Automatisierte Hardware-Software Partitionierung am Beispiel eines eingebetteten, echtzeitfähigen Stereobildanalysesystems in Kraftfahrzeugen

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

von Dipl.-Ing. Jens Kaszubiak

geb. am 10.10.1977 in Blankenburg/Harz

genehmigt durch die Fakultät für Elektrotechnik und Informationstechnik

der Otto-von-Guericke-Universität Magdeburg

Gutachter:

Prof. Dr.-Ing. habil. Bernd Michaelis

Prof. Dr.-Ing. Christian Diedrich

Promotionskolloquium am 18.02.2008

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Elektronik, Signalverarbeitung und Kommunikationstechnik der Otto-von-Guericke-Universität Magdeburg.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. habil. Bernd Michaelis, der die Anregung für diese Arbeit gab und mir jederzeit mit stetem Interesse, wertvollen Hinweisen und freundlicher Unterstützung zur Seite stand.

Bei Prof. Dr.-Ing. Christian Diedrich bedanke ich mich recht herzlich für die Begutachtung dieser Arbeit.

Den Kollegen des Lehrstuhls für Technische Informatik danke ich für die hilfreichen Hinweise und das angenehme Arbeitsklima.

Mein besonderer Dank gilt meinem Zimmerkollegen Michael Tornow für die gute Zusammenarbeit in den gemeinsamen Projekten. Den Mitarbeitern der Institutswerkstatt danke ich für die schnelle Umsetzung der verschiedenen Versuchsaufbauten und die Hilfe bei der Bestückung der Entwurfshardware. Weiterhin danke ich Herrn Dr. Thomas Schindler und Herrn Helmut Bresch für die vielen wertvollen Tipps und Hinweise bei der praktischen Umsetzung der Arbeit.

Magdeburg, 30. Oktober 2007

Jens Kaszubiak

Zusammenfassung

In der vorliegenden Arbeit wird ein Verfahren zur automatischen Partitionierung von Bildverarbeitungsalgorithmen in ein Hardware-Software Co-Design vorgestellt. Die Algorithmen entsprechen dabei dem allgemeinen Schichtenmodell der Bildverarbeitung (BV). Ziel ist die echtzeitfähige Realisierung eines BV-Algorithmus auf einem eingebetteten System mit minimalen „Kosten“. Die als C/C++ Code realisierten Funktionen des BV-Algorithmus werden auf einem PC statisch und dynamisch analysiert. Die Ergebnisse der Analyse bilden die Grundlage für die Partitionierung. Die Partitionierung selbst gliedert sich in zwei Phasen. Die erste Phase stellt die Clusterung der Funktionen zu einem Multiprozessorsystem dar. Reicht das nicht aus, um Echtzeitfähigkeit herzustellen, werden einzelne Funktionen unter Nutzung des Simulated Annealing in Logik überführt, bis die zuvor aufgestellten Echtzeitbedingungen eingehalten werden. Dabei entsteht ein System mit auf Logik und mehrere Prozessoren verteilten Funktionen, das die aufgestellten Anforderungen erfüllt.

Abstract

This work presents a system for automatic partitioning of image processing algorithms in a Hardware/Software Co-Design. The algorithms thereby correspond to the general layered model of image processing (IP). The goal is the real-time implementation of an IP algorithm on an embedded system with minimal costs. The functions of an IP algorithm implemented in C/C++ code are analyzed on a PC statically and dynamically. The results of the analysis form the basis for the partitioning. The partitioning itself is divided into two phases. The first phase constitutes the clustering of functions to a multiprocessor system. If that is not sufficient, in order to manufacture real-time conditions individual functions are transferred to logic using simulated annealing. Therefore, a system develops with logic and several processors as distributed functions, that fullfills the requirements.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

$ASIC$	Application Specific Circuit
$CISC$	Complex Instruction Set Computer
$COCOMO$	Constructive Cost Model
DSP	Digital Signal Processor
$FPGA$	Field Programmable Gate Array
IP	Intellectual property
KFZ	Kraftfahrzeug
$KKFMF$	Kreuzkorrelationsfunktion
LOC	Lines of Code
MoC	Model of Computation
NP	Non Polynomial
$RISC$	Reduced Instruction Set Computer
SoC	System on Chip
$SoPC$	System on Programmable Chip
$VHDL$	Very High Speed Hardware Description Language

Symbolverzeichnis

A	Aufwand an Logikzellen
AH	Aufrufhäufigkeit
B	Basisweite
c	Kamerakonstante
C	Kosten
ΔC	Kostendifferenz
CC	Komlexität nach McCabe
CL	Co-Level

CP	Controllparameter
f	Frequenz
f_S	Sequentieller Anteil einer Funktion
F	Funktion
G	Graph
HD	Schwierigkeit nach Halstedt
HE	Aufwand nach Halstedt
HV	Volumen nach Halstedt
KP	Kritischer Pfad
L	Level
P	Wahrscheinlichkeit
Pa	Partition
P_L	Priorität einer Funktion aus Level
P_{CL}	Priorität einer Funktion aus Co-Level
PM	Personenmonate
Q	Ähnlichkeitsmaß - Kreuzkorrelationsfunktion
SF	Geschwindigkeitsfaktor
StF	Straffaktor
t	Zeit
T	Periodendauer
Δu	Disparität

Kapitel 1

Einleitung

Bildverarbeitende Systeme versuchen die menschliche Auge-Gehirn-Interaktion mit Hilfe technischer Lösungen nachzubilden, um so den Menschen von monotonen Überwachungsaufgaben zu entlasten oder ihn bei komplexen Handlungen zu unterstützen.

Mit der Entwicklung der modernen PC-Technik steht eine preiswerte Lösung für die Signalverarbeitung der durch Kameras erzeugten Bilder zur Verfügung. Mittlerweile setzen viele Unternehmen Bildverarbeitungssysteme, vornehmlich zur Qualitätskontrolle, ein. Aber auch Gebiete, wie beispielsweise die Medizintechnik, werden immer stärker mit Bildverarbeitungssystemen ausgestattet.

Da industrielle Bildverarbeitungssysteme meist stationär aufgebaut sind, spielt der Stromverbrauch der Systeme nur eine untergeordnete Rolle. Trotz der hohen Leistung der dort eingesetzten PC-Systeme, ist eine echtzeitfähige Verarbeitung bewegter Bilder häufig nur durch den zusätzlichen Einsatz von Spezialhardware möglich. Bei mobilen Anwendungen kommt zur echtzeitfähigen Bildverarbeitung auch noch der geforderte geringe Strom- und Platzverbrauch hinzu. Daher werden für mobile Anwendungen eingebettete Systeme eingesetzt, die für den konkreten Anwendungsfall entwickelt werden.

Mobile *Assistenzsysteme* dienen zum Beispiel den Fahrern von Kraftfahrzeugen als Unterstützung in gefährlichen Situationen oder zur Entlastung von monoto-

nen Überwachungsaufgaben [16], [17] und [113]. Diese Assistenzsysteme gelten in der Automobilentwicklung als wesentlicher Bestandteil der Produktaufwertung. Man unterscheidet zwischen Fahr-Assistenzsystemen, wie ESP (Elektronisches-Stabilitäts-Programm) oder ABS (Anti-Blockier-System) und Fahrerassistenzsystemen [95]. Bei Fahr-Assistenzsystemen wird hauptsächlich das Verhalten des eigenen Fahrzeugs überwacht und gegebenenfalls korrigiert. Fahrerassistenzsysteme erfassen zudem auch noch das Umfeld des eigenen Fahrzeugs. Hiermit können Gefahren, die nicht unmittelbar auf das eigene Fahrzeug einwirken erkannt werden, um rechtzeitig entsprechende Gegenmaßnahmen einzuleiten.

Fahrerassistenzsysteme werden nach [95] in folgenden Bereichen eingesetzt:

1. Verkehrssituation (Autobahn, Stau, Stadtverkehr, Landstraße,...)
2. Mentaler Fahrzustand (müde, genervt, entspannt,...)
3. Fahraufgabe (Freizeit, Beruf, bekanntes oder unbekanntes Territorium,...)

Neben den technischen Problemen bei der Realisierung gibt es weitere Hürden, die der Markteinführung solcher Assistenzsysteme im Weg stehen [52]. Insbesondere die Produkthaftung der Hersteller verhindert eine schnelle Markteinführung. Deshalb muss eine Differenzierung zwischen Informations-/Warnsystemen, übersteuerbaren (Fahrer kann das System korrigieren) und nicht übersteuerbaren Informationssystemen (Fahrer kann das System nicht korrigieren) vorgenommen werden. Aufgrund der Produkthaftung werden von den Herstellern vorwiegend Informations- und Warnsysteme angestrebt.

1.1 Hardware-Software Partitionierung

Die in den Assistenzsystemen eingesetzten Bildverarbeitungssysteme produzieren mit ihren Kameras eine große Anzahl Daten, die im Normalfall im $MByte/s$-Bereich liegt. Aus der Forderung nach einem echtzeitfähigen System mit geringem Platz- und Energieverbrauch ergibt sich der Bedarf nach einem eingebetteten System. Die hierfür verwendeten Prozessoren verfügen im allgemeinen aber nicht

über die Leistungsfähigkeit von Hochleistungsprozessoren in PCs. Daher ist es notwendig, durch weitere Spezialhardware die Datenverarbeitung stärker zu beschleunigen. Ein solches System mit Spezialhardware und Prozessoren wird als Hardware-Software Co-Design bezeichnet [40]. Die Partitionierung der Funktionen eines Algorithmus in Hardware und Software wird bisher vom Entwickler auf Basis seiner Erfahrung entschieden. Seit Anfang der 90er Jahre werden auch objektive Verfahren zur automatischen Erzeugung eines Hardware-Software Co-Designs entwickelt [120]. Diese Verfahren werden jeweils für einen bestimmten Anwendungsbereich, z.B. für die Kommunikationstechnik [45], und unter der Einschränkung eines einfachen Prozessor-Co-Prozessorsystems entwickelt. Konzepte zur parallelen Bildverarbeitung werden in [83] besprochen. Eine Lösung für die Bildverarbeitung in einem Hardware-Software Co-Design wird zum Beispiel in [56] vorgestellt. Hier ist jedoch keine automatisierte Partitionierung durchgeführt worden.

Die oben erwähnten Entwicklungen konnten die Eigenschaften von $FieldProgram-mableGateArrays$ (FPGA) zur Erzeugung eines Multiprozessorsystems mit zusätzlicher verschiedenartiger Spezialhardware auf einem Chip (SoPC) nicht berücksichtigen, da entsprechend große Schaltkreise erst seit wenigen Jahren zur Verfügung stehen. Mit diesen ist es nun möglich die Datenflüsse auf einem Chip sinnvoll zu parallelisieren, den Algorithmus anhand seiner Eigenschaften in kontrollflussorientierte und datenflussorientierte Bereiche aufzuteilen und diese dann optimal auf einem FPGA zu implementieren.

Hier setzt diese Arbeit an und beschreibt im folgenden eine Lösung für die objektive und automatische Erzeugung eines in Hardware und Software partitionierten Bildverarbeitungssystems. Als Beispiel für die automatische Hardware-Software Partitionierung eines Bildverarbeitungsalgorithmus' wird ein Assistenzsystem im Kraftfahrzeugbereich gewählt.

1.2 Aufbau der Arbeit

Die vorgelegte Arbeit besteht im Kern aus vier Teilen.

In Kapitel 2 wird der Stand der Technik für das Anliegen der Arbeit dargestellt. Zunächst wird auf die Prinzipien eines automatischen Hardware-Software Co-Designs eingegangen. Dabei werden die Eigenschaften der verschieden Prozessor- und Logikelemente vorgestellt, sowie Einblicke in die Graphentheorie gegeben. Im zweiten Teil dieses Kapitels werden verschiedene Verfahren zur Umgebungserfassung durch Kameras beschrieben.

Kapitel 3 beschäftigt sich mit der automatisierten Hardware-Software Partitionierung. Dabei wird auf Basis von C/C++ als Eingabesprache eine Designumgebung entwickelt, um Bildverarbeitungsalgorithmen auf Logikelemente und eine Anzahl von Prozessoren zu verteilen, so dass ein System mit minimalen Kosten entsteht. Die Basis der Partitionierung bildet das Simulated Annealing.

Im 4. Kapitel wird ein KFZ-Assistenzsystem zur Erfassung der Fahrzeugumgebung vorgestellt. Es basiert auf dem Prinzip der Stereophotogrammetrie. Dabei wird der Detektionsbereich in Ebenen verschiedener Entfernungen und unterschiedlicher Auflösungen aufgeteilt. Die Erkennung der Fahrzeuge erfolgt mit Hilfe eines Clusteralgorithmus. Zur Weiterverfolgung werden Kalman-Filter eingesetzt.

Das 5. Kapitel stellt die Ergebnisse der automatischen Partitionierung anhand des Beispielalgorithmus aus Kapitel 4 vor. Dabei entsteht ein Hardware-Software Co-Design mit Teilen des Algorithmus als Logikrealisierung und Teilen in einem Multiprozessorsystem mit vier Prozessoren.

Im Anschluss folgt eine Zusammenfassung der gesamten Arbeit.

Kapitel 2

Grundlagen - Stand der Technik

In diesem Kapitel wird ein kurzer Einblick in die Themengebiete gegeben, die zu einem besseren Verstehen der folgenden Abschnitte führen. Die unterschiedlichen Hardwarearchitekturen und deren Eigenschaften werden im Überblick dargestellt. Weiterhin werden mögliche Ansätze zur Entwicklung automatisierter Hardware-Software Co-Designs vorgestellt und ein Einblick in die Umgebungserfassung mittels Kameras gewährt. Es erfolgt eine Bewertung des Standes der Technik.

2.1 Hardware-Software Co-Design

Hardware-Software Co-Design Systeme entsprechen dem Wunsch nach optimaler Ressourcenausnutzung und Verarbeitungsgeschwindigkeit unter minimalen Kosten [40]. Die Verteilung des Algorithmus zwischen Hardware und Software hängt in hohem Maße von den Randbedingungen des Systems ab. Alle Randbedingungen kann man unter Implementierungskosten und Geschwindigkeit zusammenfassen. In Abbildung 2.1 ist ein Kosten-Geschwindigkeitsgraph dargestellt [40]. Durch die Vielzahl von Randbedingungen ist eine optimale Einteilung zwischen Hardware und Software nur schwer zu finden. Aus diesem Grund wurden hierfür verschiedene computerbasierte Algorithmen entwickelt, die hier im Überlick vorgestellt werden sollen.

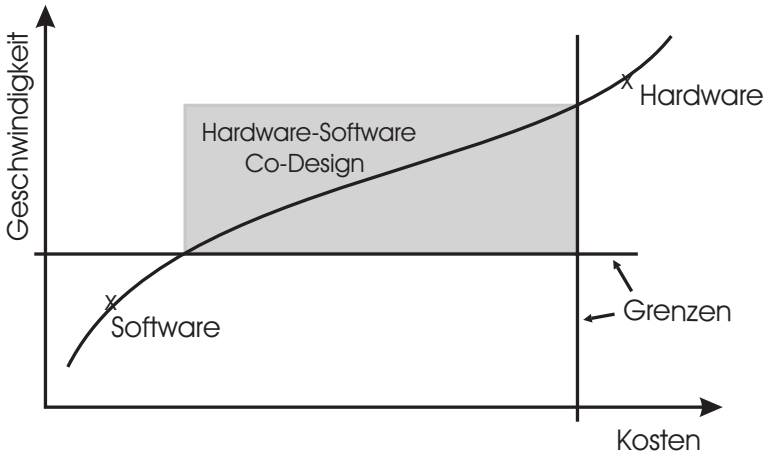

Abbildung 2.1: Kosten-Geschwindigkeitsgraph nach [40]

2.1.1 Motivation

Wie zuvor erwähnt, dient das Hardware-Software Co-Design (im folgenden als HW-SW Co-Design abgekürzt) zur Implementierung eines Algorithmus in Hardware und Software. Solche Aufteilungen sind dann sinnvoll, wenn eine gewöhnliche Implementierung des Algorithmus auf einem Prozessor nicht die gewünschten Resultate liefert. Das können zum Beispiel ein zu hoher Stromverbrauch oder eine zu geringe Verarbeitungsgeschwindigkeit großer Datenmengen sein. Diese Anforderungen werden an Bildverarbeitungssysteme in mobilen Anwendungen, zum Beispiel in Fahrzeugen oder in der Videotelefonie, gestellt.

Im folgenden werden verschiedene Hardwarekomponenten vorgestellt, die für die digitale Signalverarbeitung in einem HW-SW Co-Design geeignet sind.

2.1.2 Hardwarekomponenten

Ein Rechnersystem für die digitale Signalverarbeitung besteht meist aus mindestens einer *Arithmetic Logic Unit* (ALU) und Peripherieelementen, die der Kommunikation mit der Außenwelt dienen. Häufig befinden sich auch Speicherbausteine im System. Komplexe Datenverarbeitungseinheiten besitzen entweder Prozessoren (auch Prozessorenarrays) oder Logikelemente oder auch beide Architekturen. Logikelemente werden zu Entwicklungszwecken bzw. in Kleinserien

häufig auf PLDs (Programmable Logic Devices) realisiert. In [102] wird eine Gegenüberstellung von Prozessor und PLD vorgenommen. Die Umsetzung der Algorithmen auf PLDs unterscheidet sich von der Realisierung auf einem Prozessor. Grundsätzlich lässt sich aber jeder Algorithmus, der als Logik in einem PLD realisierbar ist, auch als Software auf einem Prozessor verwirklichen und umgekehrt. Die Verarbeitungsgeschwindigkeiten unterscheiden sich dabei oft deutlich.

Rechnerarchitekturen

Grundsätzlich existieren nach Flynn [108] vier unterschiedliche Rechnerarchitekturen, die im folgenden vorgestellt werden. **SISD** (Single Instruktion Single Data) Strukturen sind Einprozessorsysteme, die für jeden Befehl ein Datum verarbeiten (von-Neumann Rechner). Ein **MISD** (Multiple Instruktion Single Data) System arbeitet mit unterschiedlichen Befehlen einen identischen Datensatz ab. Diese Klasse wird allgemein als „Leer“ angesehen. Ein **SIMD** (Single Instruktion Multiple Data) Rechner besteht aus mehreren Verarbeitungseinheiten, von denen jede unterschiedliche Daten verarbeitet, wobei jedoch der ausgeführte Befehl auf jeder Einheit gleich ist (Beispiel: Vektorrechner). **MIMD** (Multiple Instruction Multiple Data) Strukturen können verschiedene Daten mit unterschiedlichen Befehlen verarbeiten.

Die CPU eines Prozessors arbeitet im Normalfall einen Befehl pro Verarbeitungsschritt ab. In einem Prozessor wird der Algorithmus in elementare Befehle aufgeteilt, die nacheinander ausgeführt werden. Die Algorithmierung ist kontrollflussorientiert. Man bezeichnet das Programmiermodell auf Grund der zeitlichen Sequenzialisierung als *Computing in Time* [102]. Zur Datenverarbeitung existieren verschiedene Prozessorarchitekturen - CISC (Complex Instruction Set Computer), RISC (Reduced Instruction Set Computer) und DSP (Digital Signal Processor).

RISC und CISC Architekturen können nach dem von-Neumann-Prinzip oder mit einer Harvard-Architektur arbeiten. Eine generelle Unterscheidung ist hier nicht möglich und hängt hauptsächlich von der Entwicklerphilosophie ab. So hat z.B. der ARM7 eine von-Neumann-Architektur und der ARM9 eine Harvard-

Architektur. In Pentium Prozessoren finden intern eine Harvard-Architektur bis zum L1-Cache und dann eine von-Neumann-Architektur Anwendung.

CISC: CISC-Prozessoren [73] sind universelle Prozessoren. Beim CISC Modell hat die CPU üblicherweise wenige Register. Es finden sich aber viele Befehle, darunter auch sehr mächtige, die z.B. in einer Schleife gleich mehrere Register bearbeiten. Die Befehle haben meist ungleiche Befehlslängen. Die häufigsten benutzten nur ein Byte, weniger häufige zwei oder drei Bytes. Vorteil sind ein kurzer Maschinencode und ein damit reduzierter Zentralspeicher sowie eine geringe Anzahl an Zugriffen auf den Programmspeicher. Die aufwendige Dekodierung der Befehle führt zu einer geringeren Verarbeitungsgeschwindigkeit, die aber mit hohen Taktfrequenzen ausgeglichen wird, was aber wiederum die Verlustleistung steigert [84]. Üblicherweise werden diese Befehle durch Mikrocode realisiert. Ein bekannter Vertreter dieser Baureihe ist die Intel x86-Familie.

RISC: Da festgestellt wurde, dass durch die Verwendung von Hochsprachencompilern nur 10% des Befehlssatzes von CISC-Prozessoren in 80% der geschriebenen Programme genutzt werden [81], sind RISC-Prozessoren mit einem wesentlich geringeren Befehlssatz entwickelt worden. Als Ausgleich besitzt dieser Prozessortyp erheblich mehr Register, so dass häufiger schnelle Register-Register Operationen als langsame Speicher-Register Operationen ausgeführt werden. Die wenigen Befehle machen das Design einfacher und somit billiger in der Herstellung. Der Assemblercode ist jedoch durch die wenigen Befehle länger als bei CISC. Diese Prozessorart ist heute im eingebetteten Bereich stark verbreitet. Ein bekannter Vertreter dieser Architektur ist der Motorola PowerPC [84]. Merkmale von RISC-Prozessoren sind: Wenige, schnell zu dekodierende Befehle, einheitliches Befehlsformat und schnelle Dekodierung über festverdrahtete Logik.

Die Trennung von CISC und RISC gibt es in der Form heute nur noch selten, da die RISC Prozessoren immer aufwendiger werden und daher sich dem CISC Prinzip annähern. Die klassischen CISC Prozessoren (Intel Pentium) übernehmen währenddessen immer mehr Anleihen der RISC-Welt. Nach dem Dekodieren der Befehle werden diese im Pentium in einfachere RISC Befehle übersetzt, umgruppiert und zwei Rechenwerken, die reine RISC Maschinen sind, zugeführt.

DSP: Als DSP bezeichnet man im allgemeinen die Rechnerstrukturen, die eine effiziente Implementierung von Algorithmen der digitalen Signalverarbeitung ermöglichen [30]. Die Haupteinsatzgebiete liegen in Telekommunikationssystemen mit hohem Datenaufkommen oder anderen Audio- und Bildsignalverarbeitungsaufgaben. Die Architektur berücksichtigt einige Besonderheiten der digitalen Signalverarbeitung wie hohe Datendurchsätze, präzis organisierte Datenströme und schnelle Additionen, Subtraktionen und Multiplikationen. Weiterhin ist es möglich verschiedene Operationen wie eine Addition und eine Multiplikation parallel durchzuführen. DSPs können, wie die zuvor beschriebenen Prozessortypen, frei programmiert werden, sind jedoch nicht für einen Multitaskingbetrieb vorgesehen. Durch ihre Architektur sind sie aber wesentlich besser für schnelle datenflussorientierte Anwendungen geeignet. Für spezielle Anwendungen, wie z.B. JPEG-Kodierungen, wurden auch speziell dafür optimierte DSPs entwickelt. Diese sind dann beschränkter in ihrer Programmierung.

