die als statische Kontrollstrukturen identifiziert werden, für die parallele Ausführung optimiert. Dies geschieht über Transformationen, die die Datenlokalität steigern oder die Abhängigkeiten zwischen den einzelnen Durchläufen der Schleife minimieren. Ein Beispiel für einen kompletten Ansatz, in dem zusätzlich zur Parallelisierung die Kommunikation behandelt wird, kann [81] entnommen werden.

Viele automatischen Lösungen setzen auf die Hilfe des Programmierers. Spezielle Regionen, die parallelisiert oder beschleunigt werden sollen, müssen gesondert markiert werden.

Ein ähnlicher Ansatz wie in ALMA wurde bereits in [82] durchgeführt. Der Hauptunterschied liegt in der Wahl der Eingangssprache und dem Fehlen von feingranularer Parallelisierung.

Vollständig automatisierte Lösungen, die zufriedenstellende Ergebnisse liefern, existieren auch nach Jahren der Forschung immer noch nicht. Parallelität kann sich auf unterschiedlichste Arten äußern und ist auch für Programmierer nicht immer sofort ersichtlich.

4.3. Generierung von analysierbarem C-Code

Wie in Abschnitt 2.2.2 dargestellt, existieren viele verschiedene Möglichkeiten, um einen Algorithmus darzustellen. Ziel dieses Ansatzes ist es, zunächst unabhängig von der später eingesetzten Hardware den Algorithmus zu entwickeln. Dies beschränkt die Auswahl der Sprache in Abbildung 2.7 auf die rechte Seite, welche weiter weg von der Hardware abstrahiert wurde. Da der Schwerpunkt zudem auf mathematisch beschreibbaren Algorithmen liegt, sind Computeralgebrasysteme und array-basierte Sprachen die beiden passendsten Kategorien. Für diese Arbeit wurde der Fokus auf die Sprachen MATLAB® und Scilab gesetzt, da sie für die Entwicklung von Daten- oder Kontrollflusslastigen Algorithmen besser geeignet sind als reine Computeralgebrasysteme, die sich besser für die algebraische und analytische Analysen eignen.

Ziel dieses Codegenerierungsschrittes ist es, einen Algorithmus, der in einer vornehmlich mathematischen Beschreibungssprache vorliegt, effizient auf einem eingebetteten System auszuführen. Dabei sollen die Ressourcen möglichst effizient ausgenutzt werden, so dass der Speicher und die vorhandenen verarbeitenden Einheiten gut ausgenutzt werden. Der Programmablauf soll dabei an die Gegebenheiten der Hardware angepasst werden, indem Sprünge oder Fließkommaberechnungen nur soweit zum Einsatz kommen, wie es unbedingt vonnöten ist. Da zudem zur Ausführung von array-basierten Programmiersprachen eine schwergewichtige Laufzeitumgebung benötigt wird, wurde mit C-Code eine Darstellungsform des Programms gewählt, die am besten mit dem Umfeld der eingebetteten Architekturen kompatibel ist.

Daraus ergeben sich die folgenden Anforderungen an die Codegenerierung:

1. Eingangscode einlesen und in eine Darstellung bringen, auf der weitere Optimierungen angewendet werden können.

2. Die Ausführung des Programms an die Einschränkungen des Zielsystems anpassen:
   a) Reduktion von Entscheidungen und Dynamik zur Laufzeit, da dies einen großen Overhead hinsichtlich der Performanz auf eingebetteten Systemen bedeutet.
   b) Fixierung von Variablengrößen und Datentypen, um den Speicherverbrauch zu reduzieren.
   c) Entfernen von unnötigem Code, der bei der Ausführung weggelassen werden kann.
   d) Effiziente Implementierung von Funktionen, die in der neuen Sprache deutlich aufwendiger sein müssen.
3. Erzeugung von korrektem C-Code, der das gleiche Verhalten hat wie der ursprüngliche Algorithmus.
4. Unterstützung von zusätzlichen Anweisungen in Form von Pragmas, um bereits früh das Wissen des Programmierers über den Algorithmus oder die Zielarchitektur einfließen zu lassen.
5. Der Code soll erst im Anschluss weiter an die Zielhardware angepasst werden und muss in diesem Schritt noch nicht weiter dafür optimiert werden.
6. Für die späteren Abhängigkeitsanalysen bezüglichen des Daten- und Kontrollflusses muss der Code für eine statische Analysierbarkeit optimiert werden.

4.3.1. Codegenerierung aus arraybasierten Programmiersprachen

Die wichtigsten Eigenschaften der Sprachen wurden bereits in Abschnitt 2.2.2.3 zusammengefasst. Im Folgenden wird das Vorgehen bei der Transformation von array-basierten Programmiersprachen hin zu C dargestellt.

4.3.1.1. Auflösen von dynamischen Entscheidungen

MATLAB® und Scilab gehören zu Skriptsprachen, die mit einem Interpreter ausgeführt werden. Kennzeichnend ist, dass der Quellcode des Programms nicht von einem Compiler direkt in ausführbaren Maschinencode übersetzt wird, sondern dass der Code sequentiell abgearbeitet wird. Dazu wird der Code nach dem Einlesen analysiert und die entsprechenden Anweisungen zeilenweise durchgeführt. Auf diese Weise muss der Programmierer Entscheidungen, die die Auswahl des Datentyps oder die Form bzw. Größe einer Matrix betreffen, nicht selber treffen, sondern werden dynamisch zur Laufzeit getroffen. So kann eine Matrix, die zunächst mit einer bestimmten Anzahl an Elementen und Dimensionen erstellt oder verwendet wurde, durch Zugriff auf ein Element in einem bislang nicht definierten Bereich vergrößert werden. Dies beschleunigt die Entwicklung von Algorithmen, da sich der Programmierer nicht damit befassen muss, verlangsamt aber die Ausführung des Algorithmus. Da das Laufzeitsystem zudem noch einen großen Overhead mit sich bringt, ist es nicht sinnvoll, diesen Ansatz auf einem eingebetteten

System zu verfolgen. Die im Folgenden vorgestellten Methoden können die Laufzeitentscheidungen bereits zur Designzeit treffen.

Speicherallokation
Variablen werden in den arraybasierten Sprachen immer auf dem Stack angelegt, welcher in den Sprachen auch die einzige Einstellung hinsichtlich des Speichers ist. In der Sprache C stehen mehr Möglichkeiten zur Verfügung. So muss zunächst entschieden werden, ob eine Variable global oder lokal erzeugt werden soll. Globale Variablen sind von überall im Programm aus zugreifbar. Sind sie bereits initialisiert, werden sie in der Datensektion abgelegt, ansonsten in der BSS-Sektion (von *Block Started by Symbol*). Dies kann ein wichtiges Kriterium bei Systemen mit wenig Speicher sein. Lokale Variablen hingegen werden auf dem Stack erzeugt und sind nur so lange gültig wie die umschließende Funktion existiert. Für die spätere Parallelisierung werden lokale Variablen bevorzugt, da für parallele Zugriffe auf globale Variablen aufwendigere Synchronisationsmechanismen benötigt werden. Für die Generierung von effizientem C-Code müssen Variablen in den verschiedenen Speicherbereichen generiert werden können. Diese Optimierungen sind jedoch noch nicht auf das jeweilige Zielsystem hin optimiert, sondern bringen Vorteile bei der Ausführung auf allen (eingebetteten) Systemen.

Typinferenz
Alle numerischen Datentypen, die in MATLAB® oder Scilab verwendet werden, werden ebenso von der Ausgangssprache C unterstützt. Dies sind neben Fließkommazahlen mit einfacher (32 Bit) und doppelter Genauigkeit (64 Bit) noch ganze Integer-Zahlen mit 8, 16, 32 und 64 Bit, sowohl mit (*signed*) als auch ohne (*unsigned*) Vorzeichen. Eine Umwandlung der Datentypen rein aufgrund der Funktionalität ist deshalb nicht erforderlich. Die standardmäßige Verwendung von Fließkommazahlen mit doppelter Genauigkeit ist auf eingebetteten Systemen jedoch nicht sinnvoll, da der Speicherbedarf oft unnötigerweise ansteigt und eine Hardwareunterstützung des Datentyps oft nicht vorhanden ist. Ziel der Generierung von effizientem Code ist es also, möglichst kleinere Datentypen zu verwenden, ohne an Genauigkeit bei der Berechnung zu verlieren.

Um den kleinstmöglichen und damit effizientesten Datentyp einer Variablen zu bestimmen, müssen alle Zugriffe auf die Variable untersucht werden. Standardmäßig wird der größtmögliche Datentyp mit doppelter Genauigkeit verwendet, jede Einschränkung im Code verringert aber die Größe. Im einfachsten Fall wird der Typ oder der Wertebereich durch den Programmierer angegeben. Diese Einschränkung wird auf die gesamte Laufzeit bezogen und legt den Datentyp fest. Kommen keine Vorgaben und damit Einschränkungen über den Wertebereich einer Variablen durch den Benutzer zum Einsatz, kann eine Analyse der eingesetzten Operationen durchgeführt werden. Im einfachsten Fall ist das eine Zuweisung von einer ganzen Zahl. Einfache mathematische Operationen wie Addition, Subtraktion oder Multiplikation, nicht aber die Division, mit anderen ganzen Zahlen haben keine Auswirkungen auf den Datentyp, so dass in C in der Regel eine Integer-Zahl verwendet werden kann. Einziges Problem hierbei ist der verwendete Wertebereich: bei Berechnungen, deren Ergebnisse nicht mehr mit der ganzen Zahl dargestellt werden können, kommt es zu Fehlern durch einen Überlauf der Variablen. Operationen, die

die Genauigkeit einer Variablen verändern, können ebenfalls zur Festlegung von Datentypen verwendet werden. Als Beispiel seien hier die Modulo-Operation oder die Betragsberechnung genannt. Beide liefern als Ergebnis eine ganze Zahl, abhängig von weiteren Operationen auf die Zahl kann der Datentyp auf eine ganze Zahl festgelegt werden. Die Betrachtung des Kontextes erlaubt ebenfalls Rückschlüsse auf mögliche Datentypen: die Zählvariable von For-Schleifen kann üblicherweise als ganze Zahl dargestellt werden. Schleifendurchläufe haben immer ganze Zahlen und die häufig verwendeten Zugriffe auf Matrizen über Indizes werden in der Regel auch mit ganzen Zahlen durchgeführt. Die Eingangssprachen erlauben hier zwar Zugriffe mit reellen Zahlen, dies wird vom Programmierer jedoch selten erwartet.

Die Bestimmung des Wertebereichs einer Variablen ist ohne Mithilfe des Programmierers nur in wenigen Fällen möglich. Der einfachste Fall sind Variablen, die quasi als Konstanten verwendet werden und nur einmal geschrieben werden. Hier kann der kleinstmögliche Datentyp verwendet werden, der die Zahl noch darstellen kann. Bei For-Schleifen ist es oft möglich, die Anzahl an Durchläufen bereits statisch zu bestimmen. Dies gibt somit die Obergrenze der Zählvariablen.

Komplexe Zahlen haben keinen eigenen Datentyp in C und müssen über gesonderte Strukturen dargestellt werden. Da dies Auswirkungen auf sämtliche Operationen hat, kommen imaginäre Zahlen nur explizit zum Einsatz, das heißt, dass sie bereits in der Eingangssprache gesondert markiert werden müssen. Operation wie die Quadratwurzel aus einer Zahl erwarten standardmäßig nur positive Zahlen als Eingangsdaten und nur bei expliziter Verwendung von imaginären Zahlen können auch Wurzeln von negativen Zahlen berechnet werden.

Forminferenz

Die standardmäßige Verwendung von Matrizen für alle Variablen hat ähnliche Auswirkungen wie die dynamische Auswahl des Datentyps. In C beschriebene Matrizen brauchen je nach Darstellung mehr Speicher, da noch zusätzliche Zeiger für den Datenzugriff erzeugt werden. Es ist daher sinnvoll, die kleinstmögliche Größe der Matrix zu bestimmen, die die benötigten Werte aufnehmen kann. Die bereits im vorigen Abschnitt beschriebenen Methoden zur Bestimmung ganzer Zahlen können ebenfalls verwendet werden, um rein skalare Zahlen zu erkennen.

Zunächst werden alle Indizes betrachtet, mit denen auf die Matrix zugegriffen wird. Können hierbei Maxima bestimmt werden, so ergeben sich hieraus die maximalen Größen der Matrizen. Häufig kann es jedoch vorkommen, dass in Programmen in MATLAB® nie auf einzelne Elemente einer Matrix zugegriffen wird, sondern immer die gesamte Matrix weiterverwendet wird und am Ende auch alle Ergebnisse in einer Matrix von Interesse sind. Bei dieser Art der Datenverarbeitung können die einzelnen Operationen, die mit den Matrizen durchgeführt werden, Aufschluss über die Form einer Matrix geben. So kann beispielsweise aus einer zweidimensionalen Matrixmultiplikation geschlossen werden, dass die Spaltenzahl der ersten Matrix mit der Zeilenzahl der zweiten Matrix übereinstimmen muss. Andere Operationen wie die Addition oder Subtraktion können nur ausgeführt werden, wenn beide Matrizen die gleiche Form haben. Durch Analyse sämtlicher Operationen auf die Matrizen können also viele Informationen durch Relationen mit anderen Ma-

trizen bestimmt werden. Im Falle von bekannten Funktionen wie der Bestimmung des Eigenwertes ist die Form des Ergebnisses bereits bekannt und kann als neuer Zustand verwendet werden.

MATLAB® und Scilab unterstützen Matrizen, die ihre Größe zur Laufzeit verändern können. So kann beispielsweise eine Matrix zum Sammeln von Ergebnissen einer Berechnung verwendet werden, bei der jede Iteration einen neuen Vektor als Ergebnis liefert und als neue Spalte in der Matrix gespeichert wird. Die tatsächliche Größe der Matrix hängt also unter Umständen von den Eingangsdaten ab. Das gleiche Verhalten in C abzubilden erfordert einen großen Overhead, da beim Hinzufügen von neuen Ergebnis-Vektoren zu einem bestehenden Array immer ein neues Array mit der neuen Größe erstellt werden muss und alle Daten reinkopiert werden müssen. Da durch die vorigen Analysen die maximale Größe solcher Matrizen bestimmt werden kann, werden Matrizen mit dynamischer Größer über zwei Arrays abgebildet. Das erste Array wird mit den maximalen Größen in jeder Dimension initialisiert und wird zur Speicherung der Daten verwendet. Ein zweites Array wird eingesetzt, um die aktuelle Größe jeder einzelnen Dimension bereitzustellen. Dieses zweite Array kann nun zum Setzen von Obergrenzen in Schleifen eingesetzt werden, um somit den Zugriff auf den aktuell genutzten Bereich der Datenmatrix zu gewährleisten.

```
1  int a_data[100][100];
2  int a_size[2];
3  int i, j;
4
5  a_size[0] = 50;
6  a_size[1] = 50;
7
8  for (i = 0; i < a_size[0]; ++i) {
9    for (j = 0; j < a_size[1]; ++j) {
10     a_data[i][j] = i*j;
11   }
12 }
13
14 ...
15
16 a_size[0] = 80;
17 a_size[1] = 80;
18
19 for (i = 0; i < a_size[0]; ++i) {
20   for (j = 0; j < a_size[1]; ++j) {
21     a_data[i][j] = (i == j);
22   }
23 }
```

Quellcode 4.1: Beispiel für eine Matrix mit variabler Größe in C

In Quellcode 4.1 ist ein Beispiel in C dargestellt, das den Umgang mit einer Matrix mit variabler Größe zeigt. Die ursprüngliche Variable *a* im MATLAB®-Script wird als die beiden Variablen *a_data* und *a_size* dargestellt. *a_data* wird mit einer Größe

von $100 \cdot 100$ initialisiert, was der maximalen Größe im MATLAB®-Code entspricht. Der Vektor *a_size* gibt die aktuell genutzte Größe der beiden Dimensionen an. Nachdem die Größen in Zeile 5 und 6 auf 50 gesetzt wurden, arbeitet die verschachtelte For-Schleife aus Zeile 8 und 9 nur auf einer Matrix mit der Größe $50 \cdot 50$. Später im Code kann die Größe beispielsweise auf 80 gesetzt werden, wie in den Zeilen 16 und 17 dargestellt. Die For-Schleifen in den Zeilen 19 und 20 arbeiten nun auf einer Größe von $80 \cdot 80$. Für Fälle, in denen nicht alle Elemente des genutzten Bereichs geschrieben werden, muss sichergestellt werden, dass alle Elemente zunächst mit 0 initialisiert werden. Dieses Verhalten ist in C nicht festgelegt und hängt von der Zielarchitektur und dem Compiler ab. Für einen portablen generischen C-Code muss diese Initialisierung also mit generiert werden.

4.3.1.2. Anpassungen an die Ausgangssprache

Die Unterschiede in den beiden Sprachen C und MATLAB® sorgen dafür, dass mathematische Konstrukte teilweise deutlich anders dargestellt werden. Als Beispiel soll eine simple Matrixmultiplikation herangezogen werden. Die beiden Matrizen A und B sollen multipliziert werden und das Ergebnis soll in der Matrix C gespeichert werden. In MATLAB® kann dies recht kurz durch $C = A * B$ dargestellt werden. Die Darstellung in C erfordert noch etwas an Vorarbeit, um das gleiche auszudrücken.

Loopify - Erzeugung von Schleifen
Matrizen werden in C als Arrays dargestellt. Eine zweidimensionale Matrix A bestehend aus ganzen Zahlen wird als `int A[n][m]` definiert, wobei *n* für die Anzahl an Zeilen und *m* für die Anzahl an Spalten steht. Soll nun ein Zugriff auf alle Elemente einer Matrix erfolgen, wird dazu für jede Dimension eine For-Schleife benötigt. In diesem Beispiel also eine äußere Schleife mit *n* Durchläufen und eine innere Schleife mit *m* Durchläufen.

Simplify - Vereinfachung des Programmablaufs
Die im vorigen Schritt erzeugten Schleifen sind nicht in jedem Fall notwendig, da beispielsweise eine Initialisierung mit 0 deutlich effizienter über den `memset` Befehl erledigt werden kann. Der Simplify Schritt hat zum Ziel, alle unnötigen Schleifen zu entfernen und durch einfacheren Code zu ersetzen.

Optimierungen für Analysierbarkeit und weitere Verarbeitung
Alle Transformationen, mit denen die dynamischen Entscheidungen zur Laufzeit in statischen Code übersetzt werden, sind zum einen notwendig, um effizienten C-Code zu erzeugen. Darüber hinaus sorgen sie aber auch dafür, dass sich das Laufzeitverhalten des erzeugten Codes besser analysieren lässt. Variablen haben feste Größen, so dass kein Overhead durch dynamische Speicherverwaltung betrachtet werden muss. Direkte Nutzung der Variablen sorgt dafür, dass keine Pointerarithmetik beachtet werden muss. Diese erschwert die statische Analyse, da der Programmablauf von Eingangsdaten abhängen kann. Des Weiteren kann bei der Codegenerierung eine Konstantenpropagation durchgeführt werden. Diese findet bereits bei der Analyse der Datentypen statt und kann verwendet werden, um die Anzahl der Parameter von Funktionen zu reduzieren oder Berechnungen zu vereinfachen, indem Terme bereits statisch berechnet werden können.

Eine weitere Eigenheit betrifft die Parameter von Funktionen. In MATLAB®/Scilab werden sie explizit als Eingangs- oder Ausgangsparameter spezifiziert. Dies ist in C nicht möglich, beim Aufruf mit „call-by-reference“ kann eine Variable immer als Ein- und Ausgangsparameter verwendet werden. Die zusätzlichen Richtungsinformationen können im C-Code aber als Pragmas und somit als Hinweise für die weitere Verarbeitung angegeben werden.

4.3.1.3. Anpassungen an Zielhardware

Auch wenn der hier generierte C-Code noch nicht an eine bestimmte Hardware angepasst werden soll, gibt es bei der Umwandlung von array-basierten Sprachen hin zu C Optimierungen, die in diesem Schritt deutlich effizienter durchgeführt werden können. Der wichtigste Punkt betrifft dabei das Layout der Daten im Hauptspeicher. Nehmen wir als Beispiel eine zweidimensionale Matrix M in MATLAB®: bei üblicher Syntax wird mit $M(1,2)$ auf das zweite Element der ersten Zeile zugegriffen. In C-Code greift man über $M[1][2]$ dagegen auf das dritte Element der zweiten Zeile zu. Indizes beginnen in MATLAB® bei 1, in C dagegen bei 0. Zusätzlich steht die erste Zahl in MATLAB® für die Zeile, die zweite für die Spalte. In C ist es umgekehrt. Um den C-Code so darzustellen, wie ihn ein C-Entwickler schreiben würde, werden alle Indizes umgerechnet und beginnen bei 0. Der Start bei 1 würde es erfordern, entweder im C-Code immer mit Offsets arbeiten zu müssen, die zwar vom Compiler herausoptimiert werden können, für den Anwender aber immer präsent sind. Alternativ könnte auch zusätzlicher Speicher reserviert werden, der aber nicht verwendet wird. Dieser Overhead ist aber mit Hinblick auf die anvisierten eingebetteten Systeme keine gute Option. Die Adressierung, wie sie im C-Code vorgenommen wird, entspricht dem Layout der Daten im Hauptspeicher. Bei Algorithmen, die erst Daten aus einer Dimension einlesen und erst anschließend aus einer anderen, kann es von Vorteil sein, die Daten in der Reihenfolge einzulesen, wie sie im Speicher stehen. Diese erhöhte Datenlokalität sorgt dafür, dass vorhandene Caches besser ausgenutzt werden können. Dies gilt vor allem, wenn ganze Zeilen aus dem Hauptspeicher gelesen werden können. Die Umwandlung von array-basierten Sprachen hin zu C bietet nun die Möglichkeit, alle Zugriffe auf die Daten in einer für die Zielplattform optimierten Reihenfolge darzustellen. Dazu kann für einzelne Variablen angegeben werden, in welcher Reihenfolge die einzelnen Dimensionen im Speicher hinterlegt werden sollen. Ein Beispiel ist in den Quellcodes 4.2 und 4.3 dargestellt. Die ursprüngliche Variable im MATLAB®-Code hat die Dimensionen 480x640x4, welche im C-Code ebenfalls als 480x640x4 dargestellt werden würde. Mit der Anweisung `cmd_var_dimorder` kann diese Reihenfolge jedoch verändert werden, so dass die Variable mit den Dimensionen 4x640x480 generiert wird. Alle Zuweisungen auf die einzelnen Werte der Matrix werden bei der Codegenerierung berücksichtigt, so dass die Änderung der Dimensionsreihenfolge keinen Einfluss auf die Funktionalität, sondern nur auf die Anordnung der Daten im Speicher hat. Auf diese Weise kann bereits auf dieser Algorithmus-Ebene die Performanz der Anwendung auf der späteren Zielplattform optimiert werden. Aufgrund der höheren Abstraktion ist das Verändern der Reihenfolgen unter Berücksichtigung aller Zugriffe deutlich effizienter als wenn diese Optimierung erst auf Code-Ebene durchgeführt wird.

```
1 %CMD?: cmd_var_dimorder(ret_image, [3 2 1]);
```

```
2  ret_image = uint8(rand(480,640, 4)*256);
```

Quellcode 4.2: Beispiel für MATLAB®-Code mit Anweisung, Dimensionen einer Variablen zu vertauschen

```
1  uint8_t ret_image_data[4][640][480];
2  size_t i1, i2, i3;
3
4  for (i3 = 0u; i3 < 4u; i3++) {
5    for (i2 = 0u; i2 < 640u; i2++) {
6      for (i1 = 0u; i1 < 480u; i1++) {
7        ret_image_data[i3][i2][i1] = (round(rand_mt() * 256.0));
8      }
9    }
10 }
```

Quellcode 4.3: Beispiel für eine generierte Variable mit vertauschten Dimensionen

Neben dem Layout der Daten im Hauptspeicher wird bei der Codegenerierung auch der Umgang mit großen Variablen geregelt. Ohne eine Sonderbehandlung würden Variablen in der Regel auf dem Stack innerhalb von einer Funktion generiert, mit der Gefahr, dass dieser überlaufen kann und somit das Programm nicht weiter ausgeführt werden kann. Bei der Codegenerierung werden drei alternative Möglichkeiten zur Auswahl geboten:

1. Die dynamische Speicherverwaltung auf dem Heap: im C-Code wird der Speicher bei Bedarf mittels `malloc` reserviert und nach Verwendung mit `free` wieder freigegeben. Aufgrund des beschränkten Speichers eingebetteter Systeme ist diese dynamische Allokation mit dem Risiko verbunden, dass nicht genügend Speicher für alle Variablen vorhanden ist. Deshalb sollte von dieser Speicherverwaltung im eingebetteten Umfeld abgesehen werden.
2. Globale Variablen: Anstatt große Variablen auf dem Stack zu erzeugen werden sie als global markiert, indem sie außerhalb von Funktionen deklariert werden. Das sorgt dafür, dass sie bereits vom Compiler beim Programmstart in einem anderen Speicherbereich allokiert werden. Auf diese Weise ist bereits beim Kompilieren bekannt, ob die Daten in den vorhandenen Speicher passen oder nicht. Da diese Speicherbereiche jedoch nicht wieder freigegeben werden können, stehen diese auch nicht für eine spätere Wiederverwendung zur Verfügung.
3. Statische Variablen: mit dem Schlüsselwort `static` können Variablen innerhalb von Funktionen als statisch deklariert werden. Das hat zur Folge, dass Variablen bei erneutem Aufruf der Funktion ihren letzten Wert vom letzten Aufruf behalten. Die Initialisierung von statischen Variablen erfolgt nur ein Mal. Statische Variablen werden vom Compiler gleich behandelt wie globale Variablen und auch im gleichen globalen Speicherbereich angelegt. Aufgrund der Beibehaltung des alten Werts wird in der Codegenerierung zusätzlich mit `memset` dafür gesorgt, dass bei wiederholten Aufrufen nicht die alten Werte verwendet werden. Im Vergleich zur Lösung mit den

globalen Variablen haben die statischen jedoch den Vorteil, dass die Lesbarkeit zunimmt, da Variablen dennoch nur in den Funktionen verwendet werden können, in denen sie deklariert wurden.

Standardmäßig werden alle Variablen mit einer Größe von mehr als 256 Byte als statische Variablen generiert.

4.3.2. Optimierungen für die Parallelisierung

Die Generierung des generischen C-Codes wird hier nicht als eigene Ebene für die Parallelisierung betrachtet, ist aber dennoch eng mit ihr verknüpft. Die Parallelisierung auf Algorithmus-Ebene wird bei der Codegenerierung durchgeführt und näher in Abschnitt 5.2 beschrieben. Teile der Parallelisierung auf Code-Ebene werden ebenfalls hier durchgeführt, diese Beschreibung kann Abschnitt 5.3 entnommen werden.

Im Folgenden werden noch weitere Features der Codegenerierung beschrieben, die bei der späteren Parallelisierung eingesetzt werden können.

4.3.2.1. Funktionen nur für die Codegenerierung

Um Funktionen zu implementieren, die zwar wichtig für die Codegenerierung sind, aber keinen Einfluss auf den ursprünglichen MATLAB®- oder Scilab-Code haben sollen, wird auf die Kommentarfunktionalität der Eingangssprachen gesetzt. In MATLAB® werden Codezeilen, die mit `%` beginnen und in Scilab werden Zeilen, die mit `//` beginnen, ignoriert. Wird eine Zeile nun mit dem Kommentarzeichen begonnen und mit `CMD?:` fortgesetzt, wird der nachfolgende Code nur bei der Codegenerierung ausgewertet, von den array-basierten Sprachen aber ignoriert.

4.3.2.2. Pragmas

Pragmas sind Statements im Quellcode, um Anweisungen an den Compiler zu übergeben. Bei der Codegenerierung werden sie verwendet, um zusätzliche Informationen, die bei der Umwandlung der array-basierten Sprachen hin zu C-Code gewonnen wurden, an die Parallelisierung zu übergeben. Unterstützt werden die folgenden Anweisungen:

PERFINFO
: Mit dem Format „`PERFINFO n_abs n_rel`“ wird die Anzahl an Ausführungen eines Codeblocks angegeben. `n_abs` steht dabei für die absolute Anzahl an Aufrufen, `n_rel` steht für die relative Anzahl im Vergleich zu den übergeordneten Blöcken. Im einfachsten Fall können die Werte direkt angegeben werden, da die Umwandlung des Codes bereits darauf abzielt, möglichst viele For-Schleifen mit konstanten Grenzen zu generieren. Diese Größen können in der Regel direkt aus der Größe der zu bearbeitenden Matrizen abgeleitet werden. Aufrufe, die nicht auf diese Art und Weise statisch analysiert werden können, wie beispielsweise Schleifen, die von Funktionsaufrufen begrenzt werden oder If-Blöcke, die eine bestimmte Verteilung zwischen dem true- und dem false-Pfad aufweisen, können durch Profiling auf dem

Host-System ermittelt werden. Dazu wird zunächst C-Code generiert, der bei allen zu messenden Blöcken jeweils den Start und das Ende markiert und entsprechende Zähler ansteuert. Nach dem Ausführen auf dem Host können die ermittelten Zahlenwerte in einer Datei gespeichert werden und anschließend bei einem erneuten Aufruf der Codegenerierung geladen werden. Die Informationen des Pragmas werden bei der Generierung von Performanzwerten für die einzelnen Blöcke ausgewertet und liefern die Basis für die Abschätzung der Zeit für die Parallelisierung.

KILLVAR

Die Anweisung „`KILLVAR var_name`“ wird vor einem Ausdruck platziert um zu markieren, dass der aktuelle Wert der Variablen nicht mehr benötigt wird. Ein häufiger Anwendungsfall ist die Platzierung vor einer Schleife, in welche die angegebene Matrix-Variable vollständig mit neuen Werten überschrieben wird. Diese Anweisung kann sehr gut während der Codegenerierung erstellt werden, da die Zugriffe auf die einzelnen Elemente der Matrizen bereits für die Konvertierung benötigt werden. Die Analyse später im Quellcode ist deutlich aufwändiger, da bei Schleifen alle Zugriffe auf die einzelnen Elemente der Matrix analysiert werden müssen, um sicherzustellen, dass auch wirklich die gesamte Matrix überschrieben wird. Das Pragma wird vor der Parallelisierung ausgewertet und sorgt dafür, dass an der Position des Pragmas eine Pseudo-Definition der Variablen erzeugt wird. Diese sorgt dafür, dass keine Abhängigkeit zwischen dem Wert vor der Anweisung und nach der Anweisung erzeugt werden.

Pragmafunktion

Mit der eigens definierten Funktion `cgen_pragma(String)` können Pragmas direkt in der Array-Sprache definiert werden, die nach der Codegenerierung an der entsprechenden Stelle im C-Code platziert werden. Der Haupteinsatzzweck im Workflow ist das Setzen von Randbedingungen für den Scheduler. Dabei werden zwei wichtige unterschieden: Zuweisungen und Cluster.

Mittels „`cgen_task_alloc n`“ wird der nächste Block auf Kern n zugewiesen und kann vom Scheduler nicht mehr verändert werden. Dies kann verwendet werden, um eine bestimmte Parallelisierung zu erzwingen, bei der alle relevanten Blöcke manuell auf bestimmte Kerne festgelegt werden. Auf diese Weise kann ein Ergebnis der Parallelisierung bereits im Quellcode verankert werden und muss nicht bei Änderungen am Quellcode erneut angepasst werden. Ein weiterer Anwendungsfall kommt beim Zugriff auf die Hardware zum Tragen. Je nach Plattform kann es sein, dass nur ein bestimmter Kern Zugriff auf eine bestimmte Ressource hat. Diese Information ist dem automatischen Scheduler nicht bekannt, so dass die Zuweisung bereits im Quellcode erfolgen kann. Schließlich kann das Pragma auch bei sicherheitskritischen Anwendungen verwendet werden, bei denen die Randbedingung existieren kann, dass bestimmte Codeteile nicht auf dem gleichen Kern ausgeführt werden dürfen. Dies soll verhindern, dass unerwünschte Einflüsse wie Änderungen des Timings durch einen anderen Task auf dem gleichen Kern entstehen können.

„`cgen_task_cluster`“ kann verwendet werden, um die automatische Parallelisierung eines Tasks und dessen Kindern zu verhindern. So markierte Tasks werden als eine Einheit vom Scheduler betrachtet und können nur im Ganzen auf einen Kern zugewiesen werden. Übliche Einsatzzwecke sind Code-Abschnitte, die bei Paral-

lelisierung eine hohen Kommunikationsaufwand erzeugen und deshalb nicht auf mehreren Kernen ausgeführt werden sollen. Diese Fälle können zwar bei der automatischen Parallelisierung ebenfalls erkannt und verhindert werden, sind dann aber auf komplexere Algorithmen und genauere Modelle der Hardware angewiesen. Weitere Fälle ist Code, der mit unbekannten oder auch so genannten „Black Box"-Funktionen geschrieben wurde. Können der Datenfluss und die Seitenfunktionen einer Funktion nicht statisch analysiert werden, so muss immer davon ausgegangen werden, dass eine parallele Ausführung, bei der die Aufrufreihenfolge verändert wurde, unbekannte Seiteneffekte hat. Deshalb muss eine sequentielle Ausführung mit einem Cluster-Pragma erzwungen werden.

4.3.2.3. Der Entscheidungsmechanismus

Auch wenn die Transformationen bereits bei der Generierung des sequentiellen Codes angewendet werden, sollen sie auch erst später im Workflow ausgewählt werden können. Dazu wird ein Mechanismus benötigt, mit dem Optionen für die Codegenerierung über den C-Code und darüber hinaus propagiert werden können, ohne Einfluss auf die tatsächliche Funktionalität zu haben. Für diese Art an Hinweisen, die erst später von einem Compiler oder ähnlichem ausgewertet werden, eignen sich so genannte Pragmas, die in C durch `#pragma` gekennzeichnet werden. Unbekannte oder nicht unterstützte Pragmas werden von Compilern ignoriert und haben keine Auswirkungen. MATLAB® unterstützt diese Art von Pragmas nicht, so dass sie über CMD-Kommentare (siehe Sektion 4.3.2.1) realisiert werden. Der Entscheidungsmechanismus setzt auf Pragmas in folgender Form:

```
%CMD?: dec = cgen_decision(<default>, '<name>', '<description>',
'<datatype>', '<visual>', <min>, <max>);
```

dec ist die Entscheidungsvariable, die den Wert der Entscheidung enthält. Sie ist eine normale Variable in den array-basierten Programmiersprachen und kann im Weiteren entweder direkt im Code verwendet werden oder dient als Parameter von Funktionen, die nur bei der Codegenerierung angewendet werden (siehe Sektion 4.3.2.1).

default gibt den Standardwert der Entscheidung an, der verwendet wird, sollte der Wert nicht an anderer Stelle gesetzt worden sein.

name ist der interne, eindeutige Name der Entscheidungsvariablen.

description beschreibt die Funktion der Entscheidungsvariablen im aktuellen Kontext. Diese Beschreibung kann für eine spätere, grafische Darstellung verwendet werden.

datatype definiert den Datentyp der Variablen. Unterstützt werden Integer, Float und Boolean.

visual gibt an, wie die Entscheidung grafisch dargestellt werden soll. Abbildung 4.2 zeigt eine beispielhafte Darstellung. Mögliche Optionen sind dabei:

checkbox erstellt eine boolesche Variable, die mit Hilfe einer Checkbox aktiviert werden kann.

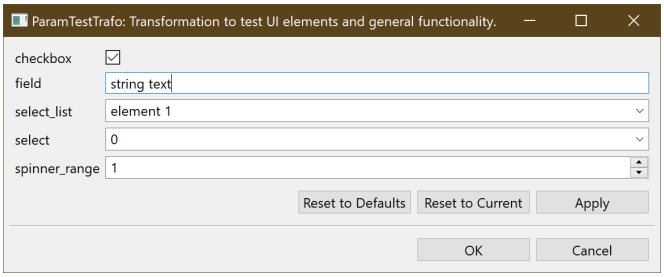


Abbildung 4.2.: Beispiel für grafische Darstellung von Entscheidungsvariablen

field um eine Textbox zu erstellen, mit der ein Freitext als String übergeben werden kann.

select_list generiert eine Auswahlliste mit allen Parametern, die nach dem Schlüsselwort angegeben werden.

select/select_range erlaubt es, vordefinierte Werte als Option für eine Integer-Variable in einer Liste auszuwählen.

spinner_range kann für Integer-Variablen mit einem wohldefinierten Wertebereich verwendet werden. Als grafisches Element wird dazu ein so genannter Spinner verwendet, der neben einem Textfeld für den Wert noch Buttons zum Erhöhen und Verringern des Werts mitbringt.

group kann verwendet werden, um mehrere Parameter zusammenzufassen und in einem gemeinsamen Einstellungsdialog bereit zu stellen.

min, max geben im Falle von Integer und Float die minimalen und maximalen Werte der Variablen an.

Im Standardfall wird der über `default` angegebene Wert für die Variable verwendet, welche anschließend zur Auswahl von Optionen oder Code-Alternativen verwendet werden kann. Der Wert kann jedoch beim Aufruf des Codegenerators überschrieben werden, indem der eindeutige Name der Variablen und ein zu setzender Wert übergeben werden. Dies kann ebenfalls über eine Datei mit Wert-Schlüssel Paaren im Format einer Initialisierungsdatei: `Schlüssel=Wert` realisiert werden. Diese Mechanik erlaubt es nun, Dateien, die erst später im Workflow erzeugt wurden, bei einem erneuten Aufruf zur Codegenerierung zu verwenden.

4.3.3. Phasen der Codegenerierung

In den folgenden Abschnitten wird näher auf die Implementierung der tatsächlichen Codegenerierung eingegangen. Alle beschriebenen Komponenten bilden die Basis für das mittlerweile kommerziell vertriebene Software-Werkzeug *emmtrix Code Generator* [48]. Das Werkzeug zur Umwandlung von MATLAB® zu C wurde in C++11 entwickelt und

nutzt dazu aktuelle Features aus der Boost Bibliothek [83]. Die einzelnen Phasen, die vom Tool nacheinander abgearbeitet werden sind in Abbildung 4.3 dargestellt.

Abbildung 4.3.: Einzelne Phasen der MATLAB® nach C Konvertierung

4.3.3.1. Parsen

In der ersten Phase werden MATLAB®-Skripte eingelesen und auf korrekte Syntax überprüft. Dabei wird auf die Kombination von Flex [84] und Bison [85] gesetzt. Die Arbeit nutzt dabei Konzepte, die bereits in [74] etabliert wurden.

Flex (*fast lexical analyzer generator*) ist ein Programm, mit dem so genannte Tokenizer erzeugt werden können. Diese werden verwendet, um reinen Text in logisch strukturierte Abschnitte (Tokens) zu unterteilen. In der Datei MatlabLexer.l werden die Zeichen bzw. -ketten beschrieben, die die logischen Strukturen von MATLAB® definieren. In Quellcode 4.4 ist ein Auszug aus der Datei dargestellt. Zeile 1 bis 4 beschreiben Schlüsselwörter, mit denen Kontrollstrukturen wie If-Blöcke, For-Schleifen oder Funktionen eingeleitet oder beendet werden. In Zeile 6 bis 9 werden mathematische Operatoren beschrieben. Flex wird nun verwendet, um für diese Beschreibung einen Tokenizer zu generieren, der den eingelesenen Text in der beschriebenen Granularität weiterreicht.

```
1 "function"  { BEGIN INITIAL; set_type(TokenType_Keyword); return(
      FUNCTION); }
2 "if"  { begin_of_statement(); BEGIN INITIAL; set_type(TokenType_Keyword
      ); return(IF); }
3 "for" { begin_of_statement(); BEGIN INITIAL; set_type(TokenType_Keyword
      ); return(FOR); }
4 "end" { BEGIN INITIAL; set_type(TokenType_Keyword); return(END); }
5
6 "&" { BEGIN INITIAL; set_type(TokenType_Operator); return('&'); }
7 "-" { BEGIN INITIAL; set_type(TokenType_Operator); return('-'); }
8 "+" { BEGIN INITIAL; set_type(TokenType_Operator); return('+'); }
9 "*" { BEGIN INITIAL; set_type(TokenType_Operator); return('*'); }
```

Quellcode 4.4: Auszug aus der Datei MatlabLexer.l

GNU Bison wird nun verwendet, um einen Parser für die MATLAB®-Grammatik zu generieren. Diese wird in der Datei MatlabParser.y definiert, ein Auszug ist in Quellcode 4.5 dargestellt. Sie bildet eine hierarchische Beschreibung der Sprache, auf deren obersten Ebene sich die „Translation Unit“ befindet. Diese besteht aus einer Liste an „Statements“, die beispielsweise Funktionen, For-Schleifen oder Zuweisungen darstellen können. Zusätzlich werden Zeichen zum Beenden von Ausdrücken, Zeilen und Funktionen

definiert. Die Datei ist in der erweiterten Backus-Naur-Form geschrieben, welche eine als ISO/IEC 14977:1996(E) standardisierte Metasprache zur Darstellung der Syntax von Programmiersprachen ist. Der mit Hilfe dieser Beschreibung generierte Parser liest die Tokens des Lexers ein und erzeugt eine interne Zwischendarstellung, die als Basis für alle weiteren Optimierungen verwendet wird.

4.3.3.2. Sparse conditional Konstantenpropagation

Die Konstantenpropagation ist eine wichtige Optimierungsphase, die großen Einfluss auf die Datentypen und die Verwendung von Variablen hat. Ziel ist es, alle Datentypen so zu wählen, dass möglichst wenig Speicher benötigt wird, aber keine Genauigkeit bei den Berechnungen verloren geht. Des Weiteren sollen alle unnötigen Ausdrücke im Programm entfernt werden, um den Quellcode möglichst effizient zu gestalten. Die entfernten Ausdrücke umfassen hierbei Schleifen, deren Ergebnisse statisch berechnet werden können, Funktionsaufrufe, „toter" Quellcode und nicht-verwendete Funktionsparameter. Wichtig für die Parallelisierung auf den späteren Ebenen ist, dass auf diese Weise unnötige Abhängigkeiten reduziert werden und das Potential für die Parallelisierung auf Task- und Daten-Ebene verbessert wird.

Bei der Konstantenpropagation werden für jeden Ausdruck die folgenden Informationen bereitgestellt:

1. Konstanter Wert: werden Variablen auf feste, konstante Werte gesetzt, so wird diese Information über alle Ausdrücke hinweg durchpropagiert. Sind alle Eingänge von einem Ausdruck bekannt, so kann häufig auch der Wert des Ausdrucks berechnet werden.
2. Matrix-Größe/-Form: Die Größe und Form einer Matrix-Variablen können durch verschiedene Ausdrücke definiert oder ermittelt werden. Im einfachsten Fall wird ein Array bereits mit einer bestimmten Dimensionierung erzeugt. Dies kann direkt über die Variable oder über einen Rückgabewert einer Funktion erfolgen. So erzeugt beispielsweise der Funktionsaufruf `zeros(4, 2)` eine 4x2-Matrix. Da in MATLAB® zudem dynamische Größenänderungen zur Laufzeit möglich sind, können die Informationen zur Matrix-Größe in späteren Teilen des Programms verändert werden. Wird beispielsweise auf Element (5,3) der vorher mit Nullen initialisierten Matrix zugegriffen, wird die intern propagierte Information ab diesem Ausdruck auf eine Größe von 5x3 aktualisiert.
3. Datentyp: Der Standard-Datentyp in MATLAB® ist Double mit 64 Bit Genauigkeit. Dieser wird verwendet, wenn nichts weiteres im Quellcode definiert wird. Es ist jedoch möglich, andere ganzzahlige Datentypen oder Fließkommas mit niedrigerer Genauigkeit explizit zu verwenden.
4. Wertebereich: Mathematische Operation, vor allem mit konstanten Werten, können den Wertebereich einer Variablen definieren. So können Additionen von positiven Zahlen immer nur weitere positive Zahlen erzeugen. Obere und untere Schranken können häufig durch die Konstantenpropagation ermittelt werden. Operationen wie die Multiplikation können aber dafür sorgen, dass der Wertebereich sehr schnell nur noch mit $\pm$ unendlich angegebenen werden kann.

```
1  %require "2.7"
2  %name-prefix "yymatlab_"
3
4  %start translation_unit
5  %%
6
7  expression_binop_component
8      : expression '+' expression { $$ = new my_smart_ptr<CExpression>(
           MatrixOperators::makeop_comp_binary(POS(@$), ScalarOperators::
           AddSat, *$1, *$3)); delete $1; delete $3; }
9      | expression '-' expression { $$ = new my_smart_ptr<CExpression>(
           MatrixOperators::makeop_comp_binary(POS(@$), ScalarOperators::
           SubSat, *$1, *$3)); delete $1; delete $3; }
10     ;
11
12 function_statement
13     : FUNCTION function_declare eostmt statement_list END eostmt  { (*
           $2)->Function->Body = *$statement_list; (*$2)->Function->Pos =
            POS(@$); $$ = $2; delete $statement_list; }
14     ;
15
16 statement
17     : empty_statement
18     | expression_statement
19     | for_statement
20     | function_statement { $$ = new my_smart_ptr<CStatement>(*$1);
           delete $1; }
21     ;
22
23 statement_list
24     : { $$ = new my_smart_ptr<CBlock>(make_node<CBlock>(POS(@$))); }
25     | statement_list statement { if ($2) { (*$1)->Statements.push_back
           (*$2); delete $2; } (*$1)->Pos = POS(@$); }
26     ;
27
28 translation_unit
29     : statement_list { param.func_container->getDefault()->
           addStatements(*$1); param.func_container->getDefault()->setPos
           (POS(@$)); delete $1; }
30     ;
31
32 %%
33 #include <stdio.h>
34
35 extern char yytext[];
36 extern int column;
```

Quellcode 4.5: Auszug aus der MATLAB®-Grammatik für GNU Bison

5. Teilbarkeit /Ergebnis der Modulo-Operation: wenn der Wertebereich und die Werte, die eine Variable annehmen kann, hinreichend genau bestimmt werden können, so kann die Teilbarkeit durch die Modulo-Operation bestimmt werden: Ergibt die Modulo-Operation mit 0 keinen Rest, so kann ein ganzzahliger Integer zur Repräsentation der Variablen verwendet werden.

Die spärlich bedingte Konstantenpropagation (engl. *sparse conditional constant propagatio*n, SCCP) ist ein bekannter Algorithmus, der erstmals von Wegman und Zadeck [86] vorgestellt wurde. Er arbeitet auf der SSA-Form der Zwischendarstellung und betrachtet alle Abhängigkeiten und Ausdrücke. Der Algorithmus konzentriert sich auf ϕ-Funktionen, die in der SSA-Darstellung für den Zusammenfluss von Kontrollflüssen stehen. Jede ϕ-Funktion hat einen Eingang für jede Kontrollkante, von der der Kontrollfluss kommen könnte. Das Ergebnis der ϕ-Funktion wird laut Definition von der jeweils genommenen Kante bestimmt. Der SCC versucht nun bei Verzweigungen im Kontrollfluss, die beispielsweise durch If-Blöcke, Schleifen oder direkte Sprünge im Quellcode erzeugt werden, zu bestimmen, welche Abzweigung genommen wurde. Hängt die Bedingung nur von konstanten Werten ab, so kann auch der Wert der zugehörigen ϕ-Funktion bestimmt werden und unnötiger Code entfernt oder existierender Code vereinfacht werden. Diese Eliminierung von totem Code, der niemals erreicht werden kann, reduziert die Problemgröße für alle späteren Phasen und sollte deshalb immer angewendet werden. Zu beachten ist, dass eine aggressive Eliminierung alle Ausdrücke im Programm entfernt, die keine Seiteneffekte haben. Das bedeutet, dass Ergebnisse, die im Programm berechnet werden, jedoch nicht über beispielsweise `printf` ausgegeben werden, ebenfalls entfernt werden. Alle Codeteile, die also auf jeden Fall Teil des Programms und der späteren Parallelisierungsschritte sein sollen, können zusätzlich über Pragmas markiert werden, um das Entfernen zu verhindern.

Diese Optimierung wird hier aber auch verwendet, um die Größe der einzelnen Matrix-Variablen zu bestimmen. Damit hat die Optimierung zudem großen Einfluss auf den Speicherbedarf des generierten C-Codes. Dabei findet die Größenbestimmung in zwei Schritten statt. Beim reinen SCC wird immer von der maximalen Größe ausgegangen, die notwendig für die Darstellung sämtlicher Variablen ist. Nachgelagert ist eine weitere Phase, die die Größen anhand der finalen Operationen bestimmt und auf Konsistenz überprüft.

Der Entscheidungsmechanismus sorgt dafür, dass die Entscheidungsvariable von außen auf einen bestimmten Wert gesetzt werden kann. Dies kann durch eine zusätzliche Datei oder per Aufruf auf der Kommandozeile erfolgen. Das Setzen eines Werts sorgt dafür, dass die Konstantenpropagation mit dem gesetzten Wert durchgeführt wird und somit Teile des Codes unnötig werden können oder Parameter von Optimierungsschritten gesetzt werden können.

Die Informationen, die in dieser Phase ermittelt werden, können in allen darauffolgenden weiterverwendet und aktualisiert werden.

4.3.3.3. Matrixoptimierung

Diese Optimierungsphase zielt darauf ab, die Ausführungsgeschwindigkeit des Codes zu optimieren. Sie arbeitet noch auf einer internen Darstellung, die an die ursprüngliche array-basierte Sprache angelehnt ist. Dazu werden die folgenden Schritte ausgeführt:

Anweisungsfusion
Das Fusionieren von Anweisungen hat zum Ziel, die Anzahl an Schleifen zu reduzieren und temporäre Variablen einzusparen. Sie kommt beispielsweise in Fällen wie in Quellcode 4.6 dargestellt ist zum Einsatz. Ohne Fusion würde C-Code wie in Quellcode 4.7 generiert werden. Die Variable `roi_data` wird in einer zweifach verschachtelten For-Schleife mit Daten aus `image_data` initialisiert (Zeile 2-6) und anschließend in einer anderen zweifach verschachtelten For-Schleife gelesen (Zeile 13).

```
1 roi = image(101:200, 101:200);
2 s =  sum(sum(roi);
```

Quellcode 4.6: MATLAB®-Beispiel für Anweisungs-Fusion

```
1  double roi_data[100][100];
2  for (i4 = 0; i4 < 100; i4++) {
3    for (i3 = 0; i3 < 100; i3++) {
4      roi_data[i4][i3] = image_data[i4 + 100][i3 + 100];
5    }
6  }
7
8  tmp1 = 0.0;
9
10 for (i4 = 0; i4 < 100; i4++) {
11   tmp2 = 0.0;
12   for (i5 = 0; i5 < 100; i5++) {
13     tmp2 += roi_data[i4][i5];
14   }
15   tmp1 += tmp2;
16 }
17
18 s_data = tmp1
```

Quellcode 4.7: C-Code ohne Anweisungsfusion

Mit der Anweisungsfusion kann die Variable `roi_data` eingespart werden, indem anstatt die Variable zu lesen direkt die eigentliche Zuweisung verwendet wird. Das Ergebnis der Fusion ist in Quellcode 4.8 dargestellt. In Zeile 6 wird nicht wie in Quellcode 4.7 auf die

Variable `roi_data`, sondern direkt auf die entsprechenden Elemente von `image_data` zugegriffen. Auf diese Weise kann sowohl die Variable `roi_data` als auch die erste zweifach verschachtelte For-Schleife eingespart werden. Dies reduziert den Speicherbedarf und die Laufzeit durch Vermeidung von unnötigen Kopieroperationen.

```
1  tmp1 = 0.0;
2
3  for (i4 = 0; i4 < 100; i4++) {
4    tmp2 = 0.0;
5    for (i5 = 0; i5 < 100; i5++) {
6      tmp2 += image_data[i4 + 100][i5 + 100];
7    }
8    tmp1 += tmp2;
9  }
10
11 s_data = tmp1;
```

Quellcode 4.8: C-Code mit Anweisungsfusion

Entfernen von totem Code

Zur Detektion von totem Code wird ein Algorithmus basierend auf dem „Mark-and-Sweep"-Algorithmus eingesetzt. Dieser besteht aus zwei Phasen: in der ersten werden alle Codeteile, die bei einer Tiefensuche besucht werden, markiert. Alle nicht markierten Teile werden in der zweiten Phase aus dem Programm entfernt.

Integer-Konvertierung

Ein Fließkommadatentyp eines Ausdrucks oder einer Variablen wird automatisch in einen Integer-Datentyp umgewandelt, wenn:

1. Der Wert immer durch 1 teilbar ist, d. h. die Modulo-Operation mit 1 eine 0 zurück liefert.
2. Der Wertebereich der Zahl in den Integer-Zahlenbereich passt.

Diese Umwandlung reduziert dabei nicht nur den Speicherbedarf von Variablen, sondern verbessert gerade auf eingebetteten Systemen zudem die Laufzeit. Operationen mit ganzen Zahlen erfordern weniger Ressourcen als Operationen mit Fließkommazahlen, für welche gerade eingebettete Prozessoren oft keine eigenen Ausführungseinheiten bereitstellen.

Zusätzlich werden in diesem Schritt die Anzahl an expliziten Casts im Code optimiert. Dazu werden zunächst unnötige Casts entfernt, die keinen Einfluss auf die Semantik haben. Diese können aufgrund der Umwandlung der Datentypen noch Teil des Programms sein. Anschließend werden notwendige Casts hinzugefügt, um implizite Casts im Code zu vermeiden.

Operationsoptimierung
Ziel dieser Optimierung ist es, die Operationen in der Matrixdarstellung zu optimieren. Dazu wird zunächst eine Konstantenfaltung durchgeführt, die Formeln bereits bei der Kompilierung ausrechnet, falls sie nur aus konstanten Werten bestehen. Durch die vorgelagerte Konstantenpropagation können auf diese Weise unnötige Anweisungen entfernt werden, die dann nicht mehr auf späteren Ebenen betrachtet werden müssen. Anschließend wird eine Peephole-Optimierung auf MATLAB®-Ebene angewendet. Hierbei werden Operationen in einem Guckloch oder Fenster (engl. *peephole optimization*) auf logischer Ebene betrachtet. Dabei können die folgenden Fälle optimiert werden:

- Nullsequenzen: Unnötige Operationen entfernen
- Kombination: Anstatt mehrere einzelne Operationen zu haben, wird stattdessen eine gleichwertige verwendet.
- Algebraische Gesetze: Anwendung von grundlegenden Regeln der Algebra (Assoziativgesetz, Distributivgesetz und Kommutativgesetz), um Ausdrücke zu vereinfachen.

Verschieben von Initialisierungscode
Werden bei der Analyse Code-Teile erkannt, die bei der Ausführung exakt ein Mal ausgeführt werden müssen und somit Initialisierungscode darstellt, so wird dieser Code in dieser Phase an den Anfang des Programms verschoben. Dies optimiert die Laufzeit des Programms für Fälle, in denen der Hauptteil in einer Schleife ausgeführt wird. Ein gutes Beispiel hierfür ist die Berechnung der Twiddle-Faktoren, welche bei der Fast Fourier Transformation (FFT) benötigt werden. Die Twiddle-Faktoren können nach der Initialisierung als konstante Werte angesehen werden, auf die während der eigentlichen FFT-Berechnung nur lesend zugegriffen werden muss.

4.3.3.4. Loopify

Die Loopify-Phase wandelt die interne Matrix-Repräsentation von Ausdrücken in die interne C-Repräsentation um. Der Name Loopify soll zeigen, dass Matrix-Anweisungen in der Regel als verschachtelte Schleifen (loops) im C-Code dargestellt werden. Während der Transformation werden die folgenden Aufgaben durchgeführt:

1. Matrix-Variablen werden durch C Variablen ersetzt.
2. Matrix-Anweisungen werden durch C-Konstrukte ersetzt, beispielsweise durch verschachtelte Schleifen.
3. Weitere Optimierungen werden angewendet, um die Anzahl an Schleifen noch weiter zu optimieren. Hierbei ist es möglich, Optimierungen auf Code-Ebene (siehe Abschnitt 5.3) durchzuführen. An dieser Stelle können Loop-Unroll und Loop-Fission angewendet werden, um den C-Code bereits besser für die spätere Anpassung vorzubereiten.

4. Einfügen von Verweisen auf den ursprünglichen MATLAB®-Code als Kommentare im C-Code. Auf diese Weise kann für jede Zeile im C-Code nachverfolgt werden, aus welchem ursprünglichen Code sie generiert wurde.

Die Optimierung von Schleifen und somit der Darstellung des ursprünglichen Algorithmus als C-Code ist eine wichtige Aufgabe, um das Optimierungspotential auf der Task-Ebene möglichst hoch zu halten.

4.3.3.5. C-Optimierung

Nach der Loopify-Phase befindet sich die interne Repräsentation in der C-Darstellung, in welche sich weitere Optimierungen vor der tatsächlichen Ausgabe durchführen lassen.

Entfernen unnötiger Dimensionen von Variablen
Bei der Generierung von Datentypen können Dimensionen mit einzelnen Elementen entstehen, beispielsweise A_data[1][10]. Diese nutzlosen Dimensionen können entfernt werden, so dass die Variable zu A_data[10] vereinfacht werden kann.

Entfernen nicht-verwendeter Variablen
Ungenutzte Variablen können aus verschiedenen Gründen im Code sein:

- Sie wurden bereits im Eingangscode definiert und auch dort schon nicht genutzt.
- Durch Konstantenpropagation werden Teile des Codes, bzw. Verwendung von Variablen unnötig, die im ursprünglichen Code noch benötigt wurden.
- Durch Optimierungen an der Verwendung von Schleifen kann es passieren, dass Zählvariablen von Schleifen oder auch temporäre Werte nicht mehr benötigt werden, da sie bei der Fusion von Schleifen keine Verwendung mehr haben.

Alle diese Variablen können entfernt werden, da sie im generierten C-Code nicht mehr eingesetzt werden und nur unnötig die Codegröße aufblähen.

Entfernen von Dummy-Befehlen
Durch die systematische Generierung der Strukturen des späteren C-Codes können Konstrukte entstehen, die keinen Effekt auf die Ausführung des Codes haben. Zu den häufigeren gehören Schleifen mit genau einer Iteration oder Zuweisungen von Variablen auf sich selbst (z.B. $a = a$). Diese Anweisungen können ohne weitere Auswirkungen auf den Code entfernt werden.

4.3.3.6. C-Ausgabe

Im letzten Schritt werden die Ausgabedateien generiert:

- C-Quelldateien: Die gesamte Funktionalität der ursprünglichen MATLAB®-Anwendung wird in einer C-Datei ausgegeben. Diese umfasst sämtliche Funktionen, die

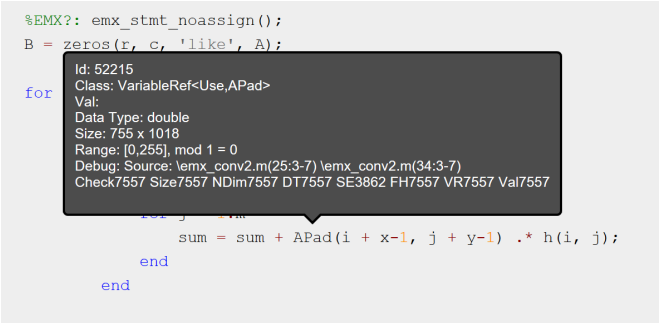


Abbildung 4.4.: Mouseover über eine Variable

im ursprünglichen Skript verwendet wurden: sowohl Funktionen aus Dateien in der Form `funktions_name.m`, die nur die Funktion mit dem entsprechenden Namen implementiert und die zusätzlich zur Hauptdatei entwickelt wurden als auch Standardfunktionen, die Teil der Bibliothek des Codegenerierungswerkzeuges ist. Zusätzlich werden in *.h- und *.c-Dateien Hilfsfunktionen ausgegeben, die im generierten C-Code eingesetzt werden. Diese Funktionen umfassen dabei die formatierte Ausgabe von Matrizen auf der Konsole, Generierung von Zufallszahlen, Handhabung von Dateien, Messen von Zeiten und andere, mathematische Funktionen. Nur die tatsächlich verwendeten Dateien werden in das Ausgangsverzeichnis kopiert.

- Bericht über die Codegenerierung: alle Analyseergebnisse, die bei der Konvertierung des Codes gewonnen wurden, werden in einem HTML-Bericht zusammengefasst. Dieser stellt das MATLAB®-Programm und den generierten C-Code gegenüber und kann Informationen zu Variablen und Funktionen als Mouseover oder über Farben darstellen.

  In Abbildung 4.4 werden die Informationen dargestellt, die bei Mouseover über eine Variable angezeigt werden. Über `Val` wird der Wert der Variablen ausgegeben, was jedoch nur möglich ist, wenn die Variable an der aktuellen Stelle einen berechenbaren, konstanten Wert hat. `Data Type` gibt den im MATLAB®-Skript verwendeten Typ an. Die Größen der Dimensionen werden im Feld `Size` angegeben. Und schließlich wird im Feld `Range` der Wertebereich der Variablen dargestellt. Zusätzlich zu den ermittelten Grenzwerten (sofern dies möglich ist) wird außerdem das Ergebnis der Modulo-Operation mit 1 oder einem ganzzahligen Vielfachen davon angegeben, wenn der Rest 0 beträgt. Diese Information wird benötigt, um den C-Datentyp effizienter bestimmen zu können. So gibt die Modulo-Operation mit 1 und Rest 0 zusammen mit dem Wertebereich der Variablen an, ob ein ganzzahliger Integer-Wert zur Darstellung der Variablen eingesetzt werden kann.

  In Abbildung 4.5 ist dargestellt, wie die Genauigkeit der Analyseergebnisse farblich dargestellt werden. Für das hier gezeigte Beispiel des Wertebereichs bedeutet grün, dass der Wertebereich der Anweisung bestimmt werden konnte. Gelb zeigt auf, dass nur ein Teil berechnet werden konnte und beispielsweise nur der maximale Wert bestimmt werden konnte, das Minimum aber als negativ unendlich angenommen

```
if (issame)
    APad = emx_padding(A, left, right, top, bottom, padtype);
else
    top = n-1;
    bottom = n;
    left = m-1;
    right = m;
    r = r + n - 1;
    c = c + m - 1;

    APad = emx_padding(A, left, right, top, bottom, padtype);
end

%EMX?: emx_stmt_noassign();
B = zeros(r, c, 'like', A);

for x = 1:r
    for y = 1:c
        sum = cast(0, 'like', APad);

        for i = 1:n
            for j = 1:m
                sum = sum + APad(i + x-1, j + y-1) .* h(i, j);
            end
        end

        B(x, y) = sum;
    end
end
```

Abbildung 4.5.: Farbliche Darstellung der Genauigkeit der berechneten Wertebereiche

werden muss. Rot bedeutet schließlich, dass sowohl Minimum als auch Maximum als plus oder minus unendlich angenommen werden müssen. Dies kann sich durch die eingesetzten mathematischen Operationen oder auch durch eine Bedingung im Code ergeben.

- Mapping Datei: in einer mit der Endung .map generierten Datei wird im JSON-Format die Beziehungen zwischen Anweisungen im C-Code und im MATLAB®-Skript dargestellt. Für jede Anweisung im C-Code gibt es eine Referenz auf die ursprüngliche Stelle im MATLAB®-Skript.
- Makefile und Abhängigkeits-Datei: optional können zusätzlich ein Makefile sowie eine dazugehörige Abhängigkeitsdatei mit der Endung .d generiert werden, mit denen das generierte Programm über GNU Make[87] im gcc-Compiler[88] kompiliert werden kann.

4.3.4. Datentyp-Inferenz

MATLAB® und Scilab sind Skriptsprachen mit dynamischer Typisierung. Das bedeutet, dass Typen und Operatoren zur Laufzeit überprüft und nicht statisch in der Sprache festgelegt werden. Für eine effiziente Ausführung des generierten C-Codes ist es notwendig, dass Datentypen und Operatoren statisch und bei der Kompilierung bekannt sind. Dynamisches Anpassen zur Laufzeit in C ist nur über Erstellen neuer Variablen mit großem Overhead für Kopieroperationen möglich und ist für Algorithmen mit klar definierten Ein- und Ausgangsdaten überflüssig. Deswegen kommt bei der Codegenerierung eine umfassende Typinferenz-Bestimmung zum Einsatz, die den Datentyp und die maximale Größe jeder Variablen bestimmt. Sollte dies nicht möglich sein, so muss der Typ im Eingangsskript spezifiziert werden. Ansonsten wird die Codegenerierung mit einer Fehlermeldung beendet.

Der Typ einer Variablen kann in die folgenden Informationen unterteilt werden:

1. Statischer Variablenstatus: Dieser Status bestimmt, ob eine Variable global oder lokal ist. Er wird als Markierung während der Codegenerierung propagiert und muss am Ende eindeutig sein. In Quellcode 4.9 ist ein Beispiel für einen mehrdeutigen Zustand dargestellt. Die Variable `a` ist zunächst nur als lokal markiert, kann aber abhängig von `x` als global markiert werden. Damit kann der Status nach dem If-Block nicht mehr eindeutig bestimmt werden und eine Fehlermeldung wird generiert.

```
1 a = 1; % Local (default at beginning of a function)
2 if (x)
3   a = 2; % Local
4   global a;
5   a = 3; % Global
6 else
7   a = 4; % Local
8 end
9 a = 5; % Ambiguous
```

Quellcode 4.9: Beispiel für einen mehrdeutigen Variablenstatus

2. Datentyp: Die Datentypinferenz bestimmt für jeden Ausdruck den tatsächlichen Datentyp, z.B. struct, double oder int32. Dabei wird zwischen zwei Darstellungen unterschieden: Matrixdatentypen und C-Datentypen. Sollte kein expliziter Datentyp im Skript angegeben sein, so wird standardmäßig jeder Ausdruck erst als Matrixdatentyp double behandelt. Jede Anweisung im Skript kann Einfluss auf den Datentyp haben. Dabei ist zu beachten, dass Integer-Datentyp eine höhere Priorität haben als Fließkommazahlen. Wird beispielsweise ein `int32(15)` auf eine Double-Variable addiert, so ist das Ergebnis vom Datentyp int32. In Tabelle 4.1 sind sämtliche Matrixdatentypen aufgelistet.

   Der C-Datentyp kann sich vom Matrixdatentyp unterscheiden, um die Effizienz des generierten Codes zu steigern. Dazu werden die bereits ermittelten Werte für den

Matrixdatentyp	Beschreibung
double	Fließkommazahl mit doppelter Genauigkeit (standard)
single	Fließkommazahl mit einfacher Genauigkeit
int8, int16, int32, int64	Integerzahl mit Vorzeichen
uint8, uint16, uint32, uint64	Integerzahl ohne Vorzeichen
logical	Boolescher Ausdruck
struct	Struktur

Tabelle 4.1.: Auflistung aller Matrixdatentypen

Wertebereich und die Ergebnisse der Modulo-Operation herangezogen, um bevorzugt Integerdatentypen zu erzeugen.

3. Matrixgröße: bei der Größeninferenz wird die Größe einer Variablen zu jedem Zeitpunkt bestimmt. Dabei besteht die Größe eines Ausdrucks aus der Anzahl der Dimensionen sowie deren minimaler und maximaler Größe. Die Größe einer Matrix kann in der Regel über die verwendeten Ausdrücke bestimmt werden. Einige Beispiele sind in Tabelle 4.2 dargestellt.

Ausdruck	Matrixgröße
1	1 x 1
`zeros(3,3)`	3 x 3
`ones(4,3,2)`	4 x 3 x 2
$[1,2,3]$	1 x 3
$[1;2;3]$	3 x 1
`zeros(n,4)`	$[1,3]$ x 4 wenn für `n` gilt: `round(rand()*3)`

Tabelle 4.2.: Größeninferenz aus Ausdrücken

Im Allgemeinen wird zwischen fixierten und dynamischen Matrixgrößen unterschieden. Eine dynamische Matrixgröße (z.B. $[1,3]$ x 4) hat mindestens eine Dimension mit minimaler und maximaler Größe. Kann die maximale Größe einer Dimension nicht berechnet werden, so wird eine Fehlermeldung ausgegeben und die Codegenerierung beendet. Abhilfe schafft hier, die Variable im Quellcode mit fixierter Größe zu markieren. Es wird dann angenommen, dass sich die Größe der Variable zur Laufzeit nicht mehr ändert und nur der Wert bei der Initialisierung ausschlaggebend ist. Matrizen mit dynamischer Größe werden im C-Code über eine Variable für die Daten mit der Endung `_data` und eine Variable mit der Endung `_size` für die aktuelle Größe jeder Dimension repräsentiert. Ein Beispiel: eine Variable wird über $A\_data[10][10]$ und $A\_size[2]$ definiert. Die maximale Größe der Variablen ist 10 x 10 und eine Zuweisung $A\_size[0] = 5$ bedeutet, dass von der ersten Dimension von A_data aktuell nur 5 Elemente in Benutzung sind.

Der Typ und die Größe einer Variablen kann nicht immer statisch bestimmt werden, da sie beispielsweise von Eingangswerten abhängen können. Die folgenden zusätzlichen Funktionen wurden implementiert, um der Inferenzanalyse zusätzliche Hinweise oder Einschränkungen mitzuteilen.

Funktion	Beschreibung
`cgen_fixedsize(var)`	Mit dieser Funktion gekennzeichnete Variablen verändern durch Arrayzugriffe nicht ihre Größe. Bei Variablen ohne diese Markierung werden Zugriffe zur Bestimmung der maximalen Größe verwendet.
`cgen_size`	Mit dieser Funktion können Größen einzelner Dimensionen auf feste Größen fixiert werden. Sollte es bei der Analyse zu Inkompatibilitäten mit der spezifizierten und der benötigten Größen kommen, wird eine Fehlermeldung generiert.
`cgen_size_noscalar`	Zusätzlich zur Angabe der Größen in `cgen_size` werden diese Variablen niemals als skalare Werte betrachtet.
`cgen_dynamic`	Die Größe von mit dieser Funktion markierten Variablen wird bei der Konvertierung bestimmt. Dies ist das Standardverhalten und die Funktion kann verwendet werden, um nach den vorigen Einschränkungen zur Bestimmung der Variablengröße wieder das normale Verhalten wiederherzustellen.
`assert`	Mittels dieser Anweisungen können Angaben über den erwarteten Datentyp gemacht werden. So kann beispielsweise der erwartete Datentyp oder die Größe angegeben werden. Kommt die automatische Analyse auf andere Werte, so bricht die Konvertierung mit einer Fehlermeldung ab.

Tabelle 4.3.: Funktionen zur Größen- und Datentyp-Inferenz

4.3.5. Fallbeispiel: Kalman-Filter

Um die Relevanz der Codegenerierung zu evaluieren, wurde sie mit einer Anwendung aus der Industrie getestet. Im Rahmen einer studentischen Arbeit wurde bei der Firma Bosch sowohl die hier beschriebene Codegenerierung, im Folgenden als „Code Generator“ bezeichnet, als auch der MATLAB® Coder von Mathworks eingesetzt, um die Anwendung von MATLAB® nach C zu konvertieren.

Als Testanwendung wurde der Kalman-Filter [89] für die Datenfusion verwendet. Der Algorithmus ist in zwei Phasen unterteilt: im ersten Schritt, der Aktualisierung der Zeit („*Predict*“), wird der Zustand der aktuellen Variablen inklusive ihrer Unsicherheiten abgeschätzt. Im nächsten Schritt, der Aktualisierung der Messung („*Correct*“), werden die aktuellen Messwerte verwendet, um die Zustandsvariablen und die Kovarianz zu korrigieren. Durch diesen rekursiven Ansatz können Fehler über die Zeit minimiert werden oder bei echtzeitkritischen Anwendungen nur der jeweils letzte Zustand mit einbezogen werden.

Unter Sensordatenfusion [90] versteht man die Verknüpfung von Daten aus verschiedenen Sensorquellen. Daraus können genauere Aussagen über das System gemacht werden, welches für die Bildverarbeitung oder die automatische Navigation ausgenutzt werden kann.

4.3.5.1. Hintergrund Kalman-Filter

Die folgenden Formeln wurden [91] entnommen und bilden die Basis für den diskreten Kalman-Filter.

Die Aktualisierung der Zeit setzt sich aus der Abschätzung des Zustandsvektors x (Formel 4.1) und des Kovarianzvektors P (Formel 4.2) zusammen. $\bar{\hat{x}_k}$ wird als *a priori* Zustandsschätzung mit dem Wissen vor Schritt k definiert, $\hat{x}_k$ als *a posteriori* Zustandsschätzung nach Messschritt k.

$$\bar{\hat{x}_k} = A\hat{x}_{k-1} + Bu_{k-1} \tag{4.1}$$

$$\bar{P_k} = AP_{k-1}A^T + Q \tag{4.2}$$

Dabei werden die Werte des aktuellen Zustands k aus dem vorherigen Zustand $k-1$ berechnet. Matrix A entspricht hier dem Zustand des letzten Zeitschritts, Marix B steht für den optionalen Kontrolleingang, der mit u gewichtet wird und Matrix Q repräsentiert das Prozessrauschen.

Die Aktualisierung der Messwerte setzt sich aus der Kalman-Matrix (auch „Kalman gain“ genannt) K (Formel 4.3), dem Zustandsvektor x (Formel 4.4) und dem Kovarianzvektor P (Formel 4.5) zusammen.

$$K_k = \bar{P_k} H^T \left(H \bar{P_k} H^T + R \right)^{-1} \tag{4.3}$$

$$\hat{x}_k = \hat{\bar{x}_k} + K_k \left(z_k - H \hat{\bar{x}_k} \right) \tag{4.4}$$

$$P_k = \left(I - K_k H \right) \bar{P_k} \tag{4.5}$$

Matrix H stellt den Zustand der aktuellen Messung z_k dar, R die Messunsicherheiten und I steht für die Einheitsmatrix mit den entsprechenden Dimensionen.

Anhand der Grundformeln lassen sich die nötigen mathematischen Operationen ableiten, die für die Beschreibung des Algorithmus verwendet werden. Da hauptsächlich mit Matrizen gerechnet wird, sind dies die Matrix-Multiplikation, die Matrix-Addition und die Matrix-Inversion. Die Größen der Matrizen können sich abhängig von den Eingangsdaten verändern und sind deshalb nicht in jedem Fall konstant.

4.3.5.2. Beschreibung der Arbeit

Die folgenden Ergebnisse wurden im Rahmen einer Masterarbeit [92] zusammen mit der Firma Bosch erarbeitet. Aus diesem Grund liegt der Quellcode der Anwendung nicht vor und es kann hier nicht näher darauf eingegangen werden.

In dieser Arbeit wurde die hier vorgestellte Generierung von sequentiellem C-Code aus MATLAB® heraus mit dem *Embedded Coder* [63] direkt von Mathworks verglichen. Alle Zeiten wurden durch Ausführung auf einem Renesas 32-Bit RISC-Mikroprozessor mit 2 Kernen ermittelt. Jeder Kern hat eine Fließkommaeinheit mit doppelter Präzision und 16 KiB Instruktions-Cache. Außerdem verfügt das Zielsystem über 1344 KiB an Hauptspeicher. Die Anwendung wurde im Rahmen dieser Arbeit nicht parallelisiert, sondern nur auf einem einzelnen Kern des Prozessors ausgeführt.

Im Folgenden werden die Unterschiede hinsichtlich der verschiedenen Matrix-Operationen, sowie die Handhabung bei der Übersetzung des Kalman-Filter näher erläutert.

4.3.5.3. Matrix-Operationen

Der größte Unterschied zwischen dem *Embedded Coder* und der hier entwickelten Codegenerierung liegt in der Indizierung der einzelnen Elemente einer Matrix. Der *Embedded Coder* bildet alle Matrizen auf Vektoren mit einer Dimension ab und verlagert die Berechnung des passenden Index in den tatsächlichen Zugriff. Die hier entwickelte Codegenerierung verwendet für jede Dimension eine eigene Variable, welche mit den umschließenden For-Schleifen verwaltet werden. Beispielhaft soll hier eine Matrix-Multiplikation, die in MATLAB® als $C = A * B$ geschrieben werden kann, verglichen werden.

In Quellcode 4.10 wird der vom Embedded Coder generierte C-Code dargestellt. Die bereits erwähnte Berechnung der Indizes ist in den Zeilen 3 und 5 zu sehen. Aus den Variablen *i*, *i_0* und *i_1* wird der tatsächliche Index für den Zugriff auf die entsprechende Matrix berechnet.

```
1  for (i = 0; i < 15; i++) {
2    for (i_0 = 0; i_0 < 15; i_0++) {
3      matrix_mul_15_Y.Out1[i + 15 * i_0] = 0.0;
4      for (i_1 = 0; i_1 < 15; i_1++) {
5        matrix_mul_15_Y.Out1[i + 15 * i_0] += matrix_mul_15_P.A[15 * i_1
             + i] *  matrix_mul_15_P.B[15 * i_0 + i_1];
6      }
7    }
8  }
```

Quellcode 4.10: Matrix-Multiplikation generiert vom Embedded Coder

Als Vergleich ist der generierte C-Code des hier vorgestellten Konvertierers in Quellcode 4.11 dargestellt. Wie zu sehen ist, erfolgen alle Array-Zugriffe in den Zeilen 3 und 5 über die Zählvariablen der For-Schleifen und müssen nicht noch weiter verrechnet werden.

```
1  for (i6 = 0; i6 < 15; i6++) {
2    for (i5 = 0; i5 < 15; i5++) {
3      C_data[i6][i5] = 0.0;
4      for (i7 = 0; i7 < 15; i7++) {
5        C_data[i6][i5] += A_data[i7][i5] * B_data[i6][i7];
6      }
7    }
8  }
```

Quellcode 4.11: Matrix-Multiplikation generiert vom Matrix-Frontend

Die unterschiedliche Darstellung führt nicht nur zu einer besseren Lesbarkeit, da der Code kompakter dargestellt werden kann, sondern auch zu Leistungsunterschieden bei der tatsächlichen Ausführung auf der Hardware. Die Werte für Laufzeit und Speicherverbrauch sind in der Tabelle 4.4 dargestellt. Sie wurden mit der Compiler-Optimierung O2 ermittelt. Die zusätzliche Berechnung der Indizes benötigt in diesem Beispiel 21 % mehr Zeit, hat aber keinen Einfluss auf den Speicherverbrauch.

	Laufzeit [μs]	Speicherverbrauch [Byte]
Embedded Coder	517,15	5404
Code Generator	428,79	5404

Tabelle 4.4.: Laufzeit und Speicherverbrauch der Matrix-Multiplikation

Der Overhead der For-Schleifen durch die Berechnung und den Vergleich der Zählvariablen kann durch (partielles) Entrollen der Schleifen optimiert werden. Dabei wird die Anzahl der Schleifendurchläufe verringert, indem die Schleifenvariable bei jedem Durchlauf in größeren Schritten inkrementiert wird und Anweisungen aus ursprünglich unterschiedlichen Durchläufen hintereinander ausgeführt werden. Wird eine Schleife komplett entrollt, entfällt der komplette Overhead durch die Schleife und alle Anweisungen

werden hintereinander abgearbeitet. Das Entrollen ist dabei aber ein Kompromiss aus erreichter Performanz und dem dafür benötigten Speicher. Beide Codegenerierer unterstützen die Option, wobei sich die Funktionsweise leicht unterscheidet. Beim *Embedded Coder* wird sie global bis zu einem einstellbaren Schwellwert an Schleifendurchläufen automatisch auf die innerste Schleife angewendet, wohingegen der Code Generator Pragmas im Quellcode unterstützt, um nur gezielt einzelne Schleifen zu entrollen.

Der Vergleich der Laufzeit mit und ohne Entrollen der Schleifen kann Tabelle 4.5 entnommen werden. Die Laufzeit des erzeugten Codes der beiden Generierungstools mit entrollten Schleifen ist nahezu gleich, da der Overhead durch die aufwendigere Berechnung der Indizes beim *Embedded Coder* entfällt. Leider wurden keine Tests hinsichtlich des veränderten Speicherverbrauchs durchgeführt, so dass hier kein Vergleich erfolgen kann.

	Laufzeit ohne entrollen [μs]	Laufzeit mit entrollen [μs]
Embedded Coder	517	282
Code Generator	429	279

Tabelle 4.5.: Schleifen entrollen bei der Matrix-Multiplikation

Schließlich wurde noch der erzeugte Code für die Matrix-Inversion verglichen. Die ermittelten Werte für die Laufzeit sowie den Speicherbedarf sind in Tabelle 4.6 dargestellt. Der *Embedded Coder* erzeugt an dieser Stelle einen optimierten C-Code, bei der Optimierungen direkt auf der Algorithmus-Ebene integriert wurden, welche der Code Generator nicht nutzt. Auf diese Weise ist der erzeugte Code des *Embedded Coder* um 47 % schneller, obwohl auch hier die zusätzlichen Berechnungen der Indizes hinzukommen. Die Optimierungen können nach weiterer Evaluation ebenfalls im Code Generator implementiert werden. Der Speicherbedarf der beiden Lösungen ist in der Summe von RAM und Stack in etwa gleich. Die unterschiedliche Verteilung ist Einstellungssache: Variablen können beim Code Generator abhängig von der Größe entweder im RAM oder auf dem Stack angelegt werden. Diese sollten abhängig vom Bedarf und der Verfügbarkeit auf der Hardware entsprechend gewählt werden.

	Laufzeit [μs]	RAM-Bedarf [Byte]	Stack-Bedarf [Byte]
Embedded Coder	603,58	3.640	1.878
Code Generator	888.01	5.404	12

Tabelle 4.6.: Laufzeit und Speicherverbrauch der Matrix-Inversion

4.3.5.4. Kompletter Kalman-Filter

Nach Betrachtung der einzelnen Operationen, aus denen sich der Kalman-Filter zusammensetzt, wurde im nächsten Schritt der ganze Algorithmus übersetzt. In Tabelle 4.7 ist ein Vergleich ohne das Entrollen der Schleifen dargestellt. Es zeigt sich, dass die Optimierungen über den ganzen Algorithmus hinweg noch stärker greifen, als nur allein bei der Matrix-Inversion. Der erzeugte Code aus dem Matrix-Frontend benötigt 29 % länger bei der Ausführung als der Code des Embedded Coder. Bei der Speichernutzung wurden

leider nicht alle Optionen des Matrix-Frontends ausgenutzt, so dass der Bedarf an Hauptspeicher mehr als doppelt so groß ist. Große Unterschiede gibt es noch bei der Anzahl an generierten C-Codezeilen. Der Embedded Coder benötigt nahezu viermal so viele Zeilen wie das Matrix-Frontend. Hieran lässt sich die unterschiedliche Ausrichtung der beiden Codegeneratoren erkennen: der Code des Embedded Coder ist direkt für das eingebettete System erzeugt worden, wohingegen der Code des Matrix-Frontends für die Analysier- und Parallelisierbarkeit in weiteren Tools optimiert wurde. Hierfür ist kleinerer Code besser geeignet und die Performanz soll erst später durch weitere Optimierungsschritte verbessert werden.

	Laufzeit [μs]	RAM [Byte]	Stack [Byte]	Codezeilen
Embedded Coder	4199	19.329	15.109	4.247
Code Generator	5420	42.776	1.572	1.101

Tabelle 4.7.: Laufzeit und Speicherverbrauch des Kalman-Filters

Abschließend wurden weitere Messungen mit entrollten Schleifen durchgeführt. Die Ergebnisse sind in Tabelle 4.8 dargestellt. Dabei ist zu beachten, dass die Werte für den *Embedded Coder* noch zusätzlich mit dem *Code Generation Advisor* optimiert wurden und somit noch etwas besser sind, als rein durch das Entrollen der Schleifen. Die Performanz bleibt vergleichbar bei etwa 31 % langsamerer Ausführung des vom Code Generator erzeugten Codes. Die Anzahl der Codezeilen wurden durch den Advisor beim *Embedded Coder* bereits optimiert und auf 2.644 Zeilen reduziert. Es zeigt sich, dass das Entrollen beim *Embedded Coder* für etwa 600 zusätzliche Zeilen Code sorgt, beim Code Generator hingegen nur für etwa 100 Zeilen.

	Laufzeit [μs]	Anzahl Codezeilen
Embedded Coder	3213	3.233 (2.644)
Code Generator	4210	1.202

Tabelle 4.8.: Laufzeit und Speicherverbrauch des Kalman-Filters mit entrollten Schleifen

4.3.5.5. Zusammenfassung der Ergebnisse beim Kalman-Filter

Aus der ersten Evaluation des Code Generators mit einer echten Anwendung aus der Industrie ergaben sich einige wichtige Erkenntnisse. Zunächst ist die generelle Kompatibilität mit der eingesetzten Darstellung von Algorithmen in Unternehmen gezeigt worden. Die Bedienung ist dabei so eingängig, dass Code schnell erzeugt und mit anderen Werkzeugen verglichen werden kann. Die Performanz ist bei grundlegenden Operationen wie der Matrix-Multiplikation mindestens so gut wie kommerziell verfügbare Lösungen, hat durch die Darstellung von Matrizen in manchen Fällen sogar Vorteile. Bei komplexeren Anwendungen greift die größere Anzahl an Optimierungen des *Embedded Coders* und der generierte Code führt bekannte Algorithmen effizienter aus. Diese Optimierungen können jedoch ebenfalls im Code Generator integriert werden und behindern nicht den generellen Ansatz, besser analysierbaren Code zu erzeugen. Diese Unterschiede im Ansatz

werden bei der allgemein besseren Lesbarkeit und der geringeren Anzahl an Codezeilen ersichtlich. Der Code benötigt nur ein Viertel so viele Codezeilen und die einzelne Indizierung der Dimensionen sorgt für ein besseres Verständnis des Codes. Die Unterschiede sind in Tabelle 4.9 noch einmal zusammengefasst. Dabei ist ein Vorteil durch + und ein Nachteil durch - gekennzeichnet.

	Embedded Coder	Code Generator
Lesbarkeit	-	+
Laufzeit	+	-
Anzahl Codezeilen	-	+
Verifikationsmöglichkeit	+	-
Hardware-Kompatibilität	+	-

Tabelle 4.9.: Unterschiede zwischen Embedded Code und Code Generator

Als Teilergebnis der multigranularen Optimierung kann festgehalten werden, dass die Konvertierung von MATLAB®-Skripten hin zu C bereits viele Möglichkeiten sowohl auf der Algorithmus- als auch auf der Code-Ebene bietet. Häufig eingesetzte Anwendungen wie Kalman-Filter haben wiederkehrende Elemente, für die verschiedene Implementierungen bereitgestellt werden können. Die abstraktere Darstellung erlaubt zusätzlich effizientere Optimierungen auf Code-Ebene, indem einfache Matrixoperationen wie die Multiplikation oder die Inversion zusammen betrachtet werden können, um Schleifen einzusparen oder durch Transformationen noch besser an eine nachfolgende Verarbeitung angepasst werden können.

Die Optimierung auf Algorithmus-Ebene ist im Beispiel des Kalman-Filters bei Durchführung der Masterarbeit im *Embedded Coder* weiter fortgeschritten als im Code Generator. Es zeigt dennoch deutlich das Potential dieses Optimierungsschrittes, da hier ein Performanzgewinn von rund 30 % erreicht werden konnte.

4.4. Generierung von C-Code für parallele Zielplattformen

Der parallelisierte Algorithmus soll im abschließenden Codegenerierungsschritt speziell an die ausgewählte Zielarchitektur angepasst werden. Dazu sollen existierende Speicherhierarchien sowie die Anzahl der Ausführungseinheiten mit ihrer Kommunikationsinfrastruktur ausgenutzt werden. Dies erfordert, dass der erzeugte Code auf die verschiedenen Hardwareeinheiten partitioniert und in den entsprechenden Speichern abgelegt werden kann.

Als Ausgabe soll C-Code generiert werden, der mit dem Compiler der Zielarchitektur kompatibel ist. Spezielle Hardwareeigenschaften wie der Aufbau der Speicherhierarchien, die Anzahl an Ausführungseinheiten sowie die Kommunikationsinfrastruktur sollen effizient ausgenutzt werden.

Daraus ergeben sich die folgenden Anforderungen:

- Es soll Code generiert werden, der auf die Zielplattform zugeschnitten ist: es können passende Bibliotheken referenziert werden und direkte („low-level“) Zugriffe auf die Hardwarekomponenten sind erlaubt.
- Compiler können unterschiedliche Anforderungen an den Quellcode stellen, daher müssen unterschiedliche Darstellungen hinsichtlich der Anzahl und der Verteilung auf mehrere Dateien zur Verfügung gestellt werden.
- Der generierte Code soll funktional identisch mit dem Eingangscode sein und das gleiche Verhalten zeigen, wie vor den Optimierungen. Aus anderen Transformationen gewonnene Informationen sollen soweit möglich eingesetzt werden. Dadurch soll sichergestellt werden, dass sowohl automatisierte Transformationen als auch Nutzereingaben verarbeitet werden können und dennoch korrekter Code erzeugt wird.
- Für jede Ausführungseinheit auf dem System muss eine eigene Anwendung erzeugt werden. Dazu gehört neben der Verwaltung der Zuständigkeit auch ein konsistenter Kontrollfluss aber auch eine Synchronisation zwischen diesen einzelnen Strängen.
- Die Partitionierung der Daten und damit die Aufteilung/Vervielfältigung der einzelnen Variablen muss gewährleistet werden.

Für die effiziente Programmierung von eingebetteten Systemen sind noch weitere Hardwareeigenschaften relevant. Dazu gehören die Speicherhierarchien, Hardwareunterstützung von Datentypen, die Kommunikationsinfrastruktur oder die unterstützten SIMD-Befehle.

Der Zugriff auf diese Elemente kann direkt in der Programmiersprache C erfolgen, reduziert aber die Übersichtlichkeit und Verständlichkeit des generierten Codes. Abhilfe schafft die Verwendung einer *Programmierschnittstelle* (API, engl. *application programming interface*), bei der die Implementierungsdetails durch Funktionsaufrufe gekapselt werden und nicht mehr in der Hauptdatei der Anwendung auftreten.

Existieren verschiedene Speicher mit einer klaren Hierarchie und unterschiedlichen Zugriffszeiten, so ist eine optimierte Speicherverwaltung sinnvoll. Diese muss in der Regel explizit im Code durchgeführt werden, da Compiler, die den ausführbaren Code erzeugen in der Regel keinen Einblick mehr in den ausgeführten Algorithmus haben.

4.4.1. Parallelisierung des Kontrollflusses

Damit für jeden Prozessor ein funktionsfähiges Programm erzeugt werden kann, muss das sequentielle Programm, das bereits mit Kommunikationsanweisungen und Kernzuweisungsinformationen angereichert ist, weiter modifiziert werden. Der Kontrollfluss ist zunächst rein sequentiell und stellt die Abfolge an Basisblöcken des Programms dar. Diese kann der Reihe nach von einem Prozessor ausgeführt werden. Bei rein sequentiellen Basisblöcken ist die Aufteilung auf mehrere Kerne trivial: alle Basisblöcke können direkt auf verschiedene Prozessoren verschoben werden. Dabei muss lediglich die ursprüngliche, relative Reihenfolge zwischen den Blöcken erhalten bleiben. Datenabhängigkeiten wurden bereits über Kommunikationsanweisungen aufgelöst: werden die Basisblöcke in

der vorgegebenen Reihenfolge ausgeführt, so wird auch die Kommunikation korrekt ausgeführt.

Anders sieht es aber bei Kontrollstrukturen aus, die für Verzweigungen im Kontrollfluss sorgen: befinden sich Blöcke, die auf unterschiedlichen Prozessoren ausgeführt werden sollen, innerhalb der gleichen Kontrollstruktur, wie beispielsweise einer Schleife, so können die Blöcke nicht einfach aufgeteilt werden. Dies würde den Kontrollfluss auf den individuellen Prozessoren verändern, da nicht jeder die gleichen Bedingungen hat. Das Duplizieren des Kontrollflusses ist eine wirksame Möglichkeit, um das Auftreten von Deadlocks zu verhindern. Wird sichergestellt, dass vor dem Duplizieren bereits alle Datenabhängigkeiten mit Hilfe von expliziten Kommunikationsinstruktionen aufgelöst wurden, so werden Sende- und Empfangsbefehle nach Anpassung des Kontrollflusses immer unter den gleichen Bedingungen ausgeführt. Dies verhindert Deadlocks, da zu jedem Sendebefehl unter den gleichen Randbedingungen der zugehörige Empfangsbefehl ausgeführt wird.

Es werden im Folgenden zwei Methoden vorgestellt, wie der Kontrollfluss angepasst werden kann.

4.4.1.1. Synchrone Vervielfältigung

Die Grundidee bei der synchronen Vervielfältigung ist es, den Kontrollfluss auf den verschiedenen Prozessoren zu synchronisieren. Wird auf einem Prozessor ein erneuter Schleifendurchlauf durchgeführt, so muss das synchron auch auf allen anderen beteiligten Prozessoren gemacht werden. Bei For- und While-Schleifen sowie If-Blöcken wird über eine Evaluation eine Entscheidung getroffen, auf welchem Zweig die Ausführung fortgesetzt wird. Diese Evaluation ist in der Regel an Variablen oder Funktionsaufrufe gekoppelt. Diese Informationen sind prozessorabhängig und liegen nicht unbedingt jedem Prozessor vor. Anstatt nun alle Daten zu kommunizieren, die für die Auswertung der Bedingung notwendig sind, wird nur das Ergebnis der Evaluation kommuniziert. In der Programmiersprache C entspricht das dem Wert 0 für ein negatives Ergebnis oder einem Wert ungleich 0 für ein positives Ergebnis. Es muss also nur ein einzelner Wert übertragen werden. Dieser wird bei jedem weiteren Durchlauf der Schleife erneut übertragen, so dass die Ausführung der Schleife über mehrere Prozessoren hinweg immer komplett synchron erfolgt. Diese Art der Vervielfältigung ist in Abbildung 4.6 dargestellt. Die ursprüngliche For-Schleife wird in eine For-Schleife und in eine While-Schleife aufgeteilt, wobei die Bedingung des Test-Blocks nur in der For-Schleife evaluiert wird und das Ergebnis in jeder Iteration an die While-Schleife geschickt wird.

4.4.1.2. Asynchrone Vervielfältigung

Die synchrone Ausführung sowie der Kommunikationsaufwand begrenzen die erreichbare Performance bei der Ausführung der Anwendung. Sie führen zu ständig wiederkehrenden Wartezeiten, wenn die Rechenlast der beteiligten Prozessoren nicht exakt gleich ist. Um eine bessere Performanz durch parallele Ausführung zu erreichen, wird daher eine asynchrone Ausführung bevorzugt. Dies kann durch eine Vervielfältigung der Kon-

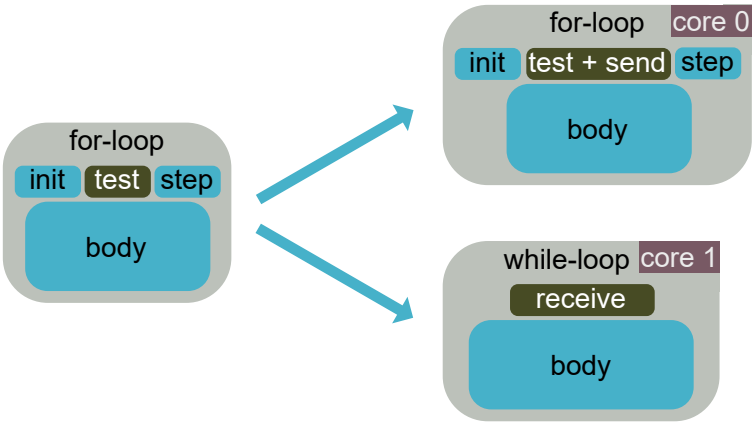


Abbildung 4.6.: Synchrones Duplizieren einer For-Schleife

trollstruktur erreicht werden. Diese Vervielfältigung ist jedoch nicht in jedem Fall möglich, folgende Bedingungen müssen erfüllt sein:

- Es dürfen keine Seiteneffekte rein durch die Schleifenstruktur entstehen. Wird beispielsweise als Bedingung für einen Schleifendurchlauf eine Funktion aufgerufen, so darf diese keine weiteren Auswirkungen auf Variablen oder sonstige Ausgaben haben. Diese würden ebenfalls vervielfältigt und hätten ein anderes Verhalten als im sequentiellen Fall.
- Schleifenvariablen dürfen nicht innerhalb der Schleife verändert werden. Wird die Zählvariable verändert, so muss dies anderen Prozessoren mitgeteilt werden. Dies hat zur Folge, dass die Schleifen wieder synchron ausgeführt werden und der Vorteil der vervielfältigten Schleife entfällt.

Sind diese Bedingungen erfüllt, kann die Schleife auf allen beteiligten Prozessoren kopiert werden. So lange keine weiteren Abhängigkeiten zwischen den Basisblöcken innerhalb der Schleifen besteht, können diese nun unabhängig voneinander ausgeführt werden und können in der Zeitplanung getrennt betrachtet werden. Ein Beispiel ist in Abbildung 4.7 dargestellt. Ist der Body der ursprünglichen For-Schleife auf zwei unterschiedliche Kerne verteilt und sind die Bedingungen erfüllt, so kann die Schleife asynchron dupliziert werden. Der Kopf der Schleife kann bei beiden Schleifen kopiert werden und nur der Body der beiden unterscheidet sich.

4.4.2. Abstraktion durch Einsatz von APIs

Der größte Vorteil beim Einsatz einer API ist das Verbergen von Implementierungsdetails: In der Regel ist es nicht notwendig, die Details einer Realisierung zu kennen, wenn die Funktionalität klar über die API geregelt wird. Als Beispiel sollen hier die eingesetzten Funktionen zur Kommunikation hergenommen werden:

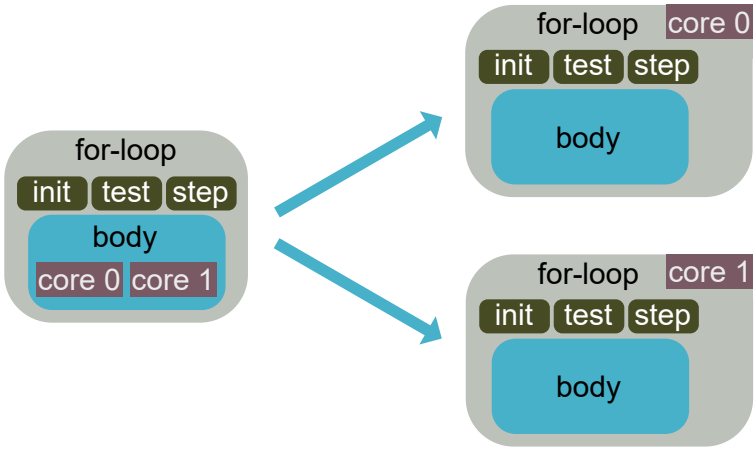


Abbildung 4.7.: Asynchrones Duplizieren einer For-Schleife

```
1 void COMMAPI_Init(int coreId);
2 void COMMAPI_Finalize();
3
4 static inline void COMMAPI_Send(void * buf, int size, int sender, int
      receiver, int id, int index);
5 static inline void COMMAPI_Recv(void *buf, int size, int sender, int
      receiver, int id, int index);
```

Quellcode 4.12: Auszug aus Kommunikations-API

Die hier dargestellten Prototypen der Funktionen zeigen auf, mit welchen Parametern und Datentypen die Funktionen aufgerufen werden müssen. Neben dieser technischen Darstellung wird noch eine Spezifikation der Funktionalität benötigt. Für diese vier Funktionen können die wichtigsten Eigenschaften wie folgt zusammengefasst werden:

- Beim Programmstart muss für jeden Kern der Zielplattform die Funktion `COMMAPI_Init` ausgeführt werden, um die Kommunikationsinfrastruktur zu initialisieren.
- Beim Beenden der Anwendung muss umgekehrt die Funktion `COMMAPI_Finalize` aufgerufen werden, um alle Ressourcen wieder freizugeben.
- Nach Aufruf der Sende-Funktion können Daten in der Puffervariablen verändert werden, ohne die Daten des Empfängers zu beeinflussen.
- Kehrt die Funktion zum Empfang der Daten wieder zurück, so befinden sich gültige Daten in der Puffervariablen.

Die API liefert hier keine Details zur tatsächlichen Implementierung. So kann bei Aufruf der Initialisierungsfunktion beispielsweise ein gemeinsamer Speicherbereich zum Datenaustausch definiert werden oder mit Funktionen des POSIX Standards [37] Prozesse erzeugt werden, die mittels ebenfalls erzeugter Pipes kommunizieren. Beim Transfer der

Daten bedingen die Angaben zum Verhalten noch weitere Implementierungsdetails. So muss bei einem System mit geteiltem Speicher beispielsweise sichergestellt werden, dass Daten entsprechend kopiert werden, um das Verändern der Daten eines anderen Kerns zu verhindern. Die Ausführungszeit der Funktion kann also sehr stark von der Implementierung abhängen, da unter Umständen sogar mehrere Kopieroperationen oder Zugriffe auf Kommunikationsstrukturen wie Busse oder Netzwerke benötigt werden.

Die Abstraktion vereinfacht die Verwendung der Funktionen deutlich, da sie sich bei korrekter Implementierung auf jedem System identisch verhalten. Zusätzlich erlauben verschiedene Implementierungen auf dem gleichen System hier einen schnellen Wechsel, um Unterschiede beim Speicherbedarf, der Leistungsaufnahme oder der Latenz ausnutzen zu können.

Im Kontext dieser Arbeit wird die Abstraktion durch APIs verwendet, um hinsichtlich Effizienz optimierte Realisierungen der Kommunikationsfunktionen für die verschiedenen Zielarchitekturen einzusetzen.