Multiprozessorsysteme

Die Entwicklung immer schnellerer Prozessoren schreitet nach wie vor voran. Es zeigt sich aber, dass das Gesetz von Moore nicht mehr allein durch die Erhöhung der Prozessortaktrate zu erfüllen ist, sondern Multiprozessorsysteme aufgebaut werden müssen [3]. Ältere Systeme beschränkten sich auf das Auslagern von speziellen Algorithmen auf mathematische Co-Prozessoren. Die Grafikkarte eines Standard-PCs kann ebenfalls als Co-Prozessor angesehen werden, der den Hauptprozessor von rechenintensiven Spezialaufgaben entlastet. Transputer waren die ersten Parallelrechner mit mehreren parallelen Recheneinheiten und einem entsprechenden Kommunikationsnetzwerk [24].

Weitere Parallelrechnerkonzepte sind die Vektorrechner (SIMD-Architektur). Dabei wird ein Befehl in einer Zeiteinheit auf mehrere unabhängige Daten angewendet. Das ist möglich, da der Befehl auf mehreren ALUs bzw. in einer Pipeline gleichzeitig ausgeführt wird. Beispiel eines solchen Rechners ist zum Beispiel Cray-1 [92].

Neuere Entwicklungen setzen auf komplette und unabhängig arbeitende Prozes-

soreinheiten, wie der AMD Opteron oder der Intel Core-Duo mit jeweils zwei Prozessorkernen. Dadurch kann die Rechenleistung vergrößert werden, ohne die Taktraten zu erhöhen. Eine andere Möglichkeit besteht in der Auslagerung weiterer rechenintensiver Aufgaben auf extra hierfür entwickelte Co-Prozessoren. Neben der Grafikkarte kann so auch ein physikalischer Co-Prozessor zum System hinzukommen, der in aufwendigen Spielen die Bewegungen der Spielfiguren in Echtzeit berechnet. Durch die Spieleindustrie ist auf der Hardwareseite ein enormer Entwicklungsbedarf entstanden, um die graphischen Anforderungen der Spieler zu erfüllen. Spielconsolen wie *XBox* oder *Playstation* spielen dabei eine große Rolle. So wird in der Playstation2 ein Mehrprozessorchip eingesetzt. Der Cell-Chip besteht aus einer Prozessoreinheit, die die Steuerung übernimmt, und 16 weiteren Prozessorelementen, die parallel Daten verarbeiten können. Mit einer Taktrate von 4,6 GHz soll dieser Chip bis zu einem Terraflop Daten verarbeiten können. Ein weiterer Parallelrechner in einer anderen Kategorie ist z.B. der „Earth Simulator“ mit 640 Knoten und jeweils 8 Prozessoren. Er gehört mit einer Verarbeitungsgeschwindigkeit von 36 Terraflops zu den leistungsfähigsten Computern der Welt [42]. Konfigurierbare Multiprozessoren bieten preiswerte Alternativen zu den beschriebenen Systemen. Sie lassen sich in Grenzen [10] oder auch völlig frei konfigurieren [100].

Parallelrechner arbeiten, wie bereits erwähnt, mit mehreren untereinander vernetzten Prozessoren.

In Multiprozessorsystemen können die Prozessoren über zwei verschiedene Varianten miteinander kommunizieren. Entweder über das Versenden von Nachrichten (message passing system) durch ein entsprechendes Verbindungsnetzwerk, oder über den Austausch gemeinsamer Variablen durch einen gemeinsamen Speicher (shared memory system) zwischen den einzelnen Prozessoren.

Die Leistung eines Parallelrechnersystems hängt davon ab, wie gut der Algorithmus auf der Zielhardware implementiert wurde. Dabei kommt es insbesondere auf den zu erwartenden Kommunikationsaufwand des Zielsystems an. Der Geschwindigkeitsfaktor S_n bezeichnet die Laufzeitverbesserung eines Systems mit n Prozessoren gegenüber einem System mit einem Prozessor.

$$S_n = \frac{t_1}{t_n} \qquad S_n \leq n \tag{2.1}$$

Theoretisch kann der Geschwindigkeitsfaktor $S_n = n$ betragen. Da aber nicht alle Programmteile parallelisiert werden können und auch ein gewisser Kommunikationsaufwand entsteht, wird die Effizienz immer einen Wert kleiner als Eins annehmen. Durch die Aufteilung der Programmlaufzeiten in einen sequentiellen und einen parallelen Teil, lässt sich nach Amdahl's Gesetz [9] die Zeit t_n nach Gleichung 2.2 beschreiben.

$$t_n = t_{seq} + \frac{t_{par}}{n} \tag{2.2}$$

$$f_S = \frac{T_{seq}}{T_{seq} + T_{par}} \qquad 0 \leq f_S \leq 1 \tag{2.3}$$

Ist f_S der sequentielle Anteil eines Programms (Gleichung 2.3), dann ergibt sich für S_n folgende Gleichung:

$$S_n = \frac{1}{f_S + \frac{1-f_S}{n}} \tag{2.4}$$

Beträgt also der sequentielle Anteil eines Algorithmus 20% so kann S_n maximal fünf sein, unabhängig von der Anzahl der eingesetzten Prozessoren.

Logikschaltkreise

FPGAs (Field Programmable Gate Arrays) sind eine Unterklasse der PLDs (Programmable Logic Devices). Man kann sie auch als Spezialprozessoren bezeichnen, in denen das Programm direkt in Hardware umgesetzt und abgearbeitet wird. Bei der Umsetzung einer Softwarefunktion in Logik können viele Verarbeitungsstufen zeitgleich parallel in einem Verarbeitungsschritt geschehen [101]. Die Algorithmierung erfolgt datenflussorientiert, d.h. der Algorithmus wird als einzige Instruktion mit dauernder Ausführung in strukturierte Logik übersetzt. Durch die räumliche

Sequenzialisierung wird dieses Programmiermodell als *Computing in Space* [102] bezeichnet.

Auf der Grundlage der Datenflussorientierung ergeben sich zwei Parallelisierungsebenen:

1. Unabhängige Daten können zeitgleich in parallelen Funktionseinheiten bearbeitet werden.
2. Während einer Operation im zweiten Verarbeitungsschritt können bereits Operationen in den ersten Stufen mit einem nachfolgenden Datensatz ausgeführt werden (Pipelining).

Vergleich der Hardwarekonzepte

In Abbildung 2.2 sind die verschiedenen Konzepte der Datenverarbeitung noch einmal zusammenfassend dargestellt. Abbildung 2.2.a zeigt den Datenfluss wie er in einem einfachen von-Neumann Rechner existiert.

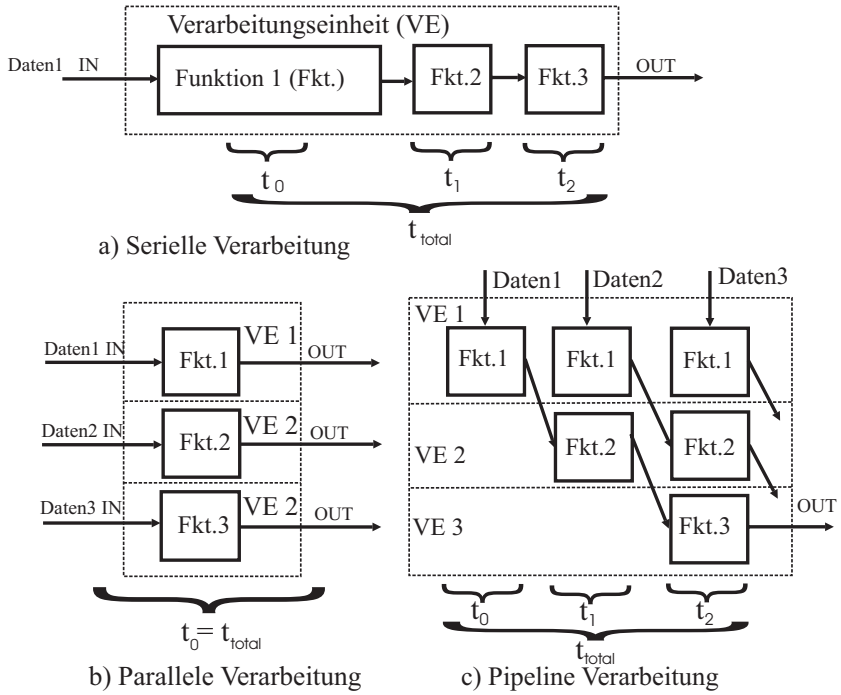

Abbildung 2.2: Verarbeitungsstrukturen

Abbildung 2.2.b stellt die parallele Abarbeitung von verschiedenen unabhängigen Datenflüssen dar, wie sie in FPGAs und Parallelprozessorsystemen erfolgt. In Abbildung 2.2.c ist ein Pipelineprinzip zu sehen, das in modernen Prozessoren (Befehlspipeline) und auch in FPGAs (Datenpipeline) realisiert wird.

Es lässt sich sagen: Programmierbare Logik hat den Vorteil einer schnellen Datenverarbeitung, wenn der zu implementierende Algorithmus datenflussorientiert ist, d.h. ein kontinuierlicher Datenstrom durch die Schaltung fließt. Einfache ereignisorientierte Datenflüsse lassen sich mit Hilfe von State Maschinen in Logik implementieren. Der Vorteil des Prozessors liegt in der Abarbeitung komplexer ereignisorientierter Datenströme. Hier ist es möglich, auf verschiedene Ereignisse entsprechend zu reagieren und in die dazugehörigen Unterprogrammroutinen zu wechseln. Diese Arbeitsweise ist besonders bei Betriebssystemen wichtig, weil dort neue Tasks aufgerufen, aber auch alte wieder geschlossen werden müssen. Solche Funktionen würden in programmierbarer Logik nicht sinnvoll realisierbar sein, da sie dort fest integriert sind. Es ist nicht möglich, neue Logik während des Betriebes einfach hinzuzufügen, ohne den kompletten Chip neu zu programmieren. Nach [99] soll eine Teilapplikation in programmierbarer Hardware integriert werden, wenn eine der folgenden Bedingungen zutrifft:

- Die Teilapplikation wird in regelmäßigem zeitlichen Abstand aufgerufen, wobei der exakte Zeitpunkt oder die exakte Zeitdifferenz zwischen zwei Aufrufen wichtig sind.
- Die Teilapplikation selbst, d.h. der einzelne Aufruf, erfordert wenige Operationen, die jedoch mit hoher Geschwindigkeit auszuführen sind.
- Die Teilapplikation kann parallel zu anderen Programmteilen ablaufen, d.h. es existieren keine begrenzenden Interaktionen oder Nebenbedingungen, die durch Sequentialität des Prozessors implizit erfüllt werden.

2.1.3 Graphen

Zur Partitionierung von Algorithmen werden häufig Graphen verwendet. Sie sollen deshalb kurz vorgestellt werden. In [58] und [28] werden diese Grundlagen

weiter ausgeführt. In [108] und [44] wird die Graphentheorie im Zusammenhang mit der parallelen Datenverarbeitung erläutert.

Ein Graph G wird durch seine Knoten und Kanten beschrieben $G = (V, E)$. Punkte eines Graphen heißen Knoten. Diese sind durch Kanten verbunden. Der Grad eines Knotens x mit $x \in V$ bezeichnet die Anzahl der mit einem Knoten verbundenen Kanten *Grad* $g(x)$. Sind die Richtungen der Kanten nur unidirektional, ist das ein gerichteter Graph. Wobei hier die Kanten als Bögen bezeichnet werden.

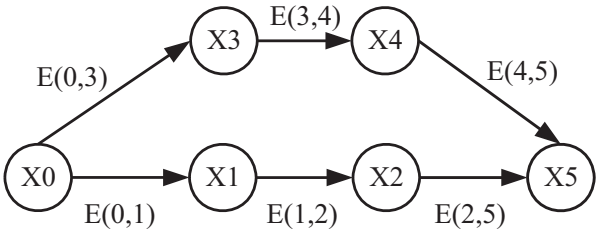

Abbildung 2.3: Gerichteter Graph mit 6 Knoten und der Länge G^3

Als Weg wird ein Graph G bezeichnet, der die Knoten $V = \{x_0, x_1, ..., x_n\}$ und die Kanten $E = \{(0,1); (1,2); ...; (n-1, n)\}$ besitzt. Die Anzahl der Kanten eines Weges wird als seine Länge bezeichnet. Ein Weg der Länge n wird mit G^n bezeichnet. Eine *Eulersche Tour* beschreibt einen Weg, wobei jede Kante eines Graphen nur einmal verwendet wird. Erreicht man dabei wieder den Ausgangsknoten, so entsteht ein *Eulerscher Kreis*. Dabei gilt der Satz: Der Graph besitzt einen Eulerschen Kreis genau dann, wenn jeder Knoten einen geraden Grad hat. Wird ein Weg gefunden, in dem jeder Knoten nur einmal verwendet wird, so ergibt sich ein *Hamiltonscher Kreis*. Der Durchmesser $\delta(G)$ beschreibt die maximale Distanz zwischen zwei beliebigen Knoten. Die Knotenkonnektivität $vc(G)$ ist die minimale Anzahl von Knoten, die gelöscht werden müssen, um den Datenfluss des Graphen komplett zu unterbrechen. Für die Kantenkonnektivität $ec(G)$ gilt das äquivalente Prinzip. Weitere Details zu Graphen sind in Anhang A dargestellt.

Ziel ist es, den Funktionsgraphen eines Algorithmus optimal auf den Graphen der Zielhardware abzubilden. Das führt zu minimalen Verarbeitungskosten. Verschiedene mögliche Partitionierungsverfahren werden im folgenden erläutert.

2.1.4 Partitionierung von Hardware und Software

Die Partitionierung dient dazu einen Algorithmus mit N Funktionen F_i auf eine Architektur mit n verschiedenen Verarbeitungselementen PE [40] aufzuteilen. Ein Verarbeitungselement kann entweder ein Prozessor oder ein Logikelement sein. Ziel ist es, eine zuvor aufgestellte Kostenfunktion zu minimieren. Kenngrößen für die Kosten können hier die Verarbeitungszeit auf dem Prozessor t_{SW} oder in der Logik t_{HW}, der Kommunikationsoverhead t_C zwischen den Verarbeitungselementen, der Umfang der Software S_{SW} und die durch die benutzte Logik verbrauchte Hardware S_{HW} sein. Weitere Ergänzungen sind möglich. Die Größen werden in einer Kostenfunktion C zusammengefasst (Gleichung 2.6). Das partitionierte System PE_{Ges} ist wie folgt beschrieben:

$$PE_{Ges} = PE_0 \cup PE_1 \cup ... \cup PE_{n-1} \tag{2.5}$$

$$C = a_0 \cdot S_{PE0}(F_0) + ... + a_{n-1} \cdot S_{PE_{n-1}}(F_{N-1}) + b \cdot t_{SW} + c \cdot t_{HW} + d \cdot t_C \tag{2.6}$$

Dabei ist F_0 der Algorithmus mit S_0 als Größe von F_0 in Partition 0, S_1 als Größe von Partition 2 bis hin zu Partition $n-1$. Die Faktoren a_n, b, c, d repräsentieren die Gewichte der Kostenfaktoren in der Kostenfunktion C. Die Kosten sind dabei eine virtuelle Zahl, die aus der gewichteten Addition realer Größen, wie Logikzellenverbrauch, Verarbeitungszeit oder Kommunikationszeit zusammengesetzt ist. Ziel der Partitionierung ist es, die entstehenden Kosten zu minimieren.

Das Partitionierungsproblem ist NP (Non-deterministic Polynomyal time) - vollständig, da es für N Funktionen und n Verarbeitungselemente $O(n^N)$ mögliche Partitionierungen gibt und es deshalb nicht effizient lösbar ist. Daher wurden für die automatische Partitionierung verschiedene Heuristiken entwickelt, um die Partition mit den minimalen Kosten zu finden. In [88] werden die verschiedenen Ansätze erläutert. Dabei wird immer ein Kompromiss zwischen Rechenaufwand und Güte (Kosten) der Partitionierung eingegangen. Man unterscheidet zwischen konstruktiven und iterativen Algorithmen. Konstruktive Algorithmen sind Ver-

fahren, die eine Partition durch schrittweises Hinzunehmen von Objekten erzeugen. Eine gültige Partitionierung ist erst am Ende des Verfahrens vorhanden. Iterative Algorithmen starten mit einer beliebigen Anfangspartitionierung und verbessern diese iterativ. Hier ist bei jedem Iterationsschritt eine gültige, wenn auch nicht optimale Lösung vorhanden. Im folgenden werden einige Verfahren kurz vorgestellt.

Random Mapping

Dieses Verfahren ist ein konstruktives Verfahren und weist jeder Funktion F_i zufällig einen Prozessor bzw. ein Logikelement zu. Dabei ist es unerheblich, welche Eigenschaften die einzelnen Funktionen besitzen. Das Ergebnis ist eine willkürliche Verteilung der Funktionen auf die verschiedenen Prozessoren und Logikelemente. Da die Funktionseigenschaften keinen Einfluss auf die Partitionierung haben, entsteht eine nicht optimale Partitionierung. Die Kosten der Partitionierung werden am Ende einer Iteration bewertet. Genügen sie nicht den Anforderungen, wird ein weiterer Iterationsschritt durchgeführt, in dem die Funktionen zufällig neu auf Prozessoren und Logikelemente verteilt werden. Die Verwendung des Random Mapping in diesem Zusammenhang ist sehr aufwendig und besitzt eine hohe Zeitkomplexität, deshalb wird dieses Verfahren häufig für die Generierung der Startpartition für die im Folgenden vorgestellten Verfahren verwendet. In diesem Fall wird das Random Mapping nur einmal durchlaufen. Die Zeitkomplexität beträgt dann $O(n)$. Die Zeitkomplexität ist die Anzahl von Schritten, die der Algorithmus zur Lösung des Problems benötigt.

Hierarchische Clusterung

Bei der hierarchischen Clusterung werden die Funktionen F_i schrittweise zu größeren Clustern gruppiert [66]. Dabei steht zu Beginn für jede Funktion ein eigener Cluster zur Verfügung. Mit Hilfe der Zusammengehörigkeit zwischen den Funktionen werden diese dann zu größeren Partitionen vereint. Die Zusammengehörigkeit der Funktionen lässt sich durch einen Graphen $G(F, E)$ mit der Gewichtung der Kanten E ermitteln. Das Verfahren wird so lange durchgeführt, bis die gewünsch-

te Anzahl an Clustern erreicht ist. Es besitzt eine Zeitkomplexität von $O(n^2)$.

Kerninghan-Lin Algorithmus

Der Kerninghan-Lin Algorithmus ist ein iteratives Verfahren [55]. Es minimiert die Kanten zwischen zwei Partitionen. Dabei wird für jedes Objekt der Kostengewinn bestimmt, wenn man es in eine andere Partition verschiebt. Es wird das Objekt verschoben, welches den größten Kostengewinn verursacht. Der Algorithmus kann nicht aus einem lokalen Minimum entweichen. In einer Iteration wird jede Funktion einmal umgruppiert und die Kosten für die jeweilige Partition werden vermerkt. Am Ende der Iteration wird die Partition mit den geringsten Kosten für den nächsten Iterationsschritt genutzt. Diese Routine wird so lange ausgeführt bis keine Partition mit geringeren Kosten zur Verfügung steht. Der Algorithmus besitzt eine Zeitkomplexität von $O(n^3)$.

Lineare Programmierung

Zur Erzeugung einer Partition mittels linearer Programmierung wird ein ILP (*Integer Linear Program*) erstellt. Die Zugehörigkeit einer Funktion F_i zu einem Verarbeitungselement PE_k wird durch eine binäre Variable $x_{F,k} = 1$ ausgedrückt.

$$x_{F,k} \in \{0,1\} \qquad 1 \leq F \leq N,\ 1 \leq k \leq n \tag{2.7}$$

$$C_k = \sum_{k=1}^{j} (x_{F,k} \cdot c_{F,k}) \quad 1 \leq k \leq n \tag{2.8}$$

Die Kostenfunktion C_k soll minimiert werden. Dabei sind die Kosten $C_{i,k}$ für die Zugehörigkeit von F_i zu PE_k gegeben. Weitere Nebenbedingungen können hinzugefügt werden. ILPs werden exakt mit Branch-and-Bound Algorithmen [85] gelöst. So wird das Problem in einen Suchbaum transformiert und in verschiedene Zweige (Branches) aufgeteilt. Zweige, deren Verfolgen bereits frühzeitig zu einer Verletzung von Randbedingungen (Bound) führen, werden eliminiert (Ausästen).

ILP sind exakte Verfahren, die im ungünstigsten Fall zu einer exponentiellen Zeitkomplexität führen. Daher ist diese Variante nur für kleine Probleme anwendbar.

Simulated Annealing

Beim Simulated Annealing [57] kann ein Objekt mehrfach umgruppiert werden. Es ist dann möglich, im Gegensatz zum Kerninghan-Lin Algorithmus, ein lokales Minimum wieder zu verlassen. Ausgehend von einer Startpartition wird eine simulierte Temperatur T langsam verringert. T entspricht dabei der Akzeptanzschwelle unterhalb derer die Kostenveränderung ΔC zwischen vorhergehender Partition und aktueller Partition liegen muss. Eine neue Partition wird durch zufälliges Auswählen und Umpartitionieren einer Funktion F_i erzeugt. Die neue Partition kann angenommen oder abgelehnt werden. Die dazugehörige Funktion P für das energetische Minimum lautet wie folgt:

$$P(\Delta C, T) = min(1, e^{-\frac{\Delta C}{T}}) \tag{2.9}$$

Damit wird die Wahrscheinlichkeit einer kostensteigernden Umgruppierung ΔC mit abnehmender Temperatur immer geringer. Der Partitionierungsalgorithmus wird bei einer Temperatur solange durchgeführt, bis keine Verbesserung der Kosten nach einer Anzahl von Iterationsschritten mehr auftritt. Dann wird die Temperatur verringert. Der gesamte Prozess arbeitet, bis die minimale Temperatur erreicht wird. Die Zeitkomplexität kann zwischen exponentiell und konstant variieren. Das Simulated Annealing wird sehr häufig für die Lösung kombinatorischer Probleme verwendet, da es robust, leicht zu handhaben und für eine große Anzahl von Problemen anwendbar ist.