4.4.3. Realisierung der Generierung parallelen Codes

4.4.3.1. Parallelisierung des Kontrollflusses

Duplizieren von If-Blöcken

Zur Parallelisierung von If-Blöcken werden die folgenden Annahmen gemacht:

- Ein If-Block besteht aus bis zu drei einzelnen Blöcken: ein Basisblock für die Bedingung, ein `then`-Block, wenn die Bedingung `true` ist und ein optionaler `else`-Block, wenn die Bedingung `false` ist.
- Die Bedingung wird nur auf einem Kern ausgeführt. Sollte die Bedingung innerhalb einer (komplexen) Funktion ausgewertet werden, so besteht immer die Möglichkeit, die Funktion vor dem If-Block zu evaluieren, das Ergebnis in einer temporären Variablen zu speichern und diese in der Bedingung des If-Blocks auszuwerten.
- Eine effiziente Parallelisierung eines If-Blocks ist nicht trivial, da zwar der `then`- und der `else`-Block keine Abhängigkeiten untereinander haben können, in der Praxis aber entweder der eine oder der andere Block ausgeführt wird. Im Weiteren werden alle Parallelisierungen betrachtet, bei denen Teile oder die kompletten `then`- und `else`-Blöcke auf unterschiedlichen Kernen ausgeführt werden.

Sind alle diese Annahmen erfüllt, so wird das synchrone Duplizieren wie folgt angewendet:

- Der Ausdruck im Bedingungsblock wird vor den If-Block gezogen und das Ergebnis der Evaluation wird in der temporären Variable `sync` gespeichert.
- Die Bedingung evaluiert die neue Variable `sync`.
- Das Schreiben der Variable `sync` und damit die Auswertung des Ausdrucks erfolgt auf dem Kern, der die niedrigsten Kommunikationskosten zur Berechnung der Bedingung hat.

- Für alle anderen Kerne wird eine Kopie des If-Blocks erstellt, deren Bedingung der Empfang des Evaluationsergebnis ist.

Auf diese Weise wird sichergestellt, dass die eigentliche Bedingung exakt ein Mal evaluiert wird und abhängig von diesem Ergebnis führen die anderen Kerne den entsprechenden Code aus.

Ein Beispiel ist in Quellcode 4.13 dargestellt. Der sequentielle If-Block in Zeile 6 - 12 wird so parallelisiert, dass Variable b auf Kern 0 und Variable c auf Kern 1 berechnet wird. Das Ergebnis res aus Zeile 14 soll wieder auf Kern 0 berechnet werden. Im parallelen Code ab Zeile 16 wurde eine neue Variable sync1_p0 (Zeile 22) eingeführt, die das Ergebnis der Evaluation speichert (Zeile 24). Anschließend wird der Wert in Zeile 25 gesendet und in Zeile 27 auf Kern 0 evaluiert. Der kopierte If-Block auf Kern 1 empfängt in Zeile 40 das Ergebnis der Evaluation direkt im neuen Bedingungsblock und führt abhängig vom Wert den entsprechenden `then`- oder `else`-Block aus.

```
1  /*** sequential ***/
2  float a = rand();
3  int b, c;
4  int res;
5
6  if (a > 0.5) {
7    b = func1();
8    c = func2();
9  } else {
10   b = func3();
11   c = func4();
12 }
13
14 res = b + c;
15
16 /*** parallel ***/
17
18 // core 0
19 float a_p0 = rand();
20 int b_p0, c_p0;
21 int res_p0;
22 int8_t sync1_p0;
23
24 sync1_p0 = a < 0.5;
25 SendSync(sync1_p0, 0, 1);
26
27 if (sync1_p0) {
28     b_p0 = func1();
29     Recv32(&c_p0, 1, 0);
30 } else {
31     b_p0 = func3();
32     Recv32(&c_p0, 1, 0);
33 }
34
35 res_p0 = b_p0 + c_p0;
36
```

```
37  // core 1
38  int c_p1;
39
40  if (RecvSync(0, 1)) {
41    c_p1 = func2();
42    Send32(&c_p1, 1, 0);
43  } else {
44    c_p1 = func4();
45    Send32(&c_p1, 1, 0);
46  }
```

Quellcode 4.13: Synchrones Duplizieren eines If-Blocks

Duplizieren von Schleifen

Das in Abschnitt 4.4.1.2 dargestellte asynchrone Duplizieren ist die bevorzugte Methode für das Duplizieren von Schleifen, da kein Kommunikationsoverhead durch Synchronisation der Schleifen erzeugt wird. Damit sie durchgeführt werden kann, müssen zunächst die beiden beschriebenen Bedingungen erfüllt sein: es dürfen keine Seiteneffekte auftreten und die Schleifenvariable darf nicht innerhalb der Schleife verändert werden.

Die Abwesenheit von Seiteneffekten kann im allgemeinen Fall nicht beantwortet werden, da selbst einfache Schreib- oder Lesezugriffe auf Variablen auf Hardware-Register erfolgen können und damit weitere Effekte auslösen können. Um die Methode dennoch in gängigen Anwendungen benutzen zu können, werden die folgenden vereinfachenden Annahmen getroffen:

- Variablen stehen nur für Daten im Speicher und Lese- bzw. Schreibzugriffe haben keine Seiteneffekte. Stehen sie dennoch für Zugriffe auf Register, so muss dies über Pragmas oder in weiteren Hilfedateien hinterlegt werden.
- Pointer werden nur als Zeiger auf Daten verwendet und vereinfachen den Zugriff auf eine definierte Stelle im Speicher, die sich zur Laufzeit nicht mehr verändert. Damit können sie auf die gleiche Weise behandelt werden wie einfache Variablen. Jegliche Pointerarithmetik wird nicht unterstützt und die entsprechenden Programmteile werden nicht parallelisiert.
- Funktionsaufrufe haben immer Seiteneffekte, unabhängig davon, ob diese tatsächlich analysiert werden können oder nicht. Diese Annahme kann getroffen werden, da in den vorangegangenen Phasen bereits analysiert wurde, ob die Werte der Funktionen konstant sind oder nicht. Konstante Werte wurden also bereits verwendet und die entsprechenden Funktionen kommen im Code gar nicht mehr vor. Im Umkehrschluss bedeutet dies, dass alle anderen Funktionen entweder Seiteneffekte haben oder von Werten zur Laufzeit abhängig sind. Ersteres sollte bereits ausgeschlossen sein, letzteres würde bedeuten, dass entweder das Ergebnis kommuniziert werden müsste und damit eine synchrone Duplizierung notwendig wäre oder die Funktion auf allen Kernen ausgeführt werden müsste. Letzteres ist im Regelfall noch aufwendiger, da alle Kerne alle Daten zur Berechnung benötigen und der Kommunikationsaufwand unter Umständen noch größer wäre als beim synchronen Duplizieren.

- Im Schleifenkopf werden nur einfache arithmetische Operationen wie die Addition oder die Multiplikation verwendet. Nur in diesen Fällen kann sichergestellt werden, dass keine nicht betrachteten Seiteneffekte auftreten können.

Sind alle Bedingungen und Annahmen bei einer Schleife erfüllt, so kann sie asynchron dupliziert werden. Ein einfaches Beispiel ist in Quellcode 4.14 dargestellt. Alle Randbedingungen und Annahmen sind in diesem Beispiel erfüllt, so dass im parallelen Fall die Schleife bzw. ihr Kopf vollständig dupliziert werden kann. Das Ergebnis kann in diesem Fall auch als Kombination aus Loop Fission und Umbenennung von Variablen betrachtet werden.

```
1  /*** sequential ***/
2  int i;
3  int a[10], b[10];
4
5  for (i = 0; i < 10; i++) {
6    a[i] = i;
7    b[i] = i * i;
8  }
9
10 /*** parallel ***/
11
12 // core 0
13 int i_p0;
14 int a_p0[10];
15
16 for (i_p0 = 0; i_p0 < 10; i_p0++) {
17   a_p0[i_p0] = i_p0;
18 }
19
20 // core 1
21 int i_p1;
22 int b_p1[10]
23
24 for (i_p1 = 0; i_p1 < 10; i_p1++) {
25   b_p1[i_p1] = i_p1 * i_p1;
26 }
```

Quellcode 4.14: Asynchrones Duplizieren einer For-Schleife

Wird der Kontrollfluss innerhalb von Schleifen von Sprüngen durch `break` oder `continue` verändert, so wird die Bedingung um die Ausdrücke herum, wie im vorigen Paragraphen beschrieben, dupliziert. Das bedeutet, dass das Ergebnis der Bedingung an alle beteiligten Kerne gesendet wird. Dadurch wird eine direkte Synchronisation zwischen den Schleifen erzeugt und alle werden exakt mit der gleichen Anzahl durchlaufen. Ein Beispiel ist in Quellcode 4.15 dargestellt. Die sequentielle Schleife (Zeile 5) wird asynchron dupliziert (Zeile 20 und 33), der If-Block aus Zeile 6 wird wie im vorigen Abschnitt beschrieben parallelisiert. Die `continue`-Anweisung aus Zeile 7 wird im parallelen Fall kopiert (Zeile 24 und 35) und durch den duplizierten If-Block auf beiden Kernen gleich häufig ausgeführt.

```
1  /*** sequential ***/
2  int i;
3  int a[10], b[10];
4
5  for (i = 0; i < 10; i++) {
6    if (i % 3 == 0) {
7      continue;
8    }
9    a[i] = i;
10   b[i] = i * i;
11 }
12
13 /*** parallel ***/
14
15 // core 0
16 int8_t sync1_p0;
17 int i_p0;
18 int a_p0[10];
19
20 for (i_p0 = 0; i_p0 < 10; i_p0 = i_p0 + 1) {
21   sync1_p0 = i_p0 % 3 == 0;
22   SendSync(sync1_p0, 0, 1);
23   if (sync1_p0) {
24     continue;
25   }
26   a_p0[i_p0] = i_p0;
27 }
28
29 // core 1
30 int b_p1[10];
31 int i_p1;
32
33 for (i_p1 = 0; i_p1 < 10; i_p1 = i_p1 + 1) {
34   if (RecvSync(0, 1)) {
35     continue;
36   }
37   b_p1[i_p1] = i_p1*i_p1;
38 }
```

Quellcode 4.15: Asynchrones Duplizieren einer For-Schleife mit `continue`

Bei While-Schleifen wird ein ähnliches Vorgehen durchgeführt: da bei einer While-Schleife in der Regel keine Zählvariable existiert, muss überprüft werden, ob die Anweisungen innerhalb des Schleifenrumpfs die Anzahl an Iterationen verändern können. Dabei werden `break` und `continue` auf die gleiche Weise behandelt wie bei For-Schleifen, allein Schreibzugriffe auf Variablen, die das Ergebnis der Evaluation der Bedingung verändern, müssen betrachtet werden. Seiteneffekte werden analog wie bei For-Schleifen betrachtet.

4.4.3.2. Abstraktion der Kommunikationsinfrastruktur

Wie in Abschnitt 4.4.2 beschrieben wurde, werden APIs eingesetzt, um die tatsächliche Implementierung für eine spezielle Hardware hinter klar definierten Funktionen zu verbergen. Dabei kann die Implementierung sehr individuell für die jeweilige Zielhardware optimiert werden, was natürlich bei der Parallelisierung mit einbezogen werden muss. Für die Kommunikation wird für jede Implementierung von einem bestimmten Kern zu einem anderen Kern eine Kommunikationstabelle erstellt, die die Kosten pro Byte über verschiedene Nachrichtengrößen hinweg betrachtet. Die Zahlenwerte werden durch Messung auf der Hardware ermittelt und beinhalten alle Optimierungen durch den eingesetzten Compiler sowie eventuelle Effekte durch die Caches auf der Hardware.

Als Beispiel soll die Ringpuffer-Implementierung für die Jetson TX2 Architektur verwendet werden. Der heterogene Prozessor besteht aus vier ARM Cortex-A57 Kernen, zwei Denver Kernen (ARM Kerne von Nvidia) und einer GPU auf Pascal Basis. Für den Einsatz unter Linux wurde die Kommunikation über einen Ringspeicher realisiert. Dazu wird für jede mögliche Kommunikation ein Puffer im geteilten Speicher erstellt, in den ein sendender Kern Daten schreibt und der Empfänger Daten lesen kann. Da die beiden CPU-Kerne Cortex-A57 und Denver unterschiedliche Taktfrequenzen und zeitliches Verhalten haben, werden zur Abbildung der Kommunikation vier Kommunikationstabellen mit allen Richtungen benötigt: Cortex-A57 zu Cortex-A57, Cortex-A57 zu Denver, Denver zu Cortex-A57 und Denver zu Denver. Zwei Beispiele sind in Quellcode 4.16 und Quellcode 4.17 dargestellt. Im direkten Vergleich zeigt sich, dass Transfers zwischen ungleichen Kernen deutlich mehr Overhead benötigen: die maximalen Kosten sind um den Faktor 3,8 höher, die minimalen um 2,6 und der maximale Durchsatz liegt nur bei 39%. Diese Kosten müssen bei der Parallelisierung auf Task-Ebene mit betrachtet werden, um eine effiziente Parallelisierung zu gewährleisten.

```
1  {
2    "version"    : 100,
3    "platform"   : "jetson_tx2_lnx",
4    "api"        : "ringbuffer",
5    "comm_table" : {
6      "cortex-a57" : {
7        "cortex-a57" : {
8          "max_cost"       : 75.689197,
9          "min_cost"       : 0.111639,
10         "max_throughput" : 8957460480,
11         "cost_per_byte_table" : {
12           "1" : 75.689197449,
13           "2" : 39.253440387,
14           "4" : 19.680111594,
15           "8" : 9.899770369,
16           "16" : 5.384329104,
```

```
17              "32" : 2.835287489,
18              "64" : 1.480635655,
19              "128" : 0.780721452,
20              "256" : 0.423100044,
21              "512" : 0.271351028,
22              "1024" : 0.187189595,
23              "2048" : 0.139940746,
24              "4096" : 0.127730030,
25              "8192" : 0.119621101,
26              "16384" : 0.114120667,
27              "32768" : 0.114745167,
28              "65536" : 0.111757055,
29            }
30          }
31        }
32      }
33  }
```

Quellcode 4.16: Auszug aus Kommunikationstabelle für Cortex-A57 nach Cortex-A57 Transfers auf dem Nvidia Jetson TX2

```
1  {
2    "version"     : 100,
3    "platform"    : "jetson_tx2_lnx",
4    "api"         : "ringbuffer",
5    "comm_table" : {
6      "cortex-a57" : {
7        "denver2" : {
8          "max_cost"        : 287.775581,
9          "min_cost"        : 0.286927,
10         "max_throughput" : 3485204480,
11         "cost_per_byte_table" : {
12           "1" : 287.775581091,
13           "2" : 135.415335083,
14           "4" : 70.923424404,
15           "8" : 36.388708846,
16           "16" : 9.516348525,
17           "32" : 9.008936938,
18           "64" : 4.608271698,
```

```
19              "128" : 2.388052060,
20              "256" : 1.883960803,
21              "512" : 1.247343035,
22              "1024" : 0.687456003,
23              "2048" : 0.601825699,
24              "4096" : 0.497804912,
25              "8192" : 0.485456774,
26              "16384" : 0.460976539,
27              "32768" : 0.480743197,
28              "65536" : 0.422314822
29          }
30        }
31      }
32    }
33  }
```

Quellcode 4.17: Auszug aus Kommunikationstabelle für Cortex-A57 nach Denver Transfers auf dem Nvidia Jetson TX2

Unterschiedliche Implementierungen für das gleiche System können neben der reinen Funktionalität noch verschiedene Vor- und Nachteile haben. Ein wichtiger Punkt ist die Art und Weise, wie der Empfänger auf Daten wartet. Beim aktiven Warten, auch als *busy waiting* oder als Realisierung mittels eines Spinlock bekannt, prüft der Kern ständig, ob neue Daten vorliegen. Er ist damit voll ausgelastet, ohne dass das Programm weiter voranschreiten kann, so lange die Daten noch nicht angekommen sind. Der Vorteil dieser Implementierung ist, dass die Daten sehr schnell weiterverarbeitet werden können, sobald sie angekommen sind. Dafür ist die Leistungsaufnahme vergleichsweise hoch, da der Kern nicht in einen Modus mit niedrigerer Leistungsaufnahme versetzt werden kann. Stattdessen können Realisierungen eingesetzt werden, bei denen der Kern in einen sparsameren Modus versetzt wird, bis er durch einen Interrupt oder vom Betriebssystem aufgeweckt wird, wenn die Daten angekommen sind. Dies geht jedoch zu Lasten der Reaktionszeit.

Andere Faktoren, die bei verschiedenen Realisierungen noch relevant sind, sind der Speicherbedarf, der beispielsweise durch Puffer entsteht und Abhängigkeiten zu existierenden Bibliotheken, die zur Compile-Zeit vorliegen müssen.

4.5. Zusammenfassung der Codegenerierung

In diesem Kapitel wurden die beiden Phasen der Codegenerierung beschrieben, die bei der ebenenübergreifenden Parallelisierung verwendet werden. Die erste Phase hat zum Ziel, effizienten C-Code aus MATLAB®-Code zu generieren, und die zweite Phase hat

zum Ziel, parallelen C-Code aus sequentiellem C-Code zu generieren. Auch wenn diese Hauptaufgaben nicht direkt mit den Optimierungen auf den einzelnen Ebenen übereinstimmen, können bereits zwei Ebenen abgedeckt werden:

- Optimierungen auf der Algorithmus-Ebene werden bereits in der ersten Codegenerierungsphase vollständig abgedeckt. Hier können durch die Verwendung der abstrakteren Sprache MATLAB® einzelne Algorithmen erkannt werden und somit verschiedene Implementierungen für verschiedene weitere Optimierungen abgedeckt werden.
- Die Code-Ebene kann sowohl in der ersten als auch in der zweiten Phase optimiert werden. In der ersten Phase können einzelne Code-Transformationen noch einfacher durchgeführt werden, da die Strukturen in den Matrix-Sprachen leichter analysierbar sind. In der zweiten Phase hingegen können gezielter Transformationen ausgewählt werden, um den Code für die Algorithmus- und Code-Ebenen besser vorzubereiten. Weitere Details zu diesem Vorgehen sind im Kapitel 5.3 beschrieben.

5. Realisierung der ebenenübergreifenden Parallelisierung

Nachdem im letzten Kapitel die beiden Codegenerierungsphasen beschrieben wurden, werden in diesem Kapitel die Parallelisierungen auf den vier Ebenen (Algorithmus, Code, Task und Daten) im Detail diskutiert. Zunächst wird jedoch eine abstrakte Spezifikation der Zielplattform vorgestellt, die alle wichtigen Informationen für die einzelnen Ebenen formal beschreibt.

5.1. Spezifikation der Zielplattform

Ziel der multigranularen Optimierung ist es, von einer abstrakten, mathematischen Beschreibung eines Algorithmus auszugehen, bei dessen Implementierung kein Wissen über die spätere Hardware und deren Ausführungseinheiten verwendet wird. Mit zunehmender Optimierung über die verschiedenen Ebenen werden Details über die verwendete Hardware immer wichtiger, so dass eine klare, formale Beschreibung aller wichtigen Eigenschaften notwendig wird. Diese wird in diesem Kapitel näher spezifiziert.

5.1.1. Architekturbeschreibung

Zur Spezifikation der Eigenschaften einer Zielplattform, wie sie in Abschnitt 3.5 dargestellt werden, kommt eine ADL zum Einsatz, die auf Version 1 der *Software-Hardware Interface for Multi-Many-Core* (SHIM™ [93])-ADL basiert. Sie wurde ursprünglich von der Multicore-Association spezifiziert, ist aber seit Januar 2020 ein IEEE-Standard[94].

Ein Beispiel ist in Quellcode 5.1 dargestellt. Die XML-basierte Beschreibung hat dabei den folgenden Aufbau:

- Auf der obersten Ebene wird die Systemkonfiguration spezifiziert und der Name der Plattform sowie die verwendete SHIM-Version angegeben. [Zeile 2]
- Im Haupt-ComponentSet [Zeile 3] werden die wichtigsten Komponenten der Plattform beschrieben. Diese umfassen die Ausführungseinheiten sowie das Speichersubsystem der Plattform.
- Die wichtigsten Komponenten des Zielsystems werden über weitere ComponentSets definiert. Im Beispiel sind das der „Quad-Core ARM Cortex-A57 MPCore“ [Zeile 4], der „Dual-Core Denver 2“ [Zeile 17] und die „Jetson-TX2 GPU“ [Zeile 25].

- Jede Komponente kann aus MasterComponents wie Prozessorkernen, beispielsweise „Cortex-A57“ [Zeile 5 und Zeile 10] oder Beschleunigern, wie der GPU „Jetson-TX2 GPU“ [Zeile 26], bestehen. Wichtige Eigenschaften sind zudem die Einträge „arch“ für die Architektur wie beispielsweise „armv8-a“ [Zeile 5] sowie die Einträge unter „archOption“. Dieses Feld, welches in SHIM als optional spezifiziert ist, wird eingesetzt, um weitere wichtige Eigenschaften eines Prozessors zu beschreiben:
  - Über „class“ wird die Art der Ausführungseinheit der Komponente dargestellt. Unterschieden wird zwischen Prozessor, GPU und FPGA.
  - Über den „coretype“ wird der tatsächliche Architekturtyp der Ausführungseinheit festgelegt. Dieser Eintrag kann verwendet werden, um gleiche Ausführungseinheiten in der Plattform erkennen zu können.
  - Bei Systemen mit GPU- oder FPGA-Beschleuingern wird ein Prozessorkern als „control_core“ markiert. Dieser wird eingesetzt, um die Datensynchronisation mit den Beschleunigern zu gewährleisten.
- Des Weiteren werden zu jeder MasterComponent die entsprechenden Instruktions- und Daten-Caches angegeben. Für den Cortex-A57 #0 sind dies der DCache [Zeile 6] und der ICache [Zeile 7]. Zu jedem Cache wird der Typ, also Daten oder Instruktionen, die Größe in kiB sowie das Cache-Kohärenz-Protokoll angegeben.
- Als letzten Eintrag kann für jede MasterComponent noch die Taktfrequenz in Hertz angegeben werden.
- Für die Systemkonfiguration kann schließlich auch noch eine Taktfrequenz in Hertz angegeben werden. Dieser Wert ist nur sinnvoll, wenn alle MasterCompenents der Plattform mit der gleichen Frequenz betrieben werden. Bei gemischten Systemen wie im Beispiel kann der Wert auf 0 gesetzt werden.

```
1  <?xml version="1.0" encoding="utf-8" standalone="yes"?>
2  <SystemConfiguration name="Nvidia Jetson TX2 Development Kit"
       shimVersion="1.0">
3    <ComponentSet name="Jetson TX2">
4      <ComponentSet name="Quad-Core ARM Cortex-A57 MPCore">
5        <MasterComponent name="Cortex-A57 #0" id="cortex-a57_0"
             masterType="PU" arch="armv8-a" archOption="class:processor,
             coretype:cortex-a57,gpu_control_core" pid="0" endian="LITTLE
             ">
6          <Cache name="" id="cortex-a57_0_l1_dcache" cacheType="DATA"
               cacheCoherency="HARD" size="32" sizeUnit="KiB"/>
7          <Cache name="" id="cortex-a57_0_l1_icache" cacheType="
               INSTRUCTION" cacheCoherency="HARD" size="48" sizeUnit="KiB
               "/>
8          <ClockFrequency clockValue="2000000000"/>
9        </MasterComponent>
10       <MasterComponent name="Cortex-A57 #1" id="cortex-a57_1"
             masterType="PU" arch="armv8-a" archOption="class:processor,
             coretype:cortex-a57" pid="3" endian="LITTLE">
11         <Cache name="" id="cortex-a57_1_l1_dcache" cacheType="DATA"
               cacheCoherency="HARD" size="32" sizeUnit="KiB"/>
```

```
12          <Cache name="" id="cortex-a57_1_l1_icache" cacheType="
                INSTRUCTION" cacheCoherency="HARD" size="48" sizeUnit="KiB
                "/>
13          <ClockFrequency clockValue="2000000000"/>
14        </MasterComponent>
15        <Cache name="" id="cortex-a57_l2_cache" cacheType="UNIFIED"
              cacheCoherency="HARD" size="2" sizeUnit="MiB"/>
16      </ComponentSet>
17      <ComponentSet name="Dual-Core Denver 2">
18        <MasterComponent name="Denver 2 #0" id="denver_0" masterType="PU"
               arch="armv8-a" archOption="class:processor,coretype:denver2
              " pid="1" endian="LITTLE">
19          <Cache name="" id="denver_0_l1_dcache" cacheType="DATA"
                cacheCoherency="HARD" size="64" sizeUnit="KiB"/>
20          <Cache name="" id="denver_0_l1_icache" cacheType="INSTRUCTION"
                cacheCoherency="HARD" size="128" sizeUnit="KiB"/>
21          <ClockFrequency clockValue="2000000000"/>
22        </MasterComponent>
23        <Cache name="" id="denver_l2_cache" cacheType="UNIFIED"
              cacheCoherency="HARD" size="2" sizeUnit="MiB"/>
24      </ComponentSet>
25      <ComponentSet name="Jetson-TX2 GPU">
26        <MasterComponent name="Jetson-TX2 GPU" id="jetson_tx2_gpu"
              masterType="PU" arch="pascal" archOption="class:gpu,
              lang:cuda" pid="6" endian="LITTLE">
27          <ClockFrequency clockValue="1300000000"/>
28        </MasterComponent>
29      </ComponentSet>
30      <SlaveComponent name="SDRAM" id="SDRAM" size="8" sizeUnit="GiB"
            rwType="RWX"/>
31    </ComponentSet>
32    <ClockFrequency clockValue="0"/>
33  </SystemConfiguration>
```

Quellcode 5.1: ADL eines NVIDIA® Jetson™ TX2

5.1.2. Performanzwerte

Neben der reinen Spezifikation des Aufbaus der Zielplattform sind noch Informationen zur Performanz der vorhandenen Ausführungseinheiten wichtig.

Dazu wird für jeden Prozessortyp einer Zielplattform ein Profiling aller Instruktionen durchgeführt und die Ergebnisse in einer JSON-Datei abgespeichert. Ein Beispiel ist in Quellcode 5.2 dargestellt. Hier werden für die Architektur (arch [Zeile 2]) ARMv8-A und den Coretyp „Cortex-A57“ [Zeile 3] ein Auszug der profilierten Instruktionen dargestellt. Alle Zahlenwerte sind in Anzahl Taktzyklen angegeben. Die Namen der Instruktionen setzen sich aus der Instruktion, beispielsweise „add“, der Bitbreite und einem nachgestellten „_fp“ für Fließkommaoperationen zusammen.

Durch das Messverfahren können drei verschiedene Fälle unterschieden werden:

1. Ganze Zahlenwerte wie 1 für „add16“ [Zeile 6] oder 2 für „add32_fp“ [Zeile 8] treten auf, wenn die Ausführungseinheit oder die ALU die entsprechende Anzahl an Taktzyklen zum Ausführen der Operation benötigt.
2. Relativ große Anteile von ganzen Zahlen, in der Größenordnung von Zehnteln, wie beispielsweise 0,5 für „add32“ [Zeile 7] treten auf, wenn die ALU mehrere dieser Instruktion gleichzeitig ausführen kann. Bei 0,5 können also 2 Instruktionen gleichzeitig ausgeführt werden. Aufgrund der Messmethodik gibt es Schwankungen bei diesen Zahlen, die durch Optimierungen des Compilers, Caches oder parallel ablaufenden Tasks auf dem Zielsystem auftreten können.
3. Relativ kleine Anteile von ganzen Zahlen in der Größenordnung von Hundertsteln, wie beispielsweise 0,04 bei „and16“ [Zeile 12] treten auf, wenn die Instruktion immer in Kombination mit anderen Instruktionen ausgeführt werden muss und selbst kaum zur Laufzeit beiträgt. Bei einer „and“-Instruktion dauert beispielsweise eine mögliche Set-Instruktion und die entsprechenden Speicherzugriffe verhältnismäßig länger. In Datenblättern von Prozessoren sind für diese Arten von Instruktionen dennoch Werte wie „ein Taktzyklus“ angegeben, die die Hardware tatsächlich benötigt. Für die Abschätzung der Performanz sind die ermittelten Werte aber sinnvoller, da der Einfluss auf die gesamte Ausführungszeit kleiner ist.

Noch ein Hinweis zu Kosten für Typumwandlungen, im Beispiel zu sehen in den Zeilen 24 bis 29. Angegeben ist immer ein Cast von einer Bitbreite zu einer anderen oder von Integer auf Fließkommazahlen. Ob durch einen Cast überhaupt eine weitere Operation ausgeführt werden muss oder nicht kann im Einzelfall ohne Kontext nicht entschieden werden und wird maßgeblich durch Compiler-Optimierungen beeinflusst. So kann ein Cast von 32 auf 64 Bit keinerlei zusätzliche Zeit benötigen, da die Variable bereits vorher mit 64 Bit gespeichert wurde. Casts mit 8 Bit hingegen benötigen in der Regel Zeit, da auf gängigen 32 und 64 Bit Systemen eine Operation benötigt wird, um die richtigen 8 Bit zu adressieren. Größere Kosten wie für `cast832_fp` können auftreten, wenn die Daten erst noch an die richtigen Stellen im neuen Datentyp kopiert werden müssen. Die Typumwandlungen sind ein wichtiges Thema, da der aus MATLAB® heraus generierte Code (siehe Abschnitt 4.3) in der Regel viele Casts beinhaltet. Die Nutzung von möglichst effizienten Datentypen mit geringerem Speicherverbrauch macht die Umwandlung von Datentypen notwendig.

```
1  "type": "static",
2    "arch": "armv8-a",
3    "coretype": "cortex-a57",
4    "default": 2,
5    "instructions": {
6      "add16": 1,
7      "add32": 0.5,
8      "add32_fp": 2,
9      "add64": 0.5,
10     "add64_fp": 2,
```

```
11      "add8": 1,
12      "and16": 0.04,
13      "and32": 0.04,
14      "and64": 0.04,
15      "and8": 0.05,
16      "call0": 1,
17      "call0_fp": 2,
18      "call16": 0.08,
19      "call32": 1,
20      "call32_fp": 3.5,
21      "call64": 1,
22      "call64_fp": 3,
23      "call8": 1,
24      "cast816": 1,
25      "cast832": 1,
26      "cast832_fp": 13,
27      "cast864": 1,
28      "cast864_fp": 13,
29      "cast88": 1,
30      "div16": 1,
31      "div32": 1,
32      "div32_fp": 2,
33      "div64": 1,
34      "div64_fp": 2,
35      "div8": 1
36    }
```

Quellcode 5.2: Beispiel für gemessene Laufzeiten von Instruktionen eines ARM Cortex-A57

5.2. Parallelisierung auf Algorithmus-Ebene

Die Parallelisierung auf Algorithmus-Ebene hat zum Ziel, bekannte Algorithmen zu erkennen, um eine Vorauswahl aus bereits existierenden optimierten Realisierungen treffen zu können, sowie den Code auf die spätere Anwendung von Code-Transformationen vorzubereiten.

Die Code-Übersetzung aus Abschnitt 4.3 dient als Vorbereitung für die späteren Parallelisierungen auf Code- und Task-Ebene, wird aber gleichzeitig auch für die Parallelisierung auf Algorithmus-Ebene verwendet. Dabei werden zwei Fälle unterschieden: Ausnutzung

von Eigenschaften von bekannten Algorithmen und die Verwendung unterschiedlicher Implementierungen für verschiedene Anwendungszwecke.

Durch Verwendung der zusätzlichen Abstraktion der array-basierten Programmiersprachen, können selbst komplexere Algorithmen als einfacher Funktionsaufruf mit einem klar definierten Verhalten verwendet werden. Dies erleichtert die Erkennung von Algorithmen ungemein, da Programmiersprachen wie C in der Regel keine allgemein eingesetzte Standardimplementierung haben, sondern meist verschiedene Implementierungen aus Bibliotheken oder komplette Eigenentwicklungen existieren. Algorithmen automatisch rein anhand des Quellcodes erkennen zu können, ist nahezu unmöglich, da für bekannte Algorithmen unzählige Implementierungen existieren können. Diese können sich in vielen Teilen unterscheiden, gemein ist nur das Verhalten der Anwendung. Das Verhalten von Algorithmen bzw. Implementierungen zu analysieren ist ein sehr komplexes Thema, und wie bereits das Halteproblem (siehe [95]) zeigt, kann für beliebige Algorithmen nicht einmal bestimmt werden, ob sie zu einem Ende finden oder nicht.

Die gesamte Erkennung bekannter Algorithmen basiert auf den in MATLAB® definierten Funktionen. Die Codegenerierung ist so aufgebaut, dass für jede Funktion eine MATLAB®-Datei als Grundlage für die Konvertierung verwendet wird. Alle Elemente des generierten C-Codes stammen somit ursprünglich aus bestehenden MATLAB®-Implementierungen. Zusammen mit dem im Abschnitt 4.3.2 beschriebenen Entscheidungsmechanismus können nun verschiedene Parallelisierungsstrategien auf Algorithmus-Ebene realisiert werden, indem MATLAB®-Funktionen durch speziell implementierte Versionen realisiert werden.

Die Erkennung eines bekannten Algorithmus während der Codegenerierung läuft wie folgt ab: wird ein Funktionsaufruf im Code erkannt, so wird die passende Funktion, die über den Namen und die Parameter eindeutig identifiziert werden kann, in der aktuellen Datei gefolgt vom aktuellen Verzeichnis und schließlich in der bereitgestellten Bibliothek gesucht. Liegen mehrere Realisierungen vor, so wird die zuerst gefundene Realisierung übernommen. Ein Nachweis, dass die Funktionalität aller Realisierungen immer identisch mit der tatsächlichen MATLAB®-Funktion ist, findet während der Codegenerierung nicht mehr statt und liegt auch außerhalb von dessen Aufgabenbereich. Bei der Entwicklung der Funktionsbibliothek können die existierenden MATLAB®-Funktionen mit ihren Dokumentationen als Referenz verwendet werden, mit der die Ergebnisse verglichen werden können. Die Implementierung in MATLAB® ist dabei abstrakt genug, so dass keine speziellen Optimierungen für den gewählten Prozessor angewendet werden können. Um sicherzustellen, dass der generierte C-Code Plattform-agnostisch ist, wird der generierte C-Code auf dem Host-System kompiliert, ausgeführt und die Ausgaben direkt mit der ursprünglichen Anwendung verglichen. Auch wenn die Kombination aus MATLAB®- und C-Code erlaubt ist, wird im hier vorgestellten Ansatz nur von MATLAB®-Skripten als Eingabe ausgegangen und es kommen keine speziell optimierten Implementierungen in C zum Einsatz. Als Parameter zur Optimierung des generierten Codes können die folgenden Optionen verwendet werden:

- Vorgabe der Anzahl unabhängiger Teile, in die der Algorithmus aufgeteilt werden soll. Dies kann als Optimierung für die Anzahl an Kernen des Zielsystems eingesetzt werden. Der hier generierte Code ist aber noch nicht parallelisiert, sondern

bietet mit unabhängigen Codeteilen die Möglichkeit zur Parallelisierung auf Task-Ebene.

- Anpassungen der eingesetzten Datentypen: standardmäßig kommen in MATLAB® Datentypen mit doppelter Genauigkeit (Double) zum Einsatz. Durch einfache Anpassungen können jedoch einfache Genauigkeiten oder sogar der Einsatz von ganzzahligen Datentypen erzwungen werden. Dies hat Auswirkungen auf die Genauigkeit der erzeugten Ergebnisse und muss im Einzelfall entschieden werden, ob eine geringere Auflösung noch ausreichend ist. Die Optimierung zielt auf die Eigenschaften des gewählten Prozessors ab, da nicht von der Existenz von Fließkommaeinheiten ausgegangen werden kann. Müssen diese über Software realisiert werden, so sind sie deutlich langsamer. Auch hier gilt, dass die Auswahl von anderen Datentypen immer noch generischen C-Code generiert und keine Beschränkung auf ein Zielsystem vorliegt.
- Einsatz rekursiver Funktionen: durch Rekursion lassen sich manche Algorithmen deutlich effizienter im Code darstellen und die Laufzeit optimieren. Abhängig von der tatsächlichen Verschachtelungstiefe kann es aber zu einem deutlich höheren Speicherbedarf kommen, so dass diese Option immer mit Hinblick auf den verfügbaren Speicher des Zielsystems betrachtet werden muss.

5.2.1. Ausnutzung von bekannten Algorithmen-Eigenschaften

Als Beispiel für die Ausnutzung der Eigenschaften von Algorithmen soll hier die MATLAB®-Funktion `fft`, die Berechnung der schnellen Fourier-Transformation (engl. *fast fourier transform*, abgekürzt FFT), dienen. Die Fourier-Transformation ist eine mathematische Berechnung, die dazu dient, eine Funktion im Zeitbereich in eine Funktion im Frequenzbereich umzuwandeln. Lineare Operationen in einem Bereich haben äquivalente Operationen im anderen Bereich, können aber wesentlich einfacher berechnet werden. Ein Anwendungsbeispiel ist die Bearbeitung von Audiodaten durch Filterfunktionen, die oft einfacher in der Frequenzdomäne dargestellt und berechnet werden können. Die zeit- und wertkontinuierliche Fourier-Transformierte muss für die Berechnung auf dem Computer in diskrete Werte zerlegt werden. Die schnelle Fourier-Transformation ist ein Algorithmus zur Berechnung der diskreten Fourier-Transformation.

Gemäß Handbuch kann die MATLAB®-Implementierung mit den Parametern `X`, `n` und `dim` aufgerufen werden, wobei `X` für die Daten als Vektor, Matrix oder mehrdimensionale Matrix, `n` für die Anzahl an Punkten und `dim` für die Dimension, entlang welcher die FFT berechnet werden soll, steht. Das Verhalten der Funktion ist demnach für verschiedene Größen der Eingangsdaten und der Anzahl an Punkten der FFT klar definiert. MATLAB® setzt dabei auf eine Implementierung von [96], die näher in [97] beschrieben wird. Für die automatische Übersetzung nach C-Code, wie in Abschnitt 4.3 beschrieben, wurde nun auf eine Referenzimplementierung gesetzt, die die gleiche Funktionalität wie die tatsächliche MATLAB®-Funktion besitzt. Der Entscheidungs-Mechanismus aus Abschnitt 4.3.2.3 kann nun innerhalb dieser Implementierung verwendet werden, um den Algorithmus mit mehreren FFTs mit weniger Punkten zu realisieren. Dazu wird eine Variable vom Typ Integer verwendet, die angibt, wie viele unabhängige FFTs generiert werden sollen. Der

zugehörige MATLAB®-Code ist in Quellcode 5.3 dargestellt. Die Integer-Variable zum Aufteilen auf die FFTs ist dabei in dem Eingangsparameter M der Funktion gespeichert. Der Algorithmus wird dabei in die folgenden Schritte aufgeteilt:

1. Neu-Anordnung des Eingangs: das zwei-dimensionale Eingangs-Array x wird zunächst in einer neuen Variable x2 gespeichert. Zusätzlich wird in Zeile 2 die Funktion an den Codegenerator übergeben, die Variable x2 in M Teile aufzuteilen. Dies wird benötigt, damit die FFTs mit weniger Punkten auch nur auf Teilmengen des Arrays arbeiten müssen.
2. FFT über die Reihen: Als nächstes werden M FFTs auf den Reihen der Variablen x2 ausgeführt. Da x2 bereits in M Teile aufgeteilt wurde, kann diese Berechnung nun auch auf M unabhängige Aufrufe aufgeteilt werden.
3. Berechnung der Twiddle-Faktoren: Unter „Twiddle-Faktor“ versteht man bei der Berechnung von FFTs alle trigonometrischen konstanten Koeffizienten, mit denen die Daten während des Algorithmus multipliziert werden. Aufgrund ihrer Struktur ist es dabei einfacher, sie zu Beginn der Anwendung ein Mal zu berechnen anstatt sie direkt konstant in den Quellcode zu schreiben. Bei dieser aufgeteilten FFT werden die Faktoren ebenfalls in sinnvolle Untermengen aufgeteilt, so dass jede Teil-FFT nur einen kleineren Bereich von allen benötigt.
4. FFT über die Spalten: Anschließend werden N FFTs über die Spalten des Arrays berechnet.
5. Neu-Anordnung der Ausgangsvariablen: im letzten Schritt werden die Ergebnisse in x2 wieder in einer einzelnen Matrix y gesammelt, die gleichzeitig auch als Ausgang der Funktion gilt.

Eine schematische Darstellung der Aufteilung der Daten der FFT ist in Abbildung 3.2 dargestellt.

```
1  function y = fftifft_decomposition(x, M, inverse)
2    %CMD?: cgen_var_split(x2, M, 1)
3
4    P = length(x);
5    N = P/M;
6
7    % step 1: Re-arrange input.
8    x2 = complex(zeros(M,N));
9    x2(:) = x(:);
10
11   % step 2: FFT-N on rows of U.
12   for m = 1:M
13     x2(m,:) = fftifft(x2(m,:), inverse);
14   end
15
16   % step 3: Twiddle factors.
17   %CMD?: cgen_perf_initcode()
18   twiddels = fftifft_decomposition_init(M, N, inverse);
19   x2 = x2 .* twiddels;
20
```

```
21    % step 4:  FFT-M on columns of U.
22    for n = 1:N
23      x2(:,n) = fftifft(x2(:,n), inverse);
24    end
25
26    % step 5:  Re-arrange output.
27    y = complex(zeros(size(x)));
28    for m = 0 : M-1
29      for n = 0 : N-1
30        y(m*N+n+1) = x2(m+1,n+1);
31      end
32    end
33  end
```

Quellcode 5.3: Realisierung der FFT Dekomposition

Um die Eigenschaften von bekannten Algorithmen zu unterstützen, muss demnach wie folgt vorgegangen werden:

1. Implementierung des Algorithmus in MATLAB®, so dass die identische Funktionalität gewährleistet wird.
2. Anpassen des Algorithmus, um die Eigenschaften anhand von Parametern einstellbar zu machen
3. Umsetzung der Parameter als Entscheidungs-Variablen, um die Auswahl erst auf einer späteren Ebene treffen zu können.

5.2.2. Auswahl von Realisierungsalternativen

Ähnlich wie bereits in Abschnitt 5.2.1 dargestellt, wurde auch die Auswahl der Realisierungsalternativen umgesetzt. Hintergrund ist, dass die gleiche Funktionalität auf unterschiedliche Arten implementiert werden können, die sich hinsichtlich der Parallelisierung oder der Performanz auf dem Zielsystem stark unterscheiden können. Mit Hilfe des Entscheidungs-Mechanismus kann nun die Auswahl der Realisierung über Optionen zu einem späteren Zeitpunkt erfolgen. Alle unterstützten MATLAB®-Funktionen werden durch Implementierungen in der Sprache MATLAB® realisiert. Dabei können einzelne Funktionen auf verschiedene Art und Weise realisiert werden; eine Entscheidungs-Variable kann dann eingesetzt werden, um die passende Realisierung auszuwählen. Ein Beispiel ist in Quellcode 5.4 dargestellt. Die Funktion `sort` ist dabei eine Hüllenfunktion, bei der abhängig von der Entscheidungsvariablen p (Zeile 5) ein unterschiedlicher instabiler Sortieralgorithmus ausgewählt wird. Im Beispiel ist der Code für Quicksort (Zeilen 18 bis 27), Selectionsort (Zeilen 29 bis 44) und Combsort (Zeilen 48 bis 68) aufgelistet. Alle drei Algorithmen liefern nach der Ausführung einen sortierten Vektor als Ergebnis. Aufgrund der Instabilität der Algorithmen ist nicht sichergestellt, dass die Reihenfolge von gleichwertigen Elementen nach dem Sortieren beibehalten wird. Bei reiner Ausführung mit Zahlen hat dies jedoch keine Auswirkungen auf das Ergebnis.

Die drei Algorithmen unterscheiden sich in der Realisierung im grundlegenden Ablauf: Quicksort setzt auf rekursive Funktionsaufrufe, die auf Teilmengen angewendet werden.

Selectionsort arbeitet mit zwei verschachtelten For-Schleifen, bei denen in jeder äußeren Iteration das jeweils kleinste Element gesucht wird und an den Anfang des Vektors geschrieben wird. Combsort ist ein von Bubblesort abgeleiteter Algorithmus, der in einer Schleife jeweils 2 Elemente mit einem Abstand (gap) miteinander vergleicht und bei Bedarf vertauscht. In jeder Iteration wird der Abstand verkleinert bis ein letzter Durchgang mit einem Abstand von 1 ausgeführt wird. Durch den immer kleineren Abstand kann sichergestellt werden, dass zuerst grob falsch einsortierte Elemente näher an die Zielposition gebracht werden und erst später mit kleinerem Abstand an die richtige Stelle gebracht werden.

```
function x = sort(x)

    %CMD?: p = cgen_decision(2, 'sort', 'Sorting algorithm', 'int', '
        select', 1, 3);

    switch p
      case 1
        x = quicksort(x);
      case 2
        x = selectionsort(x);
      case 3
        x = combsort(x);
      otherwise
        disp('no valid sorting algorithm selected')
    end

end

function [vector] = quicksort(vector)
    if numel(vector) <= 1 % vectors with one or less elements are
        sorted
        return
    else
        pivot=vector(1);
        vector = [ quicksort( vector(vector < pivot))...
                   vector(vector == pivot)...
                   quicksort(vector(vector > pivot)) ];
    end
end

function x = selectionsort(x)
    n = length(x);
    for j = 1:(n - 1)
        % Find jth smallest element
        imin = j;
        for i = (j + 1):n
            if (x(i) < x(imin))
                imin = i;
            end
        end

        % Put jth smallest element in place
```

```
41          if (imin ~= j)
42              x = swap(x,imin,j);
43          end
44      end
45
46  end
47
48  function x = combsort(x)
49      shrink = 1.3; % shrink > 1
50
51      n = length(x);
52      gap = n;
53      swapped = true;
54      while ((gap > 1) || (swapped == true))
55          % Update gap
56          gap = max(floor(gap / shrink),1);
57
58          % Bubble sort with given gap
59          i = 1;
60          swapped = false;
61          while ((i + gap) <= n)
62              if (x(i) > x(i + gap))
63                  x = swap(x,i,i + gap);
64                  swapped = true;
65              end
66              i = i + 1;
67          end
68      end
69  end
```

Quellcode 5.4: Auswahl an Sortieralgorithmen

Die Algorithmen haben unterschiedliche Anforderungen an die Ressourcen bei der Ausführung. Selectionsort und Combsort sind so genannte in-place Algorithmen, das bedeutet, sie benötigen bei der Ausführung keinen weiteren Speicher zur Zwischenspeicherung von Daten. Quicksort hat aufgrund der rekursiven Realisierung einen Speicherbedarf von $\mathcal{O}(n \log n)$, wobei n die Anzahl Elemente im Vektor ist. Dafür unterscheiden sie sich in der Laufzeit. Selectionsort hat eine konstante Laufzeit von $\mathcal{O}(n^2)$, während Combsort im Idealfall nur $\mathcal{O}(n \log n)$ benötigt. Quicksort braucht sowohl im Idealfall als auch im Durchschnitt $\mathcal{O}(n \log n)$ und nur im schlechtesten Fall $\mathcal{O}(n^2)$.

Das in Quellcode 5.4 dargestellte Beispiel kann verwendet werden, um die in MATLAB® integrierte Funktion `sort` zu realisieren. Wird die Funktion verwendet, ohne weitere Einstellungen zu tätigen, so wird für die Variable p eine 2 angenommen und somit ein Selectionsort realisiert. Die Optimierungen während der Codegenerierung, die in Abschnitt 4.3.3 beschrieben werden, sorgen dann dafür, dass sämtlicher nicht genutzter Code nicht im C-Code erscheint und dass alle konstanten Werte propagiert werden. Die generierte Funktion `sort` als C-Code ist in Quellcode 5.5. Da die Entscheidungsvariable einen konstanten Wert von 2 hat, werden alle anderen Fälle des Switchs unnötig und können entfernt werden. Dadurch wird der Switch unnötig und es bleibt rein ein Aufruf von `selectionsort`

übrig. Nur in den generierten Kommentaren und in der Beibehaltung des Pragmas für die Entscheidung können Teile des ursprünglichen MATLAB®-Codes erkannt werden. Auf diese Weise kann sichergestellt werden, dass immer möglichst effizienter C-Code generiert wird, aber in einer späteren Ebene dennoch die Entscheidung über den tatsächlich eingesetzten Algorithmus getroffen werden kann. Möglich ist dies beispielsweise auf der Task-Ebene, in der bereits ein Hardware-Modell zur Abschätzung der Laufzeit von einzelnen Codeteilen vorhanden ist. Mit dieser Information kann die beste aus den möglichen Realisierungen für die ausgewählte Zielplattform ausgewählt werden.

```
1  void sort(int32_t x_data[6]) {
2    // custom_sort.m(3:12-84):  p = cgen_decision(2, 'sort', 'Sorting
         algorithm', 'int', 'select', 1, 3);
3    #pragma CGEN_DECISION 2.0 "main.custom_sort.sort" "Sorting algorithm"
          "int" "select" 1.0 3.0
4
5    // custom_sort.m(5-12):  switch p
6
7    // custom_sort.m(7:7-28):  x = selectionsort(x);
8
9    selectionsort(x_data);
10 }
```

Quellcode 5.5: Sortierfunktion mit Standardwerten

Um verschiedene Realisierungen von Algorithmen bereitzustellen, müssen die folgenden Schritte durchgeführt werden:

1. Implementierung aller Realisierungsalternativen in MATLAB®.
2. Implementierung einer Hüllenfunktion mit Namen der MATLAB®-Funktion, die anhand einer Entscheidungsvariablen die entsprechende Realisierung auswählt.
3. Anwenden der beschriebenen Codegenerierung, um C-Code zu bekommen, der keinen Overhead durch die Entscheidungsmöglichkeiten beinhaltet.