Evolutionäre Algorithmen

Ein evolutionärer Algorithmus ist ein Optimierungsverfahren, das als Vorbild die biologische Evolution hat. Durch Manipulation des Erbgutes werden dort Anpassungen der Organismen an ihre Umgebung vorgenommen. Die Gesamtpopulation der Organismen ist das zu partitionierende System, in dem die Organismen die

einzelnen Partitionen darstellen. Die Umgebung stellt die Randbedingungen, an die sich das System anzupassen hat, dar. Die Population kann durch die Auswahlverfahren Selektion, Rekombination, Kreuzung und Mutation iterativ verbessert werden. Die Mutation ist zufällig und dient der Erzeugung von Varianten, um gegebenenfalls ein lokales Minimum wieder verlassen zu können. Bei der Rekombination werden verschiedene Teile getrennt und mit anderen Teilen wieder zusammengeführt. Dabei werden eng verknüpfte Funktionen seltener getrennt als locker verbundene. Bei der Kreuzung werden die guten Eigenschaften zweier Teile kombiniert, um so ein neues Teil mit verbesserten Eigenschaften zu erhalten. Die Selektion steuert die Suchrichtung der Evolution, indem sie festlegt, welche Faktoren eine höhere Gewichtung bekommen. Eine Fitnessfunktion gibt den Grad der Eignung der aktuellen Partition an. Evolutionäre Algorithmen können sehr gut für komplizierte Probleme eingesetzt werden, für die es keine anderen Lösungsverfahren gibt.

Das Simulated Annealing und die evolutionären Algorithmen werden häufig für komplizierte Optimierungsaufgaben verwendet. Sie wurden in verschiedenen Arbeiten miteinander verglichen ([67], [75] und [37]). Generell lässt sich sagen, dass beide Verfahren ähnlich gute Ergebnisse liefern. Die Untersuchungen haben gezeigt, dass das Simulated Annealing eine schnellere Annäherung an ein optimales Ergebnis als die evolutionären Algorithmen ermöglicht. Mit dem Simulated Annealing lässt sich also in einer geringeren Zeit eine optimale Partitionierung finden. Generell hängt der Einsatz des Optimierungsverfahrens jedoch vom Einsatzgebiet ab. Nach Wiangtong [119] sind die Ergebnisse einer Partitionierung für ein HW-SW Co-Design System mit einem evolutionären Algorithmus dem Simulated Annealing unterlegen.

Die vorgestellten Verfahren dienen als Entscheidungsgrundlage innerhalb automatischer Hardware-Software Co-Design Systeme [21], wie sie im nächsten Abschnitt vorgestellt werden.

2.1.5 Automatisierte Co-Design Systeme

Automatisierte Hardware-Software Co-Design Systeme sollen die Vorhersagbarkeit für Energieverbrauch, Geschwindigkeit und Kosten von eingebetteten Systemen mittels entsprechender Analysen und Schätzmethoden verbessern. Ziel ist es, die Algorithmen zwischen Hardware und Software so aufzuteilen, dass ein entsprechendes Gütekriterium erreicht wird. Das HW-SW Co-Design lässt sich nach Abbildung 2.4 in verschiedene Problemfelder aufteilen.

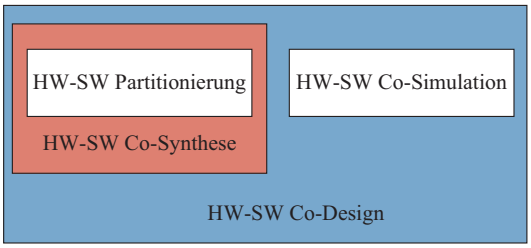

Abbildung 2.4: Bestandteile des HW-SW Co-Designs

COSYMA und VULCAN

Zwei frühe Systeme für ein Hardware-Software Co-Design waren VULCAN von der Universität Stanford [41] und COSYMA, entwickelt an der Universität Braunschweig [45]. Sie beinhalten den kompletten Designfluss von der Spezifikation bis zur Synthese.

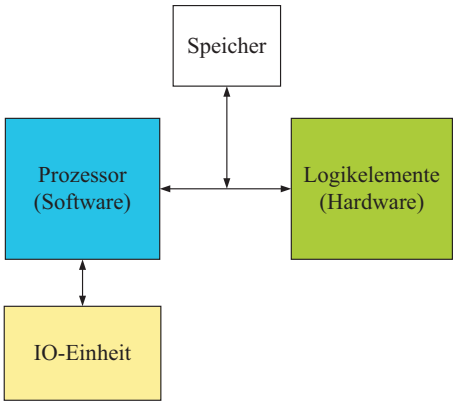

Abbildung 2.5: Hardware-Software System im COSYMA Ansatz

Die Zielarchitektur besteht aus einer Prozessor - Co-Prozessor Architektur mit einem einzelnen Prozessor und mehreren Logikelementen, die durch einen gemeinsamen Bus und einen gemeinsamen Speicher miteinander verbunden sind (Abbildung 2.5). Das Ziel von COSYMA und VULCAN ist die Geschwindigkeitssteigerung vorhandener Programme durch Auslagerung spezieller Funktionen in Hardware. Als Eingabesprache wird ein Derivat von *C* verwendet. Hierbei werden Erweiterungen für die parallele Verarbeitung eingefügt. Die Funktionen des Algorithmus werden in einen Graphen überführt. Der Graph wird analysiert und dann partitioniert. COSYMA geht dabei von einem komplett als Software auf einem Prozessor realisierten System aus. Es werden so lange Funktionen in die Logik transferiert, bis die Systemanforderungen erfüllt sind. In Abbildung 2.6 ist der Designflow des COSYMA Systems zu sehen.

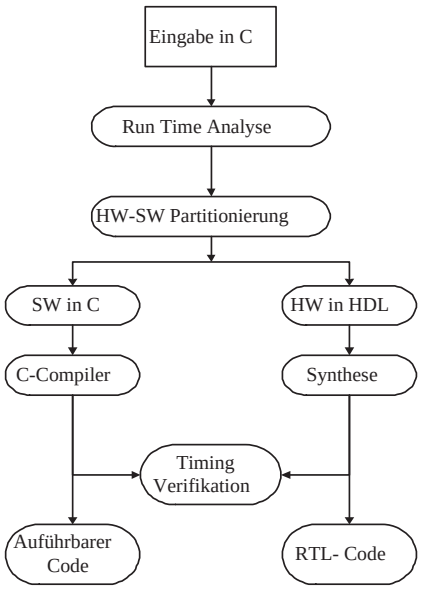

Abbildung 2.6: Das Cosyma System

VULCAN geht hingegen von einer komplett als Logik vorliegenden Startpartition aus und transferiert so lange Funktionen in die Software bis die Systemanforderungen gerade noch erfüllt sind. Beide Systeme nutzen einfache Verarbeitunsmodelle und setzen z.B. den Prozessor in einen Wartezustand, während die Logik Daten verarbeitet.

Polis

Das Polis-System ist ein von der Universität Berkeley entwickeltes System zur Co-Synthese von partitionierten Algorithmen [14].

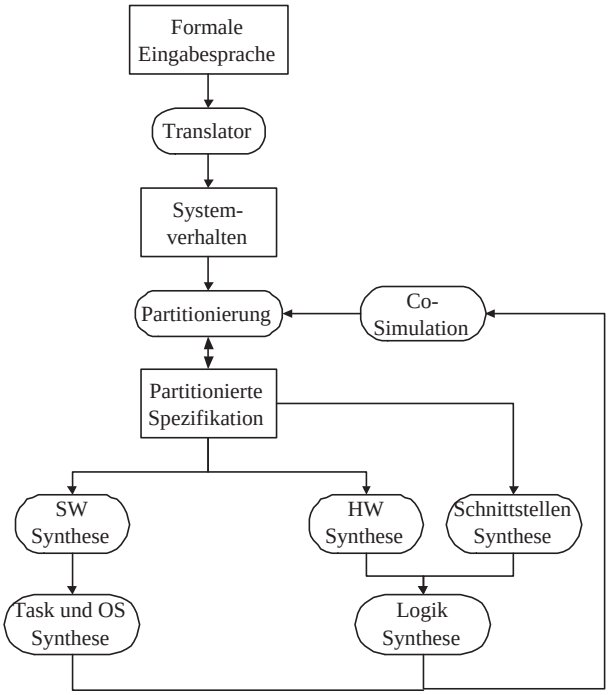

Abbildung 2.7: Das Polis System

Es wurde hauptsächlich für kontrollflussorientierte Systeme entworfen. Die Eingabe erfolgt graphisch mit Hilfe von *Finite State Maschinen* (FSM). Hier werden speziell entwickelte Co-Design FSM (CFSM) genutzt. CFSM kommunizieren mit anderen CFSM asynchron mit Hilfe von Nachrichten. Die Eingabesprache ist Esterel.

Das Polis-System dient hauptsächlich zur Co-Simulation des Algorithmus. Die Partitionierung wird dem Entwickler überlassen. Ausgehend von dem über die CFSM eingegebenen Algorithmus und der manuellen Partitionierung wird der entsprechende Hardware- und Softwarecode erzeugt. In Abbildung 2.7 ist der Designflow des Polis-Systems zu sehen. Dabei wird, ausgehend von den CFSMs, das Verhalten des Systems ermittelt. Nach der manuellen Partitionierung werden daraus die Software, die Hardware und die Schnittstellen synthetisiert. Das Ge-

samtsystem wird dann in einer Co-Simulation evaluiert, deren Ergebnisse dem Entwickler für die Verbesserung der Partitionierung zugeführt werden. Die Partitionierung wird also iterativ verbessert, bis die Randbedingungen eingehalten werden.

Ptolemy

Ptolemy ist eine Designumgebung zur Simulation von heterogenen Systemen [51]. Es unterstützt und integriert verschiedene *Models of Computation* (MoC) und ist somit sehr gut für eine Hardware-Software Co-Design Simulation und Evaluation geeignet. Dabei wird jedes Subsystem einzeln modelliert und in das Gesamtsystem durch entsprechende Transitionsmodelle integriert.

Chinook

Das Chinook-System [87] beschäftigt sich mit der automatischen Synthese von Schnittstellen zwischen Hardware und Softwarekomponenten.

DK Design Suite

Die DK Design Suite ist ein kommerzielles Designwerkzeug der Firma Celoxica. Die Vorgehensweise der Synthese ist ähnlich wie im COSYMA und VULCAN Ansatz. Die Eingabesprache ist HandelC. Sie basiert auf C/C++, die um Hardwarekomponenten erweitert wurde und die Beschreibung nebenläufiger Anweisungen ermöglicht. Dabei wird auch hier eine Prozessor - Co-Prozessor Architektur aufgebaut. Die Nutzung mehrerer Prozessoren ist nicht vorgesehen. Das System erlaubt ebenfalls eine Co-Simulation und die Synthese der Hardware- und Softwarekomponenten sowie der entsprechenden Schnittstelle zwischen beiden Komponenten.

SoPC - FPGA

Der Trend zu *Systems on Chips* (SoC) in Verbindung mit immer größer werdenden FPGAs hat eine neue flexible Klasse von Systemen hervorgebracht - die

Systems on Progammable Chips (SoPC). Hier werden komplette Systeme auf einem einzelnen FPGA realisiert. Durch den integrierten Speicher können diese Systeme nicht nur aus Logik, sondern auch aus einer Anzahl von Prozessoren bestehen, die als IP-Core bereit stehen. Diese IP-Cores können zusammen geschaltet und mit eigener Logik ergänzt werden. Das so entstehende System ist ebenfalls ein HW-SW Co-Design. Jedoch beschränkt sich die Designunterstützung hier hauptsächlich auf das Bereitstellen verschiedener IP-Cores und einer Entwicklungsumgebung zum Zusammenfügen der einzelnen Elemente [27]. Eine Co-Simulation bzw. eine automatische Partitionierung des Algorithmus ist derzeit nicht verfügbar.

SystemC

Beim COSYMA Ansatz und bei der DK Design Suite werden als Ausgangsbasis C-Programme genutzt, die mit einer speziellen C-Sprache geschrieben wurden, um die Hardwaresynthese einzelner Funktionen zu vereinfachen. Eine weitere auf C basierende Sprache für diesen Anwendungsbereich ist SystemC. SystemC ist eine C++ Bibliothek, die die Beschreibung nebenläufiger Datenflüsse in einem Programm ermöglicht. Durch den Einsatz von C^x, HandelC und SystemC für die Beschreibung der zu partitionierenden Programme in den einzelnen HW-SW Co-Designumgebungen wird der Trend hin zu einem auf C/C++ und seinen Ablegern basierenden HW-SW Co-Designsystem deutlich. Daher soll auch in der vorliegenden Arbeit eine C/C++ Beschreibung des zu partitionierenden Algorithmus als Ausgangspunkt dienen.

2.1.6 Diskussion

Die hier vorgestellten HW-SW Co-Designsysteme beziehen sich hauptsächlich auf ein System mit einem Prozessor und einen Co-Prozessor (COSYMA, VULCAN), da zum Zeitpunkt der Entwicklung vielseitige und gleichzeitig flexible Parallelrechnerarchitekturen, wie sie heute mit FPGAs möglich sind, noch nicht absehbar waren. Andere Systeme, wie Polis oder Ptolemy, beschäftigen sich eher

mit der Co-Simulation und Verifikation als mit der automatischen Partitionierung. Die vorgestellten Systeme lösen dabei immer nur ein Teilproblem des HW-SW Co-Design. Systeme wie die DK Design Suite bieten eine Komplettlösung für Prozessor-Co-Prozessorsysteme an. Hier wird jedoch eine speziell entwickelte Eingabesprache verwendet. Aktuelle Systeme, basierend auf einem SoPC-Builder, eignen sich sehr gut zur Realisierung eines HW-SW Co-Designs. Sämtliche Spezifikationen und auch die Partitionierung müssen jedoch vom Entwickler übernommen werden. Aufgrund dessen ist zu erkennen, dass gerade die automatische Partitionierung große Probleme mit sich bringt und nur für spezifische Problemstellungen realisierbar ist.

Für Bildverarbeitungsalgorithmen sind FPGAs sehr sinnvoll einsetzbar, weil hier durchaus verschiedene datenflussorientierte Verarbeitungsstränge parallel arbeiten können. Deren Ergebnisse sind wiederum Ausgangspunkt für unterschiedliche kontrollflussorientierte Algorithmen, die nicht mehr als Logik realisierbar sind, sondern einen separaten Prozessor benötigen. Durch die geforderte Komplexität und die Möglichkeit, das System flexibel mit einer unterschiedlichen Anzahl an Logikkomponenten und Prozessoren (siehe Kapitel 3.2) zu implementieren, sind Systeme wie COSYMA und VULCAN nicht geeignet. Stattdessen wird die Entwicklung eines neuen HW-SW Partitionierungssystems speziell für Bildverarbeitungsaufgaben notwendig. Als Optimierungsverfahren für die Partitionierung kommen das Simulated Annealing bzw. ein evolutionärer Algorithmus in Betracht, da diese für viele aufwendige Optimierungsaufgaben eine optimale Lösung ermöglichen. Das Simulated Annealing nähert sich dabei schneller einem optimalen Ergebnis an und wird deshalb bevorzugt zur Lösung des Partitionierungsprozesses eingesetzt. Die Eignung des Simulated Annealing für Partitionierungsaufgaben wurde durch Wiangtong [119] nachgewiesen.

Das in dieser Arbeit entwickelte automatische Hardware-Software Partitionierungssystem soll anhand eines in der Arbeitsgruppe neu entwickelten bildverarbeitenden Assistenzsystems zur Überwachung des Rückraums bei Autobahnfahrten evaluiert werden. Daher werden im folgenden Abschnitt einige Grundlagen, die zum besseren Verstehen des Algorithmus aus Kapitel 4 notwendig sind, besprochen.

2.2 Assistenzsysteme für KFZ

Assistenzsysteme im KFZ sollen den Fahrer eines Kraftfahrzeugs von monotonen Überwachungsaufgaben im Straßenverkehr entlasten bzw. ihm helfen komplexe Situationen leichter zu überblicken. Sie sollen dazu beitragen gefährliche Situationen zu vermeiden bzw. zu verhindern. Dafür existiert eine Vielzahl von Ansätzen zur Umfelderfassung. In dieser Arbeit werden nur die Systeme betrachtet, die auf Kamerasensoren basieren. Das besprochene Assistenzsystem soll den rückwärtigen Verkehr in einer Entfernung von $-150m$ bis $-10m$ beobachten und die Fahrzeugposition sowie deren Geschwindigkeit ermitteln. Ziel ist es, Hindernisse bzw. sich nähernde Fahrzeuge zu erkennen und den Fahrer bei eventuell auftretenden Gefahrensituationen zu warnen.

2.2.1 Umfelderfassungssysteme

Bei optischen Sensoren wird zwischen Mono- und Multikamerasystemen unterschieden [5]. In der Gruppe der Multikamerasysteme sind die Stereokamerasysteme am stärksten verbreitet. Optische Sensoren kommen dem menschlichen Auge sehr nahe. Das Bild eines Kamerasensors ist eine zweidimensionale Darstellung der erfassten Umgebung. Durch den Einsatz von Multikamerasystemen bzw. aktiven Kamerasensoren können zusätzlich Tiefeninformationen gewonnen werden. Im Folgenden werden die einzelnen Verfahren kurz erläutert.

Aktive und passive Sensoren

Eine Aufteilung in aktive und passive Sensoren wird anhand ihrer Eigenschaften vorgenommen. Aktive Systeme senden Licht aus und messen dessen Laufzeit, ähnlich der Entfernungsmessung mit einem Laser [29] oder beim Radar [114]. Passive Systeme senden kein Licht aus, sondern verarbeiten nur das aus der aufgenommenen Szene stammende Licht. Da sie keine Strahlung aussenden, können sie sich somit auch nicht gegenseitig beeinflussen. Durch ihre Passivität sind sie, im Gegensatz zu den aktiven Verfahren, weniger stabil bei der Messaufnahme, da sie keine definierten Messsignale aussenden und somit eine eindeutige Auswertung dieser Signale nicht möglich ist.

Monokulare Umfelderfassung

Bei den optischen Time of Flight-Messverfahren, wie z.B. Photonic Mixer Devices (www.pmdtec.com), dem 3-d TOF-CMOS-Sensor (www.csem.ch) sowie dem MDSI-Verfahren [89] von Siemens handelt es sich um aktive Systeme auf Basis der Laufzeitmessung. Bei ihnen wird ein moduliertes optisches Signal auf eine Szene gesendet, so dass dann die Reflexion aus dieser Szene von den Elementen einer Matrix (vorzugsweise CMOS) aufgenommen werden kann. Am Beispiel PMD heißt das: Die eigentliche Aufgabe der Echolaufzeitmessung, d.h. die Mischung des Ziellichtes mit der Sendemodulation geschieht bereits im inhärent mischenden Photodetektor PMD. Ein Tiefpass-Differenzverstärker liefert unmittelbar mit der Autokorrelationsfunktion die Echophasenlaufzeit bzw. die Abstandsinformation. Die maximale Messentfernung hängt vorrangig direkt von der Wellenlänge des modulierten Signals (Eindeutigkeitsbereich), der optischen Sendeleistung sowie den Sensoreigenschaften (Empfindlichkeit, Störsignalfestigkeit) ab. Bezüglich der Sendeleistungen ist bei potentiell möglicher Anwesenheit von Menschen die Gefährdung der Augen zu beachten, die den Lichtintensitäten Grenzen setzt, was wiederum Reichweitenbegrenzungen zur Folge hat. Die Modulationsfrequenz der Quelle bestimmt im Zusammenhang mit der Lichtgeschwindigkeit den Eindeutigkeitsbereich der Messungen (z.B. $20MHz -> 7,5m$). In Experimentalsystemen wird an diesem Problem z.B. durch den Einsatz von Pseudo-Noise-Signalen gearbeitet (siehe auch [112]). Insgesamt stellen Sensoren nach diesem Verfahren eine relativ neue Entwicklung dar, deren Ergebnisse aufmerksam zu verfolgen sind. Der erforderliche Messbereich benötigt für das optisches Time-of-Flight Verfahren eine Infrarotbeleuchtung mit einer Intensität, die das menschliche Augen schädigen könnte. Es ist zu erkennen, dass das PMD Verfahren für große Entfernungen aufgrund der benötigten Lichtintensität und der Modulationsfrequenz nur bedingt geeignet ist. Außerdem ist der Öffnungswinkel und damit der zu beobachtende Bereich wesentlich geringer als bei herkömmlichen Kamerasensoren.

Mittels passiver, monokularer, optischer Messsysteme ist es bedingt möglich qualitative Aussagen über die Entfernung eines Objektes zu treffen. Nur wenn Größe und Form der Objekte bekannt sind, gelingt es, mithilfe der Abbildungsgröße auf dem Kamerasensor, die Entfernung eines Objektes zu ermitteln [103]. Da in

Straßenszenen sehr viele unterschiedliche Fahrzeuge zu finden sind, ist hier eine genaue Ermittlung der Entfernung der Fahrzeuge auf Basis der bereitgestellten Informationen nicht möglich. Da die genaue Entfernungsmessung nur durch eine Klassifikation des Messobjektes möglich ist, werden diese Systeme meist in Umgebungen eingesetzt, die die Anzahl der Objektklassen beschränken [31]. In der Weiterentwicklung solcher Systeme wird mithilfe des optischen Flusses eine Hindernisdetektion durchgeführt, um Roboter durch einen Raum zu steuern [25].

Stereophotogrammetrische Umfelderfassung

Bei Bildverarbeitungssystemen im Kraftfahrzeug oder autonomen Fahrzeugen und Robotern liegen meist wechselnde Licht- und Umgebungsverhältnisse vor. Übliche Kameras mit CCD-Sensoren sind zur Umfelderfassung in Kraftfahrzeugen kaum geeignet, da sie *Blooming* Effekte aufweisen und meist einen zu geringen Dynamikbereich besitzen [20]. In diesem Bereich werden deshalb bevorzugt CMOS-Sensoren [80] eingesetzt. Diese sind für den Einsatz in Fahrzeugen wesentlich besser geeignet.

Die Algorithmen zur Umgebungserkennung sind sehr aufwendig, weil sich die Umgebung mit den zu erkennenden Hindernissen ständig mit der Bewegung des eigenen Fahrzeuges ändert. Diese Vielzahl von Problemen ließen bisher den Einsatz von optischen Assistenzsystemen nur bedingt zu.

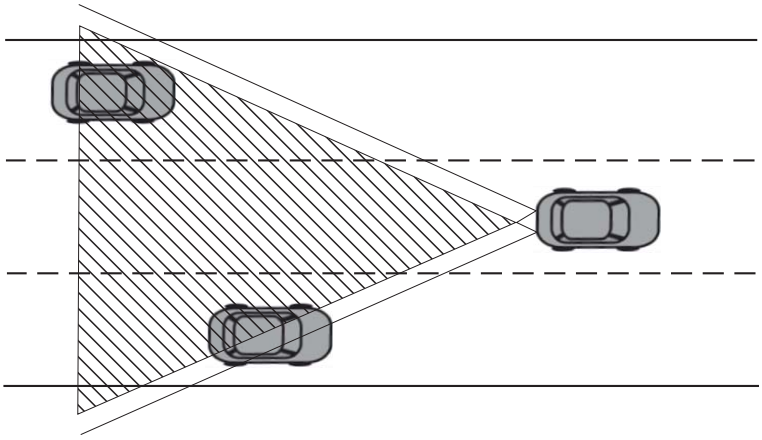

Abbildung 2.8: Detektionsbereich der Kameras

Trigonometrie : Ein Stereokamerasystem bildet grob das menschliche Sehsystem

nach und kann eine Tiefeninformation im überlappenden Sichtbereich der beiden Kameras ausgeben. Hierzu werden zwei Kameras im Fahrzeug angebracht (Abbildung 2.8). Der Detektionsbereich hängt dabei von der Kamerakonstante c und der Basisweite B zwischen beiden Kameras ab. Die Kamerakonstante wird aus der der Pixelgröße und Objektivbrennweite f berechnet. Für den geforderten Detektionsbereich von $-150m$ bis $-10m$ müssen die Kameras relativ weit auseinander angebracht werden. Im vorliegenden Fall beträgt $B = 610mm$.