5.2.3. Erfüllung der Anforderungen aus Abschnitt 3.1.1.1

A1 Anforderung A1 wird durch den Ansatz über die Codegenerierung von MATLAB®-Code kommend vollständig erfüllt. Funktionsaufrufe werden in der existierenden Codegenerierungsbibliothek gesucht und werden durch Funktionsname und die Parameter eindeutig bestimmt. Die Datentypen müssen dabei nicht spezifiziert werden, so dass erst bei der Konvertierung der passende Code erzeugt wird. Fehlerhafte Erkennung von anderen Algorithmen ist ausgeschlossen, da in MATLAB® Funktionsnamen nicht mehrfach vergeben werden dürfen. Diese Eindeutigkeit findet sich auch in der Codegenerierung wieder, da nur wenn die passende Funktion gefunden wird, überhaupt erst Code generiert werden kann. Die Codegenerierungsbibliothek vereint alle Algorithmen-Implementierungen, die optimiert oder

nicht optimiert wurden. Befindet sich keine Implementierung eines Algorithmus in ihr, so wird kein C-Code generiert. Somit gilt:

$$Bib = A_{opt} \cup A_{unopt} \tag{5.1}$$

$$alg_{unterstuetzt} \in A_{opt} \vee alg_{unterstuetzt} \in A_{unopt} \tag{5.2}$$

A2 Ein formaler Beweis für die Erfüllung von Anforderung 2 ist nicht möglich, da hier zwei Algorithmen miteinander verglichen werden müssen, die nicht zwingend die identischen Ergebnisse liefern. Ist schon der Vergleich von zwei Algorithmen, die zwar die gleichen Ergebnisse erzeugen, jedoch grundsätzlich anders implementiert sind, schon nicht im allgemeinen Fall lösbar, so wird dies durch einen unscharfen Vergleich noch komplexer. Eine allgemeine Möglichkeit zum Vergleich der Algorithmen kann über Black-Box-Tests erfolgen. Dazu wird der C-Code für die zu vergleichenden Implementierungen generiert und mit identischen Optimierungen kompiliert. Ohne Betrachtung der Implementierung, aber mit dem Wissen über die Funktionalität des Algorithmus können nun Eingangsdaten definiert werden, mit denen Ergebnisse zum Vergleich erzeugt werden sollen. Dabei hat sich die Analyse der Grenzwerte als zuverlässige Methode bewährt. Dabei wird für jeden Eingangswert der Wertebereich bestimmt und sowohl das Minimum, Maximum und soweit zutreffend der Wert 0 und jeweils ein Wert größer und kleiner 0 betrachtet. Über alle Eingangswerte hinweg können nun beide Algorithmen mit den Wertekombinationen aufgerufen und die Ergebnisse entsprechend verglichen werden. Über die Unterschiede der Werte beider Algorithmen kann der Genauigkeitsunterschied bestimmt werden. Sollten sich Ergebnisse für einzelne Werte grundsätzlich unterscheiden, so sind die Implementierungen nicht automatisiert im vorgestellten Verfahren einsetzbar, da die grundsätzliche Vergleichbarkeit vor und nach der Optimierung fehlt.

A3 Anforderung A3 wird durch das Konzept der Entscheidungsvariablen vollständig erfüllt. Sie können nachträglich durch Entscheidungen auf späteren Ebenen angepasst werden und erlauben somit eine Optimierung an die Anforderungen der späteren Ebenen. Durch die Integration in die Codegenerierung kann zudem sichergestellt werden, dass unnötiger Code, der bei bestimmten Werten der Variablen nicht mehr erreicht werden kann, bereits vor Erzeugung des C-Codes entfernt wird und nicht weiter betrachtet werden muss.

A4 Anforderung A4 wird durch die Art der Realisierung bereits erfüllt: alle Optimierungen auf Algorithmus-Ebene werden durch Implementierungen in MATLAB® sowie bei der Codegenerierung umgesetzt. In MATLAB® ist generell keine hardwarenahe Programmierung möglich, da Zugriffe auf einzelne Register und Assembler-Befehle nicht Teil der Sprache sind. Bei der Codegenerierung selbst wird reiner C-Code generiert, der auf jeder Plattform lauffähig ist. Teil der Konvertierung ist zudem eine Ausführung des generierten Codes auf dem Host-System, welcher sicherstellt, dass kein spezifischer Code für die Zielplattform zum Einsatz kommt.

5.3. Parallelisierung auf Code-Ebene

Die Parallelisierung auf Code-Ebene hat zum Ziel, den vorliegenden Quellcode so aufzubereiten, dass die nachgelagerte Parallelisierung auf Task-Ebene mehr Möglichkeiten für eine Aufteilung von Tasks auf Ausführungseinheiten bekommt.

Als Codetransformation wird im Rahmen dieser Arbeit eine Veränderung am Quellcode bezeichnet, die die Darstellung und die Reihenfolge von Berechnungen verändern kann, ohne dabei die Funktionalität des ursprünglichen Codes zu verändern. Ziel einer Transformation ist es, durch eine neue Darstellung oder Reihenfolge der Instruktionen mehr Freiheitsgrade bei der Parallelisierung oder durch eine Anpassung an die gegebene Hardware eine schnellere Ausführung zu erreichen.

Wie bereits in Abschnitt 3.2 dargestellt, ist eine automatische Exploration nur schwer realisierbar, da eine sehr große Komplexität durch Kombination von Transformationen erreicht werden kann. Im Rahmen dieser Arbeit sollen Transformationen also manuell ausgewählt und miteinander kombiniert werden. Zur Exploration werden dabei zwei Arten unterstützt, wie Transformationen ausgewählt werden können: zum einen soll die Auswahl über Pragmas im Quellcode erfolgen können. Damit ist sichergestellt, dass die Transformation auch wirklich an den passenden Positionen aufgerufen wird. Dies ermöglicht einfacheres Testen und auch eine vereinfachte Kombination von Transformationen. Nach der Auswahl sollen Transformationen automatisch angewendet werden können. Des Weiteren können Transformationen in einer zusätzlichen JSON-Datei mit den Elementen, auf die sie angewendet werden sollen, gespeichert werden. Die Datei hat den Vorteil, dass sie programmatisch mit verschiedenen Daten gefüllt werden kann und somit die schnelle Exploration von unterschiedlichen Optionen oder Kombinationen an Transformationen ermöglicht.

5.3.1. Das Transformations-Framework

Zur automatischen Verarbeitung der ausgewählten Transformationen wurden Transformationen über ein Java-Interface definiert und anschließend von einem Transformations-Handler angewendet.

5.3.1.1. Das Transformations-Interface

Über Java-Interfaces können Methoden festgelegt werden, die von jeder Transformation implementiert sein müssen. Auf diese Weise können alle Transformationen auf die gleiche Art und Weise von einer weiteren Klasse aufgerufen werden. Zusätzlich kommt noch das Konzept von so genannten „Extension Points“ in Eclipse zum Einsatz. Unter einem Extension Point versteht man ein Interface, das auf der Ebene von Eclipse-Plugins definiert wird. Dazu wird das Interface in einem Eclipse-Plugin spezifiziert, indem die Eigenschaften in einer XML-Datei gespeichert werden. Um das Interface zu implementieren, müssen Java-Klassen neben der Realisierung noch in einer XML-Datei auf Plugin-Ebene registriert werden. Dies ermöglicht es, zur Laufzeit alle aktiven Plugins zu finden, die einen bestimmten Extension Point realisieren. Auf diese Weise können neue Transforma-

tionen in Form von Eclipse-Plugins einfach zu einer bestehenden Installation hinzugefügt oder entfernt werden, ohne dass Abhängigkeiten dafür sorgen, dass ein Plugin zwingend erforderlich wird.

Die Methoden des Interfaces sind wie folgt definiert:

Analyze
: Die Methode `analyze` hat als Parameter einen Block, für den analysiert werden soll, ob die Transformation auf den Block angewendet werden kann oder nicht. Sollte es nicht möglich sein, gibt die Methode `null` zurück. Ansonsten gibt sie ein Objekt der Klasse „TransformationInformation“ zurück. Die wichtigste Information dieses Objekts sind die Parameter zur Konfiguration einer Transformation. Sie können von den Typen Integer, Float, Boolean oder String sein und können abhängig von der Transformation verwendet werden, um einzelne Parameter der Transformation anzupassen. Jeder Parameter hat neben dem Datentyp noch einen definierten Standardwert. Die `analyze`-Methode wird für jeden Block der Anwendung aufgerufen und sollte möglichst schnell `null` zurückgeben, wenn klar ist, dass eine Transformation nicht angewendet werden kann.

ShouldApply
: Die Methode `shouldApply` überprüft, ob eine Transformation angewendet werden soll. Dies kann im einfachsten Fall bedeuten, dass die gerade gesetzten Parameter mit den Standardwerten verglichen werden und bei Abweichung die Transformation angewendet werden soll. Bei komplexeren Transformationen kann die Methode aber auch dafür verwendet werden, zeitaufwendige Analysen aus der `analyze` Methode herauszuhalten und somit die Benutzbarkeit von Transformationen zu verbessern.

Apply
: Die Methode `apply` wird verwendet, um die tatsächliche Transformation anzuwenden. Als Parameter bekommt sie ein Objekt der Klasse `ApplyInformation`, welches eine erweiterte Version der Klasse `TransformationInformation` ist. Sie unterscheidet sich durch gesetzte Parameter und durch Angabe eines root-Blocks, der den Anfang der Transformation bildet. Sollte sich die Anzahl der Blöcke durch eine Transformation verändern, so muss ein neuer Block als Container für alle neuen generiert werden und als neuer root-Block in der Klasse angegeben werden.

5.3.1.2. Auswahl von Transformationen

Ähnlich wie die Auswahl von Implementierungsalternativen über den Entscheidungsmechanismus (siehe Abschnitt 4.3.2.3) erfolgt auch die Auswahl von Codetransformationen. Dazu wird nach der erfolgreichen Analyse eines Blocks die Information aus der TransformationInformation als Annotation am entsprechenden Block gespeichert. Die Parameter haben das gleiche Format wie der Entscheidungsmechanismus, so dass die Parameter auf die gleiche Art und Weise über Werte aus einer JSON-Datei oder über die grafische Oberfläche gesetzt werden können. Dabei kommt die grafische Darstellung eines hierarchischen Task Graph (siehe Abschnitt 2.3.2.3) zum Einsatz. Ein Beispiel ist in Abbildung 5.1 dargestellt. Auf der X-Achse ist die Zeit dargestellt, wobei die Breite jedes Blocks durch

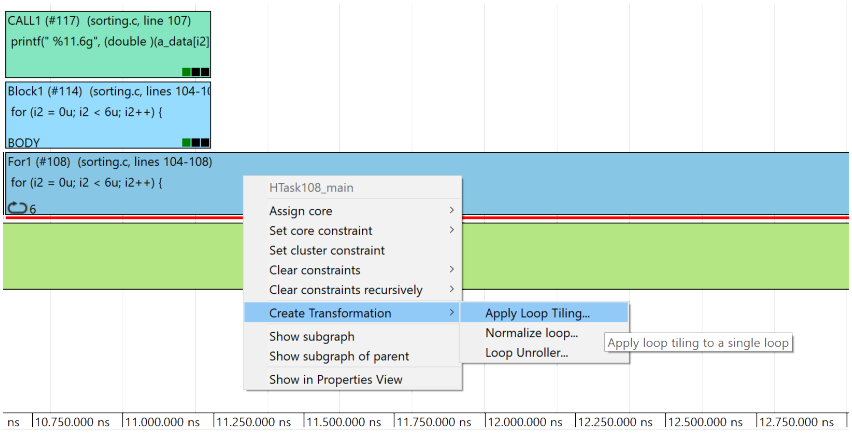


Abbildung 5.1.: Auswahl einer Transformation

eine Performanzabschätzung wie in Abschnitt 2.4 beschrieben bestimmt wurde. Auf der Y-Achse ist die Kontrollstruktur der Anwendung dargestellt. Sie setzt sich aus Funktionsaufrufen (Calls) aber auch aus den Kontrollstrukturen For-/While-Schleifen, If-Blöcken, Switch-Cases und Kompositionsblöcken zusammen. In der oberen rechten Ecke wird ein Icon angezeigt, wenn mögliche Transformationen erkannt wurden. Per Kontextmenü sind in diesem Fall Transformationen und ihre Konfigurationen auszuwählen.

Alle getroffenen Einstellungen werden in einer JSON-Datei gespeichert. Ein Beispiel ist in Quellcode 5.6 dargestellt. Im Eintrag ACTIVE_TRAFOS werden die folgenden Informationen zu jeder Transformation gespeichert:

- Die ORIGINAL_ID gibt in einer eindeutigen Nummer die Reihenfolge aller aktiven Transformationen an. Beginnend mit der 0 werden alle Transformationen in dieser Reihenfolge angewendet.
- Die BLOCK_ID gibt die eindeutige Bezeichnung des Blocks an, auf den die Transformation angewendet werden soll. Der Eintrag ergibt sich durch den Funktionsnamen sowie sämtlicher Kontrollstrukturen bis zum ausgewählten Block. Die ID aus Zeile 6 bezieht sich beispielsweise auf den For-Block mit der Nummer 5, der sich in Composite-Block (CB) Nummer 2 in der Funktion `main` befindet. Da Blöcke auch erst durch eine andere Transformation erzeugt werden können, wird zusätzlich zum Block noch der Eintrag TRAFO verwendet, um eine ORIGINAL_ID einer anderen Transformation zu speichern. Wird der Eintrag nicht benötigt, so wird der Wert auf -1 gesetzt. Die Reihenfolge, wie Transformationen angewendet werden, kann prinzipiell noch nachträglich angepasst werden, solche Abhängigkeiten zwischen Blöcken und Transformationen verhindern aber beliebige Vertauschungen.
- Über PARAMS werden alle Parameter der Transformation sowie deren Werte gespeichert. In Zeile 9 wird beispielsweise mit der booleschen Variable „ifSplitLoop“ angegeben, ob die Transformation angewendet werden soll oder nicht.

- NAME gibt schließlich den einzigartigen Namen der Transformation an.

Nach den aktiven Transformationen gibt der Eintrag BUILD (im Beispiel in Zeile 22) mit Hilfe eines gespeicherten Hash-Werts den genutzten Stand des ursprünglichen Quellcodes an. Wird dieser verändert, so sind die Blöcke unter Umständen nicht mehr gültig, da eine eindeutige Identifikation wegen Änderungen an den Kontrollstrukturen nicht mehr möglich ist.

```
1  {
2      "ACTIVE_TRAFOS": [
3          {
4              "ORIGINAL_ID": 0,
5              "BLOCK_ID": {
6                  "BLOCK": "##main.CB#2.For#5",
7                  "TRAFO": -1
8              },
9              "PARAMS": {"ifSplitLoop": true},
10             "NAME": "IfSplit"
11         },
12         {
13             "ORIGINAL_ID": 1,
14             "BLOCK_ID": {
15                 "BLOCK": "##main.CB#2.For#3",
16                 "TRAFO": -1
17             },
18             "PARAMS": {"doFission": true},
19             "NAME": "Fission"
20         }
21     ],
22     "BUILD": "B81363F4A32F958B"
23 }
```

Quellcode 5.6: Konfigurationsdatei bei Transformationen in JSON

Neben der Auswahl über eine grafische Oberfläche und der damit automatischen Speicherung der Einstellungen in einer JSON-Datei, können Transformationen auch direkt über Pragmas im Quellcode angewendet werden. In Quellcode 5.7 werden die beiden gleichen Transformationen wie in der JSON-Datei aus Quellcode 5.6 angewendet. Die Pragmas gelten dabei immer für den darauffolgenden Block, in diesem Beispiel also für die beiden For-Schleifen in den Zeilen 13 und 21. Dabei haben die drei Pragmas die folgenden Bedeutungen:

- Über `CMD_TRAFO_SHOULD_APPLY` wird über eine boolesche Variable angegeben, ob die Transformation ausgeführt werden sollte oder nicht. Das Pragma wird verwendet, um in automatisierten Tests zu überprüfen, ob die Transformation tatsächlich ausgeführt wurde. Kann sie das nicht, weil entweder die Analyse ermittelt, dass sie nicht angewendet werden kann oder kommt es bei der Transformation zu Fehlern, so schlägt der entsprechende Testfall fehl. Umgekehrt gilt für einen Wert von 0, dass die Transformation nicht angewendet werden soll, da sie von der Logik her nicht anwendbar ist.

- `CMD_TRAFO_BLOCK_NAME` kann verwendet werden, um der Transformation auf den gewählten Block einen Namen zu geben. Auf diese Weise kann die Ausführung besser vom Anwender nachverfolgt werden.
- `CMD_TRANSFORMATION` schließlich spezifiziert die anzuwendende Transformation mit ihren Parametern.

```
1  #include <stdio.h>
2
3  int main() {
4    int i, j, suma, sumb;
5    int a[8], b[8];
6
7    suma = 0;
8    sumb = 0;
9
10 #pragma CMD_TRAFO_SHOULD_APPLY 1
11 #pragma CMD_TRAFO_BLOCK_NAME for_fission
12 #pragma CMD_TRANSFORMATION Fission {"doFission": true}
13   for (i = 0; i < 8; i++) {
14     a[i] = i * i;
15     b[i] = i;
16   }
17
18 #pragma CMD_TRAFO_SHOULD_APPLY 1
19 #pragma CMD_TRAFO_BLOCK_NAME for_if
20 #pragma CMD_TRANSFORMATION IfSplit {"ifSplitLoop": true}
21   for (j = 0; j < 8; j++) {
22     if (j < 4) {
23       suma += a[j];
24     } else {
25       sumb += b[j];
26     }
27   }
28
29   printf("suma = %d, sumb = %d\n", suma, sumb);
30 }
```

Quellcode 5.7: Transformationen über Pragmas im Quellcode

5.3.1.3. Anwenden von Transformationen

Codetransformationen werden von der Klasse `TransformationHandler` angewendet. Dabei wird jede Transformation einzeln angewendet, wobei zunächst Transformationen aus Pragmas vor den Transformationen aus der JSON-Datei abgearbeitet werden. Die Abarbeitung einer Transformation erfolgt in den folgenden Einzelschritten:

1. In einer ersten Analysephase wird überprüft, ob die gewählte Transformation auch angewendet werden kann. Dazu wird die Funktion `analyze` des Transformations-Interfaces (siehe Abschnitt 5.3.1.1) aufgerufen, um alle Parameter als Transforma-

tions-Information zu extrahieren. Anschließend werden die Parameter mit den gesetzten Werten überschrieben und zusammen mit dem ausgewählten Block als Apply-Information an die Apply-Methode übergeben. Die Analyse kann an dieser Stelle fehlschlagen, wenn Änderungen am Quellcode durchgeführt wurden, die den zu transformierenden Block entfernt oder verschoben haben oder wenn die Anwendung durch andere Transformationen zu stark verändert wurde.

2. Die ausgewählte Transformation wird mit den gesetzten Parametern auf den selektierten Block angewendet. Üblicherweise sollte sie dabei nur den Block und alle darin enthaltenen Konstrukte verändern, aber auch Änderungen an den äußeren Blöcken sind möglich. Da Transformationen Teile der Anwendung grundlegend verändern können, also Blöcke, Variablen oder andere Ausdrücke hinzufügen oder entfernen können, können davon die Darstellungen des Kontrollflusses und der Datenabhängigkeiten in der SSA-Darstellung zerstört werden. Da es sehr aufwendig ist, dies in den einzelnen Transformationen selbst wieder aufzubauen, kann dem Transformations-Handler per Flag übergeben werden, ob einer oder beide Darstellungen nach Anwendung der Transformation wieder aufgebaut werden sollen. Zusätzlich wird dem Handler auch der neue Root-Block mitgeteilt, falls der ursprüngliche Block entfernt wurde und nicht mehr Teil des Programms ist.
3. In einer zweiten Analysephase nach dem Anwenden aller Transformationen werden für alle Blöcke alle analyze-Methoden aller registrierten Transformationen angewendet. Die erzeugten Transformations-Informationen werden verwendet, um Blöcke zu annotieren, bei denen die entsprechenden Transformationen angewendet werden können. Dies kann in einem späteren Schritt für die grafische Darstellung ausgewertet werden.

5.3.2. Transformationen während der Codegenerierung

Als Alternative neben dem Transformations-Framework können Transformationen auch während der Codegenerierung von MATLAB® nach C durchgeführt werden. Die während der Konvertierung durchgeführten Analysen hinsichtlich Datentypen und Datenfluss sowie die optimierte Darstellung mit verschachtelten For-Schleifen bieten eine gute Grundlage, um auch an dieser Stelle Code-Transformationen anzuwenden. Die Auswahl erfolgt über Pragmas im Eingangs-Code und über Entscheidungsvariablen (siehe Abschnitt 4.3.2.3) können die Parameter der Transformationen noch weiter angepasst werden.

5.3.3. Realisierung beispielhafter Transformationen

Im Folgenden werden einige Transformationen vorgestellt, die den Code für eine bessere Parallelisierung auf Task-Ebene vorbereiten können. Es soll nur eine Auswahl darstellen, die bei der Evaluation des Ansatzes auf der Hardware angewendet werden können.

Loop-Unroll (Schleifen entrollen)

Analyze: Während der Analysephase wird sichergestellt, dass die Anzahl an Iterationen sowie die Schrittweite der Zählvariablen für die gewählte For-Schleife bestimmt

werden kann. Zusätzlich darf die Vergleichsoperation nur eine der vier Basisvergleiche sein: kleiner (<), kleiner gleich (<=), größer (>) oder größer gleich (>=). Komplexerer Kontrollfluss durch Ausdrücke wie `break` oder `continue` oder komplexere Abhängigkeiten zwischen Schleifeniterationen werden nicht unterstützt und die Analyse bricht die Transformation ab. Ist dies alles gegeben, wird ein Parameter für den Abrollfaktor erzeugt, der als Integer-Wert von 1 bis zur maximalen Anzahl der Iterationen reicht.

ShouldApply: Die Transformation wird angewendet, wenn der Abrollfaktor auf einen Wert größer als 1 gesetzt ist.

Apply: Nach der Extraktion aller konstanten Faktoren wie Schrittweite, Teilbarkeit von Abrollfaktor und Anzahl Schleifeniterationen sowie positiver oder negativer Zählrichtung wird die ursprüngliche Schleife transformiert. Dazu wird der Rumpf der Schleife in Höhe des Abrollfaktors vervielfältigt und alle Referenzen auf die ursprüngliche Schleifenvariable werden um konstante Faktoren von eins bis zum Abrollfaktor erhöht. Ist die Anzahl an Iterationen nicht durch den Abrollfaktor teilbar, wird eine weitere Schleife generiert, die alle restlichen Iterationen beinhaltet.

Loop-Fission (Schleifenspaltung)

Analyze: Zunächst wird die Schleife hinsichtlich komplexen Kontrollflusses mittels `break` und `continue` sowie Datenabhängigkeiten über Schleifeniterationen hinweg betrachtet. Liegt dies nicht vor, werden die Ausdrücke des Rumpfblocks in Untermengen eingeteilt, die keine Abhängigkeiten untereinander haben. Nur wenn mindestens zwei Untermengen existieren, wird ein boolescher Parameter erzeugt, mit dem die Transformation aktiviert werden kann.

ShouldApply: Die Transformation wird angewendet, wenn der boolesche Parameter aktiviert wurde.

Apply: Die in der Analysephase identifizierten Untermengen des Rumpfblocks werden als Basis für die Erzeugung von neuen For-Blöcken verwendet. Für jede Untermenge wird eine neue Schleife generiert, die nur die Blöcke und Instruktionen der Untermenge im neuen Rumpf beinhaltet. Der Schleifenkopf jeder Schleife ist identisch und entspricht dem Kopf der ursprünglichen Schleife.

Loop-Tiling

Analyze: Die Analyse beschränkt sich rein auf For-Schleifen, für deren Iterationsvariable gelten muss, dass sie durch einen klar definierten Anfangswert sowie einen Maximalwert vollständig beschrieben wird und dass der Iterator immer mit einer Obergrenze auf kleiner oder kleiner-gleich überprüft wird. In so einem Fall sind die Anzahl und die Reihenfolge sämtlicher Iterationen klar definiert und ein Loop-Tiling kann angewendet werden. Als Parameter wird eine Integer-Variable generiert, die die Tiling-Größe und damit die Anzahl an Iterationen der inneren Schleife von 1 bis zur Schleifengrenze annehmen kann.

ShouldApply: Als neutrales Element dieser Transformation kann ein Wert von 1 angenommen werden. Wird dies für alle Dimensionen angegeben, so wird kein Tiling angewendet.

Apply: Die Apply-Methode generiert eine neue äußere For-Schleife, deren Grenze sich über $\frac{Iterationen_{Total}}{Iterationen_{innen}}$ berechnet. Zusätzlich wird eine innere Schleife generiert, deren Grenze bis zur gesetzten Variable geht. Sollte sich die Anzahl an Iterationen der ursprünglichen Schleife nicht ganzzahlig durch die Iteration der inneren Schleife teilen lassen, so wir eine zweite innere Schleife generiert, die alle übrigen Iterationen gemäß $Anzahl_{Tiles} * Anzahl_{InnereIterationen} * Schrittweite + Grenze_{unten}$ beinhaltet.

Variablen-Splitting Die Aufspaltung von Variablen ist eine Transformation, die in die Konvertierung von MATLAB® nach C integriert ist. Sie hat als Ziel, eine Dimension einer Matrixvariablen um einen Faktor n aufzuteilen. Aus einer Matrix `A[20]` werden mit Faktor $n = 4$ die Matrizen `A_1[5]`, `A_2[5]`, `A_3[5]` und `A_4[5]`. Alle lesenden und schreibenden Zugriffe auf die ursprüngliche Variable werden analysiert und durch Zugriffe auf die neuen Matrizen ersetzt. Voraussetzung für eine effiziente Umsetzung ist dabei die vollständige Analysierbarkeit sämtlicher Zugriffe. Denn ist nicht eindeutig bestimmbar, welcher Teil der ursprünglichen Matrix adressiert wird, da beispielsweise für einen Ausdruck wie `A[i]` statisch kein Wert für `i` bestimmt werden kann, so muss der Zugriff durch einen aufwendigeren bedingten Zugriff ersetzt werden:

```
1 if (i < 5)
2   A_1[i];
3 else if (i < 10)
4   A_2[i - 5];
5 else if (i < 15)
6   A_3[i - 10];
7 else
8   A_4[i - 15];
```

Das Variablen-Splitting entfaltet das größte Potential für eine Parallelisierung der Anwendung in Kombination mit der Schleifenspaltung. In Quellcode 5.8 ist dargestellt, wie das Ergebnis dieser Transformationen bei der Matrixmultiplikation aussieht. Die ursprüngliche Variable `c` (Zeile 2) wird in die Variablen `c_0` (Zeile 17) und `c_1` (Zeile 18) aufgeteilt. Der Zugriff in Zeile 12 wird entsprechend der Zählvariable `i2` aufgeteilt. Die Schleifenspaltungs-Transformation kann nun verwendet werden, um die äußerste Schleife in zwei unabhängige Schleifen umzuwandeln. Alle Zugriffe, die über `i2` erfolgen sind linear und können klar für `c_0` und `c_1` bestimmt werden, so dass der für die Parallelisierung optimierte Code ab Zeile 36 dargestellt wird. Beide Schleifen (Zeile 37 und Zeile 48) benötigen lesenden Zugriff auf die Variablen `a` und `b`, können ansonsten aber parallel ausgeführt werden. Abhängig von der weiteren Verwendung von Matrix `c` kann es notwendig sein, alle Daten wieder in einer einzelnen Variable zusammen zu führen. Dieser mögliche Overhead ist ab Zeile 60 dargestellt.

```
1 /* Ohne Transformationen */
2 double c[10][10];
3
4 for (i2 = 0; i2 < 10; i2++) {
5   for (i1 = 0; i1 < 10; i1++) {
6     sum1 = 0.0;
```

```
7
8       for (i3 = 0; i3 < 10; i3++) {
9         sum1 += a[i3][i1] * b[i2][i3];
10      }
11
12      c[i4][i3] = sum1;
13    }
14  }
15
16  /* Variablen-Splitting */
17  double c_0[5][10];
18  double c_1[5][10];
19
20  for (i2 = 0; i2 < 10; i2++) {
21    for (i1 = 0; i1 < 10; i1++) {
22      sum1 = 0.0;
23
24      for (i3 = 0; i3 < 10; i3++) {
25        sum1 += a[i3][i1] * b[i2][i3];
26      }
27
28      if (i2 <= 4) {
29        c_0[i2][i1] = sum1;
30      } else {
31        c_1[i2 - 5][i1] = sum1;
32      }
33    }
34  }
35
36  /* Variablen-Splitting und Schleifenspaltung */
37  for (i7 = 0; i7 < 5; i7++) {
38    for (i6 = 0; i6 < 10; i6++) {
39      sum2 = 0.0;
40
41      for (i8 = 0; i8 < 10; i8++) {
42        sum2 += a[i8][i6] * b[i7][i8];
43      }
44
45      c_0[i7][i6] = sum2;
46    }
47  }
48  for (i2 = 5; i2 < 10; i2++) {
49    for (i1 = 0; i1 < 10; i1++) {
50      sum1 = 0.0;
51
52      for (i3 = 0; i3 < 10; i3++) {
53        sum1 += a[i3][i1] * b[i2][i3];
54      }
55
56      c_1[i2 - 5][i1] = sum1;
57    }
58  }
59
```

```
60  for (i10 = 0; i10 < 5; i10++) {
61    for (i9 = 0; i9 < 10; i9++) {
62      c[i10][i9] = c_0[i10][i9];
63    }
64  }
65  for (i5 = 5; i5 < 10; i5++) {
66    for (i4 = 0; i4 < 10; i4++) {
67      c[i5][i4] = c_1[i5 - 5][i4];
68    }
69  }
```

Quellcode 5.8: Matrixmultiplikation mit Variablen-Splitting und Schleifenspaltungs-Transformationen

5.3.4. Erfüllung der Anforderungen aus Abschnitt 3.1.2.1

C1 Aufgrund des Einsatzes von mathematisch beschreibbaren Algorithmen, können alle Ergebnisse vor und nach Anwendung von Code-Transformationen verglichen werden. Alle Werte müssen identisch sein, damit sichergestellt ist, dass die Transformationen keine ungültigen Veränderungen am Code vornehmen. Werden bei einer Transformation keine Codeteile kopiert oder die Reihenfolge verändert, so kann ein formaler Vergleich der Programme über die interne Zwischendarstellung eines Compilers wie Clang erfolgen. Sie kann nach den internen Compiler-Optimierungen mit dem im LLVM-Projekt bereitgestellten Tool *llvm-diff* durchgeführt werden. Ohne Verdopplung oder Verschieben von Codeteilen muss der Kontrollfluss sowie die interne Darstellung nach Optimierungen identisch sein.

C2 Der formale Ansatz aus Anforderung C1 kann ebenfalls eingesetzt werden, um nachzuweisen, dass durch die Transformation keine weiteren Fehler hinzugekommen sind. Sollte dies aufgrund von gravierenden Änderungen am Code (z.B. Reihenfolge, Code-Duplikation, ...) nicht möglich sein, so können Black-Box-Tests wie bereits in Abschnitt 5.2.3 dargestellt, Abhilfe schaffen.

C3 Anforderung C3 wird durch das in Abschnitt 5.3.1.1 vorgestellte Interface erfüllt. Alle Optionen der Transformationen werden von außen bereitgestellt und können iterativ nach Auswertung auf einer späteren Stufe angepasst werden.

5.4. Parallelisierung auf Task-Ebene

Die automatische Parallelisierung auf Task-Ebene ist kein Hauptthema in dieser Arbeit, weswegen nur ein einzelner Algorithmus vorgestellt werden soll, der für die allgemeine Parallelisierung eingesetzt wurde. Es wurde dabei kein neuer Algorithmus entwickelt, sondern der wohlbekannte Heterogeneous Earliest Finish Time (HEFT)[98] Algorithmus eingesetzt. Dieser wurde aber gemäß den Anforderungen für diese Arbeit durch eine Möglichkeit erweitert, einzelne Tasks bereits vorab auf bestimmte Kerne festzulegen. Die-

ses Verhalten wird benötigt, um Entscheidungen bereits auf anderen Ebenen treffen zu können.

5.4.1. Grundlagen HEFT-Algorithmus

Der HEFT-Algorithmus ist ein sogenannter Greedy-Algorithmus, der nicht darauf abzielt, die optimale globale Lösung zu finden, sondern immer das nächste lokale Optimum anstrebt. Als Optimierungsziel wird die Minimierung der Ausführungszeit der parallelen Anwendung angegeben. Dabei werden die Zeiten der sequentiellen Anwendung über statische Code-Analyse ermittelt, wie es in Abschnitt 2.4 dargestellt wurde. Der Algorithmus geht wie folgt vor:

1. Bestimmung des Rangs von jedem Task. Je niedriger der Rang ist, desto eher wird ein Task einem Kern zugewiesen. Der Rang wird aus der Dauer eines Tasks sowie den Abhängigkeiten zu anderen Tasks berechnet.
2. Erstellung einer Liste aller Tasks, die bereit sind. Bereit bedeutet, dass alle Tasks, die vorher ausgeführt werden müssen, bereits zugewiesen sind.
3. Zuweisung des Tasks mit dem niedrigsten Rang.
4. Aktualisierung aller Abhängigkeiten

Nach dem letzten Schritt wird wieder mit dem ersten weitergemacht bis alle Tasks zugewiesen sind.

5.4.2. Anpassungen des HEFT-Algorithmus

Im Rahmen dieser Arbeit soll die Parallelisierung auf Task-Ebene evaluiert werden, weshalb ein rein automatischer Algorithmus nur nützlich wäre, wenn er auch das Optimum berechnen würde. Als Alternative wurde der HEFT-Algorithmus um die Möglichkeit erweitert, nutzerbasierte Randbedingungen vorzugeben. Dabei kann der Scheduler mit den folgenden Restriktionen eingeschränkt werden:

- Zuweisung: ein einzelner Task kann fest einem bestimmten Kern zugewiesen werden. Der Scheduler kann noch die Reihenfolge der Tasks auf dem Kern abhängig von den Datenabhängigkeiten verändern, aber den Task nicht mehr auf einem anderen Kern ausführen lassen.
- Ausschluss: mit dieser Einschränkung können einzelne Kerne von der automatischen Zuweisung ausgeschlossen werden. Der Scheduler kann den Task bzw. dessen Kinder noch beliebig auf die restlichen Kerne verteilen.
- Gruppierung/Cluster: diese Einschränkung sorgt dafür, dass der gewählte Task und alle Kinder wie ein einzelner Task ohne Kinder behandelt werden. Auf diese Weise kann ungewollte Parallelisierung verhindert werden.

Alle Einschränkungen gelten immer für den ausgewählten hierarchischen Task und alle darin enthaltenen Tasks. Die Einschränkungen können mit den bereits bewährten Methoden gesetzt werden: entweder über Pragmas direkt im Quellcode oder über die Auswahl

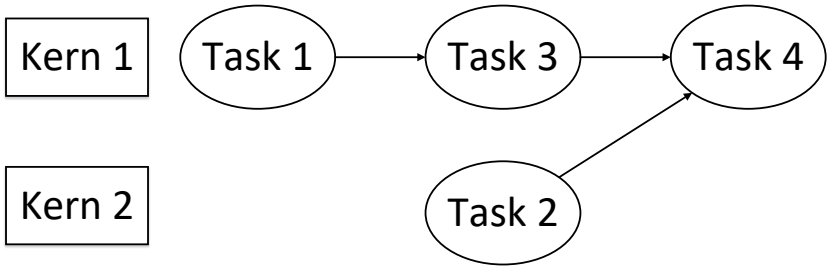


Abbildung 5.2.: Suboptimales Mapping mit dem HEFT-Algorithmus

innerhalb der grafischen Darstellung. Alle gesetzten Einschränkungen werden persistent in einer JSON-Datei gespeichert, so dass sie auch zwischen Aufrufen verfügbar bleiben.

5.4.3. Einschränkungen des HEFT-Algorithmus

Als Greedy-Algorithmus ist der HEFT nicht darauf ausgelegt, den optimalen Schedule zu bestimmen. Die schrittweisen Entscheidungen, die den besten Schedule unter Betrachtung des nächsten Tasks bestimmen, haben den Nachteil, dass spätere Tasks, die von der aktuellen Entscheidung abhängen, nicht beachtet werden. Ein Beispiel ist in Abbildung 5.2 dargestellt. Gemäß der Rang-Berechnung wird Task 1 auf Kern 1 ausgeführt. Task 2 hat keine Abhängigkeit zu Task 1 und wird deshalb auf Kern 2 gemappt. Task 3 hat eine Abhängigkeit zu Task 1 und wird ebenfalls Kern 1 zugewiesen. Schließlich benötigt Task 4 Daten sowohl von Task 2 als auch von Task 3. Abhängig von den Kosten für die Abhängigkeiten wird Task 4 nun Kern 1 oder 2 zugewiesen. In jedem Fall wird es einen Overhead zum Transfer der Daten geben. Gilt für diesen: $overhead < (T_1 + T_2 + T_3 - max(T_1 + T_3, T_2))$, so wird ein Speedup erreicht, ansonsten wird das Programm langsamer ablaufen als im sequentiellen Fall. Der Effekt wird durch setzen manueller Kernzuweisungen noch verschärft, da Vorgänger des fixierten Tasks im Idealfall ebenfalls auf dem gleichen Kern ausgeführt werden sollten, so dass manuelle Zuweisungen nur mit Bedacht eingesetzt werden sollten.

Eine mögliche Lösung liegt in der Bildung von Gruppen, indem zu Beginn des HEFT-Algorithmus bereits Abhängigkeiten zwischen Tasks betrachtet werden und Tasks mit vielen Datenabhängigkeiten bevorzugt auf dem selben Kern ausgeführt werden. Da die automatische Parallelisierung auf Task-Ebene aber nicht Ziel dieser Arbeit ist, sondern nur als Hilfsmittel eingesetzt wird, um die Tasks effizient auf die Kerne zu verteilen, wird dies im weiteren Verlauf dieser Arbeit nicht weiterverfolgt. Ineffiziente Parallelisierung kann durch manuellen Eingriff und Setzen von eigenen Randbedingungen behoben werden.

5.4.4. Scheduling mit Simulated Annealing

Aufgrund der Einschränkungen des HEFT-Algorithmus, wird als Alternative der bewährte Simulated Annealing Algorithmus [30] eingesetzt. Er ist ein heuristischer Ansatz, der bevorzugt wird, wenn die Anzahl aller möglichen Lösungen zu groß ist, um sie einzeln zu betrachten.

Die Grundidee besteht darin, einen Abkühlungsprozess zu modellieren, bei dem die Teilchen bevorzugt in energiearme Zustände wie Gitterstrukturen übergehen. Mit abnehmender Temperatur wird die Beweglichkeit der Teilchen immer geringer, bis sie schließlich in einem Minimalzustand zur Ruhe kommen. Übertragen auf das Optimierungsproblem bedeutet es, dass zu Beginn größere Sprünge von einer Lösung zur nächsten erlaubt sind und somit lokale Minima durch temporäre Verschlechterung wieder verworfen werden können, um schließlich näher an das globale Minimum zu kommen.

Zur Erzeugung von Lösungen des Optimierungsproblems wird der Metropolis-Algorithmus [99] eingesetzt. Er erlaubt es, Zustandsübergänge gemäß der Boltzmann-Verteilung zu generieren, wie sie bei thermischen Prozessen auftreten. Die so entstehende Abfolge von Zufallsvariablen kann nun verwendet werden, um iterativ Lösungen zu generieren, die jeweils auf dem vorherigen Schritt basieren.

Um das Scheduling zu optimieren, wurde die Dauer der parallelen Ausführungszeit als Kosten angenommen, die es zu minimieren gilt. Eine Lösung weist jedem Task einen Kern zu, soweit dies nicht bereits durch manuelle Auswahl vorher erfolgt ist. Datenabhängigkeiten zwischen Tasks verursachen einen Kommunikationsoverhead, der bei der Bewertung einer Lösung hinzugezogen wird.

Da dieses Verfahren im Gegensatz zum HEFT-Algorithmus nicht 'greedy' agiert, können systematische Fehler durch fehlendes Vorausschauen verhindert werden. Gerade bei Anwendungen, die aus mehr als 1000 Tasks bestehen, können dort in der Regel deutlich bessere Lösungen generiert werden, die mit dem HEFT-Algorithmus nur durch viele manuelle Zuweisungen möglich wären.

5.4.5. Erfüllung der Anforderungen aus Abschnitt 3.1.3.1

T1 Die Minimierung der Laufzeit kann als Optimierungsalgorithmus modelliert werden. Jede Entscheidung zur Zuweisung eines Kerns E_{Kern} und eines Zeitpunkts $t_{n,start}$ für einen Task T_n ist mit Kosten $C_{overhead}$ für den entstehenden Kommunikationsoverhead und C_{delay} für anfallende Verzögerungen anderer Tasks verknüpft. Die Gesamtzeit für einen Kern setzt sich dann wie folgt zusammen:

$$T_{Kern} = \sum_{i=1}^{n_{Kern}} t_i(E_{Kern}, t_{start}) = \sum_{i=1}^{n_{Kern}} (t_{i,seq} + C_{overhead}(E_{Kern}) + C_{delay}(t_{i,start})) \quad (5.3)$$

Die vorgestellten Optimierungsverfahren sind Heuristiken und können auch Ergebnisse erzeugen, deren Ausführungszeit höher als die Referenzzeit der sequentiellen Ausführung ist. Beim HEFT-Algorithmus passiert dies, da immer nur die aktuell beste Entscheidung getroffen wird, ohne Abhängigkeiten in der Zukunft zu

beachten. Beim Simulated Annealing werden schlechtere Ergebnisse bis zu einem bestimmten Wert akzeptiert. Mit der errechneten Ausführungszeit liegt aber eine feste Metrik vor, mit der keine Verschlechterungen der sequentiellen Zeit akzeptiert werden müssen und im schlechtesten Fall keine Parallelisierung durchgeführt wird. All dies setzt aber voraus, dass die Zeiten und Kosten, die als Eingangsdaten für die Optimierung verwendet werden, das tatsächliche Verhalten auf der Hardware hinreichend genau beschreiben. In Abschnitt 6.1.2.1 wird ein Verfahren vorgestellt, wie die Ergebnisse der Parallelisierung auf Task-Ebene mit Hilfe einer anderen Modellierung verifiziert werden können.

T2 Auf Task-Ebene werden keine Veränderungen am Quellcode oder den Implementierungen durchgeführt. Die einzigen Auswirkungen auf die korrekte Funktionalität können fehlerhafte Vertauschungen oder eine falsch synchronisierte Verteilung auf die verfügbaren Kerne sein. Beides setzt ein korrektes Modell voraus, das sicherstellt, dass alle Daten- und Kontrollabhängigkeiten zwischen allen Tasks vorhanden sind. Es muss also gelten:

- **T2.1** Für Variablen müssen alle Lese- und Schreibzugriffe bekannt sein.
  Dies ist gewährleistet, wenn der komplette Quellcode der Anwendung vorliegt und keine indirekten Zugriffe auf Variablen über ihre Adressen bzw. Pointer erfolgen, die durch arithmetische Operationen berechnet werden.
- **T2.2** Der Kontrollfluss der Anwendung muss eindeutig bestimmbar und korrekt modelliert sein.
  Analog zu Variablen gilt für den Kontrollfluss, dass der Quellcode vollständig vorliegen muss, um alle Funktionen analysieren zu können. Alle Sprungziele im Programm müssen ebenfalls bestimmbar sein, so dass mehrdeutige Funktionspointer nicht erlaubt sind.

T3 Für die korrekte Ausführung des parallelen Programms muss jeder Task, der innerhalb der Anwendung aufgerufen wird und damit Teil des Kontrollflusses ist, mindestens einen gültigen Prozessor zugewiesen bekommen. Zusammen mit T2 kann so eine korrekte Ausführung sichergestellt werden. Nach der Parallelisierung auf Task-Ebene wird für jeden Task diese Zuweisung überprüft. Zuweisung Z_T ist gültig für Task T, wenn einer der folgenden Fälle zutrifft:

a) $Z_T \in P_{core} = \{0, \ldots, n-1\}$, wobei n die Anzahl der Kerne des Zielprozessors ist.

b) $Z_T = P_{core}$ für Funktionen, die nicht parallelisiert werden, aber von allen Prozessoren aus aufgerufen werden können.

c) $Z_T \subset P_{core}$, wenn der Task anschließend für die parallele Ausführung aufgeteilt wird.

d) $Z_T \nsubseteq P_{core}$, ausschließlich dann, wenn der Task nicht Teil des Kontrollflusses ist und somit bei der Ausführung niemals erreicht werden kann.

5.5. Parallelisierung auf Daten-Ebene

Die Parallelisierung auf Daten-Ebene soll hier in zwei separaten Teilen diskutiert werden. Die reine Parallelisierung des Datenflusses in diesem Kapitel bezieht sich auf die Analyse von Datenabhängigkeiten im Programm, die bis auf die Kernzuweisungen unabhängig von der tatsächlichen Zielarchitektur geschehen kann. Die weitere Optimierung der Kommunikation unter Einbezug der Zielhardware und der Implementierung der Anwendung ist Teil der Optimierung des generierten Codes für die Zielplattform wie in Abschnitt 4.4 dargestellt.

Einer der wichtigsten Faktoren für den Overhead bei der Parallelisierung ist das Verteilen des Datenflusses auf mehrere Kerne. Der Datenfluss ist im Task-Graph durch Abhängigkeiten zwischen Tasks gekennzeichnet, die die Übertragung oder Synchronisation einer Variablen erfordern. Wie bereits in Abschnitt 3.1.4 dargestellt, liegt der Fokus dieser Arbeit auf Systemen mit verteiltem Speicher.

Die Aufgaben der Parallelisierung des Datenflusses können wie folgt zusammengefasst werden:

- Vervielfältigung aller Variablen, die von mehr als einem Kern benötigt werden. Da Kerne nur auf eigene Variablen zugreifen können, müssen alle mehrfach verwendeten Variablen auf allen Kernen verfügbar sein.
- Analyse des Datenflusses des Programms, um die Lese- und Schreibzugriffe auf Variablen zu erkennen, für die Kommunikation im Programm eingefügt werden muss. Dazu müssen die Datenabhängigkeiten zwischen Tasks betrachtet und alle Abhängigkeiten, die über Kerne hinweg gehen, extrahiert werden.
- Die extrahierten Abhängigkeiten müssen durch Einfügen der entsprechenden Funktionen zur Synchronisation bzw. dem Datenaustausch zwischen Kernen aufgelöst werden.

Da für die Analyse bereits eindeutige Zuweisungen von Tasks auf Kerne vorliegen müssen, kann diese Art der Optimierung erst nach der Parallelisierung auf Task-Ebene durchgeführt werden. Dabei ist die vorliegende Darstellung des Programms direkt nach der Parallelisierung auf Task-Ebene zu betrachten: Alle Tasks der Anwendung wurden vom Scheduler in einer festen Reihenfolge den verfügbaren Ausführungseinheiten zugewiesen. Das Programm befindet sich jedoch noch in einer quasi-sequenziellen Darstellung, in der das Programm noch auf einem einzelnen Kern ausgeführt werden könnte. Beim Traversieren des Kontrollflusses können daher Tasks besucht werden, die sich auf unterschiedlichen Kernen befinden und deren Startzeiten nicht in chronologischer Reihenfolge zur globalen Zeit stehen müssen.

Kommunikation umfasst den Datenaustausch und die Synchronisation zwischen unterschiedlichen Recheneinheiten. Datenaustausch wird benötigt, wenn Daten auf einem Kern geschrieben und auf einem anderen verwendet werden. Dies kann in der SSA-Darstellung (siehe Abschnitt 2.3.2.4) recht einfach überprüft werden: wird die gleiche Version einer Variablen auf einem anderen Prozessor gelesen als sie geschrieben wurde, so wird ein expliziter Datentransfer benötigt. Es existieren verschiedene Ansätze, wie diese Kommunikation im Programm platziert werden kann. Im Folgenden werden Kommunikations-

modelle und Zugriffsmuster hinsichtlich ihrer Auswirkungen auf die Entscheidungen bei der Kommunikation dargestellt.

Die Optimierung der Kommunikation und Synchronisation auf der Daten-Ebene ist entscheidend, um die korrekte Ausführung der parallelen Anwendung durch Vermeidung von Race Conditions und Deadlocks zu gewährleisten. Zusätzlich ist die optimale Platzierung wichtig, um die Parallelität, die auf den vorherigen Ebenen herausgearbeitet wurde, als Performanzgewinn auf die Hardware zu bringen.

5.5.1. Begriffserklärungen zur Kommunikation

Im Folgenden werden die im weiteren Verlauf verwendeten Definitionen zu Kommunikationsmodellen und Zugriffsmustern eingeführt.

5.5.1.1. Kommunikationsmodelle

Man differenziert zwischen synchronen und asynchronen Übertragungen und kann bei Unterscheidung zwischen Senden und Empfang die folgenden Kommunikationsmodelle definieren:

Synchrone Übertragung
: Bei der synchronen Übertragung sind Sender und Empfänger gleichzeitig mit der Datenübertragung beschäftigt. Die Daten werden nicht direkt in einen Puffer auf Empfängerseite geschrieben, sondern müssen aktiv entgegengenommen werden. Dies stellt die härtesten Anforderungen an das Scheduling der Kommunikationsinstruktionen. Da beide beteiligten Prozessoren gleichzeitig an der Kommunikation beschäftigt sind, bedeutet jede Verzögerung auf einem Prozessor die gleiche Verzögerung auf dem anderen Prozessor. Die Art der Kommunikation sorgt zudem dafür, dass die beiden betroffenen Ausführungseinheiten mit der Kommunikation direkt synchronisiert werden.

Asynchrone Übertragung
: Bei der asynchronen Übertragung wird die Kommunikation in der Regel durch Hardware unterstützt, so dass die Kommunikation unabhängig von der Last der Prozessoren erfolgen kann. Gebräuchlich ist der Einsatz von DMA-Einheiten, die die Daten von den Speichern lesen und schreiben können. Zusätzlich sorgen Puffer auf Empfangsseite dafür, dass die DMA-Einheiten nicht exakt zur gleichen Zeit mit dem gleichen Transfer beschäftigt sein müssen. Um eine korrekte Ausführung zu gewährleisten, werden noch sogenannte Barrieren im Code platziert. Sie geben die Zeitpunkte an, an denen die Übertragung erfolgreich beendet wurde. Auf Senderseite dürfen die gesendeten Daten erst nach der Barriere wieder verändert werden, auf Empfangsseite können empfangene Daten erst nach der Barriere sicher verwendet werden.

Hybride Übertragung
: Die hybride Übertragung ist eine Mischung aus synchroner und asynchroner Übertragung. Eine verbreitete Implementierung setzt auf Puffer auf Empfangsseite, die

dafür sorgen, dass kleinere Datenmengen, die in den Empfangspuffer passen, gesendet werden können, ohne dass der Empfänger aktiv die Daten entgegennehmen muss. Daten können also quasi asynchron gesendet werden. Wird die Empfangsanweisung nun erst nach dem Start der Übertragung ausgeführt, kann der Empfänger die Daten ohne weitere Verzögerung verwenden.

Sollte die Datenmenge die Kapazität des Empfangspuffers überschreiten, so fällt die Kommunikation auf eine synchrone Übertragung zurück, da der Sender darauf warten muss, dass der Empfänger die Daten aktiv entgegennimmt. Ein ähnliches Verhalten ergibt sich, wenn die Empfangsanweisung vor Beginn des Sendevorgangs ausgeführt wird. In diesem Fall muss der Empfänger auf die Daten des Senders warten und es ergibt sich wieder eine synchrone Übertragung.

Im Rahmen dieser Arbeit wird immer von einer hybriden Übertragung ausgegangen.

5.5.1.2. Zugriffsmuster

Für die Platzierung der Kommunikation sind die Anzahl an Lese- und Schreiboperationen auf eine Variable wichtig. Sie werden im Folgenden als Zugriffsmuster bezeichnet und werden in der Schreibweise „Anzahl Definitionen zu Anzahl Verwendungen“ dargestellt.

Für eine Lösung des Platzierungsproblems wird im Folgenden der Begriff der *Komposition* eingeführt. Sie ist definiert als eine Menge von Positionen:

$$K = \{p \mid p \in P\} \tag{5.4}$$

Dabei ist P die Menge aller Positionen oder Stellen im Kontrollfluss des Programms. In einfachen Fällen besteht eine Komposition aus einer einzelnen Position, bei verzweigten Kontrollstrukturen werden jedoch mehrere Positionen benötigt, um sicherzustellen, dass die notwendigen Transfers auf jedem Pfad ausgeführt werden. Bei Auswahl einer Komposition müssen die entsprechenden Kommunikationsinstruktionen in allen darin enthaltenen Positionen eingefügt werden. Verallgemeinert gilt für eine gültige Komposition:

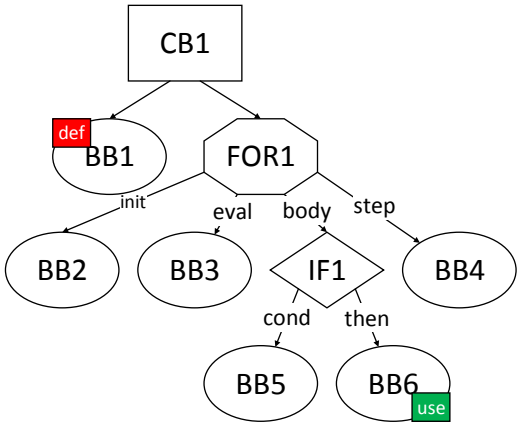
- $D_{s,n}$ sei die Menge aller Definitionen von Variable s auf Prozessor n.
- $U_{s,m}$ sei die Menge aller Verwendungen von s auf Prozessor m mit $m \neq n$.
- $\forall d \in D_{s,n}$, $\forall u \in U_{s,m}$: auf allen Pfaden von d nach u des CFG muss eine Position $p \in K$ liegen.

Einfache Definition mit einfacher Verwendung 1-zu-1 Zugriffe sind die einfachste Form und bilden die Basis für alle weiteren Zugriffe. Eine Definition auf einem Prozessor und eine Verwendung auf einem anderen bedeutet, dass genau ein Transfer ausreichend ist, um diese Abhängigkeit zu erfüllen. Eine gültige Komposition für einen 1-zu-1 Zugriff ist unter Einbezug der Dominanzrelation folgendermaßen definiert:

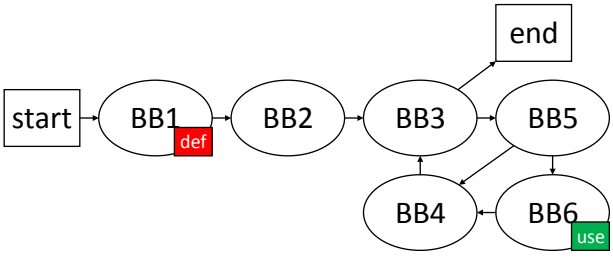
- d sei die Definition auf Prozessor n

- u sei die Verwendung auf Prozessor m ($m \neq n$)
- P sei die Menge der Positionen
- Komposition $K \subseteq P$
- $\forall p \in K, d\ dom\ p \wedge p\ dom\ u$

Eine beispielhafte Problemstellung ist in Abbildung 5.3 dargestellt: die Definition einer Variablen erfolgt in BB1, die Verwendung in BB6. Wie anhand des vereinfachten Syntaxbaums in Abbildung 5.3a zu erkennen ist, befindet sich BB6 in einer bedingten Verzweigung, die sich in einer For-Schleife befindet. Alle möglichen Pfade von BB1 nach BB6 können mit Hilfe des Kontrollflussgraphen in Abbildung 5.3b ermittelt werden. In diesem Fall existiert nur ein einzelner Pfad: $BB1 \rightarrow BB2 \rightarrow BB3 \rightarrow BB5 \rightarrow BB6$. Für eine gültige Komposition muss in diesem Fall genau eine Position auf dem Pfad von BB1 nach BB6 eingefügt werden.



(a) Vereinfachter abstrakter Syntaxbaum



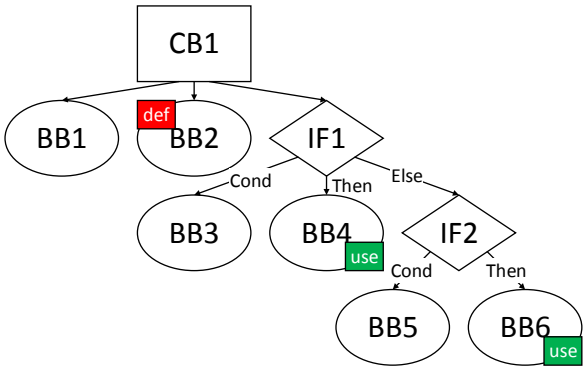
(b) Kontrollflussgraph

Abbildung 5.3.: Darstellung einer einfachen Definition mit einfacher Verwendung

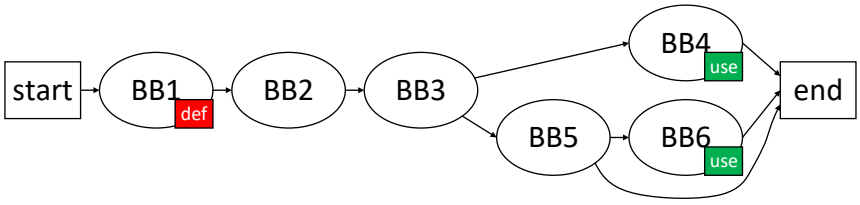
Einfache Definition mit mehrfacher Verwendung Wird eine Variable einmal auf einem Prozessor definiert und anschließend mehrfach auf einem anderen Prozessor verwendet, spricht man von einem 1-zu-n Zugriff. Es kann als n 1-zu-1 Zugriffe betrachtet werden. Für eine gültige Komposition unter Einbeziehung der Dominanzrelation gilt Folgendes:

- d sei die Definition auf Prozessor n
- U sei die Menge aller Verwendungen auf Prozessor m ($m \neq n$)
- P sei die Menge der Positionen
- Komposition $K \subseteq P$
- $\forall p \in K,\ d\ dom\ p \wedge \forall u \in U : p\ dom\ u$

Ein Beispiel ist in Abbildung 5.4 dargestellt. Die Variable wird in BB2 definiert und in BB4 und BB6 verwendet. Wie im CFG in Abbildung 5.4b besser zu erkennen ist, befinden sich die beiden Verwendungsstellen auf unterschiedlichen Pfaden, so dass Kommunikation so eingefügt werden muss, dass beide Pfade abgedeckt werden.



(a) Vereinfachter abstrakter Syntaxbaum



(b) Kontrollflussgraph

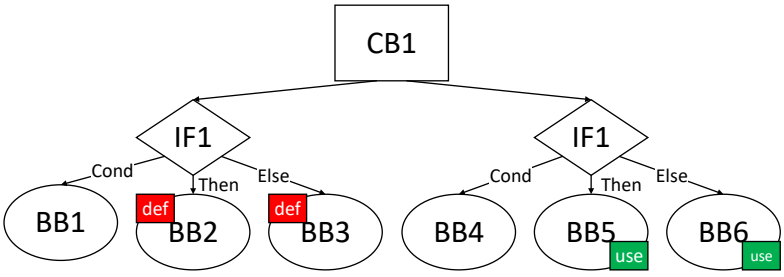
Abbildung 5.4.: Darstellung einer einfachen Definition mit mehrfacher Verwendung

Mehrfache Definition mit mehrfacher Verwendung Als verallgemeinerten Fall kann der m-zu-n Zugriff betrachtet werden. Er kann als m 1-zu-n Zugriffe betrachtet werden und

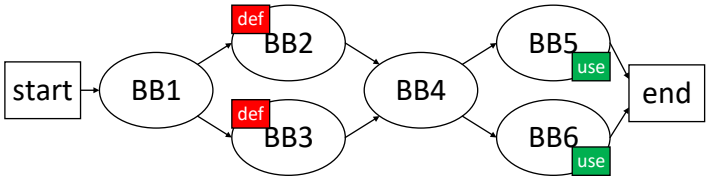
besagt, dass auf einem Prozessor eine Variable mehrfach definiert wird und auf einem anderen Prozessor mehrfach verwendet wird. Für eine gültige Komposition gilt:

- D sei die Menge an Definitionen einer Variablen s auf Prozessor n
- U sei die Menge aller Verwendungen einer Variablen s auf Prozessor m ($m \neq n$)
- $D_i \subseteq D, U_i \subseteq U$ seien so definiert, dass $\nexists(u,d) : u \prec d \wedge u \in U_i \wedge d \in D_i$
- P sei die Menge der Positionen
- Komposition $K \subseteq P$
- $\forall p \in K, \forall d \in D_i : d\ dom\ p \wedge \forall u \in U_i : p\ dom\ u$

Ein Beispiel ist in Abbildung 5.5 dargestellt. Die Variable wird in BB2 oder BB3 definiert und in BB5 oder BB6 verwendet. Wie in Abbildung 5.5b zu sehen ist, befindet sich sowohl die Definition als auch die Verwendung der Variable auf unterschiedlichen Pfaden. Insgesamt müssen von jeder Definition die Pfade zu beiden Verwendungsstellen betrachtet werden, so dass sich insgesamt vier Pfade ergeben.



(a) Vereinfachter abstrakter Syntaxbaum



(b) Kontrollflussgraph

Abbildung 5.5.: Darstellung einer mehrfachen Definition mit mehrfacher Verwendung

5.5.2. Platzierung von Sende- und Empfangsanweisungen

Die Platzierung von Sende- und Empfangsanweisungen erfolgt abhängig von den erkannten Zugriffsmustern (siehe Abschnitt 5.5.1.2) nach Algorithmen, die in Abschnitt 5.5.2.1 dargestellt werden. Die einzelnen Teilschritte sind in den darauffolgenden Abschnitten dargestellt: die Ermittlung von gültigen Positionen in Abschnitt 5.5.2.2, die Er-

mittlung aller möglichen Kompositionen in Abschnitt 5.5.2.3, die Auswahl der besten Positionen über den Kontrollfluss in Abschnitt 5.5.2.4, die Auswahl der passendsten Lösung mittels Ergebnissen des Schedulings in Abschnitt 5.5.2.5 und schließlich dem Einfügen von Kommunikationsinstruktionen in Abschnitt 5.5.2.6.

Das Verfahren wurde 2015 als Patent EP3121714A1 [1] angemeldet.

5.5.2.1. Platzierungsalgorithmen

Der Algorithmus zur Platzierung der Kommunikationsinstruktionen bei einem 1-zu-1 Zugriff ist in Quellcode 5.9 dargestellt. Zunächst wird für ein Paar aus Definition und Verwendung einer Variablen die zugehörigen IDs der *Ausführungseinheiten* (PE, engl. *processing element*) ermittelt [Zeile 1 und 2]. Sollten sie unterschiedlichen PEs zugewiesen sein [Zeile 4], so werden für beide Stellen die entsprechenden Basisblöcke (BB) gesucht. Diese bilden Anfang und Ende eines Pfades im Kontrollfluss, auf dem alle möglichen Kompositionen ermittelt werden [Zeile 8]. Mit Hilfe einer Optimierungsfunktion wird aus den möglichen Lösungen die beste ausgewählt [Zeile 9] und anschließend werden gemäß dieser Lösung die passenden Kommunikationsinstruktionen im Code eingefügt [Zeile 10].

```
1  PEDef := getAssignedPE(def)
2  PEUse := getAssignedPE(use)
3
4  if PEUse != PEDef then
5    BBDef := getBB(def)
6    BBUse := getBB(use)
7
8    allCompositions := defineAllPossibleCompositions(BBDef, BBUse)
9    selectedComposition := selectBestComposition(allCompositions)
10   insertCommunication(selectedCompositions, def, use)
11 end if
```

Quellcode 5.9: Algorithmus zur Platzierung bei einem 1-zu-1 Zugriff

Bei einem 1-zu-n Zugriffsmuster wird als Basis der Algorithmus zum Lösen des 1-zu-1 Zugriffsmusters verwendet und wie in Quellcode 5.10 dargestellt ist, erweitert. Für jede Verwendung der Variablen [Zeile 3] wird ermittelt, ob sie sich auf einem anderen Kern befindet [Zeile 5] und anschließend gleichbehandelt wie ein 1-zu-1 Zugriff. Zu beachten ist, dass die Ermittlung aller Verwendungen in der Reihenfolge des Kontrollflusses erfolgt. Dies sorgt in der SSA-Darstellung dafür, dass zusätzliche Verwendungen der Variablen später im Code keine weiteren Datentransfers benötigen, da die gleiche Version der Variablen verwendet wird. Sollte sich der Wert im Quellcode ändern, so wird dies bei einer anderen Datenabhängigkeit analysiert.

```
1  PEDef := getAssignedPE(def)
2
3  for all use in allUses(def) do
```

```
4     PEUse := getAssignedPE(use)
5     if PEDef != PEUse then
6       BBDef := getBB(def)
7       BBUse := getBB(use)
8
9       allCompositions := defineAllPossibleCompositions(BBDef, BBUse)
10      selectedComposition := selectBestComposition(allCompositions)
11      insertCommunication(selectedComposition, def, use)
12    end if
13  end for
```

Quellcode 5.10: Algorithmus zur Platzierung bei einem 1-zu-n Zugriff

Die Verallgemeinerung auf das m-zu-n Zugriffsmuster ist schließlich in Quellcode 5.11 dargestellt. Liegen mehrere Definitionen der Variablen vor, die an unterschiedlichen Stellen auf anderen PEs verwendet werden, so wird der Algorithmus in einer weiteren Schleife ausgeführt [Zeile 1] und somit m-Mal der 1-zu-n Zugriff gelöst.

```
1   for all PEDef in getAllDefs(variable) do
2     for all use in getAllUses(def) do
3       PEUse := getAssignedPE(use)
4       if PEDef != PEUse then
5         BBDdef := getBB(def)
6         BBUse := getBB(use)
7
8         allCompositions := defineAllPossibleCompositions(BBDef, BBUse)
9         selectedComposition := selectBestComposition(allCompositions)
10        insertCommunication(selectedComposition, def, use)
11      end if
12    end for
13  end for
```

Quellcode 5.11: Algorithmus zur Platzierung bei einem m-zu-n Zugriff

Da die einfacheren Zugriffsmuster bereits Teil des Algorithmus für den m-zu-n Zugriff sind, entfällt eine Erkennung oder Sonderbehandlung der unterschiedlichen Fälle. Die Optimierung ergibt sich rein aus der Anzahl an gefundenen Definitionen bzw. Verwendungen und sollten sie sich auf 1 beschränken, müssen bei der Positionsbestimmung weniger Randbedingungen erfüllt werden.

5.5.2.2. Identifikation von gültigen Positionen

Betrachten wir zunächst den einfachsten Fall mit einer einzelnen Definition und einer einfachen Verwendung. Ohne Beschränkung der Allgemeinheit kann davon ausgegangen werden, dass sich beide Zugriffe in unterschiedlichen Basisblöcken befinden. Zur Analyse der Datenabhängigkeiten mittels der SSA-Darstellung wurde bereits der Kontrollfluss mit einbezogen, so dass sichergestellt ist, dass die Verwendung erst nach der Definition

der Variablen erfolgt. Um alle möglichen Positionen für die Platzierung der Kommunikation zu ermitteln, muss der CFG zwischen Definition und Verwendung der Variablen abgeschritten werden. Dabei eignen sich jedoch nicht alle Basisblöcke für die tatsächliche Platzierung: Durch Einschränkungen, die durch die Ausgangssprache C vorgegeben werden, können manche Positionen nicht effizient für die Kommunikation genutzt werden. Ausnahmen sind Blöcke, die bestimmte Aufgaben in Kontrollstrukturen haben und nur aus einem einzelnen Ausdruck bestehen können. Bei If-Blöcken ist es die Bedingung, bei Schleifen alle Blöcke im Kopf und bei Switches der Block mit der Switch-Anweisung. Sollen mehrere Anweisungen in solch einem Block ausgeführt werden, so müssen diese mit dem Komma-Operator getrennt werden, bei dem nur der Rückgabewert der letzten Anweisung ausgewertet wird. Aus „`i = 0`“ kann beispielsweise „`i = 0, send(i, 2), i`“ gemacht werden. Auf diese Weise wird die Variable direkt nach der Initialisierung übertragen. Komplexer wird es bei Ausdrücken, in denen der Rückgabewert des Statements betrachtet wird. Soll also in einem Block mit der Anweisung „`i < 10`“ noch eine Datenübertragung stattfinden, muss sichergestellt werden, dass der Rückgabewert des letzten Ausdrucks dem ursprünglichen Wert entspricht. Ein Beispiel: „`value = i < 10, send(data, 2), value`“. Das Ergebnis der Evaluation wird in der Variablen `value` zwischengespeichert und muss nach der Kommunikationsinstruktion wiederholt werden. Die Übertragung innerhalb dieser Blöcke ist somit häufig mit zusätzlichen Instruktionen verbunden und ist aufgrund ihrer ungewöhnlichen Konstruktion auch nicht gut für die Lesbarkeit. Die Platzierung in einem Block vor oder nach einem solchen ist immer möglich und effizienter. Bei Verzweigungen im CFG, die durch Bedingungen oder direkte Sprünge wie `break`, `continue` oder `goto` erzeugt werden, sind potentiell alle Basisblöcke auf allen Pfaden für die Platzierung der Kommunikation erlaubt. Gemäß der Definition der Komposition muss sichergestellt werden, dass bei Platzierung von Kommunikationsinstruktionen hinter Verzweigungen auch auf jedem anderen Pfad eine Kommunikation platziert wird.

5.5.2.3. Suche nach möglichen Kompositionen im CFG

Wie in Abschnitt 5.5.1.2 dargestellt und in den Quellcodes 5.9, 5.10 und 5.11 mit dem Aufruf `defineAllPossibleCompositions` verwendet, müssen zunächst alle möglichen Kompositionen ermittelt werden. Beginnend von der Definition der Variablen in der SSA-Darstellung werden alle möglichen Pfade zu allen Verwendungsstellen gesucht. Die Menge aller Pfade wird im Folgenden als Pfadnetzwerk bezeichnet. Zum Aufbau eines Pfades werden alle Blöcke des CFG einer Funktion abgeschritten. Dabei werden fünf Fälle unterschieden:

1. Ein Block darf nicht überschritten werden, wenn er das Ende einer Funktion markiert. Der aktuelle Pfad führt demnach nicht zur Verwendung der Variablen und wird verworfen.
2. Ein Block darf nicht überschritten werden, wenn die untersuchte Variable darin verwendet wird. Der aktuelle Pfad wird um den Block erweitert und zum Kommunikationsweg hinzugefügt.

3. Der Block ist bereits Teil eines existierenden Pfads. In diesem Fall wird der Block dem Pfad hinzugefügt und anschließend wird der Pfad mit den Blöcken des bereits gefundenen Pfades vervollständigt und dem Pfadnetzwerk hinzugefügt.
4. Der Block wurde bereits besucht, erfüllt jedoch nicht die dritte Bedingung. Somit befindet sich der Block in einer Schleife des CFG und der aktuelle Pfad wird verworfen.
5. Sind die vorigen Bedingungen nicht erfüllt, wird der Block zum aktuellen Pfad hinzugefügt und die Suche wird in den nachfolgenden Blöcken fortgesetzt.

Verzweigt sich der CFG aufgrund von Bedingungen, so werden neue Pfade erstellt und entsprechend den beschriebenen Bedingungen behandelt. Die Tiefensuche im CFG hat eine Zeitkomplexität von $\mathcal{O}(n_b + n_k)$, wobei n_b die Anzahl der Basisblöcke und n_k die Anzahl an Kanten im CFG ist.

Das resultierende Pfadnetzwerk zwischen Definition und Verwendung der Variablen enthält alle Pfade, aus denen nun die entsprechenden Kompositionen gebildet werden können. Dazu muss aus jedem der gefundenen Pfade eine Position genommen werden, um die notwendige Kommunikation auf dem Pfad zu gewährleisten. Somit ist jede Position, die in allen Pfaden vorkommt, eine direkte Lösung und damit eine Komposition mit genau einer Position. Wird von einem Pfad eine Position gewählt, die nicht Teil eines anderen Pfades ist, so müssen weitere Positionen zur Komposition hinzugefügt werden, bis alle Pfade abgedeckt sind. Kompositionen können durch Permutationen der Positionen der einzelnen Pfade bestimmt werden. Die Laufzeitkomplexität des Permutationsalgorithmus ist mit $\mathcal{O}(k^n)$ sehr hoch, wobei n die Anzahl der Pfade und k die durchschnittliche Anzahl der Positionen pro Pfad ist. Alle gefundenen Kompositionen sind Lösungen für das 1-zu-1-Zugriffsmuster.

Zur Lösung von 1-zu-n Zugriffen wird der beschriebene Algorithmus mit allen n Verwendungsstellen als Endblöcke der Pfade aufgerufen. Das sich ergebende Pfadnetzwerk setzt sich aus allen Pfaden von der Definition der Variablen zu allen Verwendungsstellen zusammen. Wie beim 1-zu-1 Zugriffsmuster mit mehreren Pfaden müssen die Positionen der einzelnen Pfade permutiert werden, um gültige Kompositionen zu bilden. Jede zusätzliche Position in einer Komposition sorgt dafür, dass zusätzliche Sende- und Empfangsinstruktionen im Code eingefügt werden. Aufgrund bedingter Aufrufe im Code bedeutet es aber nicht, dass zur Laufzeit auch mehr Transfers stattfinden.

Der gleiche Ansatz kann auf m-zu-n Zugriffsmuster erweitert werden, indem alle m Definitionen als Startpunkte für die Tiefensuche im CFG verwendet werden. Gültige Lösungen sind auch hier Kompositionen aus Positionen, die alle gefundenen Pfade abdecken.

Um die Laufzeit des Algorithmus und gerade der Permutation zu reduzieren, können folgende Optimierungen durchgeführt werden, um die Anzahl an Positionen je Pfad zu reduzieren. Ein Block wird nur hinzugefügt, wenn

1. er die Definition der Variablen enthält.
2. er mehr als eine Eingangskante des CFG enthält und somit die Vereinigung von mindestens zwei Pfaden ist.
3. zwischen ihm und dem nächsten Block, für den Bedingung 2 gilt, nur Blöcke liegen, die keine erlaubten Positionen sind.

4. sich zwischen ihm und der nächsten Verwendungsstelle der Variablen nur nicht-erlaubte Positionen befinden.

5.5.2.4. Auswahl anhand des Kontrollflusses

Die Anzahl an möglichen Kompositionen, die das Platzierungsproblem lösen, kann sehr groß werden. Ziel ist es nun, die Kompositionen auszuwählen, die die Laufzeit der parallelen Anwendung minimieren. Dabei ist eine Möglichkeit, die Aufrufhäufigkeit der einzelnen Basisblöcke hinzuzuziehen. Blöcke, die häufiger ausgeführt werden, sorgen dafür, dass auch darin platzierte Datentransfers entsprechend häufiger ausgeführt werden. Aufgrund der Platzierung in Basisblöcken sind die Elternblöcke für die tatsächliche Anzahl an Aufrufen verantwortlich.

Wie in Abschnitt 2.4 dargestellt, kann die Anzahl an Iterationen für einfache Schleifen bestimmt oder durch Profiling gemessen werden. Sollte weder die Berechnung noch die Messung zum Ziel führen, so kann anstelle der tatsächlichen Anzahl an Iterationen ein fixer Wert verwendet werden. Bewährt haben sich die Werte 10 als Anzahl an Iterationen von Schleifen und 0,5 für die Wahrscheinlichkeiten für den `true` und `false` Pfad bei If-Blöcken. Die Zahlen sind rein als Gewicht zu verstehen, das verhindern soll, dass Kommunikation in Schleifen platziert wird. Die Kosten kleiner als 1 bei den `then` und `else` Blöcken bevorzugen eine Kommunikation innerhalb von If-Blöcken, dabei ist jedoch zu beachten, dass dieser Vorteil nur genutzt werden kann, wenn sich die Verwendung der Variablen ebenfalls in dem If-Block befindet. Ansonsten muss Kommunikation in beiden Pfaden platziert werden und es ergibt sich in der Summe doch wieder ein Gewicht von 1.

Ein beispielhafter Verlauf der Aufrufhäufigkeiten ist in Abbildung 5.6 dargestellt. Die Definition sowie die Verwendung der Variablen finden in doppelt verschachtelten For-Schleifen an Position 1 und 10 statt. Bei der hier angenommenen Anzahl von fünf Durchläufen je Schleife ergeben sich für Basisblöcke auf der untersten Ebene 25 Aufrufe. Außerhalb der Schleifen kann eine Häufigkeit von 1 und 5 ermittelt werden. Die 1 kann als kritischer Pfad betrachtet werden und bezieht sich auf Basisblöcke, die genau ein Mal ausgeführt werden. Blöcke mit 5 können als Schleifen mit fünf Aufrufen angenommen werden. Die Aufrufhäufigkeiten der Blöcke in so einer Konstellation hat häufig die Form einer Badewanne, bei der sich am Anfang und am Ende die höchsten Zahlen befinden, diese schnell abnehmen und in der Mitte meist flach sind. In diesen Fällen sollten für die Platzierung möglichst Blöcke in der Mitte der Grafik verwendet werden, bevorzugt Blöcke mit einer Aufrufanzahl von eins.

Neben der Aufrufhäufigkeit können auch andere Faktoren mit einbezogen werden, um die Güte einer Position zu bewerten:

- Zur Optimierung der Leistungsaufnahme können Positionen bevorzugt werden, die es erlauben, einen der beiden Kommunikationspartner länger in einem Modus mit verringerter Leistungsaufnahme zu halten. Das kann beispielsweise eine Position sein, die zeitlich möglichst nah an der Definition der Variablen ist, damit der empfangende Kern möglichst lang in einem sparsamen Modus verweilen kann. Bei dieser Planung muss jedoch der gesamte Verlauf der Anwendung mit einbezogen

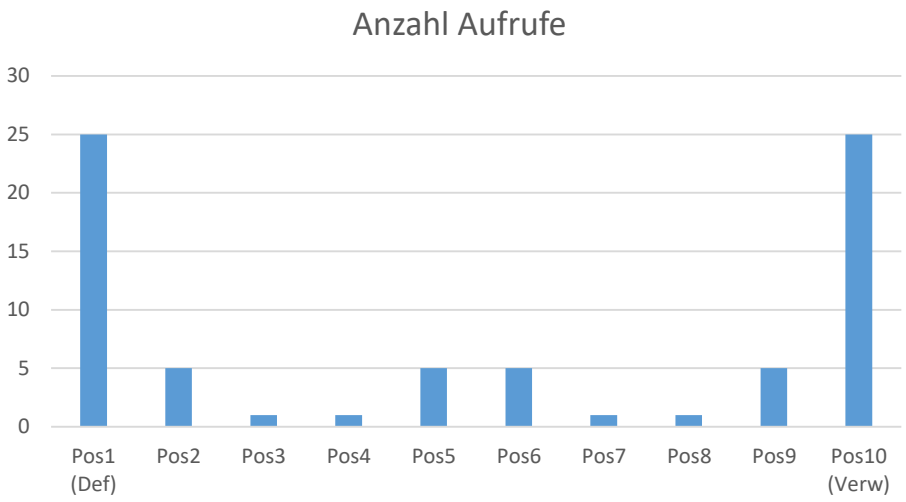


Abbildung 5.6.: Beispielhafte Aufrufhäufigkeit von Basisblöcken

werden, so dass nur während dem Scheduling alle nötigen Informationen vorhanden sind.

- Bei der Platzierung kann der Ressourcenbedarf hinsichtlich Zwischenspeicher und der Kapazität des Kommunikationsnetzwerks optimiert werden. Dazu müssen Positionen bevorzugt werden, die dem Sender erlauben, möglichst zeitnah den Speicher der gesendeten Variablen wieder freizugeben. Die Entscheidung benötigt detaillierte Informationen über die Auslastung des (lokalen) Speichers sowie die Nutzung von Puffern oder anderen Ressourcen des Netzwerks.

Die Faktoren können alle kombiniert werden, um die Kosten für die Auswahl eines Blocks festzulegen. Je nach Gewichtung können somit unterschiedliche Platzierungen mit anderer Zielsetzung generiert werden.

5.5.2.5. Auswahl mit Hilfe eines Zeitplans

Üblicherweise sorgt die Auswahl an besten Positionen über eine Gewichtung des Kontrollflusses dafür, dass mehr als eine Position in Frage kommt. Werden keine anderen Faktoren der Hardware mit einbezogen oder liefern auch sie kein eindeutiges Ergebnis, können die Ergebnisse des Schedulings mit einbezogen werden. Während dem Task-Scheduling findet keine tatsächliche Platzierung bzw. Scheduling von Kommunikation statt; die Kosten für den Datentransfer zwischen den Ausführungseinheiten werden rein als Overhead der Ausführungszeiten von Tasks hinzugefügt. Dies vereinfacht das Scheduling-Problem, sorgt aber dafür, dass zusätzliche Wartezeiten bei der Kommunikation nur unzureichend modelliert werden können.

Bei der Kommunikationsplatzierung anhand eines Zweitplans spielt das Kommunikationsmodell eine große Rolle: bei asynchroner Übertragung sollte das Senden im zeitlich ersten dieser gefundenen Basisblöcke ausgeführt werden, das zugehörige Empfangen kann erst im letzten Block mit gleicher Aufrufhäufigkeit erfolgen. Sobald ein synchroner Teil in der Kommunikation vorhanden ist, ist es wichtig, den Schedule des Programms auf allen Prozessoren mit zu betrachten. Die Kommunikation sollte so platziert werden, dass bei beiden beteiligten Prozessoren die Wartezeiten reduziert werden. Dazu können die ermittelten Werte über den Zeitpunkt der Definition und der Verwendung der Variablen verwendet werden.

Wie bereits in der Einleitung dieses Kapitels beschrieben, ist das Programm bei Einfügen der Kommunikationsinstruktionen noch nicht in einzelne Prozesse für die verschiedenen Kerne aufgeteilt. Die Ergebnisse des Schedulers sind für jeden Task in Form von Kernzuweisung sowie Start- und Endzeitpunkt des Tasks für jeden beteiligten Kern angegeben. Tasks, deren Kind-Tasks auf unterschiedlichen Ausführungseinheiten ausgeführt werden, können verschiedene Start- und Endzeiten für die unterschiedlichen Kerne haben. Um somit die Tasks mit den besten Zeiten im Vergleich zur Definition und Verwendung der Variablen zu finden, müssen für alle Tasks mit der optimalen Aufrufhäufigkeit bzw. Gewicht die Zeiten für Start und Ende betrachtet werden.

Der Algorithmus geht bei der Auswahl anhand des Schedules wie folgt vor: Zunächst werden alle möglichen Kompositionen, die das Kommunikationsproblem lösen, ermittelt. Anschließend werden sie anhand ihrer Aufrufhäufigkeit gewichtet. Je geringer das Gewicht, desto besser die Komposition. Sollte mehr als eine Komposition das gleiche Gewicht haben, so wird die beste Komposition anhand des Timings der Tasks, wie es beim Scheduling berechnet wurde, ausgewählt. Jeder Task besitzt vom Scheduling globale Zeiten für den Start und das Ende auf jedem Kern auf dem er ausgeführt wird. Durch den Kontrollfluss ist es möglich, Zeiten für Tasks zu ermitteln, die eigentlich nicht auf allen Ausführungseinheiten ausgeführt werden, indem die lokale Zeit des Kerns bestimmt wird. Dazu wird der letzte Task im Kontrollfluss gesucht, der auf dem gesuchten Kern ausgeführt wird. Das Ende dieses Tasks gibt die lokale Zeit auf dem Kern wieder. Zur Bewertung einer Position wird nun die Differenz der Zeiten von der Definition der Variablen bis zur aktuellen Position sowie von der aktuellen Position zur Verwendung der Variablen gebildet. Die Summe beider Differenzen wird als Kosten angenommen. Die Kosten können durch Faktoren weiter gewichtet werden, um besser für eine bestimmte Kommunikationsart geeignet zu sein. Bewährt hat sich bei asynchroner Kommunikation ein Faktor von 3 für die Kosten auf Senderseite, mit dem die Differenz multipliziert wird. Dies hat zur Folge, dass Positionen weiter entfernt von der Verwendung für höhere Kosten auf Senderseite sorgen und nähere Positionen im Vergleich günstiger werden. Bei asynchroner Kommunikation werden die Daten in einem Puffer gespeichert, was bedeutet, eine Position möglichst nah an der tatsächlichen Definition der Variablen ist zu bevorzugen, um den Sender möglichst früh wieder für andere Aufgaben zur Verfügung zu haben. Der Empfänger kann die Daten erst später auslesen, ohne den Sender weiter zu beeinflussen. Bei synchroner Kommunikation ist keine Gewichtung notwendig, die beste Position ist die, bei der die Differenz der lokalen Zeiten möglichst gering ist.

5.5.2.6. Einfügen von Kommunikationsinstruktionen

Wie im vorherigen Abschnitt gezeigt, muss die Kommunikation so platziert werden, dass alle Programmpfade abgedeckt werden. Die Platzierung von Sende- und Empfangsanweisungen kann dabei gemeinsam oder getrennt erfolgen. Gemeinsam bedeutet in diesem Zusammenhang, dass beide Anweisungen im gleichen Basisblock platziert werden. Für die korrekte Funktionalität muss dann sichergestellt werden, dass bei einer späteren Aufteilung des Programms auf mehrere Prozessoren, die Kommunikationsanweisungen entsprechend zugewiesen werden. (siehe Abschnitt 4.4.1) Bei einer getrennten Platzierung werden die Anweisungen in unterschiedlichen Basisblöcken platziert. Hier ist die spätere Aufteilung auf verschiedene Prozessoren offensichtlicher, da nicht mehr zwei Anweisungen, die auf unterschiedlichen Prozessoren ausgeführt werden sollen, im selben Basisblock platziert werden. Im Folgenden werden die beiden Möglichkeiten miteinander verglichen und die jeweiligen Vor- und Nachteile dargestellt.

Gemeinsame Platzierung der Kommunikation

Vorteile

- Die Platzierung der Kommunikation der Paare kann unabhängig von anderer Kommunikation von Sender und Empfänger erfolgen. Es muss also nicht erst überprüft werden, ob andere Kommunikation den neuen Transfer beeinflussen könnte.
- Durch die gemeinsame Platzierung wird zudem sichergestellt, dass die Reihenfolge der Übertragungen auf allen Kernen immer gleich ist und das Auftreten von Deadlocks wird verhindert.
- Bei synchroner Übertragung kommt hinzu, dass beide Prozessoren im Kontrollfluss an der gleichen Stelle im Programm sind und somit Wartezeiten zwischen den Kernen minimiert werden.

Nachteile

- Die Platzierungsmöglichkeiten werden durch die gemeinsame Platzierung eingeschränkt. Mit getrennter Platzierung kann eine bessere Performanz erreicht werden, da sich die einzelnen Anweisungen genauer an den Schedule anpassen lassen.
- Es wird ein zusätzlicher Schritt benötigt, der die Instruktionen auf die entsprechenden Kerne verteilt.
- Der Vorteil einer asynchronen Kommunikation, dass der Sender nicht warten muss, bis der Empfänger bereit für den Empfang ist, wird durch den synchronisierten Kontrollfluss abgeschwächt und kann unter Umständen nicht vollends ausgenutzt werden.

Getrennte Platzierung der Kommunikation

Vorteile

- Durch die getrennte Platzierung ist es möglich, eine bessere Performanz durch Verringerung der Wartezeiten zu erreichen. Da sich Anweisungen viel feingranularer platzieren lassen, können Wartezeiten auf andere Prozessoren verringert werden.
- Die genauere Platzierung erlaubt eine bessere Ausnutzung vorhandener Hardware-Puffer zur Zwischenspeicherung von Daten und kann somit dafür sorgen, dass der Speicherbedarf der einzelnen Kerne reduziert wird. Auf diese Weise können die Vorteile einer asynchronen Kommunikation optimal ausgenutzt werden.

Nachteile

- Die Platzierung muss immer unter Beachtung von anderen Transfers der beteiligen Kerne erfolgen. Transfers können sich überlappen und wenn Puffer nicht ausreichen, kann es zu Deadlocks kommen, da ein Prozessor zwar auf Daten wartet, ein anderer wegen einer blockierten Übertragung aber nie bis zum Transfer kommt. Diese Analyse kann unter Umständen recht komplex werden.
- Bei synchroner Kommunikation können die Vorteile nicht voll ausgenutzt werden, da durch die gleichzeitige Ausführung bereits große Einschränkungen bei der Platzierung bestehen. Da zudem die Kerne bei jedem Transfer synchronisiert werden, können nur einzelne Transfers davon profitieren, alle anderen müssen immer auf allen Kernen gleichzeitig ausgeführt werden.

Auch wenn die getrennte Platzierung potentiell eine höhere Performanz erreichen kann, wird in dieser Arbeit allein die gemeinsame Platzierung eingesetzt, da eine Deadlock-Freiheit oberste Priorität hat. Bei der Parallelisierung muss verhindert werden, dass neue Probleme erzeugt werden, die bei rein sequentieller Abarbeitung gar nicht auftreten können.

5.5.3. Stand der Technik der Kommunikationsoptimierung

Die Kommunikationsoptimierung für Systeme, die auf Nachrichtenaustausch basieren, ist ein weit erforschtes Thema. Es wurde hauptsächlich während der 90er Jahre bearbeitet, als Rechnerfarmen bestehend aus HPC-Clustern ein neuer Trend waren. Da sich eingebettete Systeme immer weiter hin zu Mehrkernsystemen entwickeln, können immer mehr Erkenntnisse aus dem HPC-Bereich eingesetzt werden.

Der Einfluss der verschiedenen Aspekte der Kommunikation auf die Performanz einer Anwendung wurde bereits in [100] evaluiert. Die betrachteten Eigenschaften sind die Latenzen, der Overhead und die Bandbreite. Es hat sich gezeigt, dass der Overhead den größten Einfluss hat und die Performanz um Größenordnungen beeinflussen kann. Weiter wichtig ist die Bandbreite, damit größere Datenmengen in möglichst kurzer Zeit übertragen werden können. Gerade bei Algorithmen, die mit großen Datenmengen arbeiten,

wird ansonsten die Effizienz deutlich reduziert, da Wartezeiten im Verhältnis zu Berechnungszeiten zu groß werden. Der Einfluss der Latenz ist relativ gering und wird durch gängige Algorithmen bereits gut abgefedert, indem der Zeitpunkt der Kommunikation so gelegt wird, dass die Latenz keinen großen Einfluss auf die Laufzeit hat.

Eine grobe Einteilung gängiger Optimierungen kann [101] und [102] entnommen werden:

- *Reduktion des Kommunikationsoverheads*: um einen Datentransfer zu initiieren ist ein großer Verwaltungsoverhead notwendig. Dieser kann abhängig von Implementierung und dem Zielsystem Kopieraktionen in einen Puffer, Überprüfung von Datengrößen und -typen, Aufbau eines Kanals und anderes beinhalten. Bei kleinen Datenmengen kann diese Vorbereitung die Dauer für die eigentlich Datenübertragung übersteigen. Etablierte Methoden, um diesen Overhead zu reduzieren sind:

  - *Datenaggregation* (engl. *message aggregation*): Mehrere Übertragungen von kleinen Datenmengen zwischen gleichem Sender und Empfänger werden zu einem größeren Transfer zusammengefasst, so dass der Overhead nur ein Mal anfällt.
  - *Vektorisierung* (engl. *vectorization*): Mehrere Übertragungen von einzelnen Elementen einer Matrix innerhalb einer Schleife werden zu einer großen Übertragung zusammengefasst.
  - *Verschmelzung* (engl. *coalescing*): Auch wenn Daten häufiger von einem anderen Prozessor benötigt werden, reicht eine einmalige Übertragung aus, solange sich die Daten nicht verändert haben. Dies erfordert eine zusätzliche Analyse, die zur Compile-Zeit in der SSA-Darstellung des Programms durchgeführt wird. Bei jeder Verwendung einer Variablen ist die SSA-Version entscheidend und nur wenn auf eine andere Version zugegriffen wird, ist eine erneute Übertragung notwendig.

  In [103] wurde der Einfluss der drei Methoden Entfernung von redundanter Kommunikation, Aggregation von Transfers und dem Kommunikationspipeling evaluiert. Den größten Einfluss hat die Vermeidung redundanter Kommunikation. Das Aggregieren von Transfers mit bis zu 4 KiB an Daten reduziert den Overhead im Vergleich zur Einzeltransfers merklich, da nur ein unwesentlicher Mehraufwand zum Aggregieren und Aufteilen der Daten notwendig ist.

  Ein erweiterter Ansatz zur Reduktion des Overheads und zum Zusammenfassen möglichst vieler Kommunikationen ist die Verwendung von Kommunikationsmustern [104]. Dabei werden zunächst alle nötigen Übertragungen zwischen den einzelnen Prozessoren formal beschrieben. Anschließend werden Kommunikationen mittels Musterabgleich mit bekannten Mustern wie „einer sendet an alle“, „alle senden an einen“, „alle senden an alle“ Sammelaufrufen (engl. *broadcast*) oder einfachen Sende-Empfangsmustern abgeglichen und die passendste Lösung aus einer Bibliothek ausgewählt.

- *Überlappung von Berechnung und Übertragung*: Bei der datenflussbasierten Kommunikationsplatzierung wird die Latenz der Kommunikation verborgen, indem das Senden früh und der Empfang spät platziert wird. Während der Übertragung kann weiter an davon unabhängigen Tasks gearbeitet werden. Diese Optimierung setzt

voraus, dass auf dem Zielsystem eine asynchrone Datenübertragung unterstützt wird, bei der die Kommunikation von einem dedizierten Hardware-Modul wie einer *Speicherdirektzugriff* (DMA, engl. *direct memory access*)-Einheit ausgeführt wird und der Prozessor nicht aktiv an der Kommunikation beteiligt ist. Eingesetzt in: [105]

- *Minimierung von Datenpufferung und Zugriffskosten*: weitere Optimierungen zielen darauf ab, Ressourcen wie Zwischenpuffer für Daten effizienter zu nutzen. Dazu werden Ausgangspuffer eingespart, indem Daten direkt vom Hauptspeicher gelesen und übertragen werden. Auf Empfangsseite kann das Entpacken der Daten entfallen, indem die Daten direkt aus dem Eingangspuffer heraus adressiert werden können. Bei Verwendung von Betriebssystem müssen die entsprechenden Möglichkeiten bereitgestellt werden.

Die Reduktion des Kommunikationsoverheads ist ein Thema, das in den 80er Jahren und 90er Jahren des 20. Jahrhunderts groß war [106], [107] und vor allem auf den Einsatz von HPC-Clustern abzielte.

[108] ist ein weiteres Beispiel, das auf Sammeln von Transfers und Verschmelzung setzt. Allein die Bezeichnungen werden etwas anders verwendet: *coalescing* wird als Oberbegriff für das Zusammenfassen verwendet, Verschmelzung wird im Sinne von Software Caching und einer vorgelagerten Übertragung behandelt.

Ein verwandtes Thema ist die Kommunikationssynthese, wie sie in [109] betrachtet wird. Dort wird ein Algorithmus beschrieben, der aus dem Eingangsprogramm ein Modell aus Task-Graphen und Prozessen extrahiert, um eine optimale Hardware-Topologie zu entwerfen, die die Kommunikationszeiten minimiert. [110] hat einen ähnlichen Ansatz, setzt aber stattdessen auf eine Verhaltensmodellierung der Anwendungen.

In [111] wird ein Kommunikationsmodell eingeführt, das verschiedene Ressourcen (Prozessor, Speicher, DMA-Einheiten) mit einbeziehen kann, um die Kommunikation sinnvoll während dem Mapping zu platzieren. Es arbeitet auch auf einer recht groben Darstellung und wird nicht so fein betrachtet wie es in ALMA gemacht wurde. Modelle dieser Art sind immer noch relevant und werden in komplexeren Systemen verwendet, um die Kommunikation sinnvoll zu planen [112].

In [113] wird eine Kommunikationsanalyse unter Zuhilfenahme des Kontrollflusses vorgestellt. Die Informationen über den zeitlichen Ablauf, die aus dem CFG gewonnen werden, werden in einem Kommunikationsabhängigkeitsgraph dargestellt. Auf diese Weise kann eine konfliktfreie Kommunikation gewährleistet werden, bei der der Zugriff auf geteilte Ressourcen ohne unnötige Wartezeiten erreicht werden kann.

5.5.4. Erfüllung der Anforderungen aus Abschnitt 3.1.4.1

D1 Die korrekte Ausführung der parallelisierten Anwendung kann durch einen Vergleich mit der sequentiellen Anwendung systematisch verifiziert werden. Als sequentielle Anwendung muss in diesem Fall der generierte C-Code nach der Algorithmus- und der Code-Ebene verwendet werden. Beide Ebenen erlauben grundsätzliche Änderungen an der Darstellung des Algorithmus und erschweren somit

einen direkten Vergleich mit dem generierten parallelen Code. Gemäß den Definitionen der Aufgaben je Ebene können Entscheidungen auf Task-Ebene keine funktionalen Änderungen oder fehlerhafte Ausführungen erzeugen, da alle Abhängigkeiten befolgt werden müssen und die Änderungen auf der Daten-Ebene die korrekte Ausführung sicherstellen müssen. Die folgenden Code-Änderungen müssen nach der Parallelisierung auf Daten-Ebene bei der Sicherstellung der korrekten Ausführung betrachtet werden:

D1.1 Es müssen die gleichen Operationen ausgeführt werden. Dies ist der Fall, wenn der gleiche Operationstyp mit den gleichen Eingangsdaten angewendet wird und die gleichen Kontrollabhängigkeiten bestehen.

D1.2 Variablen- und Funktionsnamen können während der Parallelisierung geändert werden, um die Kernzugehörigkeit oder duplizierte Elemente zu kennzeichnen. Beim Vergleich ist daher nur auf den Datentyp und die Verwendung zu achten.

D1.3 Alle Daten- und Kontrollabhängigkeiten, die über Prozessorgrenzen hinweg gehen, müssen durch eine entsprechende Kommunikationsinstruktion ersetzt werden. Dabei muss für eine Datenabhängigkeit ein Transfer der Variablen erfolgen und für eine Kontrollabhängigkeit der Wert der Bedingung bzw. der Evaluation.

D1.4 Die Reihenfolge von Instruktionen und Tasks muss zwischen der sequentiellen und der parallelisierten Anwendung nicht identisch sein, jedoch müssen alle Abhängigkeiten korrekt behandelt werden.

Race-Conditions und Deadlocks sind mögliche Fehler im parallelen Programm, deren Abwesenheit durch eine Erreichbarkeitsanalyse nachgewiesen werden können. Dazu wird jede Kommunikationsinstruktion als Semaphore modelliert, bei der der empfangende Prozessor erst mit der Verarbeitung fortsetzen kann, wenn die Semaphore vom sendenden Prozessor freigegeben wurde. Auf diese Weise können alle Programmteile, die potentiell gleichzeitig auf verschiedenen Kernen ausgeführt werden können, identifiziert werden. Wenn zwei oder mehr parallele Teile auf die gleiche Variable zugreifen und ein Zugriff ist schreibend, so liegt eine potentielle Race Condition vor. Dies bedeutet im Umkehrschluss, dass die Abwesenheit von Race Conditions nachgewiesen werden kann, wenn keine gleichzeitigen Zugriffe existieren und somit kein Potential für Race Conditions existiert. Die gleiche Darstellung kann ebenfalls zum Nachweis der Abwesenheit von Deadlocks eingesetzt werden: jede Kommunikation wird als Semaphorenzugriff bestehend aus einem Signalisieren und einem Wartezustand modelliert. Im Erreichbarkeitsgraph existiert also eine Verbindung zwischen den beiden, um den Übergang zwischen den verschiedenen parallelen Teilen darzustellen. Können in dieser Darstellung Stellen gefunden werden, an denen auf ein Signal kein passendes Warten erfolgt, so liegt ein potentieller Deadlock vor, der auftreten kann, wenn alle Bedingungen des Kontrollflussgraphs entsprechend erfüllt sind. Liegen keine Fälle dieser potentiellen Deadlocks vor, so ist die Anwendung frei von Deadlocks.

D2 Durch Einführung einer API, um alle Kommunikationsinstruktionen vereinheitlicht aufzurufen, aber dennoch spezielle Realisierungen der Zielhardware zu erlauben,

kann immer die effizienteste Kommunikation, die auf der Hardware verfügbar ist, eingesetzt werden. Dieser $Overhead_{API}$ kann also immer für das Zielsystem minimiert werden. Die absolute $Anzahl_{sync}$ kann nicht weiter reduziert werden, da Kommunikation nur eingefügt wird, wenn eine Variable auf einem Kern geschrieben und von einem anderen Kern gelesen wird. Die in Abschnitt 5.5.2 optimierte Platzierung sorgt aber dafür, dass Kommunikation auch nur dann stattfindet, wenn die Daten aufgrund des Kontrollflusses der Anwendung benötigt werden. Auf diese Weise wird die tatsächliche Anzahl an Transfers bei der Ausführung der Anwendung minimiert.

D3 Die Kommunikationskosten werden entsprechend den Anforderungen modelliert, um das zeitliche Verhalten auf der Hardware bestmöglich umsetzen zu können. Die Umsetzung der Funktionalität ist an die Machbarkeit auf der Zielhardware gekoppelt. Wie schon in [114] mit dem Begriff *Linearisierung* geprägt und heute unter dem Begriff der *atomic instructions* bekannt, muss sichergestellt werden, dass während der Ausführung einer solchen Instruktion nicht parallel eine andere Instruktion auf die gleichen Register bzw. Adressen zugreifen kann. Alle modernen parallelen Prozessoren haben die eine oder andere Form dieser Instruktionen. Ihre Existenz ist Voraussetzung für eine Umsetzung des vorgeschlagenen Kommunikationsmodells. Zur weiteren Realisierung muss es zudem möglich sein, dass Kerne Daten miteinander austauschen können. Dies kann durch gemeinsam nutzbaren Speicher erfolgen oder auf dedizierten Einheiten wie Bussen oder Netzwerken erfolgen. Ist dies gegeben, so kann die Kommunikations-API für die gewählte Hardware umgesetzt werden.

6. Evaluation des Ansatzes

Zur Evaluation des Ansatzes werden verschiedene Algorithmen für den Einsatz auf eingebetteten Mehrkernsystemen optimiert. Dazu werden die Algorithmen in MATLAB® implementiert und anschließend in zwei Schritten zunächst in sequentiellen und dann in parallelen C-Code konvertiert. Dabei werden Optimierungen auf allen vier definierten Ebenen durchgeführt, um den Nutzen jeder Ebene mit der Charakterisierung der Anwendung in Beziehung zu setzen. Zur Bestimmung der Performanz wird einerseits eine Abschätzung der parallelen Performanz und andererseits die Ausführung auf eingebetteten Systemen herangezogen.