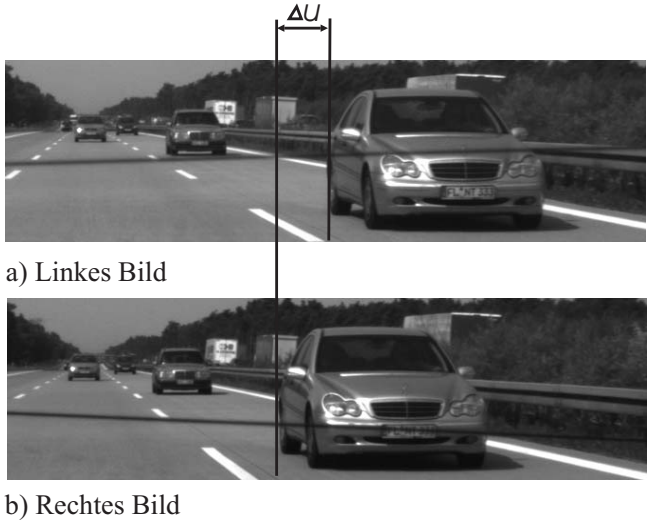

Abbildung 2.9: Disparität im Stereobild

Aus den aufgenommenen Kamerabildern muss nun eine Tiefeninformation extrahiert werden. Sie wird aus der perspektivischen Auswertung der Stereobilder berechnet. Betrachtet man jedes einzelne Bild der Stereokameraanordnung, so ist feststellbar, dass sich ein Objekt im rechten Kamerabild optisch an einer anderen Position befindet als im linken Bild (siehe Abbildung 2.9). Diese Verschiebung wird Disparität Δu genannt. Aus der Disparität und der Position des Objektes in einem Kamerabild lässt sich die Position des Objektes relativ zu einem Koordinatenursprung ermitteln.

Normalfall der Stereophotogrammetrie : Richtet man die Kameras im Stereonormalfall [5] aus, d.h. die Kameras sind exakt parallel zueinander, lässt sich mit einfachen trigonometrischen Gleichungen die reale Position eines 3-d-Punktes berechnen. Der Abstand Z zum Raumpunkt P errechnet sich im Stereonormalfall

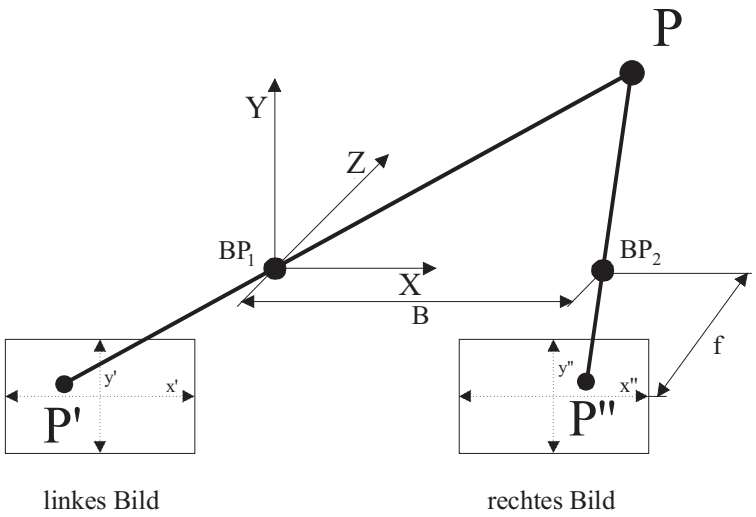

Abbildung 2.10: Lochkameramodell

auf Basis des Lochkameramodells (Abbildung 2.10) nach [63] zu:

$$Z = -\frac{c \cdot B}{\Delta u} \tag{2.10}$$

Dabei ist c die Kamerakonstante, B wird als Basisweite bezeichnet und gibt den Abstand der Projektionszentren BP_1 und BP_2 an.

$$X = \frac{x' \cdot B}{\Delta u} = \frac{x'' \cdot B}{\Delta u} - B \tag{2.11}$$

$$Y = \frac{y' \cdot B}{\Delta u} = \frac{y'' \cdot B}{\Delta u} \tag{2.12}$$

2.2.2 Diskussion

Kamerasensoren können aktive oder passive Sensoren sein. Passive Sensoren beeinflussen sich nicht gegenseitig, da sie keine Strahlung abgeben. Sie haben aber den Nachteil, weniger stabile Messergebnisse zu liefern, da sie keine definierten Messsignale aussenden und somit eine eindeutige Auswertung dieser Signale nicht möglich ist.

Passive monokulare Kamerasensoren können eine Szene zwar aufnehmen, aber aus dem Bild lässt sich nur mit entsprechendem a-priori Wissen eine Tiefeninformation gewinnen. Dafür müssen die Algorithmen der Bildauswertung sehr aufwendig sein. Insbesondere wenn sich der Kameraträger bewegt und keine festen Umgebungs- und Größenverhältnisse der zu detektierenden Objekte existieren. Hier eignen sich stereophotogrammetrisch arbeitende Sensorsysteme besser, weil mit ihnen die Position eines Objektes direkt aus dessen Verschiebung im linken und rechten Kamerabild (Disparität) abgeleitet werden kann.

Alle benötigten Eigenschaften für ein Assistenzsystem im Kraftfahrzeug in einem Sensor zu vereinen ist sehr schwierig, daher werden Assistenzsysteme zukünftig mit verschiedenen Sensoren und Sensorprinzipien arbeiten, deren Messdaten fusioniert werden. Dazu zählen die hier vorgestellten aktiven und passiven Kamerasensoren angeordnet in Mono- oder Multikamerasystemen, aber auch Radar- [114] oder Lasersysteme [29]. Im CarSense Projekt [69] wird ein Beispiel für ein solches komplexes System dargestellt.

In dieser Arbeit wird ein System mit passiven optischen Kamerasensoren, angeordnet im Normalfall der Stereophotogrammetrie, vorgestellt, weil die prinzipielle Funktionsweise eines bildverarbeitenden Assistenzsystems hiermit darstellbar ist. Sensoren, wie z.B. Radar, würden die Zuverlässigkeit des Systems weiter verbessern.

Die verwendeten CMOS-Grauwertkameras besitzen einen Megapixelsensor mit $1024x1024$ Bildpunkten. Dabei wurde eine *Region of Interest* (ROI) von $1024x500$ Bildpunkten eingestellt. Die Kameras liefern eine Bildfrequenz von $25Hz$. Die Grauwertauflösung beträgt $10bit$ und der Dynamikbereich liegt bei $120dB$. Durch die verwendeten Objektive besitzen sie eine Kamerakonstante von $c = 2358$. Die Kameras sind mit einem Basisweitenabstand von $610mm$ installiert, um den geforderten Entfernungsbereich von $-150m$ bis $-10m$ abdecken zu können.

Kapitel 3

Automatisierte Hardware-Software Partitionierung

Für automatische Hardware-Software Co-Designsysteme wurden bereits verschiedene Ansätze entwickelt (siehe Kapitel 2.1.5). Sie sind jedoch nicht für das vorliegende Problem der automatischen Partitionierung eines Bildverarbeitungsalgorithmus geeignet. Das Polis-System dient der Co-Simulation mehrerer Prozessoren und einer Anzahl von ASICs. Eine Partitionierung muss durch den Entwickler erfolgen. COSYMA und VULCAN beschäftigen sich vorwiegend mit der Partitionierung. Hier ist das zugrunde liegende System jedoch nur eine einfache Prozessor-Co-Prozessor Architektur. COSYMA wurde für den Bereich der Kommunikationstechnik entwickelt. Die dortigen Algorithmen haben aber andere Eigenschaften als die in der Steuerungstechnik oder der Bildverarbeitung. Deshalb müssen für die einzelnen Anwendungsbereiche unterschiedliche Partitionierungssysteme vorhanden sein. Durch die fortschreitende Integration von Bildverarbeitungsanwendungen in Kraftfahrzeugen werden eingebettete Systeme stärker nachgefragt. Aufgrund des Umfangs solcher Systeme wird auch hier ein Bedarf nach einer automatisierten Designumgebung für Hardware-Software Systeme geweckt. Neuere Entwicklungen bei den FPGAs ermöglichen nun den Entwurf von komplexen Single-Chip Lösungen (SoPC-System on Programable Chip) als kombiniertes Multiprozessorsystem mit Logikelementen als Hardwarebeschleuniger.

Die Struktur der Hardware steht, im Gegensatz zu den bekannten Systemen, nicht von vornherein fest, sondern wird erst abhängig vom Algorithmus und den vorhandenen Ressourcen im FPGA erzeugt. Dadurch kann eine sehr effektive Implementierung des Algorithmus entstehen.

In der Bildverarbeitung wird sehr häufig mit C/C++ oder Matlab als Beschreibungssprache gearbeitet. Deshalb soll hier die Beschreibung des Algorithmus ebenfalls mit C/C++ stattfinden. Vorteile einer C/C++ Implementierung sind die einfache Validierung des Algorithmus sowie eine sehr gute Analyse von Verarbeitungszeiten und Aufrufhäufigkeiten anhand von Beispielbildern auf dem PC.

Eine optimale Portierung des Algorithmus von einer sequentiellen Sprache, wie C/C++, in eine parallele Hardwarebeschreibungssprache, wie VHDL, ist aufgrund der unterschiedlichen MoCs (*Model of Computation*) nicht trivial und stellt einen eigenen Forschungsbereich dar [98], [115]. Dieses Thema ist daher nicht Gegenstand der vorliegenden Arbeit. Ziel ist es, auf Basis des auf einem PC in C/C++ geschriebenen Bildverarbeitungsalgorithmus eine Partitionierung in Hardwareelemente und in Softwareelemente vorzunehmen, die dann in einem FPGA realisiert werden kann. Die Partitionierung erfolgt auf Basis der vorgegebenen Randbedingungen des Systems. Für die Portierung der Softwareelemente, die in Hardware realisiert werden sollen, werden entsprechende Hardwaremodule bereitgestellt, die eine identische Funktionalität besitzen. Die Hardwaremodule sollen in einer Bibliothek gesammelt werden. Softwaremodule, die für eine Hardwareimplementierung vorgesehen sind und für die keine komplementären Hardwaremodule bereitstehen, müssen manuell portiert werden.

Die automatische Partitionierung eines C/C++ Algorithmus in ein Multiprozessorsystem mit Logikelementen auf einem Chip stellt einen neuartigen Ansatz für die Entwicklung eingebetteter Bildverarbeitungssysteme dar und soll in diesem Kapitel erläutert werden.

Im Folgenden werden zunächst die generalisierten Eigenschaften eines Bildverarbeitungsalgorithmus vorgestellt, die dann mit Hilfe des entwickelten automatischen Hardware-Software Co-Designs, basierend auf den Nebenbedingungen des Systems und den Eigenschaften der Systemhardware, zu einer effektiven Imple-

mentierung des Bildverarbeitungsalgorithmus führen.

3.1 Bildverarbeitungsalgorithmen

Die hier besprochenen Algorithmen der Bildverarbeitung können abstrahiert nach Nölle [83] (Abbildung 3.1) in ein Schichtenmodell gegliedert werden. Wesentliche Schritte der Bildverarbeitung sind die Bildgewinnung, die Vorverarbeitung und die Bilderkennung bzw. Auswertung. Die Signalquelle muss nicht zwangsläufig eine Kamera sein, es können auch andere Sensoren wie Radar oder Infrarot genutzt werden. Zwischen den einzelnen Daten, die durch die Sensoren gewonnen werden, treten im Laufe der Verarbeitungskette verschiedene Abhängigkeiten auf, die im folgenden Abschnitt dargestellt werden.

3.1.1 Schichtenmodell für Bildverarbeitungsalgorithmen

Die Datenverarbeitungsstruktur des einfachen Schichtenmodells besitzt keine Verzweigungen oder Schleifen. Bei komplexen Bildverarbeitungsalgorithmen ist dies so nicht mehr gegeben. Hier werden teilweise Iterationen durchgeführt, um die Ergebnisse des Algorithmus zu verbessern. Algorithmen dieser Art werden in dieser Arbeit nicht weiter betrachtet.

Das Schichtenmodell besteht aus folgenden Komponenten:

- Bildaufnahme
- Bildvorverarbeitung
- Merkmalsextraktion
- Interpretation und Klassifikation

Dabei sollen, im Sinne der Allgemeinheit, die Schichtenbezeichnungen großzügig interpretiert werden.

Das Schichtenmodell aus Abbildung 3.1 stellt den Datenfluss in einem Stereokamerasystem dar. Hierbei wird die Vorverarbeitung für jede Kamera parallel

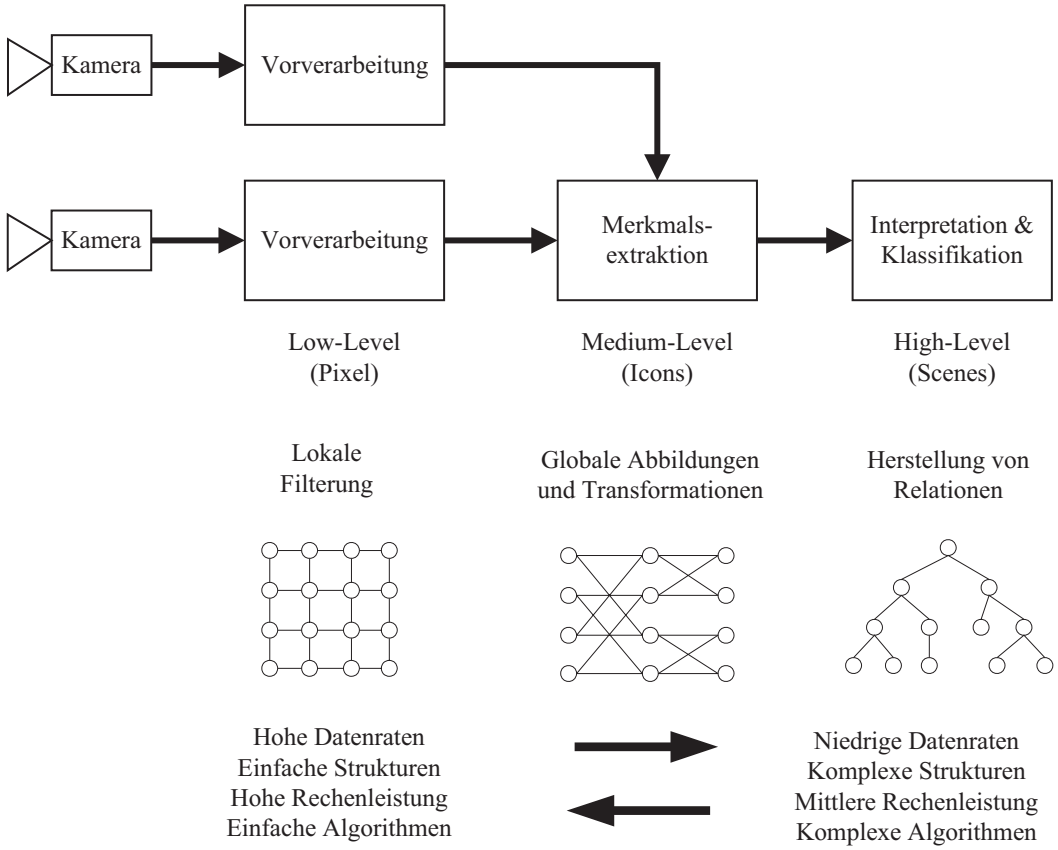

Abbildung 3.1: Schichtenmodell Bildverarbeitung für ein Stereokamerasystem nach Nölle [83]

durchgeführt. Die Datenströme werden dann während der Merkmalsextraktion zusammengeführt, um z.B. beide Bilder zu korrelieren.

Die **Bildaufnahme** erfolgt tendenziell mit digitalen Kameras, die die aufzunehmende Szene beobachten. Für bewegte Szenen sind meist mindestens 25 $Bilder/s$ zur Analyse nötig, was einen entsprechend großen Datenstrom erzeugt.

In der **Bildvorverarbeitung** werden hauptsächlich lokale Operatoren genutzt, die jeweils nur auf kleine Bildbereiche angewendet werden. Die Algorithmen besitzen eher einfache Strukturen, müssen aber hohe Datenraten verarbeiten. Dies erfordert eine entsprechend hohe Rechenleistung. Die Graphenstruktur der lokalen Algorithmen entspricht der eines d-dimensionalen Gitters oder eines d-dimensionalen Torus [108], da sie parallel auf die Daten des gesamten Bildes

oder großer Teile davon angewendet werden können. Als Rechnerstruktur ist eine SIMD Struktur bzw. entsprechende Logik gut geeignet, weil die lokalen Operatoren hier zwar die gleiche Funktion ausführen, dies aber mit verschiedenen Datensätzen tun.

Die übergeordnete Ebene ist die **Merkmalsextraktion**. Hier werden komplexere Operationen durchgeführt, die nicht mehr lokal die Daten eines bestimmten Bereichs benötigen, sondern global arbeiten und deshalb auch die Daten des kompletten Bildes bzw. die Bilder mehrerer Kameras nutzen. Dabei können Daten von räumlich getrennten Bildbereichen miteinander verarbeitet werden. Die Graphenstruktur der globalen Algorithmen kann dem eines Butterfly-Graphen entsprechen. Hier können auch noch SIMD Strukturen zum Einsatz kommen, aber auch MIMD Strukturen müssen teilweise eingesetzt werden.

Die **Interpretation und Klassifikation** stellt die obere Ebene des Schichtenmodells dar. Hier können auch Daten aus mehreren Bildern verarbeitet und in Korrelation zueinander gebracht werden. Die verwendeten Algorithmen sind sehr komplex, wobei die Datenraten eher gering sind. Paralleles Abarbeiten der Daten ist nur bedingt möglich, weil viele Datenabhängigkeiten existieren. Eine Baumstruktur spiegelt hier geeignet die Datenzusammenhänge wieder. Es können in einem Zweig völlig andere Datenströme entstehen, als in einem anderen Zweig. Sind die Zweige nicht miteinander verknüpft, können die Daten parallel verarbeitet werden. Innerhalb eines Zweiges ist es ebenfalls möglich Pipelinestrukturen in der Hardware aufzubauen. Je nach Struktur des Algorithmus und der Tiefe des Baumes ist ein MIMD (Multiprozessor)- oder ein SISD (Single Prozessor)-System die geeignete Verarbeitungsvariante.

3.1.2 Bildverarbeitungsschichten

In den folgenden Abschnitten werden die einzelnen Schichten eines Bildverarbeitungsalgorithmus noch einmal genauer untersucht, um Rückschlüsse auf die benötigte Hardwarearchitektur zu ziehen.

Bildaufnahme

Die Kamera nimmt das analoge Umgebungsbild auf und digitalisiert es. Dabei wird die Bestrahlungsstärke eines meist rechteckigen Bildbereiches gemittelt und quantisiert [50]. Die gemittelte Bestrahlungsstärke repräsentiert nach der Digitalisierung den Bildbereich als diskreten Punkt in der Bildmatrix. Dieser Punkt wird Pixel genannt und ist mit einem Grauwert GW belegt. Ein digitales Bild besitzt je nach Auflösung eine unterschiedliche Anzahl von Pixeln mit unterschiedlichen Grauwerten. Im Ortsbereich kann ein Bild nach Gleichung 3.1 beschrieben werden.

$$\begin{aligned} GW &= \{0, 1, ..., 255 - 1023\} \\ S &= [s_{x,y,n}] \\ s(x,y) &= (g_0, ..., g_n)^T \\ g_n &\in G \end{aligned} \tag{3.1}$$

Dabei ist GW die Grauwertemenge, die nach der Quantisierung einen Grauwert annehmen kann. Häufig sind das $8-10$ bit Grauwerte und demzufolge $256-1024$ Grauwertstufen. Tendenzen zu einer 12 bit Auflösung sind zu erkennen. S ist die Bildmatrix eines n-kanaligen Bildes (Ein Farbbild hat $n = 3$ Dimensionen für rot, grün und blau). s ist ein Pixel an der Position (x, y) repräsentiert durch seine Maßzahlen g. Die Auflösung des Bildes ist mit H für die horizontale Richtung und mit V für die vertikale Richtung definiert. Damit ergibt sich für $x = 0, 1...H - 1$ und für $y = 0, 1....V - 1$. Für ein Zeitreihenbild (Szene) wird die Bildmatrix um eine Zeitachse mit den diskreten Zeitpunkten $t = 0, 1, ..., T - 1$ ergänzt zu:

$$S = [s_{x,y,n,t}] \tag{3.2}$$

Eine Bildmatrix repräsentiert eine große Anzahl an Daten. So umfasst ein Bild mit $H = V = 1024$ und 8 bit Grauwerten ein Datenvolumen von einem $MByte$. Wird aus dem Einzelbild ein Zeitreihenbild mit 25 $Bildern/s$ (entspricht $40ms$ Aufnahmezeit), so erreicht der erzeugte Datenstrom bereits ein Volumen von

25 $MByte/s$. Dieser Datenstrom wird in ein Bildverarbeitungssystem eingelesen und dort bearbeitet.

Vorverarbeitung

Nach dem Schichtenmodell aus Abbildung 3.1 wird zunächst eine Vorverarbeitung eingeleitet. Dabei werden lokal Pixelwerte verändert. Bei Punktoperatoren hängt der neu zugewiesene Grauwert eines Pixels nur von seinem ursprünglichen Grauwert und der Übertragungsfunktion ($F_{x,y}$ bzw. $T_{x,y}$) ab. Dabei werden entweder der Grauwert des Bildpunktes

$$s'_{x,y} = F_{x,y}(s_{x,y}) \tag{3.3}$$

oder die Position des Bildpunktes $P_{x,y}$

$$P'_{x,y} = T_{x,y}(P_{x,y}) \tag{3.4}$$

verändert. Beispiele für Punktoperationen sind Schwellwertoperationen oder Negativbildung. Für komplexere mathematische Beziehungen bieten sich Look-up Tabellen an, weil hier die Faktoren bereits berechnet sind. Logarithmus- oder Wurzeloperationen sind typische Vertreter (Gleichung 3.5). Dabei wird der Eingangsgrauwert $s_{x,y}$ durch das entsprechende Tabellenelement ersetzt.

$$s'_{x,y} = ln(s_{x,y}) \tag{3.5}$$

Wie aus den Gleichungen 3.3 und 3.4 zu erkennen, hängt die Anzahl der Operationen je Bild von der Anzahl der Elemente in der Bildmatrix ($H \cdot V$) ab, weil die Operation für jedes eingelesene Pixel durchgeführt wird. Dadurch ergibt sich eine Verarbeitungsfrequenz f_{calc} die dem Pixeltakt f_{Pixel} entspricht (Gleichung 3.6).

$$f_{calc} = f_{Pixel} = \frac{H \cdot V}{t_{Auslese}} \tag{3.6}$$

$t_{Auslese}$ entspricht der Zeit, die benötigt wird, um den Sensor einmal komplett auszulesen. Durch die einfachen Rechenoperationen, die mit $F_{x,y}$ und $T_{x,y}$ repräsentiert werden, und die hohe Verarbeitungsfrequenz ist hier eine Implementierung in Hardware sinnvoll.