6.1. Methodik zur Evaluation

Für die Entwicklung in MATLAB® wird primär auf bereits integrierte Funktionen gesetzt, um die Algorithmen zu beschreiben. Der Fokus liegt dabei rein auf der Funktionalität der Algorithmen, die bereits in einer Art Testbench getestet werden sollen. Das Konzept aus der Hardwareentwicklung ist in Abbildung 6.1 dargestellt. Das zu testende Modul, in diesem Fall der in MATLAB® realisierte Algorithmus, wird mit ausgewählten Eingangsdaten angesteuert und generiert die zu überprüfenden Ergebnisse am Ausgang. Auf diese Weise kann die eigentliche Ausführung des Algorithmus von den Schnittstellen zum Lesen und Schreiben der Eingangs- bzw. Ausgangswerte entkoppelt werden. Die Optimierung des Algorithmus wird sich im Folgenden auch nur auf den Teil ohne die Schnittstellen konzentrieren, da diese auf dem jeweiligen System entsprechend umgesetzt werden müssen. Die Testbench wird auch eingesetzt, um die Ausführungshäufigkeiten von Schleifen und If-Blöcken zu ermitteln wie es in den Abschnitten 2.4.3.1 und 2.4.3.2 dargestellt wird. Hierbei ist wichtig, eine gute Abdeckung des Codes zu erreichen, so dass möglichst alle Teile des Algorithmus mindestens einmal ausgeführt werden. Nur in diesem Fall ist eine sinnvolle Angabe von Ausführungshäufigkeiten und -dauern möglich.

Anschließend wird der Algorithmus sowie die Testbench nach C-Code übersetzt, damit der Algorithmus potentiell in jedem Schritt der Optimierung getestet werden kann. Auch wenn die vier Ebenen prinzipiell nacheinander durchlaufen werden können, sorgen die Interaktionen zwischen den einzelnen Ebenen dafür, dass Optimierungen auf vorherigen Ebenen noch mal angepasst werden müssen. Dabei sind Sprünge bis zurück auf die Algorithmus-Ebene möglich, so dass sich ein Ablauf wie bereits in Abbildung 3.7 dargestellt ist, ergibt. Die iterative Optimierung kann beliebig viele Ebenen zurück gehen, um anschließend wieder alle Ebenen nacheinander zu bearbeiten.

Die Schrittweite und die Anzahl an Durchläufen ergibt sich durch den zu optimierenden Algorithmus und wird bei den einzelnen Anwendungen genauer beschrieben. Zur

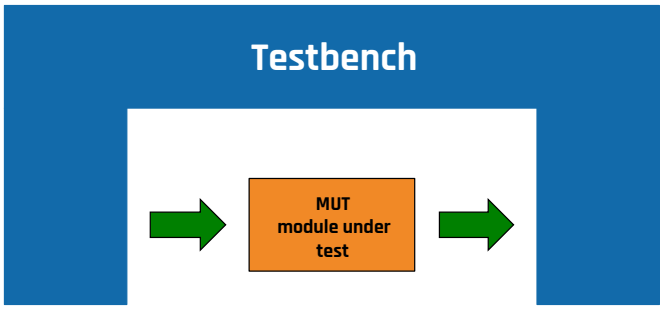


Abbildung 6.1.: Beispiel für einen Testbench-Aufbau

Evaluation der Auswirkungen der einzelnen Ebenen werden unterschiedliche Methoden eingesetzt, um die Laufzeit abzuschätzen:

Algorithmus-Ebene Auf dieser Ebene kann die Performanz nur ohne Einbezug der genutzten Zielplattform ermittelt werden. Eine grundsätzliche Aussage, ob sich die Laufzeit des Algorithmus verändert hat, kann durch Messen der Ausführungsdauer der MATLAB®-Skripte erfolgen. Dabei muss jedoch beachtet werden, dass die Ausnutzung von Eigenheiten der Algorithmen oder Auswahl einer bestimmten Realisierung auch negative Auswirkungen auf die sequentielle Ausführungszeit haben können. Positive Auswirkungen können in der Regel nur erreicht werden, wenn der Algorithmus mit kleineren Datentypen die gleiche bzw. eine ausreichende Genauigkeit erreichen kann, sich aber schneller berechnen lässt. Aus diesem Grund werden Performanzabschätzungen hier nur für Algorithmen ermittelt, bei denen die notwendige Genauigkeit der Datentypen optimiert werden.

Code-Ebene Optimierungen auf Code-Ebene können als sequentieller C-Code ausgegeben werden, so dass die Laufzeit auf der Zielplattform bestimmt werden kann. Transformationen, die die Anzahl der Tasks erhöhen, führen in der Regel zu einem geringen Overhead in sequentiellen Anwendungen. Das Entrollen von Schleifen und die Optimierung von Speicherzugriffen durch Tiling reduzieren ebenfalls die Laufzeit der sequentiellen Anwendung.

Task-Ebene Bei der Parallelisierung auf Task-Ebene wird die Optimierung mit bereits ermittelten Werten durchgeführt. Der Scheduling-Algorithmus versucht nun, die Gesamtlaufzeit durch parallele Ausführung zu reduzieren. Das Ergebnis dieses Schrittes kann daher direkt zur Performanzbestimmung verwendet werden, indem die bisherige sequentielle Ausführungszeit mit dem parallelen Schedule ins Verhältnis gesetzt wird. Die Kommunikation wird an dieser Stelle nur grob abgeschätzt, genauere Aussagen können erst nach der Parallelisierung auf Datenebene getroffen werden.

Daten-Ebene Nach der Parallelisierung auf der finalen Ebene wird der parallele C-Code der Anwendung generiert. Dieser und die Zwischendarstellung, auf der er basiert, kann wie in Abschnitt 6.1.2 dargestellt ist, evaluiert werden.

Die ebenenübergreifende Parallelisierung wurde mit verschiedenen Algorithmen durchgeführt, die verschiedene Aspekte der Charakterisierungen von Algorithmen wie in Abschnitt 6.2 dargestellt, abdecken. Anschließend werden die Ergebnisse ausgewertet und mit den Eigenheiten der Algorithmen in Verbindung gebracht.

6.1.1. Beschreibung der Zielplattformen

Für die Evaluation werden die folgenden Zielplattformen eingesetzt:

- Ein Raspberry Pi 2 wird als weitverbreitete Plattform mit vier Cortex-A7 Kernen eingesetzt. Ein Blockschaltbild des eingesetzten ARM Cortex-A7 ist in Abbildung 6.2 dargestellt. Im Rahmen der Arbeit wird der Code nur für den Armv7 jedes Kerns optimiert, die ebenfalls vorhandenen *NEON™ data engines* kommen nicht zum Einsatz. Sie verwenden den Armv7-A Befehlssatz und takten mit 900 MHz. Das Board wird mit Raspberry Pi OS 10 mit Kernel 5.10.52-v7+ betrieben. Der Arbeitsspeicher beträgt 1 GiB und stellt für alle gewählten Anwendungen keine Einschränkung dar.

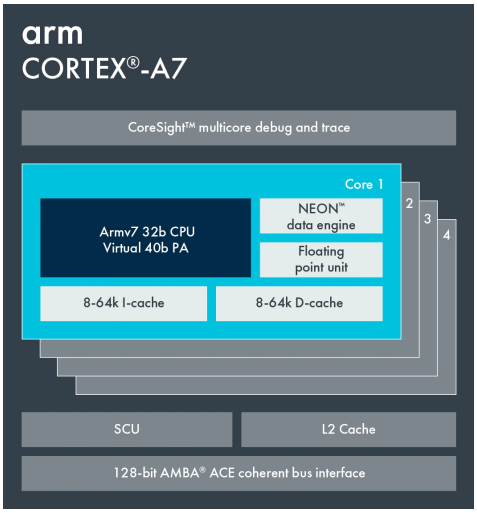


Abbildung 6.2.: Blockschaltbild ARM Cortex-A7 (Quelle: https://developer.arm.com/Processors/Cortex-A7)

- Als zweite Plattform wird ein Infineon AURIX™ TC297 drei-Kern Mikrocontroller eingesetzt. Er besitzt 8 MiB an Programmspeicher sowie 728 KiB + 2 MiB an SRAM. Der Aufbau ist in Abbildung 6.3 dargestellt. Die drei TriCore™ Prozessorkerne teilen sich den Zugriff auf die Speicher und können über Busse auf viele verschiedene Schnittstellen für den Datenaustausch mit verbundenen Sensoren und anderen Geräten zugreifen. Das System wird *bare metal*, also ohne Betriebssystem betrieben. Die Plattform ist für die Ausführung von Kontrollalgorithmen unter hohen Anforderungen der funktionalen Sicherheit aus dem Automotive-Bereich optimiert und

eignet sich schon aufgrund des vorhandenen Speichers nicht für Anwendungen, die mit großen Datenmengen umgehen müssen. Als Testsystem soll gezeigt werden, dass auch Kontroll-Algorithmen, die sich auf den ersten Blick nicht für eine parallele Ausführung eignen, vom vorgestellten Workflow profitieren können.

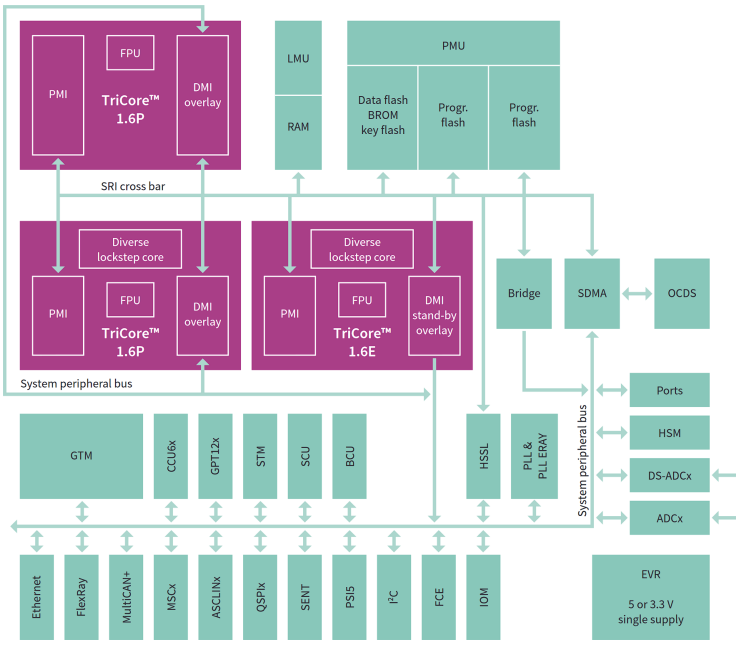


Abbildung 6.3.: Blockschaltbild Infineon AURIX TC2xx (Quelle: https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/aurix-family-tc297ta-adas/)

- Als dritte Plattform wird ein Nvidia Jetson TX2 verwendet, der eine GPU mit 256 Kernen mit sich bringt. Eine Übersicht ist in Abbildung 6.4 dargestellt. Neben der zentralen Pascal GPU sind noch zwei spezielle Denver 2 Kerne und vier Cortex-A57 Kerne verbaut. Somit kann das System sehr gut als Beispiel für ein heterogenes System mit drei unterschiedlichen Ausführungseinheiten eingesetzt werden. Der Fokus bei der Evaluation soll dabei die einfache Erweiterung des Ansatzes auf Beschleuniger mit anderen Charakteristiken (also hier der GPU) sein und wird nicht vollumfänglich mit allen verfügbaren Ausführungseinheiten durchgeführt.

6.1.2. Performanzermittlung der final optimierten Anwendungen

In diesem Abschnitt werden zwei Möglichkeiten vorgestellt, wie die Performanz des final optimierten und parallelisierten Algorithmus ermittelt werden kann. Dabei setzt eine auf die statische Analyse der parallelen Zwischendarstellung des Programms und kann

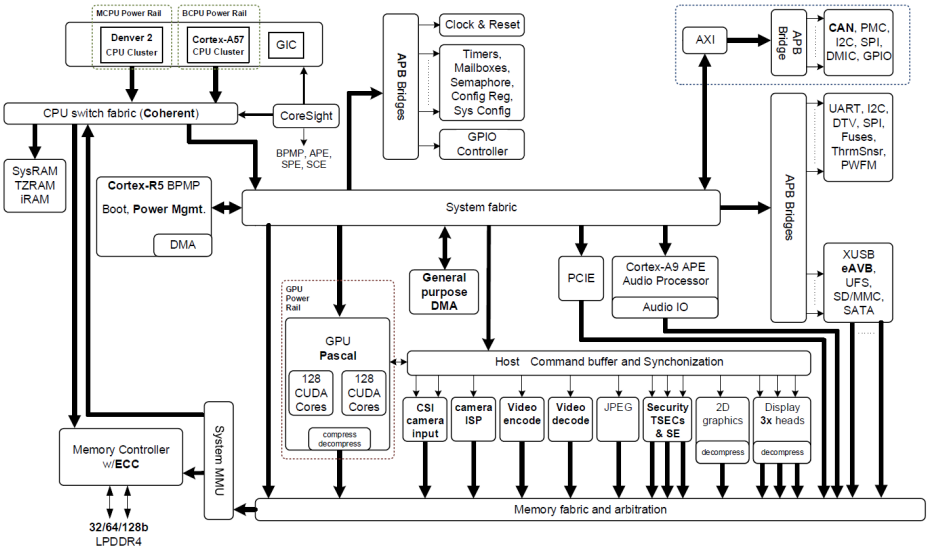


Abbildung 6.4.: Blockdiagram eines Nvidia Jetson TX2 SoC (Quelle: https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/)

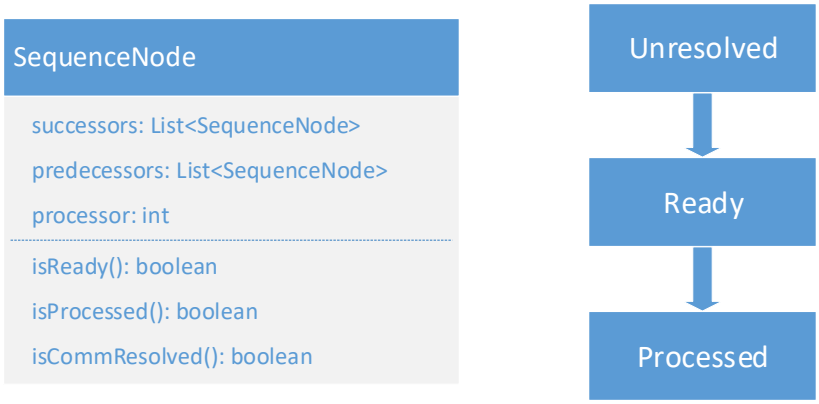
eingesetzt werden, ohne Zugriff auf die tatsächliche Hardware zu haben. Die andere Methode setzt auf Messungen der Ausführungszeiten auf der Hardware.

6.1.2.1. Parallele Performanzabschätzung

Die parallele Performanzabschätzung hat zum Ziel, die Ausführungszeit der parallelisierten Anwendung abzuschätzen. Dabei setzt sie auf die interne Darstellung des Algorithmus nach der Parallelisierung auf Daten-Ebene. Die Darstellung enthält zu diesem Zeitpunkt alle Kommunikations- und Synchronisationsinstruktionen sowie einen Prozess als Einsprungpunkt für jeden der verfügbaren Kerne. Die Zeiten für die einzelnen Tasks basieren auf den gleichen Werten, die auch zur Parallelisierung auf Task-Ebene eingesetzt werden. Da aber nun zusätzlich die Synchronisation zwischen den einzelnen Kernen betrachtet werden kann, ergeben sich weitere Verzögerungen aufgrund der in der Regel nicht vollständig balancierten Auslastung der einzelnen Kerne. Die Synchronisationsinstruktionen werden immer nur am Anfang oder Ende eines Tasks eingefügt, so dass unweigerlich Wartezeiten beim Empfänger entstehen können. Ein Teil davon kann zwar durch asynchrone Kommunikation aufgefangen werden, indem der empfangende Prozessor die Daten erst deutlich später liest, aber ein verspätetes Senden der Daten kann nicht grundsätzlich verhindert werden.

Die Abschätzung wird mit SequenceNodes modelliert. Das zugehörige Klassendiagramm ist in Abbildung 6.5a dargestellt. Für jeden Task wird ein Knoten extrahiert, der Verweise auf alle direkten Vorgänger- und Nachfolgeknoten hat. Diese werden durch Abschreiten

des Kontrollflusses auf einem einzelnen Kern ermittelt. Weitere Abhängigkeiten werden durch Kommunikation hinzugefügt: jeder sendende Knoten wird als Vorgänger von einem empfangenden markiert bzw. jeder empfangende als Nachfolger für den entsprechenden sendenden Knoten. Ein Sequenzknoten kann sich in verschiedenen Zuständen befinden, die in Abbildung 6.5b dargestellt sind. *Unresolved* bedeutet in diesem Fall, dass noch nicht alle Vorgänger im Zustand *Processed* sind. Bei einem Knoten, der auf Daten von einem anderen Kern wartet, kommt zusätzlich der Unterzustand *CommResolved* zum Einsatz, der markiert, wenn die Kommunikation zwischen den Knoten bereits aufgelöst wurde. Im Zustand *Ready* befindet sich ein Knoten, wenn alle Vorgänger abgearbeitet wurden und der Knoten vom Algorithmus weiterverarbeitet werden kann. *Processed* gibt schließlich an, dass der Knoten erfolgreich von der Performanzabschätzung verarbeitet wurde.



(a) Klassendiagramm (b) Zustandsübergänge

Abbildung 6.5.: Übersicht über SequenceNodes

Schleifen mit Kommunikation

Um die Auswirkungen von Schleifen, die Kommunikation beinhalten, auf die Gesamtlaufzeit des Algorithmus zu betrachten, werden Schleifen in drei Schritte unterteilt, die nacheinander betrachtet werden. Das Vorgehen wird eingesetzt, wenn die folgenden Bedingungen erfüllt sind:

- Eine Schleife wird parallel auf zwei Kernen ausgeführt. Dabei wird der Schleifenkopf unabhängig voneinander berechnet, jede neue Iteration kann also zu unterschiedlichen Zeiten auf den Kernen beginnen.
- Die Schleife enthält im Body mindestens einen Sendebefehl auf einem Kern und einen Empfangsbefehl auf dem anderen. Hierbei findet ein Datenaustausch statt, ohne den der empfangende Kern nicht weiterarbeiten kann. Der häufigste Anwendungsfall findet sich im Task-Pipelining, bei dem jeder Prozessor nur einen Teilschritt der gesamten Verarbeitungskette übernimmt und das Teilergebnis an einen

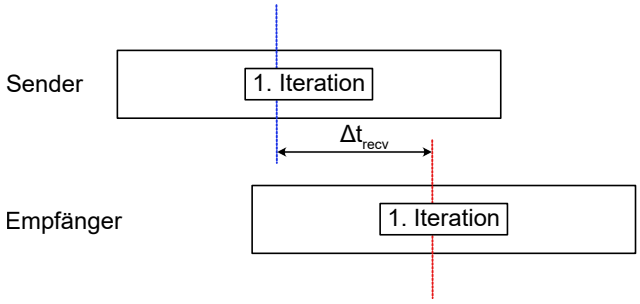
anderen Kern sendet. Durch Ausführung in einer Schleife kann ein Performanzgewinn erreicht werden, wenn alle beteiligten Kerne einen Schritt übernehmen.

Sind diese Voraussetzungen erfüllt und wird die Schleife mindestens drei Mal durchlaufen, so werden die folgenden drei Teile unterschieden:

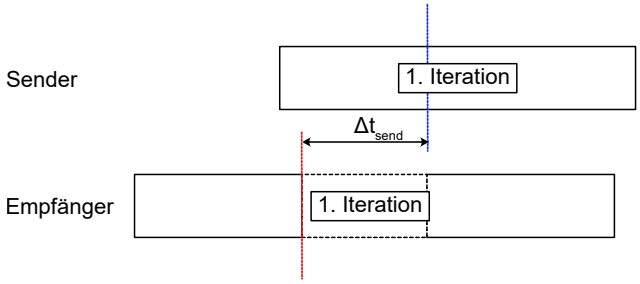
1. Zunächst wird die erste Iteration betrachtet. Es wird davon ausgegangen, dass die Schleifen zu diesem Zeitpunkt noch nicht durch Kommunikation synchronisiert sind. Damit können zwei Fälle unterschieden werden:

   a) Wie in Abbildung 6.6a dargestellt, hat der Sender die Daten gesendet, aber es dauert noch eine Verzögerung t_{recv} bis der Empfänger die Daten liest. Findet eine asynchrone Übertragung statt, so findet diese Verzögerung nur beim Empfänger statt und der Sender kann bereits weitermachen.

   b) Abbildung 6.6b zeigt den Fall, dass der Empfänger auf Daten wartet, die erst nach einer Verzögerung t_{send} vom Sender geschickt werden. Die Verzögerung findet also nur beim Empfänger statt, danach können aber wieder beide Kerne weiterarbeiten.



(a) Senden vor Empfang

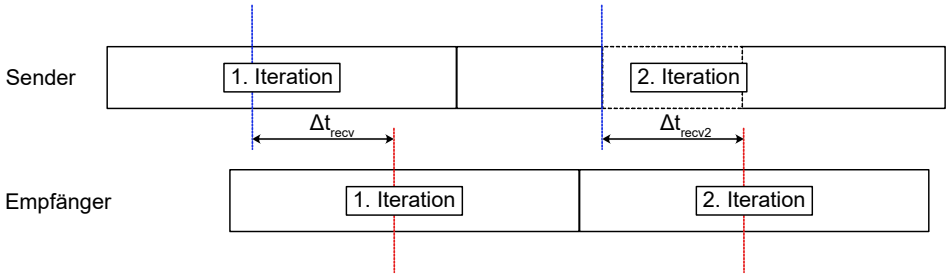


(b) Empfänger wartet auf Sender

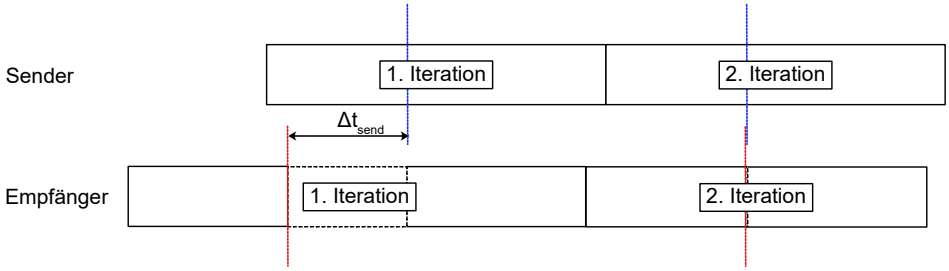
Abbildung 6.6.: Erste Iteration von synchronisierten Schleifen

In der Regel sorgt ein leichter Versatz der Startzeiten der beiden Schleifen immer für einen der beiden Fälle. Der Fall, dass sowohl Senden als auch Empfang der Daten gleichzeitig auftritt, kann vernachlässigt werden und entspricht beiden Fällen mit einer Verzögerung von 0.

2. Als nächstes wird die zweite Iteration betrachtet. Auch hier werden wieder zwei Fälle unterschieden:
   a) Konnte der Sender seine erste Iteration ohne Verzögerung ausführen, so kann er die zweite Iteration bis zum nächsten Senden ausführen, bevor er warten muss. Es kommt zu einer Verzögerung t_{send2}, da der Sender warten muss, bis der Empfänger zumindest die Daten der ersten Iteration empfangen hat. Dieser Fall ist in Abbildung 6.7a dargestellt.
   b) Musste der Empfänger in der ersten Iteration warten, so könnten beide Kerne nach dem Datentransfer mit der ersten Iteration weitermachen, bis wieder beide zum Synchronisationspunkt kommen. Beide Schleifen werden also bereits synchron bearbeitet, eventuelle Wartezeiten auf Sende- oder Empfangsseite bleiben konstant und sind auf Unterschiede zwischen den Ausführungen der Iterationen auf den unterschiedlichen Kernen zurückzuführen. Dieser Fall ist in Abbildung 6.7b dargestellt.



(a) Zweite Iteration, nachdem der Sender die Daten frühzeitig gesendet hatte



(b) Zweite Iteration, nachdem der Empfänger auf die Daten des Senders warten musste

Abbildung 6.7.: Zweite Iteration von synchronisierten Schleifen

3. Schließlich werden alle weiteren Iterationen gleichbehandelt. Nach Fall *a* der zweiten Iteration bzw. bereits mit der zweiten Iteration von Fall *b* werden beide Schleifen synchron ausgeführt und alle Verzögerungen sind identisch über alle weiteren Iterationen hinweg. Die synchronisierte Ausführung entspricht demnach der Darstellung in Abbildung 6.7b.

In der Darstellung der SequenceNodes werden so betrachtete Schleifen drei Mal dem Graph hinzugefügt. Dabei wird jeweils eine Kommunikations-Abhängigkeit zwischen den drei Stufen der Schleifen eingefügt, so dass alle möglichen Wartezeiten für t_{send} und t_{recv} modelliert werden können.

Beschreibung des Algorithmus

Quellcode 6.1 zeigt den Pseudocode des Algorithmus zur Abschätzung der parallelen Performanz. Zunächst werden alle Tasks der Anwendung als Sequenzknoten in einer Warteliste (waitingList) gesammelt (Zeile 1), wie zu Beginn von Abschnitt 6.1.2.1 beschrieben wird. Im nächsten Schritt werden alle Tasks, die keinen Vorgänger haben, in einer Bereitschaftsliste (readyList) gesammelt (Zeile 2). Diese Liste wird nun in einer Schleife elementweise abgearbeitet (Zeile 3 bis 11): beinhaltet ein Task eine Empfangs-Instruktion, so wird die minimale Startzeit des Tasks auf die Zeit des vorangegangenen sendenden Tasks gesetzt. Anschließend wird die Dauer eines Tasks berechnet. Dazu werden die gleichen Zeiten als Basis verwendet, die bereits vom Scheduler für die Parallelisierung auf Task-Ebene verwendet wurden. Auf diese Weise kann überprüft werden, ob die vom Scheduler berechnete Beschleunigung tatsächlich erreicht wurde. Wartezeiten, die durch die Kommunikation entstehen, werden durch die Beachtung der minimalen Startzeiten mitbetrachtet. Die errechnete Zeit für den Task wird anschließend zur aktuellen Zeit des zugewiesenen Kerns hinzuaddiert (Zeile 7 und 8). Nach Verarbeitung eines Tasks wird dieser aus der Bereitschaftsliste entfernt (Zeile 9) und die Nachfolger des Tasks werden aktualisiert (Zeile 10), indem der entsprechende Vorgänger als bereit markiert wird. Alle Tasks, die durch diese Aktualisierung ebenfalls bereit werden, wandern von der Warteliste in die Bereitschaftsliste (Zeile 11).

Nach erfolgreicher Abarbeitung aller Tasks sind sowohl die Warteliste als auch die Bereitschaftsliste leer und die Zeiten der einzelnen Kerne entsprechen der Ausführungszeit der gesamten parallelisierten Anwendung. Abschließend wird die berechnete Ausführungsdauer für jeden Kern einzeln ausgegeben. Der Speedup ergibt sich aus dem Quotient der sequentiellen Dauer mit der längsten Einzeldauer.

```
1  fillWaitingList()
2  fillReadyList()
3  for all entry in readyList do
4    if entry.contains(receiveInstruction) then
5      entry.minStartTime := send.getStart()
6    end if
7    calculateTaskDuration()
8    time[entry.coreId] += entry.duration
9    readyList.remove(entry)
10   updateSuccessors()
11   readyList.addAll(waitingList.getReadyNodes)
12 end for
13
14 for all coreId in allCores do
15   output(time[codeId])
16 end for
```

Quellcode 6.1: Abschätzung der parallelen Performanz

Einschränkungen statischer Performanzabschätzung
Die statische Performanzabschätzung ist nur ein grober Indikator, ob sich die Ausführungszeit einer parallelisierten Anwendung verbessert hat oder nicht. Aufgrund des Vorgehens können die folgenden Punkte nicht ohne größeren Aufwand statisch betrachtet werden:

- Abhängigkeit von Eingangsdaten: Die Ausführungszeit von Algorithmen wird sehr häufig durch die Eingangsdaten bestimmt. Unterschiedliche Daten können dafür sorgen, dass sich die Anzahl an Iterationen in Schleifen verändert oder andere Pfade von Bedingungen genommen werden. Durch Profiling, das bereits mit der sequentiellen Anwendung durchgeführt wird, wird zumindest sichergestellt, dass die Daten einem typischen Fall entsprechen und bei Bedingungen bereits Wahrscheinlichkeiten für die true und false Pfade angegeben werden.
- Optimierungen durch den C-Compiler sorgen dafür, dass nicht alle Instruktionen und Anweisungen, die sich im C-Code befinden auch tatsächlich ausgeführt werden. So können beispielsweise Schleifen komplett ausgerollt und sämtliche Logik rund um die Iterationsvariable komplett entfernt werden, wenn dies keine weiteren Seiteneffekte hat. Eine weitere Möglichkeit ist die Kostenreduzierung (engl. *strength reduction*), bei der kostenintensive Instruktionen wie Multiplikationen oder Divisionen durch günstigere Instruktionen wie Additionen oder binäres Verschieben ersetzt werden. Die möglichen Optimierungen sind nur schwer abzuschätzen, ohne sie selbst durchzuführen, weswegen eine mögliche Optimierung der Abschätzung darin bestehen würde, auf eine bereits optimierte Zwischendarstellung eines existierenden Compilers (z.B. LLVM) zu setzen, um genauere Zahlen ermitteln zu können. Das Abschätzen über die Instruktionen im Quellcode sorgt also dafür, dass eine zu hohe Ausführungszeit ermittelt wird.
- Auswirkungen von Caches: besitzt die Zielplattform Caches, so können die Kosten für Speicherzugriffe nur sehr schwer abgeschätzt werden, da zur Compile-Zeit nicht bekannt ist, welche Daten sich im Cache befinden und ob überhaupt ein Zugriff auf den Hauptspeicher erfolgen muss. Tendenziell wird die Ausführung auf der Hardware also schneller sein, als hier abgeschätzt.
- Overhead durch die Kommunikationsinfrastruktur: Die Kommunikation zwischen den einzelnen Kernen erfolgt immer über gemeinsam genutzte Ressourcen, sei es Speicher oder dedizierte Hardwareeinheiten. Zur Laufzeit können hier Schwankungen durch Arbitrierung oder exklusive Zugriffe auftreten, die nur mit großem Aufwand statisch abgeschätzt werden können. Dieser Teil der parallelen Ausführung wird daher von der Performanzabschätzung als zu gering eingestuft.

Die ersten drei Punkte treten in ähnlicher Form auch bei der statischen Abschätzung der sequentiellen Anwendung auf. Da die parallele Abschätzung aber die Zahlen des sequentiellen Falls aufgreift, eignet sie sich dennoch gut dafür, die Effizienz der Optimierungen einzuordnen.

6.1.2.2. Messung der Zeiten

Um die Einschränkungen von statischer Code-Analyse zu umgehen, bietet sich das Messen der Zeiten für die Ausführung der vom C-Compiler optimierten Anwendung auf dem Zielsystem an. Dazu wird der Quellcode so instrumentiert, dass eine Zeitmessung bei Programmstart und -ende durchgeführt wird. Um Schwankungen durch das Betriebssystem und durch Caches zu verringern, kann die Anwendung in einer Schleife ausgeführt werden und der Mittelwert nach n Ausführungen genommen werden.

Eingebettete Systeme besitzen in der Regel dedizierte Einheiten, um Zeiten akkurat ermitteln zu können. So kann bei Programmstart in der Regel entweder ein Zähler gestartet werden, der bei Programmende ausgelesen werden kann, oder ein Zeitstempel ermittelt werden, der mit einem weiteren Zeitstempel bei Programmende verglichen wird.

Eine Alternative zur Messung auf der Hardware kann das Ausführen mittels eines Simulators sein. Dabei ist zu beachten, dass zyklenakkurate Simulatoren, die zudem das Speichersubsystem inklusive der Caches simulieren, eigentlich nur bei der Prozessorentwicklung eingesetzt werden. Sie sind in der Regel sehr langsam und können bereits mehrere Minuten zur Ausführung von nur wenigen Instruktionen benötigen. Gebräuchlicher sind so genannte Instruction Set Simulatoren, die die Mikroarchitektur der zu simulierenden Prozessoren abbilden. Sie sind in der Regel nicht zyklengenau, können aber eingesetzt werden, um die Auswirkungen von Compiler-Optimierungen auf die Laufzeit zu ermitteln. Zu beachten ist zudem, dass Simulatoren häufig nur für einzelne Kerne existieren und für die Simulation paralleler Prozessoren erweitert werden müssen.

Als letzte Möglichkeit soll das Messen der Zeiten auf der Host-Plattform erwähnt werden. Aufgrund der Unterschiede bei der Hardware und dem Betriebssystem können Zeiten zwar nicht direkt verglichen werden, aber eine grobe Abschätzung der Verbesserung ist dennoch möglich. Dazu wird der parallele Code mit der gleichen Aufteilung sowie Platzierung der Synchronisations- und Kommunikationsinstruktionen ausgeführt und mit der sequentiellen Ausführungszeit verglichen. Eine schlechte Parallelisierung, bei der der Overhead sogar für eine Verlangsamung der parallelen Anwendung sorgt verhält sich auch auf dem Host so. Optimierungen, die großen Einfluss auf die Laufzeit haben, beispielsweise eine Halbierung der Zeit, werden auch auf dem Host einen ähnlichen Effekt haben. Nicht eingesetzt werden kann diese Art des Profiling bei Optimierungen, die nur kleine Auswirkungen auf die Laufzeit haben. Zum einen verhalten sich der Overhead sowie Hardware-Einflüsse wie Caches sehr unterschiedlich und zum anderen ist die Zeitmessung auf einem Desktop PC mit Betriebssystem und anderen, im Hintergrund laufenden Aufgaben, zu ungenau, um feine Änderungen nachweisen zu können. Dennoch ist es hilfreich, um eine erste grobe Einschätzung zu bekommen und die allgemeine Parallelisierung zu validieren.

6.2. Charakterisierung von Algorithmen

Die im folgenden betrachteten Algorithmen lassen sich grob in zwei Kategorien einteilen: datenflusslastige und kontrollflusslastige.

Datenflusslastige Algorithmen sind gekennzeichnet durch gleichförmige Berechnungen auf großen Datenmengen. Dies wird häufig durch Schleifen und Matrizen für die Daten dargestellt. Im Code lassen sich häufig große Blöcke oder Tasks identifizieren, die den größten Teil der Ausführungszeit ausmachen. Üblicherweise ergibt sich eine Verarbeitungskette, bei der die Daten schrittweise weiterverarbeitet werden, bis das finale Ergebnis bereitsteht. Die bekanntesten Anwendungsfelder sind die Bild- oder Audioverarbeitung, aber auch aktuelle Anwendungsfälle aus dem Bereich des Deep Learning mit neuronalen Netzen sind sehr stark vom Datenfluss dominiert.

Kontrollflusslastige Algorithmen sind durch If-Bedingungen und Berechnungen mit kleinen Datenmengen gekennzeichnet. Im Code stellt sich dies in der Regel durch eine große Anzahl an kurzen Tasks dar, die stark vernetzt sind. Solche Anwendungen finden sich häufig in der Steuerungs- und Regelungstechnik, bei der Anhand von Eingangsdaten und dem aktuellen Zustand möglichst schnell eine Entscheidung getroffen werden muss.

MATLAB® wird in beiden Fällen gerne zur Entwicklung eingesetzt. Datenflusslastige Anwendungen profitieren von dem einfachen Umgang mit Matrizen, der direkt in den Datentypen integriert ist und somit häufig den Einsatz von expliziten Schleifen einsparen kann. Bei kontrollflusslastigen Anwendungen wird das Design komplexer Systeme geschätzt, welches häufig im Zusammenspiel mit Simulink für eine einfache Simulation und Ausführung sorgt.

Viele Anwendungen haben zwar Teile aus beiden Bereichen, in der Regel dominiert jedoch eine Ausprägung die Ausführungszeit des Algorithmus. Zur Evaluation dieser Arbeit werden sowohl datenfluss- als auch kontrollflusslastige Algorithmen betrachtet, um den Unterschied im Vorgehen und der Relevanz der einzelnen Ebenen darzustellen.

6.3. Beschreibung der Testanwendungen

6.3.1. Schnelle Fourier-Transformation

Die schnelle Fourier-Transformation (engl. *fast Fourier transform*, FFT) wurde in dieser Arbeit bereits an verschiedenen Stellen als Beispiel herangezogen: bei der Parallelisierung auf Algorithmus-Ebene in Abb. 3.2, bei der Ausnutzung von bekannten Algorithmus-Eigenschaften in Abschnitt 5.2.1 und beim Verschieben von Initialisierungscode in Abschnitt 4.3.3. An dieser Stelle soll nun die Parallelisierung über alle Ebenen hinweg betrachtet werden.

Die FFT ist ein eindeutig datenflusslastiger Algorithmus, der die meiste Zeit auf Daten in Matrizen in Anzahl der Wertigkeit der Transformation angewendet wird. Der Algorithmus kommt dabei komplett ohne bedingte Ausführungen aus und ist allein von der Größe der Eingangsdaten abhängig. Im Weiteren wird ein Eingangsvektor mit 4096 Werten verwendet, um den Fokus auf den Datenfluss in der Anwendung zu setzen.

Die FFT wird normalerweise als Vorverarbeitung bei der Signalverarbeitung verwendet und bildet somit nur einen Teilschritt der Verarbeitungskette. Als Test soll hier eine reine FFT dienen, die in einer Schleife ausgeführt wird und auf Vektoren aus zufälligen Daten arbeitet, wie es in Quellcode 6.2 dargestellt ist.

(a) Eingangsbild

(b) Ausgangsbild

Abbildung 6.8.: Streifendetektion zur Weltraumschrotterkennung

```
N = 4096;
for n = 1:10
  input = rand(N,1);
  output = fft(input);
  disp(output)
end
```

Quellcode 6.2: MATLAB®-Code der FFT

6.3.2. Streifendetektion

Als zweiter Anwendungsfall wird ein Algorithmus zur Streifendetektion eingesetzt, wie er beispielsweise bei der Erkennung von Weltraumschrott von Satelliten eingesetzt wird, um frühzeitig Raumschrott zu erkennen, der mit dem Satellit kollidieren könnte. Ein Beispiel ist in Abbildung 6.8 dargestellt, bei dem die erkannten Streifen im rechten Bild farblich hervorgehoben werden.

Der Algorithmus setzt sich aus den folgenden Teilschritten zusammen, wie sie auch in Quellcode 6.3 dargestellt sind:

1. Graustufenumwandlung: Zunächst wird das farbige Eingangsbild in ein Graustufenbild umgewandelt. Dabei wird die folgende Gewichtung verwendet: $val = (0.3 * red) + (0.59 * green) + (0.11 * blue)$. Diese Gewichtung basiert darauf, dass die Wellenlänge der Farben und damit auch das menschliche Empfinden unterschiedlich ist. Rot hat die höchste Wellenlänge, grün die niedrigste und blau liegt dazwischen. Die weiteren Schritte arbeiten nicht auf den einzelnen Farbkanälen, so dass bereits frühzeitig auf eine Graustufendarstellung mit geringerem Speicherbedarf gewechselt werden kann.
2. Kantenerkennung: Zur weiteren Verarbeitung wird eine binäre Schwarz-Weiß-Darstellung der Kanten des Eingangsbildes benötigt. Dazu kommt der bewährte Can-

ny-Algorithmus [115] zum Einsatz. Er ist ein mehrstufiger Algorithmus, der mit einem Filter basierend auf der Ableitung der Gauß-Verteilung sowohl in X- als auch in Y-Richtung beginnt. Neben der Reduktion von Rauschen wird so anhand der Intensität der Gradienten das Bild gefiltert. Im nächsten Schritt werden potentielle Kanten auf Kurven mit einer Breite von einem Pixel reduziert, indem alle Pixel entfernt werden, die nicht dem Maximum entsprechen. Abschließend werden Pixel an den Kanten mit Hysterese-Schwellenwertbildung anhand des Ausschlags der Gradienten entweder behalten oder entfernt.

3. Hough-Transformation: Die Hough-Transformation ist ein ehemals patentiertes Verfahren [116], das eingesetzt wird, um geometrische Formen wie Geraden in binären Schwarz-Weiß-Darstellungen aufzufinden. Die in MATLAB® integrierte Implementierung setzt dabei auf die folgende, parametrisierte Darstellung einer Linie: $rho = x * cos(theta) + y * (sin(theta))$. Als Ergebnis liefert die Funktion *rho*, den Abstand vom Ursprung zur Linie entland eines Vektors, der senkrecht auf der Linie steht und *theta*, den Winkel zwischen der X-Achse und diesem Vektor.
4. Extraktion der Spitzenwerte: die höchsten Werte der Hough-Transformation entsprechen den gesuchten Objekten und können verwendet werden, um die Objekte im Bild zu markieren. Die MATLAB®-Funktion `houghpeaks` wird mit der Anzahl an Spitzenwerten, die zurückgegeben werden sollen (hier: 50), aufgerufen und liefert eine Matrix, die die Reihen- und Spaltenkoordinaten der Spitzenwerte beinhalten.
5. Verbinden der Hough-Linien: Abschließend werden die gefundenen Spitzenwerte mit Hilfe von Grenzwerten hinsichtlich der Länge von gefundenen Linien gefiltert und sollten sie den Kriterien entsprechen, werden sie als Lösungen im Eingangsbild eingezeichnet.

```
1 function [point1Arrays, point2Arrays] = streak_detection(img)
2   gray = rgb2gray(img);
3   [E, thresh] = edge(double(gray), 'Canny');
4
5   [H,T,R] = hough(E);
6   P = houghpeaks(H, 50);
7   [point1Arrays, point2Arrays] = cgen_houghlines(E,T,R,P);
8 end
```

Quellcode 6.3: MATLAB®-Code der Streifendetektion

6.3.3. Kontrollflusslastiger Algorithmus aus der Luftfahrt

Zur weiteren Evaluation soll ein kontrollflusslastiger Algorithmus aus der Luftfahrt dienen. Er unterliegt einem *Geheimhaltungsvertrag* (NDA, engl. *non-disclosure agreement*) und die genaue Funktionalität darf an dieser Stelle nicht weiter beschrieben werden. Die wichtigsten Charakteristiken sind:

- Der Algorithmus wird in regelmäßigen Abständen aufgerufen und trifft anhand von Eingangswerten, die von Sensoren erfasst wurden, Entscheidungen, welche Signale zur Ansteuerung von Aktoren sowie zur Protokollierung der wichtigsten Daten generieren.
- Der Algorithmus arbeitet nicht auf großen Datenmengen, so dass die meiste Zeit zur Auswertung von Bedingungen und somit dem Kontrollfluss der Anwendung zugebracht wird. Schleifen haben häufig nur zwei bis sechs Iterationen, nur in Ausnahmefällen bis zu 100 mal.
- Als Zielsystem soll der Algorithmus auf einem Infineon AURIX™ TC297 ausgeführt werden und mittels der drei verfügbaren Kerne soll die Laufzeit optimiert werden.
- Der Algorithmus besteht aus rund 1400 Tasks, deren durchschnittliche Dauer kürzer als 1 µs beträgt. Eine erste Abschätzung der Dauer auf der Zielplattform ergibt eine Gesamtdauer von 77 µs.
- Es wird üblicherweise mit skalaren Daten oder Vektoren mit weniger als 10 Elementen gearbeitet. Es kommen keine mehrdimensionalen Variablen zum Einsatz, weshalb Schleifen in der Regel auch nicht verschachtelt sind.

Im Vergleich mit den beiden datenflusslastigeren Anwendungen bringt dieser Anwendungsfall ganz neue Herausforderungen mit sich: da nicht lange Schleifen die Ausführung dominieren, muss die Parallelisierung auf Ebene der Tasks ansetzen. Aufgrund der höheren Anzahl an relativ kurzen Tasks und des vergleichsweise hohen Overheads zur Übertragung von Daten, wird eine sehr feine Parallelisierung mit einem genauen Hardware-Modell benötigt.

6.4. Parallelisierung der Testanwendungen

Anhand der Parallelisierung der FFT soll das generelle Vorgehen noch mal konkreter dargestellt werden, weshalb näher auf die Einzelschritte und Teilergebnisse eingegangen wird. Bei den anderen beiden Algorithmen liegt der Fokus dagegen rein auf den angewendeten Optimierungen je Ebene sowie dem finalen Ergebnis.

6.4.1. Schnelle Fourier-Transformation

6.4.1.1. Optimierungen auf Algorithmus-Ebene

Wie bereits in Abschnitt 5.2.1 dargestellt wurde, ist die Funktion `fft` Teil des Sprachumfangs von MATLAB®. Damit kann eine optimierte Version bereitgestellt werden, die automatisch bei der Umwandlung von MATLAB®nach C eingesetzt werden kann. Bei der Implementierung wird auf eine Hauptfunktion namens `fftifft` gesetzt, die viele verschiedene Fälle an Eingangsdaten unterscheidet und Daten entsprechend aufbereitet. Sie ist in Quellcode 6.4 dargestellt. Aufgrund der mächtigen Konstantenpropagation und Datentyp-Inferenz bei der Code-Konvertierung werden alle nicht genutzten Teile aus der Funktion entfernt und nur der relevante Teil bleibt im C-Code bestehen. Da bei Ver-

wendung der FFT in der Regel auch die inverse FFT benötigt wird, wurde zur besseren Wiederverwendbarkeit der Funktion noch ein boolescher Parameter hinzugefügt, um die gleiche Struktur auch für die Inverse verwenden zu können.

Zunächst wird in Zeile 2 und 3 der Fall abgedeckt, dass die FFT von ganzen Zahlen berechnet werden soll. Die Eingangsdaten werden in diesem Fall zu Doubles konvertiert und die Funktion `fftifft` wird rekursiv erneut aufgerufen. Als nächstes wird in den Zeilen 4 bis 11 der Fall behandelt, dass die FFT mit Arrays mit mehr als 2 Dimensionen aufgerufen wird. In diesem Fall wird mittels der MATLAB®-Funktion `reshape` eine temporäre Matrix mit 2 Dimensionen erzeugt, auf die wiederum `fftifft` angewendet wird. Der Code in den Zeilen 12 bis 19 wird verwendet, wenn die Eingangsdaten als Zeilenvektor vorliegen, um `fftifft` mit einem temporären Spaltenvektor aufzurufen. Da die FFT nur für komplexe Zahlen berechnet werden kann, wird in den Zeilen 20 bis 25 durch eine Addition eines Imaginärteils von 0 dafür gesorgt, dass die Eingangsdaten als komplexe Daten behandelt werden. Schließlich findet sich in den Zeilen 27 bis 36 der tatsächliche Aufruf der FFT. Dabei wird eine zusätzliche Entscheidungsvariable p (siehe Abschnitt 4.3.2.3) eingesetzt, um die Verarbeitung der FFT bei Bedarf auf p unabhängige Teile aufzuteilen. Wird p auf 1 gesetzt, so wird direkt die wohlbekannte FFT-Implementierung von Cooley und Tukey [117] aufgerufen. In allen anderen Fällen wird die Funktion `fftifft_decomposition` aufgerufen, die abhängig von p unabhängige Aufrufe zur FFT generiert. Die genauere Beschreibung kann Abschnitt 5.2.1 entnommen werden.

Der tatsächliche Kernel der FFT wird in der Funktion `fftifft_kernel` berechnet. Bei der Berechnung werden fünf spezielle Fälle unterschieden: 2 Faktoren und die Anzahl der Elemente ist ohne Rest durch 2 teilbar, dies ebenso für die Zahlen 3, 4 und 8 und schließlich eine Implementierung für eine Anzahl, die keine Potenz von 2 ist. Die Konstantenpropagation sorgt dafür, dass nur der relevante Teil der Anwendung im C-Code erhalten bleibt.

Die Optimierung auf dieser Ebene fokussiert sich somit auf die folgenden Punkte:

- Optimierung der benötigten Datentypen
- Minimierung der benötigten Funktionsaufrufe und temporären Variablen
- Generierung einer einstellbaren Anzahl an unabhängigen Teilen der Berechnung

Die ersten beiden Punkte ergeben ohne weitere Optimierungen bei der sequentiellen Ausführung einen Performanzgewinn von rund 2,6 %. Der dritte Punkt hat allein noch keinen Einfluss auf die Laufzeit und muss zusammen mit der weiteren Verarbeitung auf der Code- und Task-Ebene betrachtet werden.

Die etablierte Lösung von Mathworks konvertiert nicht direkt den MATLAB®-Code, sondern setzt generischen C-Code ein, bei der die Größe der FFT als Parameter genutzt wird. Der hier vorgestellte Ansatz ist flexibler und generiert Code, der tatsächlich an die verwendete Größe und den eingesetzten Datentyp angepasst ist. Alle durchgeführten Optimierungen sind unabhängig von der Zielplattform, einzig die Aufteilung in n unabhän-

```
function [y] = fftifft(x, inverse)
  if (~isfloat(x))
    y = fftifft(double(x), inverse);
  elseif (ndims(x) > 2)
    %CMD?: cgen_stmt_fuse('infunc')
    x2 = reshape(x, size(x,1), numel(x) / size(x,1));

    %CMD?: cgen_stmt_inline()
    y2 = fftifft(x2, inverse);

    y = reshape(y2, size(x));
  elseif (isrow(x))
    %CMD?: cgen_stmt_fuse('infunc')
    x2 = x.';

    %CMD?: cgen_stmt_inline()
    y2 = fftifft(x2, inverse);

    y = y2.';
  elseif (isreal(x))
    %CMD?: cgen_stmt_fuse('infunc')
    x2 = x + 0i;

    %CMD?: cgen_stmt_inline()
    y = fftifft(x2, inverse);
  else
    %CMD?: p = cgen_decision(1, 'numproc', 'Number of processors', 'int
        ', 'select', 1, '#cpu');

    if (p == 1)
      %CMD?: cgen_stmt_inline()
      y = fftifft_cooleytukey(x, inverse);
    else
      y = fftifft_decomposition(x, p, inverse);
    end
  end
end
```

Quellcode 6.4: MATLAB®-Code der Funktion `fftifft`

gige Tasks benötigt die Information über die zur Verfügung stehenden Ausführungseinheiten.

6.4.1.2. Optimierungen auf Code-Ebene

Die ersten, noch plattformunabhängigen Optimierungen auf Code-Ebene werden während der Konvertierung von MATLAB®-Code nach C-Code durchgeführt und befinden sich als extra Anweisungen an die Code-Konvertierung als Kommentare im MATLAB®-Code. Ziel ist es, die Anzahl an unabhängigen Tasks für die Task-Ebene zu optimieren. Dabei liegt der Fokus auf der Generierung von analysierbaren For-Schleifen und der Identifikation von Code-Teilen, die nur einmal bei der Initialisierung durchlaufen werden. Zusätzlich kommt *Pre-execution*, eine Ausführung zur Compile-Zeit, zum Einsatz, um konstante Berechnungen wie die Twiddle-Faktoren bereits vorab durchzuführen.