Lokale Operatoren sind eine Klasse von Operatoren, die die Grauwerte mehrerer Pixel eines Bildes (Fensterausschnitt mit einer nxn Ausdehnung ($n \in N$)) für eine Transformation benötigen. Lokale Operatoren können dabei Teil der Vorverarbeitung sein, wie Hochpass- und Tiefpassfilter (z.B. Gaußfilter), oder sie sind bereits Teil der Merkmalsextration wie Gradientenfilter (z.B. Sobel- oder Prewittoperator), welche eine Unterklasse der Hochpassfilter sind [109].

Symmetrisch aufgebaute Filter besitzen eine reelle Transferfunktion, so dass die enthaltenen Frequenzanteile nur skaliert und nicht verschoben werden [64]. Sie werden daher bevorzugt eingesetzt. Eine 2-dimensionale Filtermaske ergibt sich wie folgt:

$$h_{-m,n} = h_{m,n}, h_{m,-n} = h_{m,n} \tag{3.7}$$

In Gleichung 3.8 ist die Formel eines 2-d-Binomialfilters dargestellt.

$$s'_{x,y} = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{3.8}$$

Der Faktor 1/16 skaliert die Faltungssumme.

Die Filteroperation wird $H \cdot V - (2H - 2V + 4)$ mal aufgerufen, wenn der Filter über das komplette Bild läuft. Dabei müssen für eine nxn Maske auch nxn Multiplikationen und Additionen durchgeführt werden. Zusätzlich wird noch einmal dividiert.

Genügt für jede Rechenoperation eine Zeiteinheit, so ergibt sich die Verarbeitungsfrequenz zu:

$$f_{calc} = \frac{(HV - 2H + 2V - 4) \cdot (2nxn + 1)}{t_{Auslese}} \tag{3.9}$$

Das erfordert für einen sequentiell arbeitenden Prozessor eine sehr hohe Rechenfrequenz. In einer Hardwarelösung ist es möglich, verschiedene Verarbeitungsschritte zu parallelisieren und so die nötige Verarbeitungsgeschwindigkeit zu reduzieren (siehe auch Abbildung 2.2). So können die einzelnen Multiplikationen parallel ausgeführt werden, während die Addition in einer Pipelinestruktur dahinter liegt. Am Ende der Pipeline wird die Division durchgeführt. Mit der Aufteilung in ein paralleles Pipelinekonzept lässt sich die Verarbeitungsgeschwindigkeit wieder auf den Pixeltakt reduzieren.

$$f_{calc} = f_{Pixel} = \frac{H \cdot V}{t_{Auslese}} \tag{3.10}$$

Auch hier ist der Verarbeitungsalgorithmus sehr einfach und linear, so dass eine Hardwarelösung bei entsprechend großer Reduzierung der nötigen Verarbeitungsgeschwindigkeit genutzt werden kann.

Merkmalsextraktion

Bei der Merkmalsextraktion werden aus den vorverarbeiteten und geglätteten Bildern einer oder mehrerer Kameras die gesuchten Merkmale (Kanten, geometrische Formen, Entfernungswerte usw.) ermittelt. Für die verwendeten lokalen Operatoren gelten die zuvor beschriebenen Eigenschaften. Zu den globalen Operatoren gehören die Fouriertransformation, Texturanalyse und der optische Fluss. Dabei werden die Daten des gesamten Bildes oder mehrerer Bilder benötigt, um ein Ergebnis zu erzeugen. Diese Anwendungen sind wesentlich komplexer, weil auch hier eine große Anzahl an Daten verarbeitet wird.

Die $Kreuzkorrelationsfunktion$, als eine Funktion der Merkmalsextraktion, ermittelt die Ähnlichkeit zweier Bildbereiche und ist zunächst eine gewöhnliche

Gleichung, die gelöst wird. Damit ist sie datenflussorientiert und somit sehr gut als Hardwarefunktion realisierbar [111]. Eine eingehendere Beschreibung der Kreuzkorrelationsfunktion befindet sich in Abschnitt 4.1.1.

Die Verarbeitungsgeschwindigkeit lässt sich wieder durch ein paralleles Pipelining reduzieren. Hierbei können, in Abhängigkeit verschiedener Korrelationsaufgaben, komplexere Steuerungselemente als bei einem lokalen Operator auftreten. Diese Steuerung ist aufgrund ihrer geforderten Flexibilität wesentlich aufwendiger mit einer unflexiblen Hardware zu realisieren. Deshalb kann hier eine Prozessor-Logikarchitektur sinnvoll sein.

Die $Fouriertransformation$ [109] ist wie die Kreuzkorrelationsfunktion ein Verfahren zur Merkmalsextraktion und ist sowohl in Hardware als auch in Software zu realisieren. Eine Hardwarelösung ist in [107] vorgestellt worden. Die Fouriertransformation ist sehr gut in ihre Einzelteile separierbar und deshalb gut parallel zu implementieren.

Bei der $Texturanalyse$, als weiteren Teil der Merkmalsextraktion, wird die Textur einer Fläche im Bild analysiert. Je nach Textur müssen zur Analyse verschiedene Operatoren aus unterschiedlichen Schichten der Bildverarbeitung angewendet werden. Texturoperatoren sind in Klassen zusammengefasst. Mittelwert und Varianz sind rotations- und größeninvariant und gehören in eine Klasse. Eine andere Klasse kann mit der Orientierung und Größe besetzt werden, da diese rotations- und größenvariant sind. In Abhängigkeit der verschiedenen Operatoren kann eine Zuordnung zu den vorher festgelegten Texturen erfolgen. Für die einzelnen Operatoren kommen Algorithmen zum Einsatz, die jeder für sich einfache Rechnungen ausführen. Zusammengenommen kann sich hier eine Komplexität ergeben, die nicht mehr sinnvoll mit einer Hardwarelösung realisierbar ist. Da die Texturanalyse aber bereits auf einer Vorverarbeitung beruht und eine gewisse Datenreduktion vorgenommen wurde, verringert sich die benötigte Rechenfrequenz, was wiederum für eine mögliche Prozessorimplementierung genügen würde.

Interpretation und Klassifikation

Die bereits beschriebenen Algorithmen dienen der Bildaufbereitung bzw. der Extraktion der gesuchten Merkmale. Die Interpretation der gefundenen Merkmale und die anschließende Klassifikation erfolgt in dieser Schicht. Die Interpretation und Klassifikation beinhaltet komplexe Algorithmen mit häufigen Verzweigungen und Schleifen. Weil hier nicht mehr das gesamte Bild, sondern nur noch die in der Merkmalsextraktion gefundenen Regionen untersucht werden, reduziert sich die Zahl der Eingangsdaten sowie die Anzahl der Algorithmenaufrufe erheblich. Dafür ist jeder Algorithmus sehr komplex aufgebaut, weil er mit den extrahierten Daten eine sinnvolle Klassifikation durchführen muss. Beispiele solcher Algorithmen sind die Clusterung, die Hough-Transformation oder die Verfolgung der Objekte über mehrere Bilder hinweg. Bei der Clusterung müssen, zum Beispiel anhand der Merkmale wie Größe und Form, die verschiedenen Regionen zu einem Objekt zusammengeführt und Objektklassen zugeordnet werden. Da die Algorithmen teilweise auf die gleichen Daten der Merkmalsextraktion zurückgreifen, ist hier eine parallele Datenverarbeitung auf mehreren Prozessoren möglich. Die Algorithmen sind eher kontrollflussorientiert und deshalb gut für eine Softwareimplementierung geeignet.

HW-SW Co-Design Modell

Abgeleitet vom Schichtenmodell lassen sich verschiedene Ebenen der Partitionierung erkennen. Wie bereits erwähnt, sind lokale Operatoren einfache Algorithmen, die eine große Datenmenge verarbeiten müssen. Da sie lokal arbeiten, reicht hier meist schon ein lokaler Zwischenspeicher aus. Es können auch mehrere Operatoren auf einen Zwischenspeicher zugreifen (z.B. Zwischenspeicher einer Zeile). Somit entsteht eine *parallele Pipelinestruktur*. Durch die parallele Pipelinestruktur kann die Verarbeitungsfrequenz teilweise deutlich gesenkt werden. Für globale Operatoren muss entsprechend ein globaler Zwischenspeicher vorhanden sein. Auch hier sind Parallelitäten möglich, die entsprechend ausgenutzt werden können, um die Verarbeitungsgeschwindigkeit zu steigern. Das gilt ebenfalls für den Bereich der Interpretation und Klassifikation.

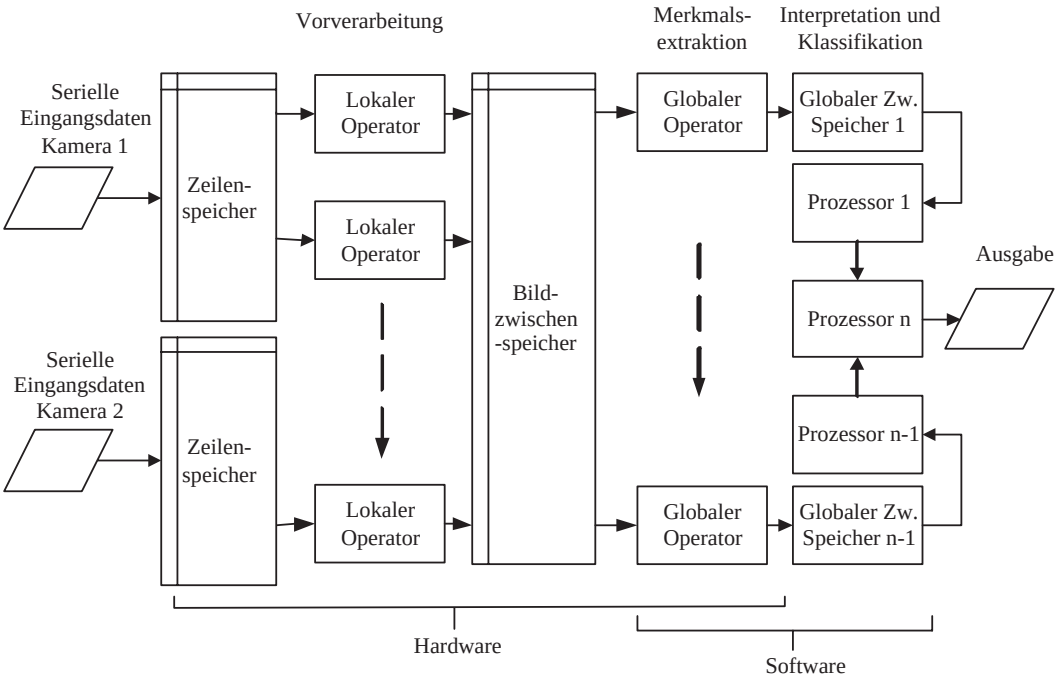

Abbildung 3.2: 2-dimensionaler Datenfluss im Schichtenmodell

Es ist also möglich, dass der Datenfluss nicht nur gerade durch alle Schichten des Modells läuft, sondern Verarbeitungsschritte parallel vorgenommen werden können. Damit kann der Datenfluss nicht nur eindimensional, sondern auch zweidimensional durch das Schichtenmodell verlaufen (Abbildung 3.2).

So teilt sich ein Bildverarbeitungsalgorithmus in verschiedene Komplexitätsgrade, deren Funktionen global nacheinander (Schichtenmodell) und lokal parallel abgearbeitet werden können. Es entsteht für den Datenflussgraphen ein paralleles Pipelining unterschiedlicher Komplexität, das auf der entsprechenden Systemhardware in Logik oder auf einem Prozessor realisiert werden muss.

Wichtigstes Entscheidungskriterium ist die Häufigkeit des Aufrufes (Auslastung) einer Funktion und deren Komplexität (Anzahl der Verzweigungen). Wird eine einfache Funktion oft genutzt, ist eine Hardwareimplementierung sinnvoll. Wird sie selten aufgerufen, reicht eine Softwarerealisierung aus. Komplex aufgebaute Funktionen sind in Hardware nur mit sehr viel Aufwand zu realisieren. Falls die Ausführungszeit dieser Funktionen auf einem Prozessor den vorgegebenen Echtzeitbedingungen nicht entspricht, kann durch eine Analyse der inneren Parallelität

des Algorithmus eine Aufteilung auf ein paralleles Rechnersystem vorgenommen werden.

3.2 Systemhardware

Der Bildverarbeitungsalgorithmus muss auf der entsprechenden Systemhardware, auf den verwendeten Prozessoren oder der zur Verfügung stehenden Logik implementiert werden, um als eingebettetes System seine Anwendung zu finden. Daher soll im Folgenden die Systemhardware, auf der das vorgestellte Hardware-Software Co-Design beruht, vorgestellt werden.

System

Das als Anwendungsbeispiel gewählte System besteht aus einem Altera Stratix FPGA mit 60.000 Logikzellen [8]. In das FPGA können nicht nur Logikelemente, sondern auch NIOS-Softcore-Prozessoren integriert werden. Ein NIOS II Softcore-Prozessor benötigt etwa 3.500 der vorhandenen Logikzellen, d.h. es können mehrere NIOS II Prozessoren mit einer jeweiligen Taktfrequenz von bis zu $60MHz$ implementiert werden. Dadurch werden verschiedene Teile des Algorithmus parallel zueinander abgearbeitet. Mithilfe des aufteilbaren internen Speichers von $600kByte$ können die Prozessoren auch unabhängig voneinander Daten und Programme verarbeiten.

Neben der Implementierung mehrerer Prozessoren und der daraus folgenden Implementierung eines MIMD-Systems ist es möglich, spezifische Funktionen unterschiedlicher Komplexität einzubinden. Zum einen wären das die bereits erwähnten Logikelemente, die unterschiedlich stark abhängig von den Prozessoren sein können. Weiterhin ist es möglich den Befehlssatz des Prozessors so zu erweitern, dass für einen bestimmten komplexen Befehl, statt einer Softwareroutine, ein Hardwarebaustein in die ALU des entsprechenden Prozessors eingebunden wird. In [36] wird für eine FFT ein Geschwindigkeitsgewinn um den Faktor 27 mit einem kundenspezifischen Befehlssatz und um den Faktor 530 mit einem externen

Logikelement mit DMA-Zugriff gegenüber einer Softwarelösung angegeben.

Durch die Nutzung eines FPGAs können die verschiedenen Algorithmen flexibel implementiert werden. Es ist auch möglich die programmierte Hardware während der Laufzeit zu wechseln oder upzudaten.

3.2.1 Hardwaremodell

Für die Partitionierung und die Erstellung des Hardwaremodells wird angenommen, dass keine Feedbackschleifen zwischen den einzelnen Schichten des Schichtenmodells der Bildverarbeitung (siehe Abbildung 3.1) vorhanden sind.

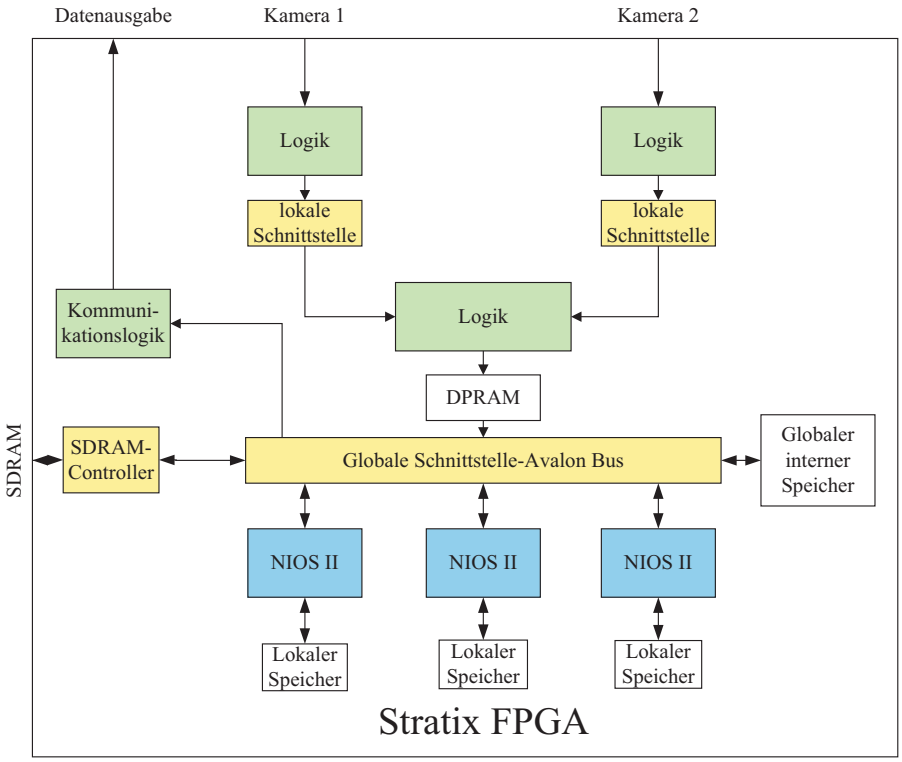

Abbildung 3.3: Hardware des Modulsystems

Demzufolge wird in Abbildung 3.3 ein Hardwaremodell mit Pipeline entworfen, bei dem einfache datenflussorientierte Algorithmen mithilfe von Logik in Hardware realisiert werden können. Dazu gehören die Bildvorverarbeitung und Teile der Merkmalsextraktion. Komplexe kontrollflussorientierte Algorithmen, wie die

Interpretation und Klassifikation aber auch Teile der Merkmalsextraktion können auf Prozessoren verwirklicht werden.

Prozessoren (NIOS II) und Logikelemente werden durch einen gemeinsamen Bus miteinander verbunden. Die Prozessoren sind dabei die Busmaster, während die Logikelemente als Busmaster oder Busslave eingesetzt werden können.

Das System sollte als Multirechnersystem mit lokalem Programm- und Datenspeicher aufgebaut werden, um unnötigen Kommunikationsaufwand auf dem gemeinsamen globalen Bus zu vermeiden. Die Logikelemente werden untereinander mit Hilfe einer lokalen Schnittstelle verbunden.

3.2.2 Globale Schnittstelle Avalon Bus

Als globaler Bus wird der von Altera entwickelte Avalon Bus [7] vorgeschlagen, der mithilfe des SoPC-Builders [122] generiert wird. Mit ihm können alle Prozessoren, Logikelemente und Speicher miteinander verbunden werden.

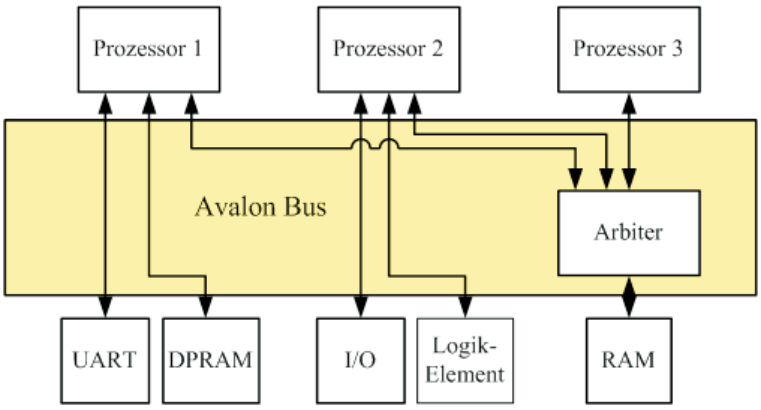

Abbildung 3.4: Avalon Multi Master Bus

Die Avalon Busarchitektur unterstützt einen Multimasterbetrieb. Dabei können mehrere Busmaster gleichzeitig auf verschiedene Buselemente zugreifen. Nur bei gleichzeitigem Zugriff zweier Busmaster auf einen Busslave entscheidet ein Arbiter über die Freigabe (Abbildung 3.4). Die Arbitrierung wird *Slave – Side – Arbitration* genannt, da der Arbiter nur an der Stelle eingesetzt wird, wo ein Slave von zwei oder mehreren Mastern genutzt wird. Der Vorteil gegenüber traditio-

nellen Busarchitekturen ist, dass das System keine gemeinsamen Busleitungen besitzt. Stattdessen wird für jedes einzelne Master-Slave-Paar eine direkte Punkt zu Punkt Verbindungen, wie in einem Netz, generiert. Der Begriff Bus ist also nicht ganz korrekt verwendet worden. Da er aber von Altera so vorgeschlagen wurde, soll er auch weiterhin genutzt werden. Sind mehrere Slaves mit einem Master verbunden, wird die jeweilige Verbindung mit Hilfe von Multiplexern hergestellt.

Durch die Multi-Master Architektur ist es möglich, hohe Übertragungsbandbreiten und maximal unabhängiges Arbeiten der Prozessoren zu gewährleisten. Ein Avalon Bussystem mit zwei unabhängigen Master-Slave-Paaren ermöglicht einen maximal doppelten Datendurchsatz gegenüber einem System mit einem gemeinsamen Bus. Das Entsprechende gilt auch für mehrere Master. Die Erhöhung des Datendurchsatzes hängt davon ab, wie oft ein simultaner Buszugriff geschieht. Durch den netzartigen Aufbau des Busses steigt die Komplexität mit der Anzahl der angeschlossenen Elemente. Somit ist es nicht möglich, eine unbegrenzt große Anzahl von Prozessoren und Logikelementen an den Avalon Bus anzuschließen.

Alle Bussignale werden mit dem globalen Systemtakt synchronisiert, so dass ein vollständig synchrones Design entsteht, weil auch die Prozessoren und Logikelemente mit diesem Takt oder mit einem daraus abgeleiteten Takt arbeiten. Der Avalon Bus lässt sich flexibel auf die Datenwortbreite der angeschlossenen Elemente anpassen, wodurch sich Systemressourcen einsparen lassen. So sind 8-,16,- oder 32 bit breite Busse möglich. Der Bus übernimmt automatisch die Adressdekodierung für die *ChipSelects* der einzelnen Slaves, die Generierung von $Wait - States$, die *Interrupt*-Priorisierung bei Anforderungen mehrerer Slaves und die Weiterleitung der Interrupts. Des weiteren können automatisch *Latenzzeiten* eingefügt oder *Databursts* durchgeführt werden [7].