Hier eine Auflistung der wichtigsten Transformationen und wo sie angewendet wurden:

- Zusammenführen von Parametern: mit der Anweisung `func_paramcanmerge(a, b)` können die Parameter a und b einer Funktion zu einer einzelnen Variablen zusammengeführt werden. Dies wird in der Funktion `fftifft_kernel`, welche die tatsächliche FFT-Berechnung durchführt, eingesetzt, um die Eingangsdaten x mit den Ausgangsdaten y zusammenzuführen. Dies ist möglich, da die Größe beider Variablen identisch ist und nach der Berechnung alle Werte der Eingangsvariablen x vollständig überschrieben sind. Im generierten C-Code findet sich keine Variable x mehr, so dass kein weiterer Speicher für eine Kopie der Daten benötigt wird. Bei der späteren Parallelisierung auf Task und Daten-Ebene sorgt dies für eine Einsparung auf jedem beteiligten Kern.
- Entrollen von Schleifen: das Entrollen von Schleifen wird hier eingesetzt, um die Erzeugung kleiner Schleifen in anderen Schleifen zu verhindern. Es wird an zwei Stellen eingesetzt: zunächst wird es in `fftifft_kernel` eingesetzt, um bei drei verschachtelten For-Schleifen zu bleiben. Wie im Beispiel Quellcode 6.5 gezeigt wird, gibt es Code-Abschnitte, die für eine bestimmte Anzahl an Faktoren implementiert sind (hier: für 3, siehe Zeile 1). Dadurch ergeben sich Matrixoperationen mit eben dieser Anzahl an Iterationen (Zeile 8 und 22). Um unnötige Komplexität bei der statischen Analyse zu verhindern, können diese Schleifen mittels Entrollens entfernt werden. Ein weiterer Einsatzzweck ist die Kombination aus Entrollen von Schleifen und der Konstantenpropagation, wie es in Quellcode 6.6 dargestellt ist. Die Anzahl an Iterationen der For-Schleife kann durch die Konstantenpropagation berechnet werden und die Parameter können in die Funktion `fftifft_kernel` hineingezogen werden, so dass sich C-Code wie in Quellcode 6.7 dargestellt ist, ergibt. Es werden fünf Funktionsaufrufe hintereinander generiert, die ohne Schleife aufgerufen werden. Durch die Konstantenpropagation sind bei allen Aufrufen nur noch zwei Parameter übrig. Zudem wurden fünf verschiedene Funktionen generiert, die sich in der Implementierung abhängig von den ursprünglichen Eingangsgrößen unterscheiden. Für die Parallelisierung auf Task-Ebene bietet dies einige Vorteile: weniger Abhängigkeiten zwischen Tasks, da die Anzahl an Variablen reduziert wurden. Multiple Funktionsaufrufe, die individuell und unabhängig voneinander parallelisiert werden können.
- Vorberechnung: in der Funktion `fftifft_init` werden die Twiddle-Faktoren, Konstanten, die bei der Berechnung der FFT benötigt werden, berechnet. Sie sind von der Größe der FFT abhängig und werden mit den trigonometrischen Funktionen Sinus und Cosinus berechnet. In Quellcode 6.8 ist ein Codeausschnitt dargestellt, in dem die Variable `factors` mit Daten gefüllt wird. Dies erfolgt in einer While-Schleife (Zeile 3), in der abhängig von $n2$ eine temporäre Variable f geschrieben wird und in Zeile 20 in `factors` übergeht. Damit die Konstantenpropagation die Anzahl an Iterationen der Schleife berechnen kann, wird mit der Anweisung `loop_maxtripcount` die Berechnungsvorschrift in Abhängigkeit zur Größe der FFT $n2$ angegeben (Zeile 2). Nun kann mit der Anweisung `scc_preexecute` (Zeile 1) die Berechnung bereits zur Compile-Zeit erfolgen. Als Ergebnis ergibt sich die folgende Initialisierung im generierten C-Code:

```
1   static const int32_t init1[6] = {2, 4, 4, 4, 4, 4} ;
```

Die Vorberechnung sorgt dafür, dass die Anzahl an Tasks reduziert wird, die nicht relevant für die Parallelisierung der Anwendung sind. Somit erhöht sich das Parallelisierungspotential der Tasks, die auf der Task-Ebene betrachtet werden.

- Fusion von Anweisungen: die Anweisung `stmt_fuse` kann verwendet werden, um temporäre Variablen zu verhindern, die Zwischenwerte von einer Anweisung an eine andere weitergeben. Mit dem Parameter `infunc` kann dies über Funktionsgrenzen hinweg gemacht werden, um zudem potentiell die Anzahl an Argumente beim Funktionsaufruf zu reduzieren. Zum Einsatz kommt es in der Funktion `fftifft`, wie sie in Quellcode 6.4 dargestellt ist. Um verschiedene Fälle hinsichtlich der Größe und der Struktur von x zu unterstützen, werden die Daten in eine angepasste Variable $x2$ überführt (Zeile 6, 14 und 22) und die Funktion `fftifft` rekursiv erneut aufgerufen. Diese Aufrufe werden mit der Anweisung `stmt_inline` kombiniert, so dass im final generierten C-Code nur ein einzelner Aufruf von `fftifft` bleibt, in dem die Eingangsdaten bereits entsprechend vorbereitet sind, ohne dass eine weitere temporäre Variable benötigt wird. Diese Optimierungen sorgen dafür, dass die multiplen unabhängigen Aufrufe der FFT, die auf der Algorithmus-Ebene erzeugt werden können, für sich selbst optimiert sind und möglichst wenig unnötige Funktionsaufrufe oder temporäre Variablen beinhalten.
- Herausziehen der Initialisierung: die Berechnung der FFT wird in einer Schleife mehrere Male hintereinander auf unterschiedliche Daten angewendet. Die Initialisierung von globalen Variablen, die während der Berechnung benötigt werden, muss dabei nur einmal erfolgen. Zur Optimierung kann die Anweisung `INIT_CODE` verwendet werden, um Code außerhalb der Hauptfunktion zu ziehen. Damit kann die Initialisierung vor der eigentlichen Berechnung und auch nur ein Mal außerhalb der Schleife erfolgen.

Alle Optimierungen, die bislang auf dieser Ebene gemacht wurden, optimieren die reine Darstellung des Algorithmus im C-Code. Dazu wurden keinerlei Informationen über die Zielplattform benötigt. Ziel ist es, die Tasks für die nächste Ebene auf das nötigste zu reduzieren und Abhängigkeiten sowie Overheads zu minimieren.

6.4.1.3. Optimierungen auf Task-Ebene

Die hierarchische Darstellung der Tasks nach der Code-Ebene mit allgemeinen Optimierungen ist in Abbildung 6.9 dargestellt. Gezeigt wird die reine Berechnung der FFT. Die Initialisierung wurde herausgezogen und wird weder in der sequentiellen noch in der

```
    if (f == 3 && mod(n, 3) == 0)
      epi3 = twiddels(1+L2*stride);

      for l=1:m
        for k=0:L:(n-1)
          for j=0:(L2-1)
            %CMD?: cgen_perf_loopunroll(3)
            data3 = x(k+j+L2*(0:2)+1, l);

            temp1   = data3(1);
            temp2   = twiddels(1+j*stride*1) * data3(2);
            temp3   = twiddels(1+j*stride*2) * data3(3);
            temp4   = temp2 + temp3;
            temp5   = temp1 - temp4/2;
            temp6   = (temp2 - temp3) * imag(epi3);

            data3(1)  = temp1 + temp4;
            data3(2)  = temp5 + complex(-imag(temp6), real(temp6));
            data3(3)  = temp5 + complex(imag(temp6), -real(temp6));

            %CMD?: cgen_perf_loopunroll(3)
            y(k+j+L2*(0:2)+1, l) = data3;
          end
        end
      end
    end
```

Quellcode 6.5: Auszug aus fftifft_kernel

```
    %CMD?: cgen_scc_loopunroll()
    for i=2:length(factors)
    y = fftifft_kernel(y, twiddels, factors(i), prod(i), inversesign,
        ispowof2);
    end
```

Quellcode 6.6: Kombination aus Entrollen von Schleifen und Konstantenpropagation

parallelen Performanzabschätzung mit einbezogen. Die Funktion `fft` besteht auf der untersten Ebene aus vier Tasks: der erste Task For1 wandelt die Eingangsdaten in den komplexen Zahlenbereich um. Der zweite Task ist die Funktion `fftifft`, die, wie in den vorigen Abschnitten beschrieben, den Hauptteil der Berechnung der FFT beinhaltet. Der dritte Task For2 transformiert das Ergebnis zurück in den reellen Raum und der letzte Task, For3, sammelt alle Ergebnisse in der Variablen *y_data*. Auf dieser Ebene bilden die Tasks eine Verarbeitungskette, bei der sie nacheinander ausgeführt werden müssen. Die Funktion `fftifft` besteht aus sechs Aufrufen der Funktion `fftifft_kernel`. Diese wurden durch Entrollen einer Schleife generiert und haben Abhängigkeiten untereinander, so dass sie immer in derselben Reihenfolge ausgeführt werden müssen. Der Algorithmus hat in dieser Darstellung kein echtes Potential für die Parallelisierung auf Task-Ebene: alle besprochenen Tasks müssen nacheinander ausgeführt werden, so dass keine Beschleunigung durch eine parallele Ausführung erreicht werden kann. Aufgrund der unausgewogenen Ausführungszeiten der Tasks ist auch kein Pipelining auf Task-Ebene für eine effizientere Ausführung möglich.

```
1       // <lib>\fftifft_cooleytukey.m(34:9-31):  cgen_scc_loopunroll()
2       // <lib>\fftifft_cooleytukey.m(36:3-79):  y = fftifft_kernel(y,
            twiddels, factors(i), prod(i), inversesign, ispowof2);
3       fftifft_kernel_1(y_data, g_twiddels_data);
4       fftifft_kernel_2(y_data, g_twiddels_data);
5       fftifft_kernel_3(y_data, g_twiddels_data);
6       fftifft_kernel_4(y_data, g_twiddels_data);
7       fftifft_kernel_5(y_data, g_twiddels_data);
```

Quellcode 6.7: Generierter C-Code bei Kombination aus Entrollen von Schleifen und Konstantenpropagation

```
1  %CMD?: cgen_scc_preexecute()
2  %CMD?: cgen_loop_maxtripcount(ceil(log2(cgen_scc_getmax(n2))));
3  while n2 > 1
4    if (rem(n2, 4) == 0)
5      f = int32(4);
6    elseif (rem(n2, 2) == 0)
7      f = int32(2);
8    elseif (rem(n2, 3) == 0)
9      f = int32(3);
10   else
11     f = n2;
12     for f2=5:2:n2
13       if (rem(n2, int32(f2)) == 0)
14         f = int32(f2);
15         break;
16       end
17     end
18   end
19
20   factors(1, end+1) = f;
21   n2 = idivide(n2, f);
22 end
```

Quellcode 6.8: Vorberechnung von konstanten Werten

Ein Wechsel zwischen den verschiedenen Zielplattformen offenbart, dass ein wenig Potential in der Parallelisierung der einzelnen Kernels besteht, das bei Plattformen mit geringem Kommunikationsoverhead einen Vorteil bringen kann. Schlug der Scheduling-Algorithmus auf dem Raspberry Pi 2 keine Parallelisierung vor, so erwartet er auf dem x86 einen Speedup von 1,52 und auf dem Infineon AURIX einen von 1,29.

6.4.1.4. Kombination aus Algorithmus-, Code- und Task-Ebene

Da die Parallelisierung auf Task-Ebene mit der bisherigen Anzahl an Tasks nicht sinnvoll ist, kann die Anzahl an Tasks auf der vorherigen Ebene optimiert werden. Dazu wird zunächst der Fokus auf den zeitraubendsten Task, in diesem Fall die Funktion `fftifft`, gesetzt. Sie nimmt bei einem Eingangssignal mit 4096 Werten rund 69 % der Ausführungszeit in Anspruch (zur Einordnung: rund 71 % bei 8192 und rund 65 % bei 1024 Werten) und bildet somit den Hot-Spot der Anwendung.

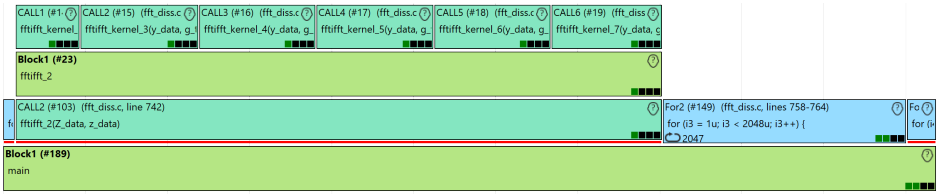


Abbildung 6.9.: FFT auf Task-Ebene

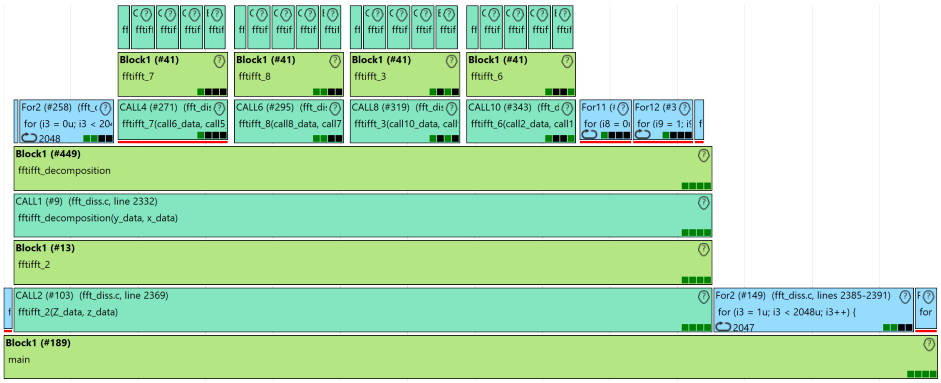


Abbildung 6.10.: FFT auf Task-Ebene mit Optimierung für 4 Kerne

Auf Algorithmus-Ebene wurde die FFT bereits für die Ausführung auf mehreren Kernen vorbereitet, indem eine Variable die Berechnung in n unabhängige Teile aufsplitten kann, die nur noch m/n-Punkte FFTs berechnen. Dafür ergibt sich ein Overhead, der beim Aufteilen und Sammeln der Daten entsteht. Diese Kopieroperationen sind notwendig, da von Systemen mit geteiltem Speicher ausgegangen wird und Kopien der Daten für jeden Kern vorliegen müssen. Für die Optimierung auf die vier Kerne des Raspberry Pi 2 ergeben sich dadurch zwei Möglichkeiten: entweder die FFT-Berechnung direkt in vier Teile aufteilen, die parallel ausgeführt werden, wie es in Abbildung 6.10 dargestellt wird oder ein zweistufiger Ansatz, bei der die FFT in zwei Aufrufe aufgeteilt wird, die anschließend ebenfalls in zwei Teile geteilt werden, wie es in Abbildung 6.11 dargestellt wird. Der Vergleich zeigt, dass der zweistufige Ansatz einen größeren Overhead verursacht. Benötigt die sequentielle Ausführung ohne Optimierung für parallele Tasks 2,28 ms, so erhöht sich die Zeit beim einstufigen Ansatz auf 2,78 ms und beim zweistufigen Ansatz auf 3,25 ms. Damit verändert sich auch der relative Anteil der tatsächlichen FFT-Berechnung an der Ausführungszeit: waren es beim Einzelaufruf noch 69 %, so sind es bei vier Aufrufen noch 47 % und beim zweistufigen Ansatz 41 %. Da sich zudem keine Verbesserung des Parallelisierungspotentials ergibt, wird im weiteren der einstufige Ansatz verfolgt.

Die Aufrufe der Funktion `fftifft` können nun parallel ausgeführt werden und sind aufgrund der Aufteilung der FFT auch in der Dauer verkürzt worden. Als nächstes sollten die nächst kürzeren Tasks betrachtet werden. Das sind in diesem Fall die beiden Schlei-

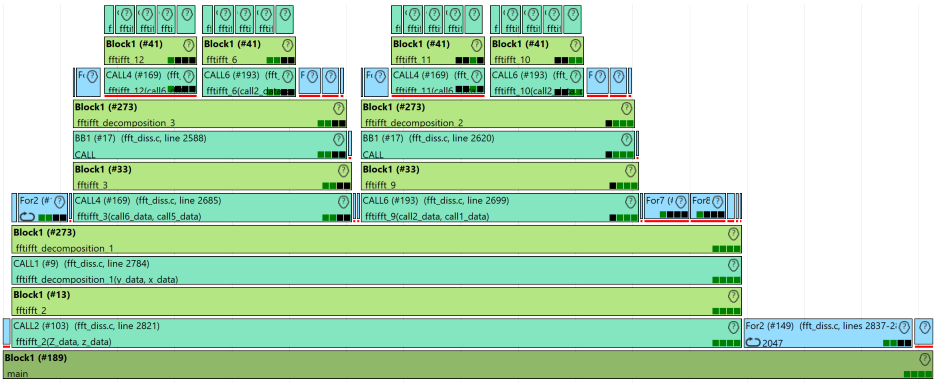


Abbildung 6.11.: FFT auf Task-Ebene mit Optimierung für 4 Kerne in 2 Stufen

fen For2(#258) und For2(#149) am Anfang und Ende der Berechnung. Schleife For2(#258) teilt die Eingangsdaten in vier Arrays auf, wobei aufeinanderfolgende Elemente in neuen Variablen gespeichert werden. Damit ist bei Annahme von verteiltem Speicher eine Parallelisierung immer mit der Übertragung des gesamten Arrays verbunden. Eine Verteilung auf zwei Kerne bringt einen kleinen Vorteil. Die Schleife am Ende kümmert sich um die Umwandlung der Ergebnisse zurück in den reellen Zahlenraum. Anwenden einer Transformation zum Aufsplitten der Variablen gefolgt von einem If-Splitting sorgt hier für eine gute Auslastung aller Kerne. Zusammen mit einer Loop-Fission Transformation bei der Umwandlung der Eingangsdaten ergibt sich eine neue Task-Struktur, wie sie in Abbildung 6.12 dargestellt ist. Zu beachten ist, dass aufgrund des Variablensplittings über Funktionsgrenzen hinweg die Funktion `fftifft_real` nun explizit aufgerufen werden muss und nicht mehr durch Inlining verborgen werden kann. Die Änderungen auf Code-Ebene sorgen dafür, dass sich die sequentielle Ausführungszeit auf 2,88 ms erhöht.

Die Schritte zurück auf die Algorithmus- und Code-Ebene sind essentiell, um eine sinnvolle Parallelisierung auf Task-Ebene zu erzielen. Ohne sie kann auf der Task-Ebene keine Parallelisierung durchgeführt werden, so dass der Scheduler nur eine sequentielle Ausführung wie in Abbildung 6.13 dargestellt ist, vorschlagen kann. Mit allen Optimierungen ist ein paralleler Schedule wie in Abbildung 6.14 dargestellt ist, möglich. Da der verwendete HEFT-Algorithmus nicht die Abhängigkeiten zu zukünftigen Tasks überprüft, müssen die vier Aufrufe der Funktion `fftifft` sowie die beiden Schleifen For2(#203) und For3(#231) mit einem Cluster-Constraint versehen werden, um eine Parallelisierung zu verhindern, die aufgrund der unterschätzten Kommunikationskosten für keinerlei Verbesserung sorgt. Ansonsten kann die Parallelisierung auf Task-Ebene automatisch erfolgen und es sind keine weiteren manuellen Optimierungen notwendig oder ersichtlich.

Der Scheduler berechnet einen Speedup von 1,72, wobei er sich auf den bereits transformierten Code mit einer sequentiellen Ausführungszeit von 2,88 ms bezieht.

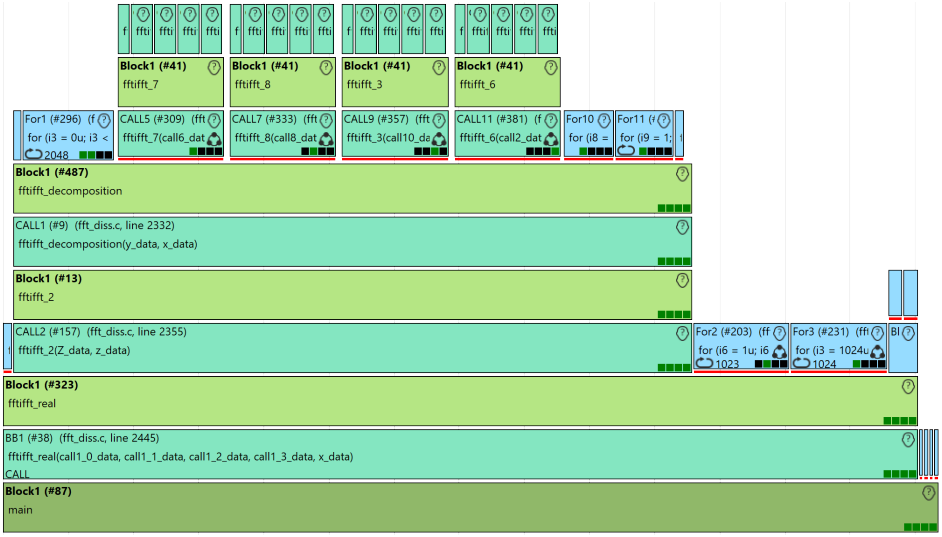


Abbildung 6.12.: FFT auf Task-Ebene mit Optimierung für 4 Kerne und angewendeten Code-Transformationen

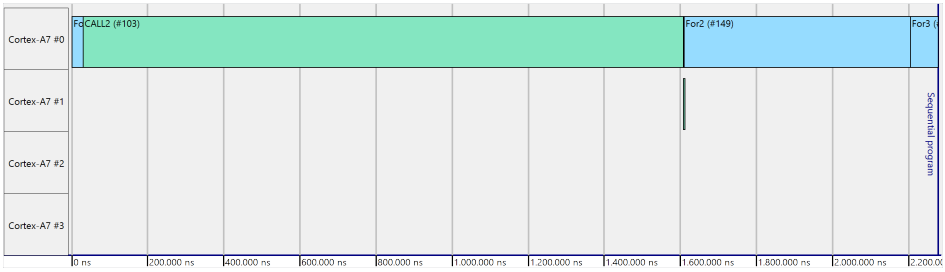


Abbildung 6.13.: Schedule der FFT ohne Optimierungen auf Algorithmus- und Code-Ebene

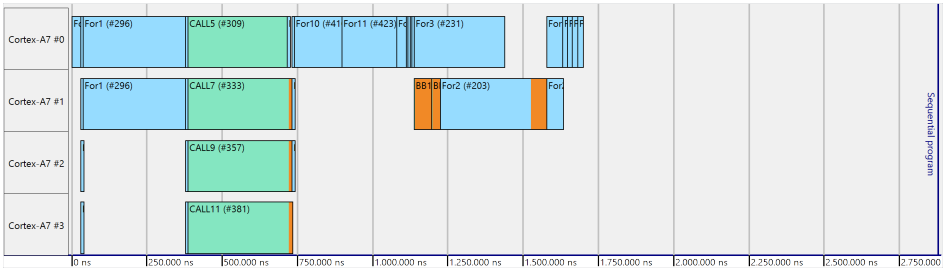


Abbildung 6.14.: Schedule der FFT mit Optimierungen für 4 Kerne

6.4.1.5. Optimierungen auf Daten-Ebene

Die Daten-Ebene hängt zunächst eng mit der Task-Ebene zusammen. Die Cluster-Constraints, die auf der Task-Ebene gesetzt wurden, verhindern eine ineffiziente Parallelisierung der gewählten Tasks, da auf Task-Ebene der Kommunikationsoverhead je Iteration der Schleife nicht genau bestimmt werden kann, wenn eine Kontrollabhängigkeit zwischen den Tasks besteht. Diese sorgt dafür, dass eine synchrone Vervielfältigung von For-Schleifen generiert wird, deren Overhead häufig größer ist als der Performanzgewinn durch die parallele Ausführung. Die Platzierung der Datentransfers, die aufgrund der parallelen Verarbeitung der Daten notwendig sind, ist bei der FFT unkompliziert und wird direkt durch den parallelen Schedule in Abbildung 6.14 vorgegeben. Alle parallel bearbeiteten Tasks wie beispielsweise CALL5 bis CALL11 arbeiten auf Daten, die sich vorher auf genau einem Kern befinden. Dieser muss die Daten also vor der eigenen Verarbeitung an alle anderen Kerne senden und nach der Verarbeitung wieder alle Ergebnisse von den anderen Kernen empfangen. Zusätzlicher Kommunikationsoverhead ist im Schedule in der Farbe orange dargestellt und zeigt auf, an welcher Stelle der Scheduler bereits Overhead mit einbezogen hat.

Das final parallelisierte Programm benötigt 25 Datentransfers, wovon 20 für die Übertragung von Datenarrays verwendet werden und fünf für die Kommunikation von Synchronisation wie in Abschnitt 4.4.1.1 beschrieben. Dabei ist zu beachten, dass die Synchronisation hier nur für die Synchronisation von If-Blöcken verwendet wurde, aber nicht bei For-Schleifen.

Die parallele Performanzabschätzung ermittelt die folgenden Werte für die einzelnen Kerne: 1,54 ms für Kern 1, 946 µs für Kern 2, 404 µs für Kern 3 und 368 µs für Kern 4. Basierend auf der ursprünglichen sequentiellen Ausführungszeit von 2,28 ms ergibt sich somit ein Speedup von 1,48, vergleicht man die sequentielle Dauer inklusive Transformationen für die parallele Ausführung, so ergibt sich ein Speedup von 1,87 mit der parallelen Ausführung.

6.4.2. Streifendetektion

Als Bildverarbeitungsalgorithmus ist die Weltraumschrott-Erkennung prädestiniert für eine Parallelisierung des Datenflusses. Durch die Aufteilung in einzelne Teilschritte können diese sehr gut einzeln optimiert werden und jeweils möglichst viele Recheneinheiten für die Verarbeitung eingesetzt werden. Da im FFT-Beispiel schon sehr genau auf das Vorgehen und die Anpassungen im Quellcode eingegangen wurde, soll anhand dieses Beispiels nur noch ein Vergleich mit einem ähnlich datenflusslastigen Algorithmus erfolgen. Im Folgenden werden deshalb allein alle Optimierungen auf den einzelnen Ebenen zusammengefasst und die notwendige Interaktion bzw. Information von einer anderen Ebene entsprechend erwähnt. Als Basis wird die sequentielle Umsetzung des Algorithmus verwendet, deren hierarchische Darstellung in Abbildung 6.15 zu sehen ist.

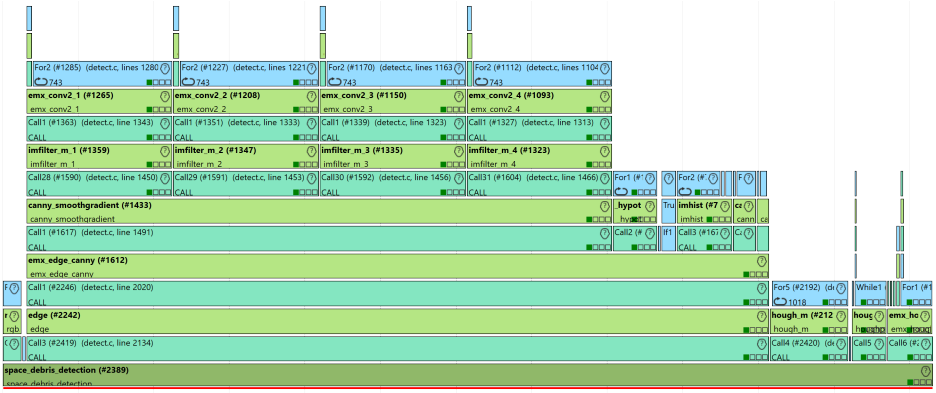


Abbildung 6.15.: Hierarchische Darstellung der Streifendetektion ohne Optimierungen

6.4.2.1. Optimierungen auf Algorithmus-Ebene

Der MATLAB®-Code der Anwendung in Quellcode 6.3 zeigt es bereits: bis auf die Extraktion der Hough-Linien mit der Funktion `cgen_houghlines` sind alle anderen Funktionsaufrufe Teil des Sprachumfangs von MATLAB®, so dass speziell optimierte Versionen während der Codegenerierung eingesetzt werden können. Neben Optimierungen der einzelnen Funktionen können zudem bereits Code-Transformationen, die während der Codegenerierung angewendet werden, an den passenden Stellen vorbereitet werden. Die folgenden Anpassungen an Bibliotheksfunktionen wurden durchgeführt, um Optimierungen bereits auf Algorithmus-Ebene zu ermöglichen:

- `rgb2gray`: Mittels der Transformationen zur Aufteilung von Matrix-Variablen und dem Aufsplitten von For-Schleifen kann die Graustufenumwandlung sehr effizient für die Ausführung auf mehreren Kernen optimiert werden. Konfigurierbar über eine Entscheidungsvariable kann die Entscheidung über die Aufteilung von den Anforderungen der späteren Code- und Task-Ebene getroffen werden.
- `edge`: Für die Kantenerkennung existieren viele verschiedene Möglichkeiten, die von der MATLAB®-Funktion abhängig von den Eingangsparametern unterschieden werden. In diesem Fall kann eine optimierte Version des Canny-Edge-Algorithmus eingesetzt werden. Dabei sorgt eine weitere Entscheidungsvariable für die gleichen Möglichkeiten wie bei `rgb2gray`. Des Weiteren können Teile des Kantenerkennungsalgorithmus mit dynamischen Größen realisiert werden, um den Speicherbedarf bei der Ausführung zu minimieren. Das Wissen über den aufrufenden Kontext kann jedoch eingesetzt werden, um die Dynamik einzuschränken und nicht die maximale Größe zu benötigen. Der Hauptteil des Algorithmus wird in der Funktion `conv2`, der Berechnung der 2-dimensionalen Faltung, verbracht. Aufgrund des internen Aufrufs kann hier eine optimierte Version der Funktion aufgerufen werden, die über einen zusätzlichen Parameter das Auffüllen (engl. *padding*) von Matrix-Variablen erlaubt. Die Funktion `conv2` wird insgesamt vier Mal aufgerufen, jeweils zwei Mal in X- sowie in Y-Richtung, wobei der zweite Aufruf die Ergebnisse des

ersten als Eingang verwendet. Die Nutzung von vereinheitlichten Array-Größen ermöglicht eine effizientere Auslastung des Speichers.

- `hough`: Auch die eigentliche Hough-Transformation profitiert von einem Aufsplitten der Daten und Schleifen.

6.4.2.2. Optimierungen auf Code-Ebene

Die Hauptaufgabe der Code-Ebene ist es, der Task-Ebene möglichst viele unabhängig zu berechnende Tasks bereitzustellen. Dies wird am besten aus der Kombination aus Variablensplitting und Schleifen-Fission erreicht. Die auf der Algorithmus-Ebene vorbereitete Transformation ermöglicht nun eine einfache Exploration der verschiedenen Optionen. Die besten Ergebnisse werden erreicht, wenn die Last zwischen den Kernen ausgeglichen über die Zeit ist und dabei möglichst wenige Daten ausgetauscht werden müssen. Für die Funktionen `hypot`, `hough` und `rgb2gray`, die jeweils nur wenige Berechnungen auf einer einzelnen Matrix durchführen, ergibt sich als beste Lösung eine Aufteilung auf vier, also alle verfügbaren Kerne des Zielsystems. Beim Canny-Edge-Algorithmus, der unabhängig voneinander in X- und Y-Richtung ausgeführt wird, hat sich gezeigt, dass eine Aufteilung auf jeweils zwei Kerne die besten Ergebnisse liefert. Durch die parallele Ausführung der beiden Richtungen können alle vier Kerne ausgelastet werden und sie jeweils auf zwei Kerne zu limitieren verringert den Kommunikationsoverhead.

6.4.2.3. Optimierungen auf Task-Ebene

Die Aufteilung auf Tasks mit Hilfe von Transformationen auf der Code-Ebene sorgt dafür, dass die Parallelisierung auf Task-Ebene weitgehend automatisch optimal gelöst werden kann, da für alle wichtigen Teilschritte bereits unabhängige Tasks für alle Kerne generiert worden sind. Lediglich für Tasks, die die Daten aufteilen bzw. einsammeln besteht Optimierungsspielraum bei der Zuweisung, indem möglichst Kerne so ausgewählt werden, dass die Kommunikation minimiert wird. Dies erfolgt im Zusammenspiel mit der Daten-Ebene, da erst dort die finale Zuweisung von Phi-Knoten, die Einfluss auf die Kommunikation haben, erfolgt.

6.4.2.4. Optimierungen auf Daten-Ebene

Durch die Art des Algorithmus ist die Platzierung auf der Daten-Ebene sehr geradlinig: Daten werden immer versendet, nachdem sie im vorigen Schritt aufgeteilt bzw. berechnet wurden und sie werden direkt vor der Weiterverarbeitung empfangen. Wichtig ist eine möglichst schnelle Datenübertragung zwischen den Kernen, um die Zeiten paralleler Datenverarbeitung zu maximieren.

6.4.2.5. Messung auf der Hardware

Der mit Hilfe dieses Verfahrens optimierte Algorithmus wurde anschließend auf einem Raspberry Pi 2 ausgeführt, um die abgeschätzte Performanz auf der Hardware zu verifizieren. Als Referenz dient die Ausführung der sequentiellen Anwendung nach der C-Code Generierung ohne weitere Optimierungen für die Hardware. Hier wird eine durchschnittliche Zeit von 2,26 ms erreicht. Die parallelisierte Anwendung benötigt 1,57 ms, was zu einem Speedup von 1,44 führt.

6.4.2.6. Einsatz von heterogenen Systemen

Mit Einsatz des Nvidia Jetson TX2 soll hier noch das Potential des Ansatzes für heterogene Systeme angerissen werden: die Optimierungen und Vorbereitungen der Anwendung auf Algorithmus- und Code-Ebene für den Raspberry Pi 2 hatten zum Ziel, möglichst viele Tasks zu generieren, die anschließend auf die Kerne verteilt werden können. Da der Code der erzeugten Schleifen sehr gleichförmig und nur mit wenig Datenabhängigkeiten zwischen den einzelnen Iterationen ist, eignet er sich auch sehr gut für die Ausführung auf einer GPU. Im Idealfall kann ein Kern der GPU eine Iteration einer Schleife berechnen und die gesammelten Ergebnisse werden anschließend in einem Array gespeichert.

Zur Programmierung der GPU muss auf die Programmiersprache CUDA[20] zurückgegriffen werden. Das Programmiermodell setzt auf so genannte Kernels, die bis zu 3-fach verschachtelte For-Schleifen darstellen, deren Eingangs- und Ausgangsdaten definiert werden und über geteilte Puffer mit Daten versorgt werden. Die Sprache setzt dabei auf C auf, hat aber eigene Befehle zur Kommunikation und zur Definition von Kernels. Am Beispiel der Faltung soll gezeigt werden, wie der C Code mit Hilfe von Transformationen auf Code-Ebene für die Ausführung auf der GPU optimiert werden kann.

In Quellcode 6.9 wird die Hauptschleife des Faltungsalgorithmus als C-Code dargestellt. Die Schleife ist zweifach verschachtelt (Zeile 8 und 9), wobei die innere Schleife zwei Zuweisungen (Zeile 10 und 17) sowie eine dritte Schleife enthält. Die äußeren beiden Schleifen können verwendet werden, um einen 2-dimensionalen Kernel für eine GPU zu generieren. Bei der parallelen Ausführung kann in jeder Iteration der zweiten Schleife ein neuer Wert für `B_data` berechnet werden.

```
1     static int32_t APad_data[1018][755];
2     double chain1_data;
3     double h_data[13];
4
5     int32_t i4, i5, i6;
6     double sum_data;
7
8     for (i4 = 1; i4 < 744; i4++) {
9       for (i5 = 1; i5 < 1019; i5++) {
10        sum_data = 0.0;
11
12        for (i6 = 1; i6 < 14; i6++) {
13          chain1_data = (double )(APad_data[i5 - 1][i6 + i4 - 2]) *
                h_data[i6 - 1];
```

```
14            sum_data += chain1_data;
15          }
16
17          B_data[i5 - 1][i4 - 1] = sum_data;
18        }
19      }
```

Quellcode 6.9: Hauptschleife der Faltung

Quellcode 6.10 zeigt diesen einfachsten Kernel. Anstelle der beiden Schleifen wird der Kernel als Funktion dargestellt, deren Parameter (Zeile 1) sowohl die Ein- und Ausgangsdaten `B_data`, `A_data` und `h_data` als auch die Charakteristiken der Schleifen darstellen. Bei den Daten wird durch die neue Endung *buf* kenntlich gemacht, dass sie über Puffer (engl. *buffer*) übertragen werden. Schleifen werden über ihre Untergrenze (lower bound, *lb*), ihre Obergrenze (upper bound, *ub*) sowie ihre Schrittweite (*stride*) definiert. Im Beispiel also über die ursprünglichen Iteratoren `i4` und `i5`, ihren Grenzen von 1 bis 744 bzw. 1019 sowie der Erhöhung um 1 je Iteration. Die Informationen werden verwendet, um die beiden Variablen `it0` und `it1` (Zeile 2 und 3) zu definieren, die über eine Bedingung (Zeile 5) bestimmen, ob eine bestimmte Berechnung ausgeführt werden soll oder nicht. Der innere Teil (Zeile 6 bis 17) entspricht quasi dem ursprünglichen Algorithmus in der innersten Schleife, jetzt aber abhängig von den beiden Variablen `it0` und `it1`. Von einer GPU können nun prinzipiell alle Iterationen unabhängig voneinander parallel ausgeführt werden. Da aber auf die Ein- und Ausgangsdaten über Puffer zugegriffen wird, wird die Performanz durch diesen Datenaustausch sehr stark begrenzt. Hinzu kommt, dass die Rechenlast je Teilstück nicht sehr groß ist: sie besteht hauptsächlich aus 13 Iterationen der For-Schleife (Zeile 12), in der neben der Indexberechnung jeweils eine Multiplikation und eine Addition ausgeführt wird.

```
1   __global__ void kernel_For14(double * B_data_buf, int *
        APad_data_buf, double * h_data_buf, int lb0, int ub0, int
        stride0, int lb1, int ub1, int stride1) {
2     int it0 = (blockIdx.y * blockDim.y + threadIdx.y) * stride0 + lb0
          ;
3     int it1 = (blockIdx.x * blockDim.x + threadIdx.x) * stride1 + lb1
          ;
4
5     if (it0 <= ub0 && it1 <= ub1) {
6       double sum_data;
7       double chain1_data;
8       int i6;
9
10      sum_data = 0.0;
11
12      for (i6 = 1; i6 < 14; i6 = i6 + 1) {
13        chain1_data = (double)APad_data_buf[i6 + it0 - 2 + 755 * (it1
              - 1)] * h_data_buf[i6 - 1];
14        sum_data = sum_data + chain1_data;
15      }
16
```

```
17          B_data_buf[it0 - 1 + 743 * (it1 - 1)] = sum_data;
18        }
19      }
```

Quellcode 6.10: Einfacher CUDA Kernel der Faltung

Mit weiteren Transformationen auf Code-Ebene können diese Unzulänglichkeiten angegangen werden:

Unter *Local Memory Caching* versteht man die Ausnutzung von lokalem Speicher der GPU, indem gezielt Daten vor deren Verwendung in den lokalen Speicher kopiert und erst die Ergebnisse nach der Verarbeitung zurück an die CPU gesendet werden. Da die Lese- und Schreibzugriffe für alle verwendeten Variablen bereits analysiert wurden, können geeignete Kandidaten durch eine Gewichtung der Zugriffe bis zum Limit des lokalen Speichers vorgeladen werden.

Eine komplexere Code-Transformation ist das so genannte *Register Tiling*. Es hat zum Ziel, die Menge an Arbeit pro Thread bzw. Task zu erhöhen. Dazu wird eine Kombination aus Schleifen-Tiling, -Verteilung und -Austausch eingesetzt. Das Ergebnis für den Kernel aus Quellcode 6.10 wird in Quellcode 6.11 dargestellt. Zunächst wird die Last erhöht, indem mit jedem Schritt der Iteratoren mehrere Berechnungen auf einmal durchgeführt werden. Ihre Berechnung wird entsprechend in Zeile 2 und 3 angepasst. Im Beispiel wurde das Tiling mit einer Größe von 2*2 durchgeführt. Das bedeutet, dass der zweidimensionale Kernel in jeder Dimension doppelt so viele Berechnungen in einer Iteration ausführen wird. Als Zwischenschritt wird dazu jeweils eine Schleife mit 2 Durchläufen eingefügt. Da diese Schleifen nur wenige Iterationen haben, können sie anschließend durch Entrollen entfernt werden, um den Overhead zu reduzieren und alle neuen Anweisungen flach auf einer Ebene zu haben. Dieses Ergebnis ist in den Zeilen 34 bis 52 dargestellt: die ursprünglich ein Mal ausgeführte Schleife wird nun vier Mal ausgeführt. Zu beachten gilt außerdem, dass eine neue Fallunterscheidung in Zeile 12 hinzugekommen ist, die Fälle abdeckt, bei denen `it0 + 1 > ub0` gilt. Dies ergibt sich durch notwendige neue Begrenzungen, die durch das Tiling hinzukommen. Je nach Algorithmus kann es sinnvoll sein, neue erzeugte Schleifen mittels Schleifen-Austausche so anzupassen, dass auf einer Schleifenebene immer die gleichen Iteratoren eingesetzt werden.

```
1   __global__ void kernel_For14(double * h_data_buf, int *
        APad_data_buf, double * B_data_buf, int lb0, int ub0, int
        stride0, int lb1, int ub1, int stride1) {
2     int it0 = (blockIdx.y * blockDim.y + threadIdx.y) * stride0 + lb0
          ;
3     int it1 = (blockIdx.x * blockDim.x + threadIdx.x) * stride1 + lb1
          ;
4
5     if (it0 <= ub0 && it1 <= ub1) {
6       int i6;
7       double sum_data;
8       double chain1_data;
9       int regTileIt0;
10      int regTileIt1;
```

```
11
12          if (it0 + 1 > ub0) {
13            double sum_data_regtile[2][2];
14            sum_data_regtile[0][0] = 0.0;
15            sum_data_regtile[0][1] = 0.0;
16            for (i6 = 1; i6 < 14; i6 = i6 + 1) {
17              chain1_data = (double)APad_data_buf[i6 + (it0 + 0) - 2 +
                    755 * (it1 + 0 - 1)] * h_data_buf[i6 - 1];
18              sum_data_regtile[0][0] = sum_data_regtile[0][0] +
                    chain1_data;
19            }
20
21            for (i6 = 1; i6 < 14; i6 = i6 + 1) {
22              chain1_data = (double)APad_data_buf[i6 + (it0 + 0) - 2 +
                    755 * (it1 + 1 - 1)] * h_data_buf[i6 - 1];
23              sum_data_regtile[0][1] = sum_data_regtile[0][1] +
                    chain1_data;
24            }
25            B_data_buf[it0 + 0 - 1 + 743 * (it1 + 0 - 1)] =
                  sum_data_regtile[0][0];
26            B_data_buf[it0 + 0 - 1 + 743 * (it1 + 1 - 1)] =
                  sum_data_regtile[0][1];
27          } else {
28            double sum_data_regtile[2][2];
29            sum_data_regtile[0][0] = 0.0;
30            sum_data_regtile[0][1] = 0.0;
31            sum_data_regtile[1][0] = 0.0;
32            sum_data_regtile[1][1] = 0.0;
33
34            for (i6 = 1; i6 < 14; i6 = i6 + 1) {
35              chain1_data = (double)APad_data_buf[i6 + (it0 + 0) - 2 +
                    755 * (it1 + 0 - 1)] * h_data_buf[i6 - 1];
36              sum_data_regtile[0][0] = sum_data_regtile[0][0] +
                    chain1_data;
37            }
38
39            for (i6 = 1; i6 < 14; i6 = i6 + 1) {
40              chain1_data = (double)APad_data_buf[i6 + (it0 + 0) - 2 +
                    755 * (it1 + 1 - 1)] * h_data_buf[i6 - 1];
41              sum_data_regtile[0][1] = sum_data_regtile[0][1] +
                    chain1_data;
42            }
43
44            for (i6 = 1; i6 < 14; i6 = i6 + 1) {
45              chain1_data = (double)APad_data_buf[i6 + (it0 + 1) - 2 +
                    755 * (it1 + 0 - 1)] * h_data_buf[i6 - 1];
46              sum_data_regtile[1][0] = sum_data_regtile[1][0] +
                    chain1_data;
47            }
48
49            for (i6 = 1; i6 < 14; i6 = i6 + 1) {
50              chain1_data = (double)APad_data_buf[i6 + (it0 + 1) - 2 +
                    755 * (it1 + 1 - 1)] * h_data_buf[i6 - 1];
```

```
51              sum_data_regtile[1][1] = sum_data_regtile[1][1] +
                    chain1_data;
52            }
53            B_data_buf[it0 + 0 - 1 + 743 * (it1 + 0 - 1)] =
                  sum_data_regtile[0][0];
54            B_data_buf[it0 + 0 - 1 + 743 * (it1 + 1 - 1)] =
                  sum_data_regtile[0][1];
55            B_data_buf[it0 + 1 - 1 + 743 * (it1 + 0 - 1)] =
                  sum_data_regtile[1][0];
56            B_data_buf[it0 + 1 - 1 + 743 * (it1 + 1 - 1)] =
                  sum_data_regtile[1][1];
57          }
58        }
59      }
```

Quellcode 6.11: CUDA Kernel mit Register Tiling

Die Transformationen wurden hier nur beispielhaft angewendet, ohne dass Eigenschaften der speziellen Hardware konkret ausgenutzt wurden. Durch den parametrisierten Ansatz ist aber eine Anpassung an den Speicher sowie die Anzahl an Rechenkernen möglich.

Die vorgestellten Codetransformationen für die GPU-Nutzung sollen einen Einblick in die Anwendbarkeit des vorgestellten Verfahrens für heterogene Systeme gewähren. Erst durch Optimierungen auf Algorithmus-Ebene und die anschließende Generierung von optimiertem C-Code, die einen Fokus auf klar analysierbare For-Schleifen hat, können die Transformationen effizient angewendet werden. Die Task-Ebene rückt etwas in den Hintergrund, aber die Bereitstellung von ausreichend Last pro Thread deckt zumindest einen Teil der Aufgaben auf der Ebene ab. Das eigentliche Scheduling wird bei GPUs vom Treiber zur Laufzeit erledigt und kann nur indirekt durch den Quellcode beeinflusst werden. Die Daten-Ebene muss schließlich die Bereitstellung der geteilten Puffer und damit die Kommunikation mit der GPU sicherstellen.

6.4.3. Kontrollflusslastiger Algorithmus aus der Luftfahrt

Wie bereits in Abschnitt 6.3.3 beschrieben, unterliegt der hier betrachtete Algorithmus der Geheimhaltung. Alle folgenden Ausschnitte und Abbildungen wurden nach einer Transformation erstellt, die alle Variablen, Funktionen und Datentypen nur über den Typ und einen Hashcode darstellt, um keine weiteren Rückschlüsse auf die Art des Algorithmus zu erlauben. Eine hierarchische Darstellung mit dieser Benamung ist in Abbildung 6.16 dargestellt. Es zeigt den Hauptteil der Verarbeitung, welcher in der Anwendung in einer Schleife ausgeführt wird. Der Algorithmus wird nicht durch eine oder mehrere große Schleifen, in denen über die Daten iteriert werden, dominiert, wie es in datenflusslastigen Algorithmen häufig der Fall ist. Die in dieser Darstellung erkennbaren Schleifen werden alle im Bereich zwischen 8 und 40 Iterationen eingesetzt. Ein Beispiel ist in Quellcode 6.12 aufgelistet. Die Schleife hat 24 Durchläufe, wobei die Zählvariable `var_e285290` auch zur Indexierung von Array-Variablen wie `var_0a993d0` oder `var_0a993d0` verwendet wird. Die Schleife ist nicht verschachtelt und die Matrizen haben in der Regel nur eine Dimension. Innerhalb der Schleife werden verschiedene Operationen gebündelt: so werden Va-

riablen auf Ergebnisse von Funktionsaufrufen gesetzt (Zeile 5), arithmetische Operationen durchgeführt (Zeile 6,7,9) und einfache Vergleiche (Zeile 8) angestellt. Die statische Performanzabschätzung ermittelt eine Ausführungsdauer von 22,22 µs für die gesamte Schleife bei Ausführung auf einem einzelnen TriCore Kern eines Infineon AURIX TC297. Dabei muss jedoch beachtet werden, dass die rein statische Abschätzung keinerlei Wahrscheinlichkeiten über die Bedingungen in der Anwendung haben und mit 50 % gewichtet wurden.

```
1  for (var_e285290 = 1; var_e285290 < 25; var_e285290 = var_e285290 + 1)
       {
2    unsigned int var_cdec530 = 127773u;
3    uint32_t var_736c030;
4
5    var_736c030 = func_9149510(var_0a993d0[(size_t)var_e285290 - 1u],
         var_cdec530);
6    var_70d62a0 = var_736c030*16807u;
7    var_add1470 = var_0a993d0[(size_t)var_e285290 - 1u] / 127773u*2836u;
8    if (var_70d62a0 < var_add1470) {
9      var_ae2d2b0[var_e285290 - 1] = 2147483647u - (var_add1470 -
           var_70d62a0);
10   }
11   else {
12     var_ae2d2b0[var_e285290 - 1] = var_70d62a0 - var_add1470;
13   }
14   var_0ca10c0[var_e285290 - 1] = (double)var_ae2d2b0[(size_t)
         var_e285290 - 1u]*4.656612875245797e-10;
15 }
```

Quellcode 6.12: Beispielhafte Schleife der kontrollflusslastigen Anwendung

In Summe setzt sich der Algorithmus aus rund 1400 kurz laufenden Tasks zusammen, die bei der Parallelisierung betrachtet werden müssen.

6.4.3.1. Optimierungen auf Algorithmus-Ebene

Der Algorithmus wurde in MATLAB® hauptsächlich durch arithmetische Operationen sowie manuell implementierte Funktionen realisiert. Die einzige Bibliotheksfunktion, die zum Einsatz kommt, ist `bin2dec`. Diese konvertiert eine als Text dargestellte Binärzahl in einen Double-Wert mit doppelter Genauigkeit. Die MATLAB®-Repräsentation ist dabei bereits optimiert, so dass für die C-Darstellung keine weiteren Optimierungen durchgeführt werden müssen. Eine optimierte Bibliothek spezieller Funktionen kann bei der Codegenerierung also nicht eingesetzt werden, so dass keine nennenswerte Optimierung bei Algorithmen dieser Art auf Algorithmus-Ebene durchgeführt werden können. Als denkbare weitere Schritte könnten weitere Algorithmen dieser Art analysiert werden, um Gemeinsamkeiten zu erkennen, die anschließend als Funktionen in einer Bibliothek gesammelt werden. Diese könnten dann für die Parallelisierung optimiert werden.