3.2.3 Lokale Schnittstelle

Die lokale Schnittstelle dient der Verbindung der Logikelemente untereinander. Schaltet man mehrere Logikelemente zusammen, entsteht eine parallele Pipelinestruktur (Abbildung 3.5). Beim Erstellen der Pipelinestruktur soll es unerheblich

sein, an welcher Stelle ein Element steht. Das bedeutet, die Eingangs- und Ausgangsschnittstelle muss bei jedem Logikelement gleich sein.

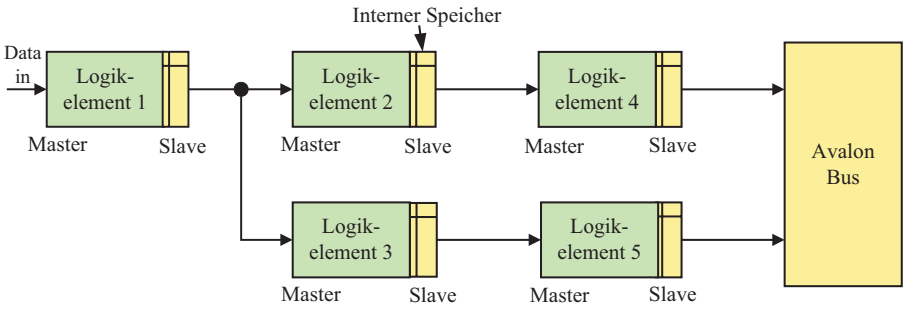

Abbildung 3.5: Lokale Datenverarbeitungsstruktur

Da auch Zwischenspeicher in Form von Registern implementiert werden müssen, erscheint es sinnvoll, die entsprechende Speicherschnittstelle samt zeitlichem Verhalten zu implementieren. Die Struktur der Logikelemente ist *datengetrieben*, das heißt ausgehend von den Kameras werden die Daten erzeugt und kontinuierlich verarbeitet. Die Logikelemente warten also auf Daten. Sobald diese vorliegen, werden sie verarbeitet. Die verarbeiteten Daten werden unverzüglich an das nächste Logikelement weitergeleitet, wobei die Leseoperation vom übernehmenden Logikelement gestartet wird. Der Eingang eines Logikelementes ist also immer ein Master, während der Ausgang immer ein Slave ist. Es besteht dabei die Möglichkeit, Daten in Speicherelementen zwischenzuspeichern. Um die analysierten Daten mit einem Prozessor (NIOS II) weiter zu verarbeiten, werden sie für das globale Netzwerk zur Verfügung gestellt. Die lokale Schnittstelle besteht aus Datenleitungen, Adressleitungen (müssen nicht verwendet werden) sowie einer Steuerleitung (*Enable*). Das Enable dient dem Master als Signal zum Auslesen der bereitgestellten Daten. Da alle Daten synchron verarbeitet werden, sind keine zusätzlichen Synchronisationsleitungen notwendig.

3.3 Entwurfsprozess

Der Entwurfsprozess für die HW-SW Partitionierung gliedert sich in zwei Teile. Zuerst wird mit Hilfe des Frameworks eine Partitionierung des zuvor entworfenen

Systems vorgenommen. Anschließend wird das Design mit der Quartus II Software implementiert. Die Quartus II Software ist eine von Altera bereitgestellte Designumgebung zum Entwurf von Logikschaltungen auf einem FPGA.

Framework

Für die Beschreibung von Hardware und Software wurden verschiedene Sprachen entwickelt. Die Beschreibung unterliegt zwei unterschiedlichen Paradigmen. Hardware wird mit nebenläufigen Konstrukten beschrieben, so dass parallele Strukturen erzeugt werden. Die Hauptvertreter dieser Sprachen sind VHDL und Verilog. Software wird mit Hilfe sequentieller Sprachen implementiert.

Die Paradigmenverschiebung zwischen parallelen und sequentiellen Beschreibungssprachen muss beim Hardware-Software Co-Design berücksichtigt werden. Zur Beschreibung des Algorithmus ist es notwendig, sich auf eine Beschreibungssprache festzulegen und Teile später in die jeweils andere Beschreibungssprache zu überführen. Das kann zu nicht optimalen Portierungen führen. Wie bereits besprochen, erfolgt die Beschreibung eines Algorithmus in der Bildverarbeitung häufig in C/C++. Daher soll auch hier auf C/C++ zurückgegriffen werden. Damit kann der komplette Algorithmus funktional am PC entwickelt und getestet werden. Danach wird dann die Analyse und die Partitionierung durchgeführt. Dabei wird, ausgehend von der Software, ein Graphenmodell des Algorithmus [79] mit den entsprechenden Modulen erstellt.

Die Softwarefunktionen der Partition mit den minimalen Kosten werden mit einem GNU Compiler für die NIOS II Prozessoren compiliert und die Hardwaremodule entsprechend synthetisiert. Die Ergebnisse werden dann auf das Zielsystem portiert (siehe Abbildung 3.6).

Für die möglichen Softwaremodule stehen in der Regel auch entsprechende Hardwaremodule in einer Bibliothek bereit. Diese sollten bereits bei der Entwicklung des Algorithmus eingesetzt werden, um eine bestmögliche und fehlerfreie Portierung zwischen Hardware und Software zu gewährleisten. Wie bereits erwähnt, ist die Portierung von Softwarecode auf eine Hardwarebeschreibungssprache auf-

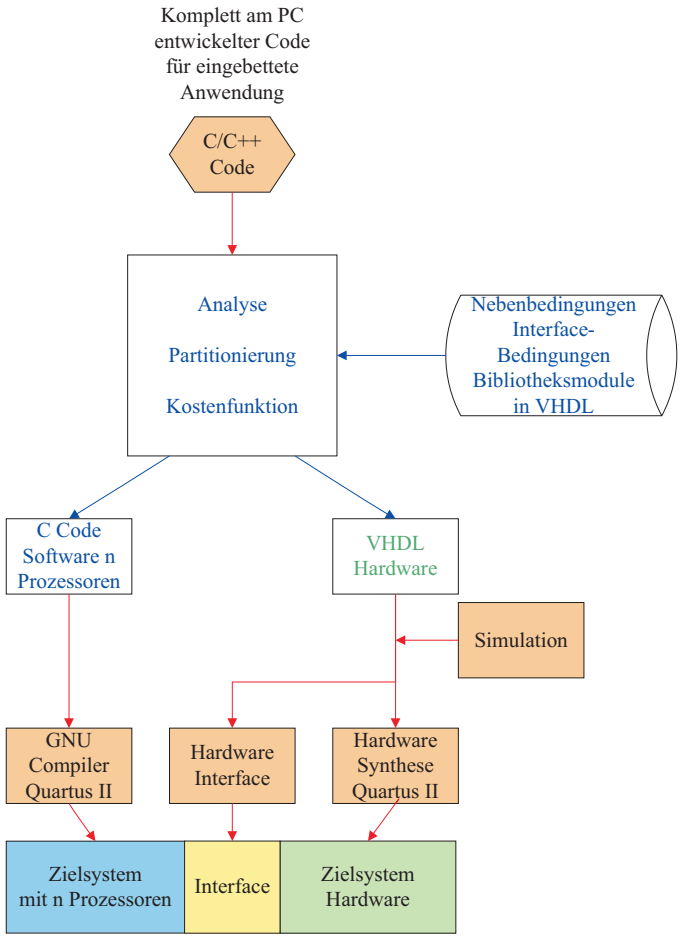

Abbildung 3.6: Entwurfsablauf

grund des Paradigmenwechsels (unterschiedliche Models of Computation) nur schwer automatisierbar und soll hier nicht betrachtet werden. Ansätze zu einer Portierung von C-Code in VHDL sind in [86] dargestellt. In dieser Arbeit werden Softwareelemente für die keine Hardwaremodule zur Verfügung stehen, die aber aufgrund des Partitionierungsprozesses für die Hardwareimplementierung vorgesehen sind, entsprechend für eine manuelle Portierung kenntlich gemacht. Die Partitionierung wird in Abschnitt 3.4 beschrieben. Alle Softwareelemente werden auf ein Multiprozessorsystem, entsprechend des Vorschlags durch das automatisierte Partitionierungssystem, aufgeteilt. Ziel des Entwurfs ist eine Partitionierung mit minimalen Kosten und eine Darstellung des Algorithmus, an der erkenntlich ist, welche Funktionen als Hardware, welche Funktionen als Software

und auf welchem Prozessor die Softwarefunktionen für eine optimale Implementierung realisiert werden sollen.

SoPC-Builder

Im zweiten Teil des Entwurfsprozesses wird die zuvor ermittelte Partitionierung auf das Zielsystem portiert. Diese Portierung wird hauptsächlich mit dem SoPC-Builder (*System on Programmable Chip*) [7] durchgeführt. Der SoPC-Builder ist ein in QUARTUS-II integriertes Tool, das den Aufbau eines eingebetteten Multiprozessorsystems in ein FPGA ermöglicht.

Mit dem SoPC-Builder werden die IP-Funktionen (*Intellectual Property*) ausgewählt (Prozessoren, Logikelemente, Kommunikationselemente, Speicher) und konfiguriert. Die IP-Funktionen werden automatisch mit dem Avalon Bus verbunden und so das komplette System erzeugt. Dabei wird die gesamte Adressdecodierung, Arbitrierung, Interrupt- und Waitstate-Steuerung, Anpassung der Datenbusbreite sowie das Clock-Domain Management generiert [26]. Alle Komponenten am globalen Datenbus können so miteinander verbunden werden. Die Komponenten mit einer lokalen Datenschnittstelle werden außerhalb des SoPC-Builders in das System implementiert. Entsprechend ihrer Möglichkeiten können sie dort konfiguriert werden.

Mithilfe der NIOS-II IDE [7] (*Integrated Design Environment*) werden die Softwareelemente für die NIOS-II-Prozessoren compiliert und der Startup-Code der Prozessoren erzeugt.

3.4 Partitionierung

Die Partitionierung ist die eigentliche Aufgabe des Hardware-Software Co-Design Frameworks. Grundlage der Partitionierung ist ein Algorithmus, der abhängig von seinen Eigenschaften auf eine heterogene Struktur aus Prozessoren und Logikelementen aufgeteilt wird. Die heterogene Struktur befindet sich auf einem oder mehreren Chips (FPGA) und wird erst zum Ende des Partitionierungsprozesses den Anforderungen entsprechend erstellt.

Der zu partitionierende Algorithmus selbst ist in Funktionen F_i aufgeteilt. Eine Funktion ist hier ein in sich geschlossenes Programmmodul, das aus einem anderen Modul heraus aufgerufen wird, an das Parameter übergeben werden können und das selbst einen Rückgabewert liefert.

Der Partitionierungsprozess kann als softwareorientierter oder als hardwareorientierter Ansatz erfolgen. Bei einem softwareorientierten Ansatz ist die Startpartition eine Softwarepartition, in der sich der gesamte Algorithmus befindet. Aus der Softwarepartition werden dann, bis zum Erreichen der Nebenbedingungen, einzelne Elemente in die Hardwarepartition überführt. Der hardwareorientierte Ansatz erfolgt wie der softwareorientierte Ansatz, nur, dass sich hier der gesamte Algorithmus in der Hardwarepartition befindet. Da im konkreten Fall der Algorithmus bereits als C/C++ Code vorliegt, wird der softwareorientierte Partitionierungsansatz weiter verfolgt.

Der Umfang des Partitionierungsprozesses hängt von der Anzahl N der Funktionen F_i des Algorithmus ab. Jede Funktion F_i kann als Hardware oder Software realisiert werden. Es existieren also für jede Funktion zwei mögliche Partitionen. Woraus sich für N Funktion insgesamt 2^N mögliche Partitionen Pa ergeben

$$K_A = Pa_N = 2^N \tag{3.11}$$

Dies entspricht auch der Aufwandskomplexität K_A. Die Aufwandskomplexität nimmt also exponentiell mit der Anzahl der Elemente zu. Das führt bei einer großen Anzahl von Elementen zu langen und aufwendigen Berechnungen. Aus diesem Grund soll hier ein heuristisches Verfahren zur Partitionierung verwen-

det werden. Somit wird zwar nicht immer das absolute Optimum gefunden, aber es kann sich diesem, mit wesentlich geringerem Aufwand, genähert werden. Wie bereits in Kapitel 2.1.4 erläutert, ist das Simulated Annealing deshalb ein geeignetes Verfahren, weil es schnell ein optimales Ergebnis findet, mit ihm lokale Minima verlassen werden können und sich dieses Verfahren auf dem Gebiet der Partitionierung bewährt hat [119].

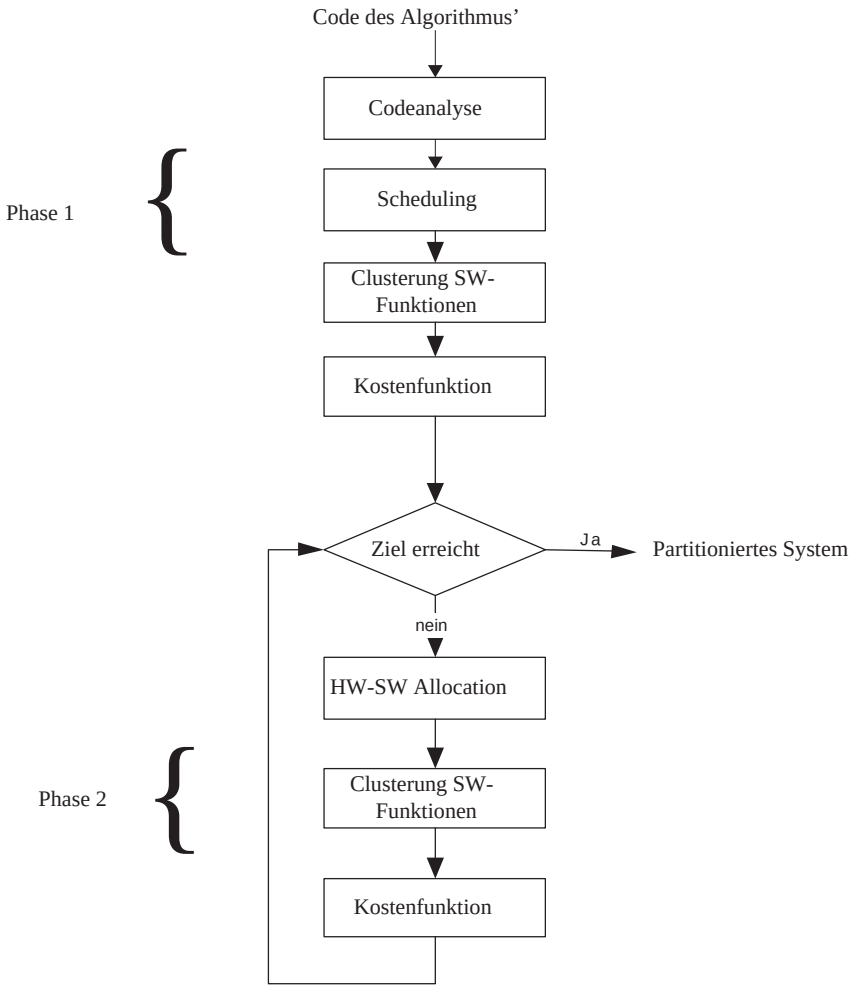

Abbildung 3.7: Verlauf der Partitionierung

Wie in Abbildung 3.7 zu sehen, wird bei der vorgestellten Partitionierung in zwei Phasen vorgegangen. Phase eins wird nur einmal durchlaufen. Hier wird überprüft, inwiefern eine Multiprozessorlösung ausreicht, um die Zielkriterien zu erreichen. Ist das nicht der Fall, wird in Phase zwei das eigentliche Simulated

Annealing in mehreren Iterationen durchlaufen, bis das Ziel der Partitionierung erreicht ist.

Die **Codeanalyse** wird nur einmal zu Beginn des Partitionierungsprozesses durchlaufen. Dabei werden die Programmausführungszeiten auf dem PC, die durchschnittlichen Ausführungshäufigkeiten und die Komplexität der einzelnen Funktion ermittelt. Weiterhin wird festgestellt, welche globalen und lokalen Variablen in den Funktionen verwendet werden und welche Variablen an die Funktionen übergeben bzw. zurückgegeben werden.

Das **Scheduling** erfolgt wie die Codeanalyse nur einmal zu Beginn des Partitionierungsprozesses. Hier wird anhand der Codeanalyse festgestellt, in welcher Reihenfolge die einzelnen Funktionen abgearbeitet werden müssen und welche Zusammenhänge sie untereinander aufweisen. Dabei werden die Funktionen nach Ausführungszeitpunkt und Ausführungszeit geordnet.

Dem Scheduling folgt die **Clusterung der SW-Funktionen**. Hier werden die Funktionen so zusammengefasst, dass sie auf verschiedene NIOS-II-Prozessoren mit minimalen Kommunikationskosten verteilt werden. Das dabei entstehende Multiprozessorsystem wird mithilfe der **Kostenfunktion** bewertet. Durch diese wird entschieden, ob eine Hardware-Software Partitionierung notwendig ist oder nicht.

Sind die gegebenen Randbedingungen nicht erfüllt, wird das Multiprozessorsystem wieder aufgelöst und eine Hardware-Software Partitionierung durchgeführt. Die **Allocation** führt die eigentliche Zuordnung der Programmteile auf die verschiedenen Logik- und Softwareelemente durch. Dafür werden auch die Logikbausteine aus einer Elementebibliothek herangezogen. Die verbleibenden Softwarefunktionen werden wieder mit der **Clusterung der SW-Funktionen** zu einem Multiprozessorsystem zusammengefasst.

Anhand einer **Kostenfunktion** wird der Implementierungsaufwand des Algorithmus für die vorliegende Partitionierung ermittelt. Die Kostenfunktion berücksichtigt Parallelitäten, Komplexitäten, Ausführungszeiten der Unterprogramme, den Bedarf an Chipfläche, die Kommunikationskosten, den Portierungsaufwand einer Softwarefunktion in Hardwarelogik und den möglichen *SpeedUp* durch die

Portierung. Dabei sollen die Kosten ein Minimum erreichen, um so eine nahezu optimale Implementierung zu ermöglichen. Liegen die Kosten der aktuellen Partition niedriger als die zuvor ermittelte kostengünstigste Partition, so wird die aktuelle Partition als die beste angesehen. Die Iterationsschleife wird nun neu durchlaufen. Dabei werden neue Partitionen generiert und deren Kosten ermittelt, bis die vorgegebenen Nebenbedingungen erfüllt sind.

3.4.1 Simulated Annealing

Bevor nun die einzelnen Schritte der Partitionierung betrachtet werden, wird das Simulated Annealing noch einmal eingehender vorgestellt.

Wie bereits zuvor erwähnt, ist die Aufwandskomplexität zur Partitionierung des Algorithmus, mit 2^N möglichen Varianten ein NP-Problem (Non-Polynomial). N steht für die Anzahl der zu partitionierenden Funktionen. Mit jeder weiteren zu partitionierenden Funktion verdoppelt sich die Anzahl der möglichen Partitionen. Jeder Partition Pa können entsprechende Partitionierungskosten C zugeordnet werden. Alle Teilkosten, die über die entsprechende Partition Pa definiert sind, werden zu C aufsummiert und nehmen Werte der Menge R an ($C \in R$). Ziel der Partitionierung ist es, die Partition mit den minimalen Kosten zu finden (Gleichung 3.12).

$$C_{opt} = minC(Pa_n) \tag{3.12}$$

Zur Lösung des NP-Problems wird das Simulated Annealing verwendet. Hiermit ist es möglich, für eine große Anzahl von Optimierungsproblemen in einer nicht exponentiellen Rechenzeit eine nahezu optimale Lösung zu finden [65].

Das Simulated Annealing basiert auf der Boltzmannverteilung (Gleichung 3.13) für das Abkühlen von flüssigem Material bis zur Erstarrung.

$$P = e^{-\frac{E}{k_B \cdot T}} \tag{3.13}$$

P ist dabei die Wahrscheinlichkeit mit der das Material bei einer bestimmten

Energie E und der anliegenden Temperatur T das thermische Gleichgewicht erreicht. k_B ist die Boltzmannkonstante.

Beim Simulated Annealing werden die Boltzmannkonstante und die Temperatur aus Gleichung 3.13 durch einen Kontrollparameter CP ersetzt. Für die Energie werden die Kosten C der Partition eingesetzt. Somit ergibt sich Gleichung 3.14.

$$P = e^{-\frac{C}{CP}} \tag{3.14}$$

Wie bereits erwähnt wird die Partitionierung nach Abbildung 3.7 durchgeführt. Nachdem Phase eins abgeschlossen ist und das Abbruchkriterium nicht erfüllt wurde, wird zu Phase zwei übergegangen. Zuerst wird der Kontrollparameter auf einen Startwert gesetzt $CP = CP_{max}$. Dann wird eine Funktion anhand ihrer Priorität (Kapitel 3.4.4) für eine Hardwarerealisierung vorgesehen. Die als Software verbleibenden Funktionen werden geclustert. Das Ergebnis ist eine neue Partition Pa_i, die sich in einer Funktion F von der vorhergehenden Partition Pa_j unterscheidet.

Für das Gesamtsystem ergeben sich daraus neue Kosten C_k. Mithilfe des $Metro-polalgorithmus$ wird entschieden, ob diese Partition angenommen wird. Beim $Metropolalgorithmus$ werden die Kosten C_k der aktuellen Partitionierung Pa_i mit den Kosten der Partitionierung Pa_j mit C_{k-1}, die zuletzt als die Partitionierung mit minimalen Kosten ermittelt wurde, verglichen (Gleichung 3.15). k stellt dabei den jeweiligen Iterationsschritt dar.

$$\Delta C = C_k - C_{k-1} \tag{3.15}$$

Ist $\Delta C \leq 0$, wird die Wahrscheinlichkeit, dass dies die Startkonfiguration für den nächsten Iterationsschritt ist auf $P = 1$ gesetzt. Ist $\Delta C > 0$, d.h. die aktuelle Partition besitzt höhere Kosten als die Partition Pa_j, wird die Funktion $P = e^{-\frac{\Delta C}{CP}}$ herangezogen und mit einer vom System erzeugten Zufallswahrscheinlichkeit Px mit $(0 \leq Px < 1)$ verglichen. Ist die Zufallswahrscheinlichkeit Px kleiner als P wird auch eine kostensteigernde Partitionierung akzeptiert. Somit ist es möglich lokale Minima zu verlassen. Die Wahrscheinlichkeit, dass $Px < P$, nimmt

mit jedem Iterationsschritt ab, weil nach einer Iteration der Kontrollparameter CP verringert wird. Damit sinkt auch die Wahrscheinlichkeit der Annahme einer kostensteigernden Partitionierung. CP wird erst verringert, wenn eine für den aktuellen Iterationsschritt passende Partitionierung gefunden wurde, weil erst dann ein neues Gleichgewicht existiert. Bei einer nicht akzeptierten Partition wird bei konstantem CP in der Nachbarschaft nach einer anderen Partition gesucht, die die Bedingung des Metropolalgorithmus erfüllt, bzw. deren Wahrscheinlichkeit $Px < P$ ist. Am Ende des Partitionierungsprozesses konvergiert CP gegen Null bzw. gegen einen fest vorgegebenen Grenzwert.

Der Kontrollparameter CP wird mit jeder Iteration in Phase 2 (siehe Abbildung 3.7) verkleinert, um damit die Wahrscheinlichkeit kostensteigernder Partitionierungen zu verringern. Der Zeitplan für das Absenken des Kontrollparameters ist dabei sehr wichtig. Ein zu schneller Abfall des Kontrollparameters kann bedeuten, dass das Simulated Annealing in einem lokalen Minimum hängen bleibt. Ein zu langsamer Abfall bedeutet, dass zu Beginn die Partitionierungen zwischen Hard- und Software lange Zeit hin und her springen, bevor sich eine geeignete Partitionierung herausbildet. Zwei Varianten werden üblicherweise für den Kontrollparameter CP verwendet - der geometrische Ansatz:

$$CP_{neu} = \alpha \cdot CP_{alt} \tag{3.16}$$

wobei α zwischen 0,9 und 0,99 liegt oder der Ansatz nach Lundy und Mess [91]:

$$CP_{neu} = \frac{CP_{alt}}{1 + \beta CP_{alt}} \tag{3.17}$$

Wobei β eine Konstante nahe Null ist.

3.4.2 Codeanalyse

Das Gütekriterium der Partitionierung ist die Minimierung der Systemverarbeitungszeit bei gleichzeitiger Verringerung der benötigten Ressourcen. Dazu muss

der in C/C++ realisierte Algorithmus analysiert werden. Hierfür wird ein Softwaremodell aufgestellt, das auf das Hardwaremodell übertragen wird.

Softwaremodell: Die Granularität der Analyse bezieht sich im beschrieben Partitionierungssystem auf die Funktionen. Das heisst, die kleinste Einheit, die in Hardware oder Software implementiert wird, ist die Funktion. Die Partitionierung wird demzufolge funktionsbasiert durchgeführt. Der Datenfluss innerhalb einer Funktion spielt bei der Analyse des Codes und beim Aufbau des Funktionsschedules keine Rolle. Daher wird eine Aufteilung der Funktionen in Hüll- und Basisfunktionen vorgenommen.

Definition Hüllfunktion: Eine Hüllfunktion ist eine Funktion, die nur Schleifen, Verzweigungen und Funktionsaufrufe zu anderen Funktionen enthält. Die Funktionen, die in einer Hüllfunktion aufgerufen werden, sind die Kinder der Hüllfunktion und können selbst wieder Hüllfunktionen oder Basisfunktionen sein.

Definition Basisfunktion: Innerhalb einer Basisfunktion werden keine weiteren Funktionen aufgerufen, d.h. es existieren keine weiteren Kinder dieser Funktion. Innerhalb einer Basisfunktion können die einzelnen Algorithmen abgearbeitet werden. Die Größe einer Basisfunktion muss vom Entwickler sinnvoll gewählt werden, um eine korrekte Partitionierung durchführen zu können. Die Gesamtmenge M_F der Basisfunktionen F_i bildet dabei das zu partitionierende Gesamtsystem (Gleichung 3.18).

$$M_F = \{\ F_0, F_1, ..., F_{N-1}\} \tag{3.18}$$

Wobei das System ein $(T + TT = N)$-Tupel aus (S_{F_S0},S_{F_S1},...,$S_{F_{ST-1}}$, H_{F_S0}, H_{F_S1},...,$H_{F_{STT-1}}$) ist. S sind dabei die auf T verschiedene Prozessoren aufgeteilten Funktionen und H die als Hardware auf TT verschiedenen Logikelementen realisierten Komponenten.

Die Hüllfunktionen dienen dem Aufbau des Verbindungssystems zwischen den Basisfunktionen. Sie werden hauptsächlich für die Entwicklung des Funktionsschedules benötigt.

Um die einzelnen Funktionen entsprechend ihrer Eigenschaften in Hardware- und

Softwarekomponenten aufteilen zu können, erfolgt eine statische und eine dynamische Analyse des Algorithmus. Statisch werden die Nebenbedingungen, die Konnektivität und die Komplexitäten der Funktionen ermittelt. Dynamische Kenngrößen sind die Aufrufhäufigkeit und die Verarbeitungszeiten der Funktionen, um festzustellen, welche Zeiten der Gesamtalgorithmus und die Einzelfunktionen benötigen. Die einzelnen Schritte sind in Abbildung 3.8 dargestellt und werden nachfolgend erläutert.

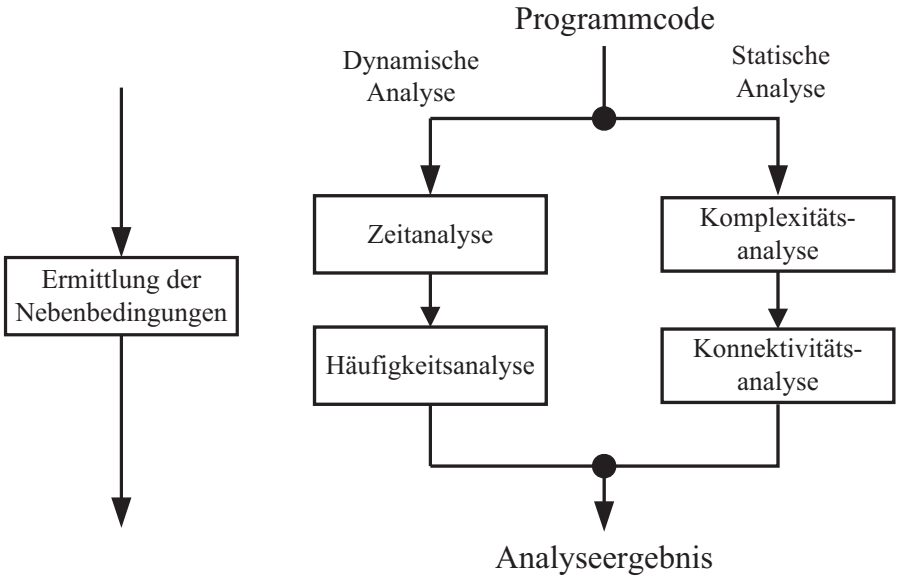

Abbildung 3.8: Analyseprozess

Zeitanalyse

Die Zeitanalyse dient dem Ermitteln der Programmausführungszeit, um mithilfe der Nebenbedingungen feststellen zu können, wie groß der Optimierungsbedarf ist und welche Unterprogramme die meiste Verarbeitungszeit benötigen.

Der Algorithmus wurde auf einem normalen Windows PC als C/C++ Code entworfen und dort getestet. Daher wird auch die Zeitanalyse auf dem PC vorgenommen. Dazu werden mit einem Analysetool zu Beginn und am Ende einer Funktion Codezeilen automatisch eingefügt, die zum Messen der PC-Zeit in den entsprechenden Zeitpunkten dienen. Die zusätzlichen Codezeilen werden nach

dem Durchlaufen des Algorithmus wieder entfernt.

$$t_{F_i} = t_{F_{istart}} - t_{F_{iende}} \tag{3.19}$$

Die Differenzen der PC-Zeiten t_{F_i} zwischen Beginn $t_{F_{istart}}$ und Ende $t_{F_{iende}}$ einer Funktion (Gleichung 3.19) werden in Ticks gemessen und zusammen mit dem entsprechenden Funktionsnamen in eine Datei geschrieben. Fehler durch eventuelle Interrupts des Betriebssystems können durch ein Heraufsetzen der Priorität des Ausführungsprogramms unter Windows begrenzt werden.

Mit einem Benchmarkprogramm lassen sich zusätzlich die Ausführungszeiten bestimmter Programmroutinen auf dem PC und auf dem NIOS-II Prozessor im FPGA vergleichen. Dazu werden für die entsprechenden Routinen die Verarbeitungszeiten auf dem NIOS-II gemessen. Da hier kein Betriebssystem im Hintergrund arbeitet, können diese Zeiten sehr genau bestimmt werden. Die Ausführungszeiten der Routinen auf einem PC werden direkt durch das Benchmarkprogramm ermittelt, weil hier verschiedene PCs zum Einsatz kommen können und so verschiedene Ausführungszeiten entstehen. Mit den Ergebnissen des Benchmarktests und der Timinganalyse kann eine Abschätzung der Funktionslaufzeiten auf der eingebetteten Hardware erfolgen. Die Ergebnisse sind dabei Schätzwerte und können nur eine beschränkte Aussage über die wahren Ausführungszeiten auf einem NIOS II Prozessor geben. Ein Ausweg wäre eine Emulation des Algorithmus auf einem entsprechenden Emulator. Dazu sind aber verschiedene Anpassungen des Algorithmus an das eingebettete System nötig (Speicherzugriffsoperationen, Schnittstellen zu den Kameras). Diese Anpassungen benötigen entsprechende Entwicklungszeit, was in diesem Stadium nicht unbedingt sinnvoll ist, solange die Schätzung durch die gewonnenen Werte des Benchmarktests ausreichend genau ist.

Häufigkeitsanalyse

Mit der Häufigkeitsanalyse wird festgestellt wie oft ein Unterprogramm aufgerufen und in welcher Zeit die Funktion abgearbeitet wurde. Die Kamera wird dabei

simuliert und der Algorithmus mit den entsprechenden Eingangsdaten durchlaufen. Die Daten können über mehrere Bilder ermittelt werden, um eine Verteilungsfunktion für die Aufrufhäufigkeit HA_{F_i}, die Verarbeitungszeit einer Funktion t_{F_i} und die Gesamtverarbeitungszeit t_{ges} eines Bildes zu bestimmen. Somit lässt sich der Erwartungswert und die Standardabweichung für t_{F_i}, HA_{F_i} und t_{ges} ermitteln. Es wird angenommen, dass die Werte gaußverteilt und unabhängig sind. Damit ergeben sich der Erwartungswert μ und die Standardabweichung σ (Gleichung 3.21) (Gleichung 3.20) für die Verarbeitungszeit einer Funktion wie folgt.

$$\mu_{t_{F_i}} = \sum_{i=0}^{n} t_{F_i} \cdot p_i \qquad mit\ p_i = \frac{1}{n} \tag{3.20}$$

$$\sigma_{t_{F_i}} = \sqrt{\sum_{i=0}^{n} (t_{F_i} - \mu_{t_{F_i}})^2 \cdot p_i} \qquad mit\ p_i = \frac{1}{n} \tag{3.21}$$

p_i ist dabei die Auftrittswahrscheinlichkeit. Da hier alle gemessenen Zeiten der gleichen Wahrscheinlichkeit unterliegen, ergibt sich p_i aus der Anzahl der Messungen n. Weiterhin ist es notwendig, den Maximalwert der einzelnen Auftrittshäufigkeiten anhand einer ausreichenden Zahl von Messungen, auch über mehrere Bilder, zu bestimmen. Für den Erwartungswert und die Standardabweichung von Auftrittshäufigkeit und Gesamtverarbeitungszeit ergeben sich analoge Gleichungen. Mit den hier ermittelten Werten kann festgestellt werden, welche Funktionen die meisten Ressourcen verbrauchen und welche das größte Optimierungspotential aufweisen.

Komplexitätsanalyse

Komplexe Funktionen sind eher kontrollflussorientiert und daher besser als Softwareprogramme auf einem Prozessor realisierbar, während weniger komplexe Funktionen durch ihre Datenflussorientiertheit auch sehr gut als Logik in Hardware implementiert werden können. Hier soll die Komplexität der einzelnen Funktionen F_i analysiert werden. Die Komplexität einer Funktion F_i ergibt einen wichtigen Anhaltspunkt zu den Kosten für die Implementierung der Funktion in Hardware

bzw. Software. Zur Ermittlung der Komplexität werden verschiedene Softwaremetriken verwendet. Es werden sowohl die Anzahl der mathematischen Operationen (Halstead-Metrik) als auch die Anzahl der Verzweigungen (zyklomatische Komplexität) ermittelt. Die Anzahl der Codezeilen (LOC) ist ein sehr ungenaues Maß, da sie maßgeblich vom Stil des Programmierers abhängt. Bei der Halstead-Metrik [43] wird die Anzahl der unterschiedlichen Operatoren hn_1 und Operanden hn_2 eines Programms gezählt. Sie bildet die Vokabulargröße hn. Die Anzahl der insgesamt verwendeten Operatoren HN_1 und Operanden HN_2 bildet nun die Implementierungslänge HN. Daraus lassen sich das Halstead-Volumen HV (Gleichung 3.22) und die Schwierigkeit HD (Gleichung 3.23) berechnen.

$$HV = HN \cdot lb(hn) \tag{3.22}$$

$$HD = \frac{hn_1}{2} \cdot \frac{HN_2}{hn_2} \tag{3.23}$$

HV dient der Ermittlung der Programmgröße und HD der Berechnung der Schwierigkeit eines Programms. Aus den berechneten Werten lässt sich der Aufwand HE einer Funktion bestimmen:

$$HE = HD \cdot HV \tag{3.24}$$

Eine weitere gebräuchliche Softwaremetrik zur Analyse von Programmcode ist die zyklomatische Komplexität CC (Gleichung 3.25) nach McCabe [77].

$$CC = E - V + 2 \tag{3.25}$$

Sie ist ein Maß für die strukturelle Komplexität einer Funktion. Dabei wird der Kontrollflussgraph des Programms zu Grunde gelegt. E ist die Anzahl der Kanten des Graphen und V die Anzahl der Knoten. Die zyklomatische Komplexität gibt die Anzahl der unabhängigen Pfade durch den Kontrollflussgraphen an. Sie kann auch direkt im Code der Funktion durch die Anzahl der dort verwendeten binären

Verzweigungen b_z ermittelt werden (Gleichung 3.26), sofern sich alle Befehle auf einem Pfad von einem Eintrittsknoten zu einem Austrittsknoten befinden [32].

$$CC = b_z + 1 \tag{3.26}$$

Zyklomatische Komplexität CC	Bewertung der Funktionskomplexität
1-10	einfache Funktion
11-20	komplexere Funktion
21-50	komplexe Funktion
>50	sehr komplexe Funktion

Tabelle 3.1: Komplexität von Programmen nach [32]

Konnektivitätsanalyse

Nachdem bisher das Verhalten der einzelnen Funktionen F_i untersucht wurde, steht hier das Zusammenspiel der Funktionen im Vordergrund. Die Konnektivität wird ermittelt, indem zunächst die Reihenfolge der Funktionsaufrufe im Hauptprogramm festgestellt wird. Die Reihenfolge stellt ein erstes Indiz für das Scheduling dar. Hierbei ist es möglich, dass verschiedene Programme, die erst spät aufgerufen werden, aufgrund der zur Verfügung stehenden Daten eigentlich schon viel eher abgearbeitet werden können. Um das im Scheduling festzustellen, muss ermittelt werden, welche Variablen an die einzelnen Funktionen übergeben werden und welche sie zurück geben. Weiterhin sind die verwendeten globalen Variablen wichtige Konnektivitätsmerkmale der Funktion. Sind alle übergebenen lokalen und globalen Variablen der Funktionen ermittelt, kann festgestellt werden zu welchem Zeitpunkt die einzelnen Variablen das letzte Mal verwendet wurden. Mit diesen Informationen lassen sich dann beim Scheduling Parallelitäten und Pipelinestrukturen erkennen, die bei der Optimierung der benutzen Hardwarestruktur Verwendung finden.

Ermitteln der Nebenbedingungen

Nebenbedingungen sind die systemimmanenten Eigenschaften. Hierzu zählen der Kommunikationsoverhead und die Kommunkationstotzeit der globalen und lokalen Schnittstellen, die Pixel-, Zeilen- und Bildfrequenz der Kameras und die Taktfrequenz von Logik und Prozessor. Der Kommunikationsoverhead ist das zusätzliche Datenvolumen, das durch die Schnittstellenkommunikation entsteht. Die Totzeit kennzeichnet die Zeit, die zwischen Bereitstellung und Übernahme der Daten entsteht. Aus den Pixel-, Zeilen- und Bildfrequenzen ergeben sich die maximalen Verarbeitungszeiten für die einzelnen Funktionen. Die Frequenzen variieren je nach Kamera und deren Auflösung. Bei Zeilenkameras entspricht die Zeilenfrequenz z.B. auch der Bildfrequenz. Die Bildfrequenzen können 10 Bilder je Sekunde aber auch 100 Bilder je Sekunde betragen. Die Nebenbedingungen müssen erfüllt sein, um eine erfolgreiche Partitionierung zu erreichen.

Überblick Analyse

In Abbildung 3.9 sind noch einmal alle während der Analyse überprüften Systemeigenschaften tabellarisch zusammengestellt.

Ziel der Analyse war es, die relevanten Informationen zum Verhalten des Algorithmus zu ermitteln, um das Gütekriterium einer minimalen Systemverarbeitungszeit bei gleichzeitiger Verringerung der benötigten Ressourcen zu erreichen. Dazu wurde der Zeit- und Ressourcenverbrauch der einzelnen Funktionen auf einem Prozessor ermittelt, um das jeweilige Optimierungspotential zu erschließen. Mit der Komplexitätsanalyse konnte festgestellt werden, inwiefern eine Portierung in Hardware möglich und sinnvoll ist. Die Ergebnisse der Konnektivitätsanalyse werden beim Scheduling zum Einordnen der Funktionen in eine Pipeline und zum Parallelisieren verwendet. Die Nebenbedingungen geben Auskunft über die zu erreichenden Ziele der Partitionierung.

Zeitanlyse Ausführungszeit PC wahrscheinliche Ausführungszeit auf NIOS II	**Häufigkeitsanalyse** Anzahl der Funktionsaufrufe Mittelwert Standardabweichung Maximum
Komplexitätsanalyse Anzahl der arith. Operationen Multiplikation Divisionen Additionen und Subtraktionen zyklomatische Komplexität LOC	**Nebenbedingungen** Schnittstellenoverhead Lokale Schnittstelle Globale Schnittstelle Externe Schnittstelle Datenvolumen der Kameras Verarbeitungsgeschwindigkeiten der Hardware
Konnektiviätsanalyse Funktionsreihenfolge im Programm verwendete globale Variablen übergebene Variablen Kinder der Funktionen	

Abbildung 3.9: Überprüfte Systemeigenschaften

3.4.3 Scheduling

Aus den in der Analyse ermittelten Daten lässt sich der Verbrauch an Rechenressourcen und Speicherplatz der einzelnen Funktionen bestimmen. Beim Scheduling wird nun auf Basis der Konnektivität festgestellt, in welcher Reihenfolge die Funktionen abgearbeitet werden müssen und ob sich mögliche Parallelitäten im Programm ergeben. Dadurch soll aus sequentiell geschriebener Software für einen Prozessor ein System mit mehreren Prozessoren und Logikelementen entstehen. Beim Scheduling wird zwischen dynamischem und statischem Scheduling unterschieden [93]. Das statische Scheduling ordnet eine Funktion einem Verarbeitungselement fest zu. Dies geschieht vor der eigentlichen Laufzeit des Programms und ist dann nicht mehr änderbar. Während beim dynamischen Scheduling die Funktion bei ihrem Aufruf an ein gerade freies Verarbeitungselement übergeben wird. Der Scheduler für das dynamische Scheduling muss mit implementiert werden. Das vergrößert den Systemoverhead und führt zu einer langsameren Abar-

beitung des Algorithmus. Da bei dynamischen Kontrollbedingungen, in Schleifen und Verzweigungen, die Ausführungszeiten der einzelnen Funktionen nur ungenau bestimmt werden können, ist es mithilfe des dynamischen Scheduling jedoch möglich, flexibel auf unterschiedliche Verarbeitungszeiten zu reagieren. Diese Vorgehensweise ist typisch für Multiprozessoranwendungen. Die im vorgesehenen System genutzten Logikelemente sind nur speziell für eine Funktion nutzbar und daher für ein dynamisches Scheduling ungeeignet (Logik, die in das FPGA geladen wird und danach bis zum Neuladen unveränderlich ist). Eine Mischform aus beiden Scheduling-Methoden ist das quasi-statische Scheduling [19]. Hierbei wird zunächst ein statisches Scheduling durchgeführt, wobei Funktionen mit stark veränderlichen Verarbeitungszeiten dynamisch zur Laufzeit geschedult werden.

In dieser Arbeit wird das statische Scheduling verwendet, da im ersten Schritt nur die Funktionen, die als Hardware realisiert werden sollen, den Logikelementen zuzuordnen sind. Wie bereits erwähnt ist für die Logikelemente ein dynamisches Scheduling ungeeignet. Die Zuordnung der Softwarefunktionen zu den entsprechenden Prozessoren wird in Kapitel 3.4.5 besprochen.

Das Scheduling wird in den folgenden Abschnitten beschrieben. Dazu wird zuerst das Flussdiagramm des zu partitionierenden Algorithmus aufgestellt. Mithilfe eines Listscheduling wird dann die Reihenfolge der Abarbeitung festgelegt. Da jedes Logikelement für nur eine Funktion verwendet wird, können Parallelitäten und Pipelinestrukturen effektiv umgesetzt werden. Hierei sind insbesondere die entstehenden Kommunikationskosten zu beachten.

Erstellen des Flussdiagramms

Die kleinste Granularität des Graphen ist die Funktion F_i. Das heißt im Flussdiagramm werden nur die Abhängigkeiten der Funktionen zueinander ermittelt. Durch die Unterteilung in Hüll- und Basisfunktionen sind nur die Kontrollflussanweisungen innerhalb der Füllfunktionen bei der Erstellung des Flussdiagramms notwendig, um Funktionsaufrufe in den Verzweigungen zu finden.

Die Datenflüsse ergeben sich aus dem Verwendungszeitpunkt einer Funktion und der letztmaligen Verwendung der in der Funktion benutzten lokalen und globalen

Variablen.

Eine geeignete Form der Graphenrepräsentation ist die Adjazenzmatrix bzw. die inverse Adjazenzmatrix [28]. Eine Adjazenzmatrix ist eine verkettete Liste, wobei jede Zeile einer Liste entspricht. Die Nummer der Zeile stellt den Startknoten für die folgende Liste dar. Es existieren genau N Zeilen für N Knoten eines Graphen. Jedes Element einer Adjazenzliste stellt eine Kante zu einem dem Startknoten nachfolgenden Knoten dar. Die inverse Adjazenzmatrix ist so wie die Adjazenzmatrix aufgebaut, allerdings befinden sich in den Adjazenzlisten die Kanten zu den Vorgängern des Startknotens.