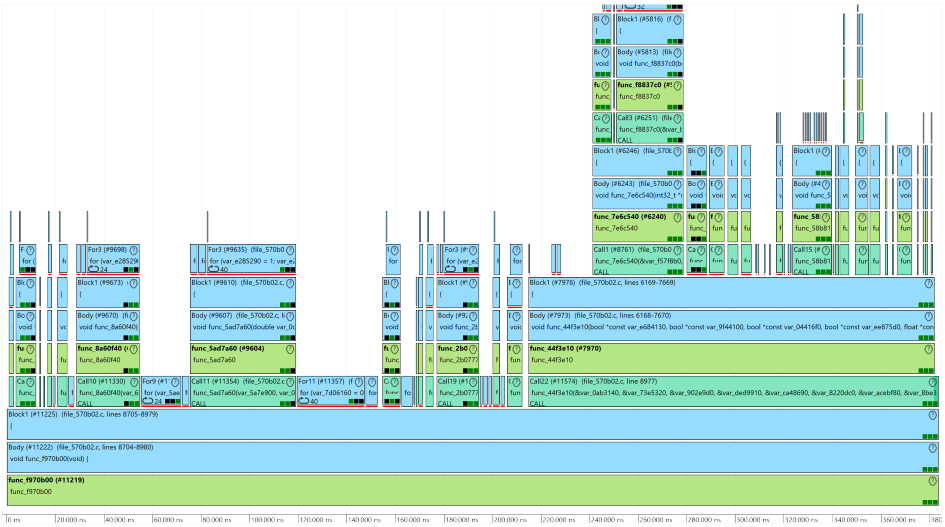


Abbildung 6.16.: Hierarchische Darstellung eines kontrollflusslastigen Algorithmus aus der Avionik

6.4.3.2. Code-, Task- und Daten-Ebene

Alle drei weiteren Ebenen werden im Folgenden sehr eng verzahnt betrachtet, da eine getrennte Darstellung den Lesefluss stören würde.

Bei der Code-Ebene liegt der Fokus auf der Darstellung im Quellcode und den Auswirkungen auf die Anzahl möglicher Tasks. Schleifen wie in Quellcode 6.12 lassen sich statisch sehr gut analysieren, da sowohl die Anzahl an Iterationen sowie alle Datenabhängigkeiten direkt ermittelt werden können. Das Zusammenfassen von Operationen in Schleifen kann während der Code-Konvertierung von MATLAB® nach C bereits automatisch übernommen werden. Wichtig ist hierbei die spärlich bedingte Konstantenpropagation (siehe Abschnitt 4.3.3.2), die gemeinsame Wertebereiche erkennen kann und somit Operationen in einer Schleife zusammenfassen kann. Da die Anzahl an notwendigen Schleifen jedoch begrenzt ist, befinden sich die meisten Operationen direkt in Basisblöcken, die durch kurze Schleifen oder If-Blöcke getrennt werden. Das hat zur Folge, dass eine große Anzahl an möglichen Tasks erzeugt wird, die große Abhängigkeiten untereinander besitzen können. Diese große Anzahl an Tasks stellt nun große Herausforderungen an die Task-Ebene: die theoretischen Möglichkeiten zur parallelen Ausführung werden zwar gesteigert, aber es bestehen sehr viele Abhängigkeiten zwischen diesen Tasks. Da aber die meisten Datentypen jedoch keine großen Daten-Arrays sind, sondern einfache, skalare Variablen, ist der Kommunikationsoverhead zum Versenden einer einzelnen Variable vergleichbar groß wie die Dauer eines Tasks. Dies limitiert die mögliche parallele Ausführung stark, da das Ausführen eines Tasks auf einem anderen Kern quasi die Laufzeit verdoppelt und somit den Gewinn durch die Parallelisierung direkt wieder verbraucht.

Auf Daten-Ebene kann dieser Overhead zwar noch mal etwas minimiert werden, indem eine einfachere Kommunikationsmethode eingesetzt wird und einzelne Transfers zusammengefasst werden, für eine gute Lösung muss dennoch noch mal der Übergang von der Code- auf die Task-Ebene betrachtet werden: auf Code-Ebene wurde der Code so generiert, dass die Reihenfolge der Operationen dem ursprünglichen Programm in MATLAB® folgt. Da bei der Konvertierung aber auch die Abhängigkeiten zwischen den einzelnen Operationen bekannt sind, können diese genutzt werden, um die Operationen, die auf gleichen Daten arbeiten, zu clustern. Auf diese Weise kann die Anzahl an Tasks sowie Abhängigkeiten, die bei der Parallelisierung auf Task-Ebene betrachtet werden müssen, drastisch reduziert werden. Von den ursprünglich 1400 Tasks müssen nun nur noch rund 300 relevante Tasks betrachtet werden.

Die Parallelisierung auf Task-Ebene für die drei verfügbaren Kerne des AURIX T297 kann somit auf ein Optimierungsproblem reduziert werden, bei dem die geclusterten Tasks auf die Kerne so verteilt werden, dass es möglichst wenig Datenaustausch zwischen den Kernen gibt, aber dennoch möglichst viele Tasks parallel ausgeführt werden können. Eine Verteilung mit dem ansonsten ausreichenden HEFT-Algorithmus (Abschnitt 5.4.1) hat sich als nicht praktikabel erwiesen, da die fehlende Betrachtung späterer Abhängigkeiten immer zu großen Overhead geführt hat, so dass die parallele Ausführung langsamer als die sequentielle wurde. Abhilfe könnte eine Erweiterung des HEFT-Algorithmus bringen, die es erlaubt, schlechte vorherige Entscheidungen wieder rückgängig zu machen. Dies wurde jedoch nicht weiter verfolgt, stattdessen konnte ein Wechsel auf Simulated Annealing (siehe Abschnitt 5.4.4) bessere Ergebnisse liefern. Als finales Ergebnis wurde die Laufzeit des Algorithmus auf der Hardware ermittelt. Dabei konnte die sequentielle Laufzeit von 77 µs auf eine parallele Dauer von 55 µs gesenkt werden. Das entspricht einem Speedup von 1,4.

6.4.4. Betrachtung heterogener Systeme

Bislang wurde die multigranulare Optimierung nur mit heterogenen Mehrkernsystemen betrachtet. Gerade bei datenflusslastigen Anwendungen bietet es sich an, spezielle Prozessoren wie GPUs oder DSPs einzusetzen, die darauf optimiert sind, möglichst viele Operationen auf größeren Datenmengen parallel auszuführen.

Mit einem multigranularen Ansatz können Algorithmen auch für diese Systeme sehr effizient optimiert werden. Das Vorgehen auf den einzelnen Ebenen hat jedoch leichte Änderungen an der Zielsetzung. Ein steter Datenaustausch zwischen Beschleuniger und CPU erzeugt einen zu großen Overhead, so dass ein großer Teil des Speedups bei der Datenübertragung bzw. Synchronisation verloren geht. Deshalb ist es sinnvoll, zunächst auf Datenebene nach den zeitkritischen Tasks zu analysieren. Die vorgestellte hierarchische Darstellung eignet sich sehr gut für diese Aufgabe. Um nun optimal von einem Beschleuniger ausgeführt werden zu können, müssen Optimierungen auf Algorithmus- und Code-Ebene angewendet werden, um das parallele Ausführungspotential zu maximieren. Dies bedeutet beispielsweise, dass Schleifen so angepasst werden, dass keine Abhängigkeiten zwischen den einzelnen Iterationen bestehen und alle theoretisch gleichzeitig ausgeführt werden können. Diese Parallelisierung kann dann abhängig vom System vom Compiler oder sogar unterstützt von einem Laufzeit-Scheduler ausgenutzt werden.

6.5. Auswertung der Ergebnisse

Bei der FFT hat sich gezeigt, dass die Algorithmus- und Code-Ebene entscheidend sind, um der Task-Ebene ausreichend Möglichkeiten zur Parallelisierung zu geben. Die Daten-Ebene hat schließlich nur wenig Optimierungspotential und muss die beste Performanz aus den Vorgaben des Schedulers herausholen. Zudem hat sich gezeigt, dass für eine Anwendung, bei der die Hauptberechnung optimal parallel verarbeitet werden kann, dennoch nur ein Speedup in der Größenordnung von 1,5 erreicht werden kann, wenn von einem System mit verteiltem Speicher ausgegangen wird. Die zusätzliche Zeit, die für die Aufteilung der Daten und die anschließende Kommunikation benötigt wird, macht in der parallelen Anwendung am Ende nahezu die Hälfte der gesamten Ausführungszeit aus. Optimierungspotential besteht somit in einem Wechsel zur Unterstützung von Systemen mit geteiltem Speicher, bei denen Variablen nicht zwingend kopiert werden müssen, sondern Zugriffe durch Arbitrierung in der korrekten Reihenfolge und unter Umständen sogar gleichzeitig erfolgen kann. Diese Erweiterung eröffnet neue Scheduling-Möglichkeiten auf der Task-Ebene, da der Overhead deutlich reduziert werden kann. Zudem erfordert sie weitere Anpassungen auf der Daten-Ebene, um die korrekte Ausführung der Anwendung zu garantieren.

Die Streifendetektion bietet als Verarbeitungskette von Daten größeres Optimierungspotential: die Aufteilung der Daten gefolgt von der Verarbeitung kann mehrfach angewendet werden, aufgrund der dadurch ebenfalls ansteigenden Kommunikation kann ein Speedup von 1,44 erreicht werden.

Ähnlich wie bei der FFT sind die entscheidenden Ebenen die Algorithmus- und die Code-Ebene, um die Verarbeitung der Daten in passende Größen zu unterteilen und die verfügbaren Kerne gut auszulasten. Auch hier könnten durch eine Erweiterung für die Nutzung von geteiltem Speicher noch weitere Optimierungen durchgeführt werden.

In Abbildung 6.17 ist dargestellt, wie sich die Struktur der Anwendung durch die Codetransformationen auf Code-Ebene verändert hat: zu sehen ist, dass Schleifen nun aufgeteilt wurden und parallel ausgeführt werden können. Am besten ist dies ersichtlich bei den Schleifen in der Faltung (`conv2`) wie beispielsweise For2#1450 oder For2#1236, aber auch bei der Funktion `rgb2gray` (ganz links) oder `_hypot`.

Im parallelen Schedule in Abbildung 6.18 zeigt das Ergebnis der Parallelisierung auf der Task-Ebene, dass die langen Schleifen aus der Faltung auf die vier verfügbaren Kerne verteilt und parallel ausgeführt werden können. Zusätzlich kann eine Vor- und Nachbearbeitung der Daten auf dem jeweiligen Kern durchgeführt werden und erst die Hough-Operationen müssen anschließend hauptsächlich von Kern 0 und Kern 1 ausgeführt werden.

Mit der Parallelisierung eines kontrollflusslastigen Algorithmus aus der Luftfahrt konnte schließlich gezeigt werden, dass der Ansatz nicht auf schleifendominierte Anwendungen beschränkt ist. Durch die genaue Bestimmung der Laufzeiten der einzelnen Tasks und die anschließende Zusammenfassung eng verzahnter Berechnungen konnte eine Beschleunigung auf der Hardware erreicht werden, die nur einen minimalen Overhead durch Kommunikation aufweist. Im Gegensatz zu den datenflusslastigen Algorithmen ist durch die Nutzung von geteiltem Speicher keine große Verbesserung zu erwarten, da der Overhead

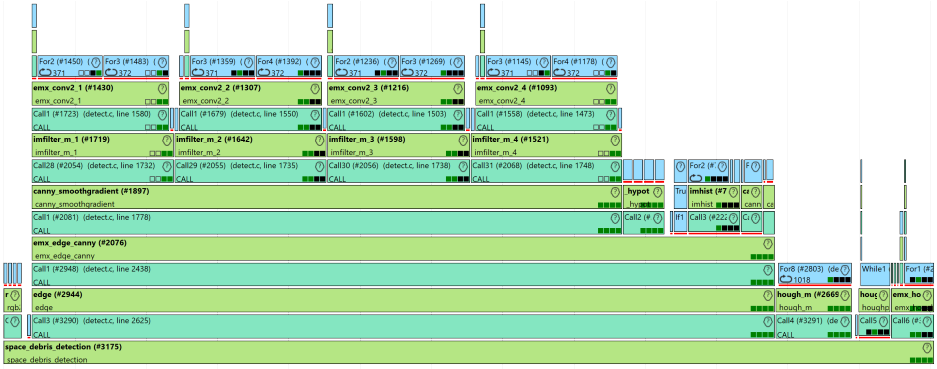


Abbildung 6.17.: Hierarchische Darstellung der Streifendetektion optimiert für 4 Kerne

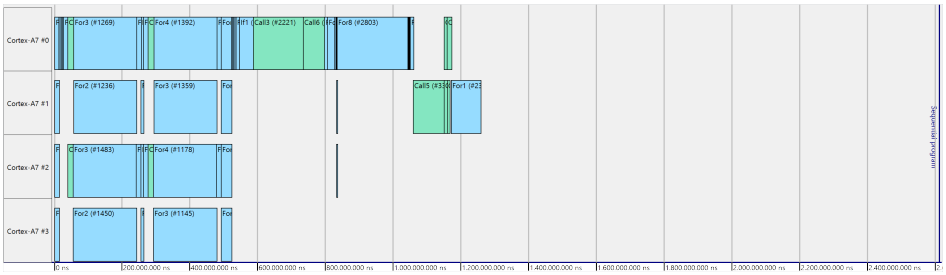


Abbildung 6.18.: Schedule der Streifendetektion für vier Kerne

für den Datentransfer in einer ähnlichen Größenordnung liegt wie die Synchronisation zwischen den einzelnen Kernen.

7. Zusammenfassung und Ausblick

Die effiziente Programmierung moderner eingebetteter Systeme wird immer komplexer, da die gestiegene Anzahl an Ausführungseinheiten in Form von Prozessorkernen oder Beschleunigern für bestimmte Operationen bereits bei der Softwareentwicklung berücksichtigt werden muss. Der vorgestellte Ansatz einer multigranularen Optimierung auf Algorithmus-, Code-, Task- und Daten-Ebene ermöglicht eine Entkopplung der eigentlichen Funktionsentwicklung von der später eingesetzten Zielhardware.

Durch den Einsatz der mathematischen Modellierung von Algorithmen in MATLAB® wird sichergestellt, dass bei der Entwicklung der reinen Funktionalität noch kein Bezug auf das spätere Zielsystem möglich ist. Außerdem können häufig verwendete Algorithmen leichter identifiziert und bereits optimierte Implementierungen in einer Bibliothek bereitgestellt werden. Der notwendige Schritt der Code-Konvertierung von MATLAB® in den auf dem eingebetteten System verwendeten C-Code ermöglicht erste Optimierungen auf Code-Ebene, indem die Repräsentation bereits für eine spätere statische Analyse sowie Parallelisierung vorbereitet wird. Weitere Optimierungen auf Code-Ebene, wie Code-Transformationen oder das 'Clustern' von Code-Abschnitten, sorgen dafür, dass die daraus generierten Tasks auf Task-Ebene bestmöglich auf die verfügbaren Ausführungseinheiten der Hardware verteilt werden können. Erst hier sind Kenntnisse über die Systemkonfiguration, wie z.B. die Anzahl der Prozessorkerne oder die Verteilung des Speichers, sowie eine Abschätzung der Performance notwendig, um die Laufzeiten der Tasks zu bestimmen. Die Optimierungen auf Daten-Ebene befassen sich schließlich mit der optimalen Platzierung von Synchronisations- und Kommunikationsinstruktionen und der effizienten Realisierung dieser Interprozesskommunikation sowie der Sicherstellung der korrekten parallelen Ausführung auf der Hardware.

Der Schwerpunkt der Arbeit lag auf der Unterstützung von Mehrkernsystemen, bei denen alle Kerne homogen programmiert werden können. Anhand der exemplarischen Unterstützung eines Grafikbeschleunigers konnte zudem gezeigt werden, dass der Ansatz auch für heterogene Systeme geeignet ist, bei denen für die Beschleuniger auch Varianten von C-Code wie CUDA oder OpenCL verwendet werden können.

Neben dem Einsatz dieses iterativen Ansatzes zur teilautomatisierten Parallelisierung von Anwendungen können die vier definierten Ebenen auch zur Einordnung anderer Parallelisierungsansätze verwendet werden. So zeigt sich, dass viele existierende Forschungsarbeiten für bestimmte Anwendungsfälle optimiert sind und mit einer Optimierung auf zwei oder drei der vorgestellten Ebenen auskommen.

Die vorgestellten Optimierungen auf den einzelnen Ebenen decken einen möglichen Entwicklungsablauf ab, mit dem vom MATLAB®-Code bis zum parallelen C-Code für die Hardware iterativ besser angepasster Code generiert werden kann. Es gibt aber noch viele Möglichkeiten, den Ansatz zu erweitern, hier eine Auswahl:

- Aufbau von Funktionsbibliotheken für verschiedene Anwendungsfälle: Die Optimierungen auf Algorithmus-Ebene können durch weitere Implementierungen für verschiedene Umgebungen wie Filter, Regler, Computer Vision oder andere Datenverarbeitungen erweitert werden. Durch die Beschränkung auf einen bestimmten Anwendungsfall können bereits Vorbereitungen für die typischerweise eingesetzte Hardware getroffen werden, z.B. durch Reduzierung des Speicherbedarfs oder durch Bereitstellen von Realisierungen, die bereits an vorhandene Beschleuniger angepasst sind.
- Automatische Auswahl von Code-Transformationen: Code-Transformationen können sehr gut automatisch auf den Quellcode angewendet werden, um diesen besser für die Parallelisierung auf Task-Ebene vorzubereiten. Mit steigender Anzahl der Transformationen steigt die Anzahl der Möglichkeiten exponentiell an, da in der Regel auch die Reihenfolge der Transformationen eine Rolle spielt. Das sich daraus ergebende Optimierungsproblem könnte durch maschinelles Lernen gelöst werden, um automatisch die beste Kombination von Transformationen für den gegebenen Anwendungsfall zu ermitteln.
- Unterstützung von geteiltem Speicher: die in dieser Arbeit vorgestellten Optimierungen auf Task- und Daten-Ebene wurden nur für die Anwendung auf Systemen mit verteiltem Speicher vorgestellt. Dieses Modell kann zwar auf allen Systemen angewendet werden und bietet durch die Datentrennung auch eine gute Lösung gegen Race Conditions und Deadlocks, kostet auf Systemen mit geteiltem Speicher aber Performanz durch unnötige Kopieroperationen. Die Unterstützung von geteiltem Speicher hat durch Reduktion von Overhead Auswirkungen auf die Performanzvorhersage der Task-Ebene, erfordert aber auch Unterstützung bei der Implementierung der Synchronisation auf Daten-Ebene, um die korrekte Ausführung zu gewährleisten.
- Spekulative Ausführung: Mit Hilfe der spekulativen Ausführung können Codeteile ausgeführt werden, bei denen noch nicht sicher ist, ob sie überhaupt ausgeführt werden müssen. So können z.B. bei einer Verzweigung beide Pfade parallel berechnet werden und erst wenn das Ergebnis der Bedingung bekannt ist, wird das Ergebnis des richtigen Pfades genommen. Durch diese überlappende Ausführung können vor allem Anwendungen optimiert werden, die sehr kontrollflusslastig sind und keine Codeteile haben, die direkt parallel ausgeführt werden können.

A. Anhang

A.1. Optimierung von lokaler Speichernutzung

Wie in Abschnitt 2.1.2.1 dargestellt, können eingebettete Systeme verschiedene Speicherhierarchien besitzen. NUMA-Architekturen haben aufgrund ihrer Struktur das größte Potential bei der Optimierung der Ausführung, da eine effiziente Aufteilung der Daten direkten Einfluss auf die Ausführungszeit haben kann. Alle folgenden Betrachtungen gehen von Systemen aus, die schnelle lokale Speicher mit langsameren, aber größeren geteilten Speichern kombinieren. Zur effizienten Nutzung dieser Speicherbereiche soll eine explizite Ansteuerung im Quellcode erfolgen. Variablen werden über entsprechende Adressen angesprochen und Daten werden bei Bedarf kopiert.

Die Speicherverwaltung kann auf unterschiedliche Arten geschehen: dynamisch oder statisch. Dynamisch sorgt für eine bessere Auslastung über die Zeit, erfordert aber komplexe Lebenszeitanalysen und eine Partitionierung des Speichers. Dies ist sinnvoll, wenn Variablen in einem schnelleren lokalen Speicher platziert werden, um die Performanz zu erhöhen. Variablen werden so lange im lokalen Speicher gehalten wie sie in Verwendung sind. Anschließend kann der Speicher für andere Variablen wiederverwendet werden und die verdrängten Daten werden in den Hauptspeicher geschrieben. Da Variablen unterschiedliche Größen haben können, werden Konzepte benötigt, um die Fragmentierung zu verhindern oder umgehen zu können. Die Wiederverwendung von Speicher, der zwischen anderen Variablen liegt, benötigt einen größeren Verwaltungsaufwand.

Die Optimierung von Daten im lokalen Speicher sind ein großes Themengebiet, auf dem schon seit Jahren aktiv geforscht wird. In dieser Arbeit wurde hier kein neuer Ansatz entwickelt, sondern es soll aufgezeigt werden, wie diese Optimierung bereits auf der Algorithmus-Ebene vorbereitet werden kann und welche Auswirkungen dies auf spätere Ebenen hat. Dazu wird beispielhaft ein im Rahmen einer Studienarbeit entwickelter Scratchpad-Allokations-Algorithmus vorgestellt.

A.1.1. Problembeschreibung

Ziel ist es, den lokalen Speicher von Kernen, auch Scratchpad Memory genannt, so mit Daten zu füllen, dass sich die Ausführungszeit des Programms verbessert. Möglich ist dies, da lokale Speicher in der Regel kürzere Zugriffszeiten als globale Speicher haben und somit Wartezeiten reduziert werden können, wenn sich wichtige Daten im lokalen Speicher befinden. Da hier nur der Einfluss innerhalb der ebenenübergreifenden Optimierung betrachtet werden soll, wurde die Komplexität des Ansatzes verringert, indem die Speicher nur statisch bei Programmstart mit Daten gefüllt werden und nicht mehr dynamisch zur Laufzeit mit neueren Daten gefüllt werden können. Dieses Verfahren ist auch

als *Non-Overlay Scratchpad Memory Allocation* bekannt, da sich die Speicheradressen aufgrund der statischen Nutzung nicht überlagern dürfen.

Das Problem lässt sich in zwei Teilschritte aufteilen: zunächst die Identifizierung der passendsten Variablen und anschließend eine effiziente Allokation des Speichers.

A.1.2. Variablenselektion mittels Energiemodell

Zur Auswahl der passendsten Variablen wurde auf das Energiemodell von Verma and Marwedel [118] aufgesetzt. Memory Objects mo_i werden nach ihrer Wertigkeit (Valenz) V sortiert, die sich aus dem Energiebedarf $E(mo_i, MM)$ sowie der Größe des Objekts im Speicher zusammensetzt wie in Formel A.1 dargestellt ist.

$$V(mo_i) = \frac{E(mo_i, MM)}{size(mo_i)} \tag{A.1}$$

Der Energiebedarf von Variablen im Speicher setzt sich wie in Formel A.2 dargestellt ist, aus der Anzahl n_r an Lesezugriffen multipliziert mit dem Energiebedarf beim Lesen aus dem Speicher $E_r(Mem)$ und den entsprechenden Werten n_w und $E_w(Mem)$ für die Schreibzugriffe zusammen.

$$E_{Var}(mo, Mem) = n_r(mo) \cdot E_r(Mem) + n_w(mo) \cdot E_w(Mem) \tag{A.2}$$

Ziel ist eine effizientere Ausführung von Algorithmen, so dass der Energiebedarf von Variablen insgesamt minimiert werden soll. Dazu wurde das Modell erweitert, indem auch noch die Operationen, in denen die Variablen verwendet werden als Gewichtung mit einbezogen wurde. Dadurch bekommen Variablen, die häufiger verwendet werden, eine höhere Wertigkeit. Dies ist in Formel A.3 dargestellt. Zur Energie $E_{LoadStore}$ für die Speicherzugriffe wird noch die Summe des Produkts aus Anzahl Aufrufen n_i und dem Energiebedarf E_{OP} von allen Operationen, in denen die besagten Variablen verwendet werden, hinzuaddiert.

$$E_{Var}(mo) = E_{LoadStore} + \sum_{i=0}^{N} n_i \cdot E_{OP} \tag{A.3}$$

Die Anzahl an Aufrufen kann durch Analyse des Kontrollflusses ermittelt werden. Dabei werden etwaige Optimierungen durch einen späteren Compiler nicht mit betrachtet. Da die Berechnung hier nur zur reinen Gewichtung der Variablen verwendet wird, kann der Einfluss jedoch vernachlässigt werden. Es kann davon ausgegangen werden, dass Variablen, die im Programm häufig lesend und schreibend verwendet werden, nicht herausoptimiert werden können und damit immer einen merkbaren Einfluss auf die Performanz der Anwendung haben.

Zur Ermittlung des Energiebedarfs einer Operation wurde auf die Zahlenbasis einer statischen Performanzanalyse wie in Abschnitt 2.4.1.1 beschrieben, zurückgegriffen. Für unterstützte Prozessoren wurde die Anzahl an Taktzyklen je Operation gemessen und in einer JSON-Datei gespeichert. Diese Zahl kann als Basis für den Energiebedarf verwendet werden. Werden sie direkt verwendet, so können sie zur einfachen Gewichtung verwendet werden. Um jedoch im Energiemodell zu bleiben, sollte die Zahl noch mit einem Faktor für die entsprechende Ausführungseinheit multipliziert werden. So kann beispielsweise davon ausgegangen werden, dass auch wenn sowohl eine Addition als auch eine Multiplikation jeweils nur einen Taktzyklus benötigt, die Kosten für die Multiplikation höher sind, da mehr Transistoren an dieser Berechnung beteiligt sind. Ein paar Faktoren haben sich dabei bewährt: ALUs können mit 2 multipliziert werden, MULs mit Faktor 3 und eine Division mit Faktor 16. Die tatsächlichen Zahlen unterscheiden sich bei allen Prozessoren abhängig von der Implementierung und der Fertigung und kann ohne detailliertes Wissen über die Hardware nicht genauer ermittelt werden.

Diese Berechnung wird für alle Variablen im Programm durchgeführt und anschließend werden alle Variablen anhand ihrer Wertigkeit in einer Liste gespeichert.

A.1.3. Allokationsalgorithmus

Auf die sortierte Liste der Wertigkeit aller Variablen wird der in Quellcode A.1 dargestellte Allokationsalgorithmus angewendet.

```
1 while RemSPMSize > 0 do
2   mo[i] := head(SortedMO)
3   RemSPMSize := RemSPMSize - size(mo[i])
4   MOSPM := MOSPM.add(mo[i])
5   AddressVector[i] := PrevAddress
6   PrevAddress += size(mo)
7 end while
```

Quellcode A.1: Allokationsalgorithmus für Scratchpad-Speicher

Der Algorithmus wird solange ausgeführt, bis der Scratchpadspeicher gefüllt ist. Zunächst wird das erste Element `mo[i]` aus der sortierten Liste `SortedMO` herausgenommen. Die Größe des verfügbaren Scratchpadspeichers `RemSPMSize` wird um die Größe des Speicherobjekts `size(mo[i])` reduziert. Anschließend wird `mo[i]` zur Liste `MOSPM` hinzugefügt, welche die bereits zugewiesenen Speicherobjekte im Scratchpadspeicher darstellt. In der Variablen `AddressVector` wird die Zieladresse von `mo[i]` auf den Anfang des leeren Bereichs im Scratchpadspeicher gesetzt, welcher in der Variablen `PrevAddress` gespeichert ist. Diese wird zunächst mit der Basisadresse des Scratchpads initialisiert und anschließend in jeder Iteration um die Größe des betrachteten Speicherobjekts erhöht. Somit wird sichergestellt, dass das nächste Speicherobjekt nicht überlappend auf das aktuelle Speicherobjekt geschrieben wird. Dieser Vorgang wird mit allen Speicherobjekten der `ListeSortedMO` wiederholt, bis der Scratchpadspeicher gefüllt ist oder die sortierte Liste keine Elemente mehr enthält.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, 3(20):33–35, 2006.

[2] Karl Rupp. 50 Years of Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data/tree/master/50yrs, February 2022. Accessed: 29-05-2022.

[3] J. Becker, T. Stripf, O. Oey, M. Huebner, S. Derrien, D. Menard, O. Sentieys, G. Rauwerda, K. Sunesen, N. Kavvadias, K. Masselos, G. Goulas, P. Alefragis, N.S. Voros, D. Kritharidis, N. Mitas, and D. Goehringer. From scilab to high performance embedded multicore systems: The alma approach. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 114–121, Sept 2012.

[4] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[5] T. Sakurai and A.R. Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25(2):584–594, 1990.

[6] Nvidia. The benefits of multiple cpu cores in mobile devices, 2010.

[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[8] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[9] Rogers Jr, Hartley. *Theory of Recursive Functions and Effective Computability*. New York, McGraw-Hill, 1967.

[10] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. In *Communications of the ACM*, page pages 366–371, 05 1966.

[11] Haskell Performance Resource. https://wiki.haskell.org/Performance, August 2018.

[12] Haskell language. https://www.haskell.org/.

[13] The scheme programming language. http://www.scheme.com/.

[14] Javascript. http://www.ecma-international.org/ecma-262/7.0/index.html.

[15] Python. https://www.python.org/.

[16] The perl programming language. https://www.perl.org/.

[17] MathWorks. MATLAB. http://de.mathworks.com/products/matlab/.

[18] Scilab Enterprises. Scilab. http://www.scilab.org/.

[19] James Martin. *Application development without programmers*. Prentice Hall PTR, 1982.

[20] NVIDIA. CUDA. https://developer.nvidia.com/cuda-zone.

[21] Khronos Group. OpenCL. https://www.khronos.org/opencl/.

[22] IEEE standard VHDL language reference manual. http://ieeexplore.ieee.org/document/4772740/.

[23] IEEE standard for systemverilog. https://standards.ieee.org/findstds/standard/1800-2012.html.

[24] Milind Girkar and Constantine D Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[25] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[26] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In Zhiyuan Li, Pen-Chung Yew, Siddharta Chatterjee, Chua-Huang Huang, P. Sadayappan, and David Sehr, editors, *Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 114–130. Springer Berlin Heidelberg, 1998.

[27] CAIRN group. GeCoS – Generic Compiler Suite. http://gecos.gforge.inria.fr/doku/doku.php.

[28] Eclipse Foundation. Eclipse. http://www.eclipse.org/, 11 2001.

[29] W3C. Extensible Markup Language, XML. http://www.w3.org/TR/2006/REC-xml11-20060816/, 2006.

[30] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[31] Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, 2006.

[32] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[33] Jeronimo Castrillon. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. PhD thesis, RTWH Aachen, 2013.

[34] The LLVM Compiler Infrastructure. http://llvm.org/.

[35] ARM CoreSight Architecture. https://developer.arm.com/Architectures/CoreSight%20Architecture.

[36] Ankunda R Kiremire. The application of the pareto principle in software engineering. *Consulted January*, 13, 2011.

[37] IEEE. Portable Operating System Interface (POSIX). http://get.posixcertified.ieee.org/, 2017.

[38] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? *arXiv preprint arXiv:1701.00854*, 2017.

[39] Todd R. Allen and David A. Padua. Debugging Fortran on a shared memory machine. Technical report, Illinois Univ., Urbana (USA). Center for Supercomputing Research and Development, 1987.

[40] David A. Huffman. The synthesis of sequential switching circuits. 1954.

[41] Perry A. Emrath, S. Chosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing'89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 580–588. IEEE, 1989.

[42] Oliver Oey. Race Conditions: Software-Parallelisierung für Multicore-Prozessoren I. *ELEKTRONIKPRAXIS*, (4), 2020.

[43] Oliver Oey. Deadlocks: Software-Parallelisierung für Multicore-Prozessoren II. *ELEKTRONIKPRAXIS*, (6), 2020.

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

[45] Steven Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, San Francisco, Calif, 1997.

[46] ANL Mathematics and Computer Science. The message passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi/index.htm, 1992.

[47] emmtrix Technologies GmbH. emmtrix Parallel Studio. www.emmtrix.com/tools/emmtrix-parallel-studio.

[48] emmtrix Technologies GmbH. emmtrix Code Generator. www.emmtrix.com/tools/emmtrix-code-generator.

[49] Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[50] Hamid Arabnejad, João Bispo, Jorge G. Barbosa, and João M.P. Cardoso. AutoParClava. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms - PARMA-DITAM '18*. ACM Press, 2018.

[51] Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. *The Journal of Supercomputing*, 76(9):6753–6785, dec 2019.

[52] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 2013.

[53] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[54] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4all: From convex array regions to heterogeneous computing. In *2nd International Workshop on Polyhedral Compilation Techniques, Impact (Jan 2012)*. Citeseer, 2012.

[55] Todor Stefanov, Hristo Nikolov, Lubomir Bogdanov, and Angel Popov. DAEDALUS framework for high-level synthesis: Past, present and future. In *2021 25th International Conference Electronics*. IEEE, jun 2021.

[56] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, may 2020.

[57] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 228–241, 2015.

[58] Saiyedul Islam, Sundar Balasubramaniam, Shruti Gupta, Shikhar Brajesh, Rohan Badlani, Nitin Labhishetty, Abhinav Baid, Poonam Goyal, and Navneet Goyal. Pattern-Based Automatic Parallelization of Representative-Based Clustering Algorithms. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, oct 2018.

[59] Marco Danelutto, Gabriele Mencagli, Massimo Torquati, Horacio González–Vélez, and Peter Kilpatrick. Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming. *International Journal of Parallel Programming*, 49(2):177–198, nov 2020.

[60] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing - ARMS-CC '17*. ACM Press, 2017.

[61] Sabri Pllana and Fatos Xhafa, editors. *Programming multi-core and many-core computing systems*. John Wiley & Sons, Inc., jan 2017.

[62] Luís Miguel Pinho, Eduardo Quinones, and Andrea Marongiu. *High-performance and time-predictable embedded computing*. River Publishers, 2018.

[63] Mathworks. Embedded Coder. https://de.mathworks.com/products/embedded-coder.html.

[64] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.

[65] J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J.M.P. Cardoso, and P.C. Diniz. The MATISSE MATLAB compiler. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 602–608, 2013.

[66] Joao.P. Cardoso, Tiago Carvalho, José.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. LARA: An aspect-oriented programming language for embedded systems. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 179–190. ACM, 2012.

[67] Joao Bispo, Luis Reis, and Joao M. P. Cardoso. Multi-target c code generation from MATLAB. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 95:95–95:100. ACM, 2014.

[68] Esin Yavuz, James Turner, and Thomas Nowotny. GeNN: a code generation framework for accelerated brain simulations. *Scientific reports*, 6(1):1–14, 2016.

[69] Harsh A. Shukla, Bulat Khusainov, Eric C. Kerrigan, and Colin N. Jones. Software and hardware code generation for predictive control using splitting methods. *IFAC-PapersOnLine*, 50(1):14386–14391, 2017.

[70] Luiz Benicio Degli Esposte Rosa and Alexandre Leite. Automatic C Code Generation for Implementation of IIR Embedded Systems in Fixed-Point Arithmetic. In *Anais da Conferência de Estudos em Engenharia Elétrica*, volume 15. Galoa, dec 2017.

[71] Goran Banjac, Bartolomeo Stellato, Nicholas Moehle, Paul Goulart, Alberto Bemporad, and Stephen Boyd. Embedded code generation using the OSQP solver. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 1906–1911. IEEE, 2017.

[72] Pantelis Sopasakis, Emil Fresk, and Panagiotis Patrinos. OpEn: Code generation for embedded nonconvex optimization. *IFAC-PapersOnLine*, 53(2):6548–6554, 2020.

[73] Nikolay P. Brayanov and Anna V. Stoynova. Evaluation of Model-Based Code Generation for Embedded Systems–Mature Approach for Development in Evolution. *International Journal of Computer and Information Engineering*, 13(8):459–464, 2019.

[74] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U. Nagaraj Shenoy, and Alok Choudhary. Handling Context-sensitive Syntactic Issues in the Design of a Front-end for a MATLAB Compiler. In *Proceedings of the International Conference on APL-Berlin-2000 Conference*, APL '00, pages 27–40. ACM, 2000.

[75] Pramod G. Joisha and Prithviraj Banerjee. An algebraic array shape inference system for MATLAB®. *ACM Trans. Program. Lang. Syst.*, 28(5):848–907, 2006.

[76] IBM. X10. http://x10-lang.org/, 2004.

[77] Google. Go language. https://golang.org/, 2009.

[78] Ericsson Computer Science Laboratory. Erlang programming language. http://www.erlang.org/.

[79] Jeffrey Werner Bezanson. Julia language. http://julialang.org/, 2012.

[80] RedMonk. The redmonk programming language rankings: June 2015. https://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/, June 2015.

[81] Tomofumi Yuki. *Beyond Shared Memory Loop Parallelism in the Polyhedral Model*. PhD thesis, Colorado State University, 2012.

[82] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 267–276, 2010.

[83] Boost C++ libraries. http://www.boost.org/.

[84] Flex. https://github.com/westes/flex.

[85] GNU Bison. https://www.gnu.org/software/bison/.

[86] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[87] GNU Make. https://www.gnu.org/software/make/.

[88] GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[89] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[90] Wald Lucien. Some terms of reference in data fusion. *IEEE Transactions on Geosciences and Remote Sensing*, 37(3):1190–1193, 1999.

[91] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical report, USA, 2004.

[92] Jianzhou Chen. Untersuchung der Effizienz von Codegenerierung für eingebettete Systeme am Beispiel eines Datenfusionsalgorithmus. Master's thesis, KIT, 2015.

[93] The Multicore Association®. Software-Hardware Interface for Multi-Many-Core (SHIM™). https://www.multicore-association.org/workgroup/shim.php.

[94] IEEE 2804-2019 - IEEE Standard for Software-Hardware Interface for Multi-Many-Core. https://standards.ieee.org/standard/2804-2019.html, January 2020.

[95] Martin Davis. *Computability and Unsolvability*. Dover Publications Inc., 1985.

[96] FFTW. http://www.fftw.org/.

[97] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, 1998.

[98] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, mar 2002.

[99] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, jun 1953.

[100] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 85–97. ACM, 1997.

[101] Vikram Adve and John Mellor-Crummey. Advanced Code Generation for High Performance Fortran. In Santosh Pande and Dharma P. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems*, number 1808 in Lecture Notes in Computer Science, pages 553–596. Springer Berlin Heidelberg, 2001.

[102] D.J. Palermo, E. Su, J.A. Chandy, and P. Banerjee. Communication optimizations used in the paradigm compiler for distributed-memory multicomputers. In *International Conference on Parallel Processing, 1994. ICPP 1994 Volume 2*, volume 2, pages 1–10, 1994.

[103] Sung-Eun Choi and L. Snyder. Quantifying the effects of communication optimizations. In *, Proceedings of the 1997 International Conference on Parallel Processing, 1997*, pages 218–222, 1997.

[104] Jingke Li and Marina Chen. Generating explicit communication from shared-memory program references. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 865–876. IEEE Computer Society Press, 1990.

[105] Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. A loop transformation algorithm for communication overlapping. *International Journal of Parallel Programming*, 28(2):135–154, 2000.

[106] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Trans. Program. Lang. Syst.*, 21(6):1251–1297, 1999.

[107] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 126–138. ACM, 1993.

[108] R. Das, M. Uysal, J. Saltz, and Y. S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, 1994.

[109] Ti-Yen Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *, 1995 IEEE/ACM International Conference on Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers*, pages 288–294, 1995.

[110] R.B. Ortega and G. Borriello. Communication synthesis for distributed embedded systems. In *1998 IEEE/ACM International Conference on Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers*, pages 437–444, 1998.

[111] L. Freund, D. Dupont, M. Israel, and F. Rousseau. Interface optimization during hardware-software partitioning. In *(CODES/CASHE '97), Proceedings of the Fifth International Workshop on Hardware/Software Codesign, 1997*, pages 75–79, 1997.

[112] F. Ferrandi, P.L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):911–924, 2010.

[113] A. Siebenborn, A. Viehl, O. Bringmann, and W. Rosenstiel. Control-flow aware communication and conflict analysis of parallel processes. In *Design Automation Con-*

ference, 2007. ASP-DAC '07. Asia and South Pacific, pages 32–37, 2007.

[114] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*. ACM Press, 1987.

[115] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[116] Paul V. C. Hough. Method and means for recognizing complex patterns, 1962.

[117] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–297, may 1965.

[118] Manish Verma and Peter Marwedel. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. Springer-Verlag GmbH, 2007.

Betreute studentische Arbeiten

[Abich, 2016] Abich, K. (2016). Evaluation von parallelen Programmierschnittstellen auf dem Raspberry Pi2 hinsichtlich automatischer Codegenerierung. Bachelorarbeit, Hochschule Karlsruhe – Technik und Wirtschaft.

[Bull, 2012] Bull, C. (2012). Erweiterung der Kahrisma Architektur um Network-on-Chip Kommunikation mit architektur-spezifischen Optimierungen. Diplomarbeit, Karlsruher Institut für Technologie.

[Chen, 2015] Chen, J. (2015). Untersuchung der Effizienz von Codegenerierung für eingebettete Systemam Beispiel eines Datenfusionsalgorithmus. Masterarbeit, Karlsruher Institut für Technologie.

[Imrani, 2013] Imrani, M. A. (2013). Konzeption und Implementierung eines SystemC Simulators für ein FPGA-basiertes Multiprozessor System. Diplomarbeit, Karlsruher Institut für Technologie.

[Kilic, 2014] Kilic, Y. (2014). Speicherverwaltung für parallele Prozessorsysteme am Beispiel der KAHRISMA Architektur. Studienarbeit, Karlsruher Institut für Technologie.

[Meder, 2011] Meder, L. (2011). Konzeption und Implementierung von Zuverlässigkeitsmechanismen für das Star-Wheels Netzwerk-on-Chip. Diplomarbeit, Universität Karlsruhe (TH).

[Meyer, 2014] Meyer, J. (2014). Kommunikationsoptimierung paralleler Programme für Systeme mit verteiltem Speicher. Masterarbeit, Karlsruher Institut für Technologie.

Eigene Veröffentlichungen

Konferenzbeiträge

[Becker et al., 2012] Becker, J., Stripf, T., Oey, O., Huebner, M., Derrien, S., Menard, D., Sentieys, O., Rauwerda, G., Sunesen, K., Kavvadias, N., Masselos, K., Goulas, G., Alefragis, P., Voros, N. S., Kritharidis, D., Mitas, N., and Goehringer, D. (2012). From Scilab to High Performance Embedded Multicore Systems: The ALMA Approach. In *2012 15th Euromicro Conference on Digital System Design*, pages 114–121.

[Bruckschloegl et al., 2014] Bruckschloegl, T., Oey, O., Rückauer, M., Stripf, T., and Becker, J. (2014). A Hierarchical Architecture Description for Flexible Multicore System Simulation. In *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*. Institute of Electrical and Electronics Engineers (IEEE).

[Goulas et al., 2012] Goulas, G., Alefragis, P., Voros, N. S., Valouxis, C., Gogos, C., Kavvadias, N., Dimitroulakos, G., Masselos, K., Goehringer, D., Derrien, S., Ménard, D., Sentieys, O., Huebner, M., Stripf, T., Oey, O., Becker, J., Rauwerda, G., Sunesen, K., Kritharidis, D., and Mitas, N. (2012). From scilab to multicore embedded systems: Algorithms and methodologies. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 268–275.

[Goulas et al., 2013] Goulas, G., Valouxis, C., Alefragis, P., Voros, N. S., Gogos, C., Oey, O., Stripf, T., Bruckschloegl, T., Becker, J., Moussawi, A. E., Naullet, M., and Yuki, T. (2013). Coarse-grain optimization and code generation for embedded multicore systems. In *2013 Euromicro Conference on Digital System Design*, pages 379–386.

[Göhringer et al., 2013] Göhringer, D., Meder, L., Oey, O., and Becker, J. (2013). Reliable and adaptive network-on-chip architectures for cyber physical systems. *ACM Transactions on Embedded Computing Systems*, 12(1s):1–21.

[Göhringer et al., 2011] Göhringer, D., Oey, O., Hübner, M., and Becker, J. (2011). Heterogeneous and runtime parameterizable star-wheels network-on-chip. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 380–387.

[Oey et al., 2024] Oey, O., Huebner, M., Stripf, T., and Becker, J. (2024). Embedded Multi-Core Code Generation with Cross-Layer Parallelization. In Bispo, J. a., Xydis, S., Curzel, S., and Sousa, L. M., editors, *15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024)*, volume 116 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:13, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Oey et al., 2018] Oey, O., Rueckauer, M., Stripf, T., Becker, J., David, C., Debray, Y., Mueller, D., Durak, U., Kasnakli, E., Bednara, M., and Schoeberl, M. (2018). Interactive Parallelization of Embedded Real-Time Applications Starting from Open-Source Scilab & Xcos. In *Embedded Real Time Software and Systems Conference 2018*.

[Oey et al., 2017] Oey, O., Rückauer, M., Stripf, T., and Becker, J. (2017). Increasing Energy Efficiency Through Semi-Automatic Parallelization of Applications for Embedded Computing Devices in the IoT Domain. In *Embedded World Conference 2017*.

[Oey and Stripf, 2015] Oey, O. and Stripf, T. (2015). Effiziente Embedded-Multicore-Programmierung. In *Embedded Software Engineering (ESE) Kongress Tagungsband*.

[Oey et al., 2021] Oey, O., Stripf, T., and Becker, J. (2021). From MATLAB to C Code for Heterogeneous Embedded Systems - Using Abstraction Levels for Specialized Optimizations. In *embedded World 2021 Conference Proceedings*, pages 412–417.

[Oey et al., 2012] Oey, O., Werner, S., Göhringer, D., Stuckert, A., Becker, J., and Hübner, M. (2012). Virtualization of heterogeneous and adaptive multi-core/multi-board systems. In *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*, pages 1–2.

[Rueckauer et al., 2013] Rueckauer, M., Munoz, D. M., Stripf, T., Oey, O., Llanos, C. H., and Becker, J. (2013). A flexible implementation of the pso algorithm for fine- and coarse-grained reconfigurable embedded systems. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6.

[Stripf and Oey, 2016] Stripf, T. and Oey, O. (2016). Effiziente Embedded-Multicore-Programmierung - Automatische Parallelisierung von Scilab/MATLAB-Anwendungen. In *MPC Workshopband 55. Tagung*.

[Stripf and Oey, 2017] Stripf, T. and Oey, O. (2017). Interaktive Parallelisierung von Anwendungen für eingebettete Mehrkernprozessoren. Parallel 2017.

[Stripf et al., 2012a] Stripf, T., Oey, O., Bruckschloegl, T., Koenig, R., Goulas, G., Alefragis, P., Voros, N. S., Potman, J., Sunesen, K., Derrien, S., Sentieys, O., and Becker, J. (2012a). A compilation- and simulation-oriented architecture description language for multicore systems. In *2012 IEEE 15th International Conference on Computational Science and Engineering*, pages 383–390.

[Stripf et al., 2012b] Stripf, T., Oey, O., Bruckschloegl, T., Koenig, R., Huebner, M., Becker, J., Rauwerda, G., Sunesen, K., Kavvadias, N., Dimitroulakos, G., Masselos, K., Kritharidis, D., Mitas, N., Goulas, G., Alefragis, P., Voros, N. S., Derrien, S., Menard, D., Sentieys, O., Goehringer, D., and Perschke, T. (2012b). A flexible approach for compiling scilab to reconfigurable multi-core embedded systems. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8.

[Stripf et al., 2016] Stripf, T., Rückauer, M., and Oey, O. (2016). Plattformübergreifende Software für Multicore & FPGAs. In *Tagungsband Embedded Software Engineering Kongress 2016*.

[Werner et al., 2012] Werner, S., Oey, O., Göhringer, D., Hübner, M., and Becker, J. (2012). Virtualized on-chip distributed computing for heterogeneous reconfigurable multi-core systems. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*,

pages 280–283.

Journals

[Becker et al., 2014] Becker, J., Bruckschloegl, T., Oey, O., Stripf, T., Goulas, G., Raptis, N., Valouxis, C., Alefragis, P., Voros, N. S., and Gogos, C. (2014). Profile-guided compilation of scilab algorithms for multiprocessor systems. In *Lecture Notes in Computer Science*, pages 330–336. Springer Nature.

[Göhringer et al., 2012] Göhringer, D., Meder, L., Werner, S., Oey, O., Becker, J., and Hübner, M. (2012). Adaptive Multiclient Network-on-Chip Memory Core: Hardware Architecture, Software Abstraction Layer, and Application Exploration. *International Journal of Reconfigurable Computing*, 2012:1–14.

[Oey and Stripf, 2017] Oey, O. and Stripf, T. (2017). Grafisch parallel programmieren. *Elektronik*, 9.2017:48–51.

[Reder et al., 2019] Reder, S., Kempf, F., Bucher, H., Becker, J., Alefragis, P., Voros, N., Skalistis, S., Derrien, S., Puaut, I., Oey, O., Stripf, T., Ferdinand, C., David, C., Ulbig, P., Mueller, D., and Durak, U. (2019). Worst-Case Execution-Time-Aware Parallelization of Model-Based Avionics Applications. *Journal of Aerospace Information Systems*, 16(11):521–533.

[Stripf et al., 2013] Stripf, T., Oey, O., Bruckschloegl, T., Becker, J., Rauwerda, G., Sunesen, K., Goulas, G., Alefragis, P., Voros, N. S., Derrien, S., Sentieys, O., Kavvadias, N., Dimitroulakos, G., Masselos, K., Kritharidis, D., Mitas, N., and Perschke, T. (2013). Compiling Scilab to high performance embedded multicore systems. *Microprocessors and Microsystems*, 37(8):1033–1049.

Patente

[1] MEYER, J. ; OEY, O. ; STRIPF, T. ; BECKER, J. : *Computer System and Method for Multi-Processor Communication*. EP3121714A1 - 2015

[2] OEY, O. ; STRIPF, T. ; RÜCKAUER, M. ; BECKER, J. : *Computer System and Method for Parallel Program Code Optimization and Deployment*. EP3208712B1 - 2016

Oliver Wolf

Multigranulare Optimierung für heterogene Multicore-Systeme

Im Kontext eingebetteter Systeme lässt sich eine zunehmende Fokussierung auf Multicore-Prozessoren beobachten, welche mit Beschleunigern für spezialisierte Aufgaben kombiniert werden. Um die optimale Performance und effiziente Nutzung zu gewährleisten, ist bei der Software-Entwicklung eine Parallelisierung erforderlich, die jedoch eine Vielzahl von Herausforderungen für Programmierer mit sich bringt.

Im Rahmen dieser Arbeit wird eine Methode präsentiert, die eine Anwendung für eingebettete Multicore-Systeme durch multigranulare Parallelisierung optimieren kann. Ziel dabei war die Entwicklung einer Methode, welche die funktionale Entwicklung von der Optimierung für die Zielplattform entkoppelt, ohne die optimale Ausnutzung aller Ausführungseinheiten zu beeinträchtigen.

Das Verfahren umfasst zwei Phasen: In der ersten Phase erfolgt die Entwicklung des reinen Algorithmus ohne Optimierung für die Zielplattform. Im zweiten Schritt wird C-Code generiert, der anschließend auf vier definierten Ebenen – Algorithmus, Code, Task und Daten – optimiert und parallelisiert wird. Dadurch soll eine möglichst performante Ausführung auf der Zielhardware ermöglicht werden. Neben den Optimierungen auf den einzelnen Ebenen wird ein ebenenübergreifendes Vorgehen eingeführt und abschließend mit Beispielen evaluiert.