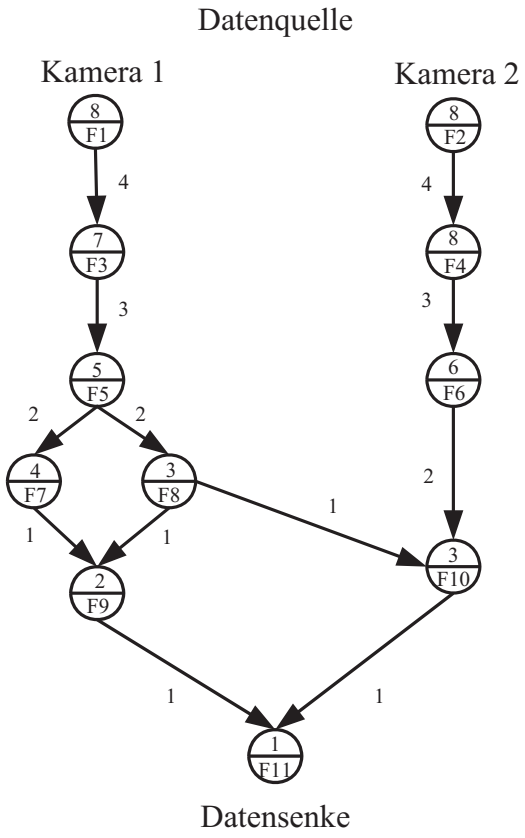

Abbildung 3.10: Beispielgraph $G(V, E)$

Für das Scheduling werden beide Formen der Adjazenzmatrix erzeugt, um die Vorgänger und Nachfolger der Knoten schnell ermitteln zu können. In Abbildung 3.10 ist ein Beispielgraph dargestellt, an dem die Erzeugung der Adjazenzmatrizen in Abbildung 3.11 vorgestellt wird. Der Beispielgraph dient auch in

den folgenden Abschnitten zur Darstellung des Partitionierungsalgorithmus. Die Funktionen werden als F_i bezeichnet und stehen im unteren Teil eines Knotens V_i. Im oberen Teil befinden sich die Verarbeitungskosten C_{P_i} für den Knoten. Die Werte an den Kanten E_{ij} des Graphen bezeichnen die Kommunikationskosten $C_{C_{ij}}$ zwischen zwei aufeinander folgenden Knoten.

Ausgangs-knoten	Adjazenz-matrix	Inverse Adjazenz-matrix
F1	→ F3	
F2	→ F4	
F3	→ F5	⇢ F1
F4	→ F6	⇢ F2
F5	→ F7 → F8	⇢ F3
F6	→ F10	⇢ F4
F7	→ F9	⇢ F5
F8	→ F9 → F10	⇢ F5
F9	→ F11	⇢ F7 → F8
F10	→ F11	⇢ F6 → F8
F11		⇢ F9 → F10

Abbildung 3.11: Adjazenzmatrix und inverse Adjazenzmatrix des Beispielgraphen $G(V, E)$ aus Abbildung 3.10

Erstellen des Schedules

Für das Scheduling existieren nur in wenigen Spezialfällen Algorithmen, die mit einer polynomialen Verarbeitungszeit zu einem optimalen Ergebnis führen [33]. Solche Spezialfälle sind zum Beispiel eine Baumstruktur des Taskgraphen oder eine Architektur mit nur zwei Prozessoren. Beide Fälle treffen auf den hier vorgestellten Fall nicht zu. Daher löst das Scheduling, wie auch das Simulated Annealing, ein NP-Optimierungsproblem.

Für das Scheduling des Funktionsgraphen sind folgende Regeln zu beachten:

Datenflussabhängigkeiten: Hier wird zwischen drei Arten unterschieden [108]: Zum einen ist es die *Flussabhängigkeit*. Das heißt eine Funktion muss erst auf die

Ergebnisse einer anderen Funktion warten. Dazu kommt die *Antiabhängigkeit*. Hier verwendet eine Funktion eine Variable, die nachfolgend genutzt wird, um das Ergebnis einer anderen Funktion abzulegen. Eine weitere Abhängigkeit ist die *Ausgabeabhängigkeit*. Das bedeutet eine Funktion gibt ihr Ergebnis an die gleiche Variable ab wie eine andere Funktion. Somit überschreiben sich die Ergebnisse gegenseitig. Daraus ergibt sich die Regel: *Zeitlich abhängige Inhalte von Variablen müssen im Datenfluss zu definierten Zeiten abgerufen bzw. geschrieben werden*

Kontrollflussabhängigkeit: Hier wird die Reihenfolge der Funktionen in Abhängigkeit zu Schleifen und Zweigen festgelegt [110]. Dabei lassen sich zwei Regeln festhalten:

1. *Funktionen die im Datenfluss vor einer Kontrollflussanweisung liegen, müssen auch vorher abgearbeitet werden.*

2. *Eine Funktion in einer Verzweigung oder Schleife wird erst gestartet, wenn die Bedingung der Kontrollflussanweisung ausgewertet ist.*

Durch die dynamische Länge einer Schleife oder die dynamische Entscheidung, welcher Zweig eines Algorithmus abgearbeitet wird, ist es schwierig konkrete Verarbeitungszeiten zu ermitteln und so ein korrektes Scheduling durchzuführen. Hier können durch die Zeitanalyse jedoch entsprechende Anhaltspunkte geliefert werden. Schätzverfahren für die Verarbeitungszeit werden im Kapitel 3.4.7 vorgestellt. Die Synchronisation der Datenübergabe an die einzelnen Logikelemente erfolgt über die lokalen und globalen Schnittstellen.

Zum Erstellen des Schedules ist das $List - Scheduling$ [93] ein geeigneter Ansatz zur Partitionierung in Hardware und Software. Das List-Scheduling ist ein heuristisches Verfahren, das in einer großen Anzahl von Fällen ein nahezu optimales Schedulingergebnis liefert [33]. Ein anderes Verfahren wird von Henkel [45] vorgestellt. Dabei werden Basisblöcke zu Objekten verschiedener Größe zusammengefasst. Das kleinste Objekt ist der Basisblock selbst und das größte Objekt ist der gesamte Algorithmus. Durch das Zusammenfassen entsteht eine endliche Anzahl von Objekten, die mit Hilfe einer Auswahlfunktion auf die entsprechenden Komponenten zugewiesen werden.

List-Scheduling: In dieser Arbeit soll auf das List-Scheduling zur Partitionierung zurück gegriffen werden.

Zu Beginn dieses Kapitels wurde das Schichtenmodell eines Bildverarbeitungsalgorithmus vorgestellt (Abbildung 3.1). Wie zu erkennen ist, wird der Komplexitätsgrad der Funktionen von der Vorverarbeitung zur Klassifikation immer größer. Die Funktionen untereinander sind aber durch eine Baumstruktur miteinander verknüpft (Abbildung 3.12).

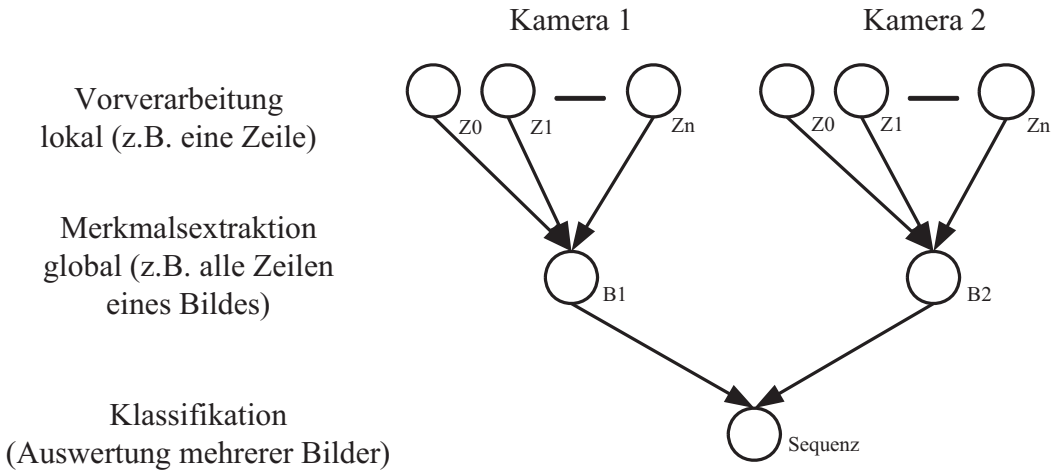

Abbildung 3.12: Baumstruktur eines Bildverarbeitungsalgorithmus

Die Ergebnisse der Vorverarbeitung (als Beispiel für Z_i Zeilen) werden an eine Merkmalsextraktion weitergereicht. Diese wird für ein komplettes Bild B durchgeführt. Die Ergebnisse mehrerer Bilder (mehrerer Kameras, oder zeitlich aufeinander folgender Bilder) werden in der Klassifikation verarbeitet. Wie in [1] bewiesen, ist ein List-Scheduling für Baumstrukturen sehr gut geeignet. Insbesondere der HLF (Highest Level First) Algorithmus führt in 90 Prozent der Fälle, in einem dynamisch geschedultem Multiprozessorsystem, zu einem Ergebnis, das maximal 5% vom Optimum entfernt ist. Andere Scheduling Algorithmen sind RANDOM und SCF. Beim RANDOM Algorithmus werden die Prioritäten zufällig vergeben und beim SCF Algorithmus werden die Co-Level zur Berechnung der Priorität herangezogen.

Wenn der Algorithmus auf einem einzelnen Prozessor realisiert ist, werden paralle-

le Funktionen, für mehrere Aufrufe, sequentiell als Schleifenanweisung ausgeführt. Die Parallelisierung von Schleifenanweisungen ist, ebenso wie die automatische Erzeugung von Logikelementen, nicht Gegenstand dieser Arbeit. In [4] und [22] sind Lösungsansätze zur Parallelisierung von Schleifen vorgestellt. In den folgenden Graphen werden Schleifen nicht explizit dargestellt, sondern als erhöhte Verarbeitungs- und Kommunikationszeit einer Funktion interpretiert.

Beim List-Scheduling werden den einzelnen Funktionen Prioritäten zugeordnet. Die Funktionen werden dann entsprechend ihrer Priorität absteigend geordnet. In Multiprozessorsystemen mit dynamischem Scheduling werden die Funktionen den freien Prozessoren zugewiesen. Im hier vorliegenden Fall wird die Funktion mit der höchsten Priorität in Hardware verschoben. Wird die so entstehende neue Partitionierung vom Simulated Annealing Algorithmus akzeptiert, beginnt eine neue Iteration der Phase 2 (siehe Abbildung 3.7). Wird die Partition zurückgewiesen, wird mit der Funktion der nächst niedrigeren Priorität fortgefahren. Die zurückgewiesene Funktion erhält eine niedrigere Priorität, die durch einen Straffaktor SF bestimmt wird.

Die **Prioritäten** werden bestimmt, indem der Datenflussgraph analysiert wird. Zunächst werden das Level und das Co-Level der einzelnen Graphenknoten bestimmt. Das Level L einer Funktion wird durch den längsten Pfad KP von der Funktion zur Datensenke beschrieben (Summe aller Gewichte). Während das Co-Level CL den längsten Pfad einer Funktion zur Datenquelle darstellt. Der Wert eines Pfades ergibt sich aus der Summe aller Gewichte entlang des Pfades (Gleichung 3.27).

$$L_{F_i} = KP_{F_i} = max \sum_{n=i}^{N} C_{P_i} \quad i \in N \tag{3.27}$$

Für den Graphen aus Abbildung 3.10 ergibt sich daraus Tabelle 3.2, ohne Einbeziehung der Kommunikationskosten $C_{C_{ij}}$. Dieser Fall tritt auf, wenn alle Funktionen auf dem gleichen Prozessor realisiert worden sind.

Wird jede Funktion auf einem eigenen Logikelement bzw. Prozessor realisiert (Startpartition), ergeben sich die Level aus Tabelle 3.3. Dabei wird generell noch

Funktion F_i	1	2	3	4	5	6	7	8	9	10	11
Level L_{F_i}	27	26	19	18	12	10	7	7	3	4	1
Co-Level CL_{F_i}	8	8	15	16	20	22	24	23	26	26	27

Tabelle 3.2: Level und Co-Level für $G(V, E)$ ohne Kommunikationszeiten

zwischen der Realisierung auf einem Prozessor bzw. auf einem Logikelement unterschieden, weil sich unterschiedliche Zeiten für die Kommunikation zwischen Hardware und Hardware, sowie zwischen Hardware und Software ergeben. Diese Unterscheidung wird hier aus Gründen der Übersichtlichkeit nicht dargestellt.

Funktion F_i	1	2	3	4	5	6	7	8	9	10	11
Level L_{F_i}	38	36	26	24	16	13	9	9	4	5	1
Co-Level CL_{F_i}	8	8	19	20	27	29	33	32	36	36	38

Tabelle 3.3: Level und Co-Level für $G(V, E)$ mit Kommunikationszeiten

Wie bereits zu erkennen ist, spielen nicht nur die reinen Verarbeitungskosten eine wichtige Rolle, sondern auch die Kommunikationskosten. Werden diese nicht berücksichtigt, können partitionierte Systeme entstehen, die einen maximalen Grad an Parallelisierung aufweisen, aber dennoch mehr Zeit benötigen als das Startsystem mit der kompletten Partition in einen Prozessor. Bei der Partitionierung existiert also ein Min-Max Problem zur Maximierung des Parallelisierungsgrades bei gleichzeitiger Minimierung der Kommunikationskosten. Im Kapitel 3.4.7 fließen die Kommunikationszeiten bereits in die Kostenfunktion für das Simulated Annealing ein, jedoch ist dort der Berechnungsaufwand wesentlich größer. Hier kann, durch die Einbeziehung der Kommunikationskosten, die Qualität der Partitionierung gesteigert werden, ohne wesentlich mehr Zeit dafür zu benötigen (siehe Berechnung der Prioritäten).

3.4.4 HW-SW Allocation

Wie aus den Tabellen zu erkennen ist, besitzen die Funktionen zu Beginn des Datenflusses in der Nähe der Datenquelle das höchste Level. Diese Funktionen sind nach Abbildung 3.1 gleichzeitig die Funktionen mit hohen Datenraten und ho-

her Datenflussabhängigkeit. Das Level nimmt mit größer werdender Entfernung von der Datenquelle ab, wohingegen das Co-Level zunimmt. Die Funktionen am Ende des Datenflusses weisen gleichzeitig die höchste Komplexität und ein niedriges Datenvolumen auf, was eher für eine Softwareimplementierung spricht. Durch die Verwendung eines List-Scheduling Algorithmus werden die bereits genannten Regeln der Daten- und Kontrollflussabhängigkeiten erfüllt. Somit stellt das List-Scheduling mit der Zuordnung von Level und Co-Level auf die Funktionen eine geeignete Basis für die Partitionierung dar. Dies gilt jedoch nur für solche Datenflussschemata, wie sie üblicherweise in Bildverarbeitungsalgorithmen auftreten.

Für eine automatische Partitionierung ist es notwendig, Funktionen nicht nur von Software in Hardware zu verschieben, sondern auch in die umgekehrte Richtung. Dazu wird für Funktionen in Software das Level zur Erzeugung einer Priorität herangezogen und für Funktionen in Hardware das Co-Level. Für eine Funktion mit einem hohem Level ergibt sich auch eine hohe Priorität (siehe Gleichung 3.30). Wird diese Funktion auf die Hardwareseite verschoben, ergibt sich daraus eine niedrige Priorität. Da das Co-Level für eine Funktion mit großem Level sehr klein ist, folgt daraus eine geringe Priorität. Die Prioritäten aller Funktionen (Funktionen in Hardware und Software) werden zur Partitionierung herangezogen. Die Berechnung der einzelnen Prioritäten wird im Folgenden erläutert.

Berechnung der Prioritäten: Die Kalkulation der Prioritäten erfolgt auf Basis der zuvor ermittelten Level und Co-Level. In der Startpartitionierung besitzt jede Funktion ihren eigenen Prozessor. Es ist also eine komplette Softwarerealisierung. Wurde eine Funktion im vorhergehenden Iterationsschritt aus der Software in die Hardware verschoben und durch den Metropolalgorithmus akzeptiert, ändert sich das Level der Funktion nach Gleichung 3.28 und das Co-Level nach Gleichung 3.29. Dabei sind $C_{C_{ij}}$ die Kommunikationskosten zwischen der Funktion F_i und deren direkten Nachfolgern. $C_{C_{gi}}$ sind die Kommunikationskosten zwischen der Funktion F_i und ihren direkten Vorgängern. Die Kommunikationskosten werden Näherungsweise mit den in der Analyse bestimmten Daten ermittelt (siehe auch Kapitel 3.4.7). Somit sind die benötigten Kommunikationskosten in den aktuellen Levelwerten integriert.

$$L_{F_i} = KP_{F_i} = max \sum_{n=i}^{N} (C_{P_i} + C_{C_{ij}}) \quad i \in N \tag{3.28}$$

$$CL_{F_i} = KP_{F_i} = max \sum_{n=0}^{i} (C_{P_i} + C_{C_{gi}}) \quad i \in N \tag{3.29}$$

Durch eine Gewichtsfunktion (Gleichung 3.30) lässt sich aus den Werten der Level und Co-Level sowie einzelner Analysewerte die Priorität der Funktionen bestimmen.

$$P_{L_i} = (1 - StF) \cdot (a_P \cdot L_{F_i} + b_P \cdot (C_{P_{i_{SW}}} - C_{P_{i_{HW}}}) - c_P \cdot CC) \tag{3.30}$$

Die oben beschriebene Gewichtsfunktion gilt für in Software realisierte Funktionen. Für in Hardware realisierte Funktionen wird L_{F_i} durch CL_{F_i} ersetzt. $C_{P_{i_{SW}}} - C_{P_{i_{HW}}}$ stellt den Geschwindigkeitsgewinn für eine mögliche Hardwarerealisierung dar. CC ist die Komplexität einer Funktion nach Gleichung 3.25 und StF ist der Straffaktor, falls die Funktion bereits einmal vom Metropolalgorithmus abgewiesen wurde. a_P, b_P und c_P, sind die Gewichtsfaktoren zur Ermittlung der Funktionspriorität. Sie werden heuristisch bestimmt. Die konkreten Werte werden im Kapitel 5 vorgestellt.

Elementebibliothek: Die auf der Hardware allozierten Elemente befinden sich in einer Elementebibliothek, da eine Portierung des C/C++ Codes in eine Hardwarebeschreibung im Rahmen dieser Arbeit nicht vorgesehen ist. Um die Elemente korrekt zuordnen zu können, müssen sie sowohl als Hardwarebeschreibung als auch als Softwarebeschreibung in der Elementebibliothek vorliegen. Dadurch kann bereits während des Entwurfs der Software auf diese Bibliothek zurückgegriffen werden. Für diese Funktionen existieren sehr genaue Verarbeitungszeiten, so dass das Scheduling in der folgenden Iteration eine höhere Genauigkeit erreicht. Sind als Hardware allozierte Funktionen nicht in der Elementebibliothek vorhanden, müssen sie manuell erzeugt werden. Die manuelle Erzeugung der Funktionen kostet Entwicklungszeit und soll daher auch in die Kostenfunktion als Faktor eingehen. Der Einsatz von $CtoVHDL$ Compilern ist möglich, wobei nicht von einer

optimalen Portierung ausgegangen werden kann. Für Funktionen, die nicht in der Elementebibliothek vorhanden sind, können zunächst nur Schätzverfahren angewendet werden, um die Kommunikationskosten, die Verarbeitungszeit der Funktion in Hardware und den Ressourcenverbrauch der Funktion in Hardware zu ermitteln. Die Schätzverfahren werden im Kapitel 3.4.7 betrachtet.

Weiterführende Untersuchungen zur automatischen Generierung einer VHDL-Beschreibung aus C/C++ Code sind nicht Teil dieser Arbeit. Um ein vollständig automatisiertes System zu erhalten, sind dazu weitere Forschungen notwendig.

3.4.5 Clusterung der Softwarefunktionen

Für die Partitionierung der Funktionen in Software und Hardware wurde ein List-Scheduling Algorithmus verwendet, der den einzelnen Funktionen eine Priorität zugewiesen und die Funktionen entsprechend ihrer Priorität in Hardware realisiert hat.

Alle Funktionen, die nicht einer Hardwarekomponente zugeordnet worden sind, werden bis zu diesem Zeitpunkt auf einem einzelnen Prozessor ausgeführt. Zwischen den einzelnen Softwarefunktionen können Parallelitäten existieren, die wiederum für einen System-Speedup nutzbar sind. Um den maximalen Speedup zu erreichen, können alle Funktionen auf getrennten Prozessoren verarbeitet werden. Wie bereits im Kapitel 3.4.3 erläutert, führt dies zu einer Erhöhung der Kommunikationskosten. Weiterhin führt die Allozierung eines Prozessors mit nur einer Funktion zu einer geringen Prozessorauslastung, was die Kosten erhöht und die Effektivität der Partitionierung verringert. Es entsteht also wieder ein Min-Max-Problem mit einer Minimierung der Kommunikationskosten bei gleichzeitiger Maximierung des System-Speedups und der Prozessorauslastung. Zur Lösung dieses Problems wird ein Clusteralgorithmus verwendet. Der Clusteralgorithmus soll verschiedene Funktionen zu einem Funktionscluster zusammenfassen und die so entstandenen Cluster entsprechenden Prozessoren zuordnen.

Clusteralgorithmus

Der Clusteralgorithmus ist eine Heuristik, die im Graphen durch Kanten verbundene Funktionen zu Clustern zusammenfasst. Die Anzahl der Cluster entspricht dabei der Anzahl der verwendeten Prozessoren. Da durch die genutzten NIOS-II Prozessoren und den Avalon Bus keine konkrete Prozessorenanzahl und Verbindungsstruktur vorgegeben ist, kann mit Hilfe des Clusteralgorithmus eine effektive Clusterstruktur gefunden werden. Verschiedene Clusteralgorithmen sind in [38] und in [121] vorgestellt. Im vorliegenden Fall wird ein Clusteralgorithmus nach Sarkar [94] verwendet.

Die Clusterung wird einmal in Phase eins durchgeführt und in Phase zwei des Partitionierungsalgorithmus in jeder Iteration. In Phase eins wird überprüft, ob eine Partitionierung in Hardware und Software durch den Aufbau eines Multiprozessorsystems vermieden werden kann. In Phase zwei werden nach einer erfolgten Portierung einer Funktion in Hardware, die als Software verbliebenen Funktionen wieder zu Clustern zusammengefasst. Erst nach der Clusterung liegt ein komplettes Hardware-Software Co-Design vor, das dann durch den Metropolalgorithmus angenommen bzw. abgelehnt wird. Die Clusterung wird nach jeder Iteration wieder aufgelöst, sofern das Ziel der Partitionierung noch nicht erreicht wurde.

Wurde die Hardwarepartitionierung durchgeführt, ergibt sich je nach eingestellten Nebenbedingungen ein partitionierter Graph, wie z.B. in Abbildung 3.13 zu sehen. In diesem Fall bilden die Funktionen F_1 bis F_6 die Menge der in Hardware realisierten Graphenknoten. Die Funktionen F_7 bis F_{11} bilden die Menge SW der als Software zu realisierenden Graphenknoten (siehe Gleichung 3.31).

$$SW = \bigcup_{i=7}^{11} F_i \tag{3.31}$$

Im vorliegenden Fall ergeben sich drei zusammenhängende Graphenmengen, mit wenigen Übergängen zwischen Hard- und Software. Es ist aber auch möglich, dass Hardwarefunktionen Daten an Softwarefunktionen übergeben und diese wieder Hardwarefunktionen aufrufen. Diese Möglichkeit tritt auf, wenn z.B. die Funktion F_4 als Softwarefunktion deklariert wurde.

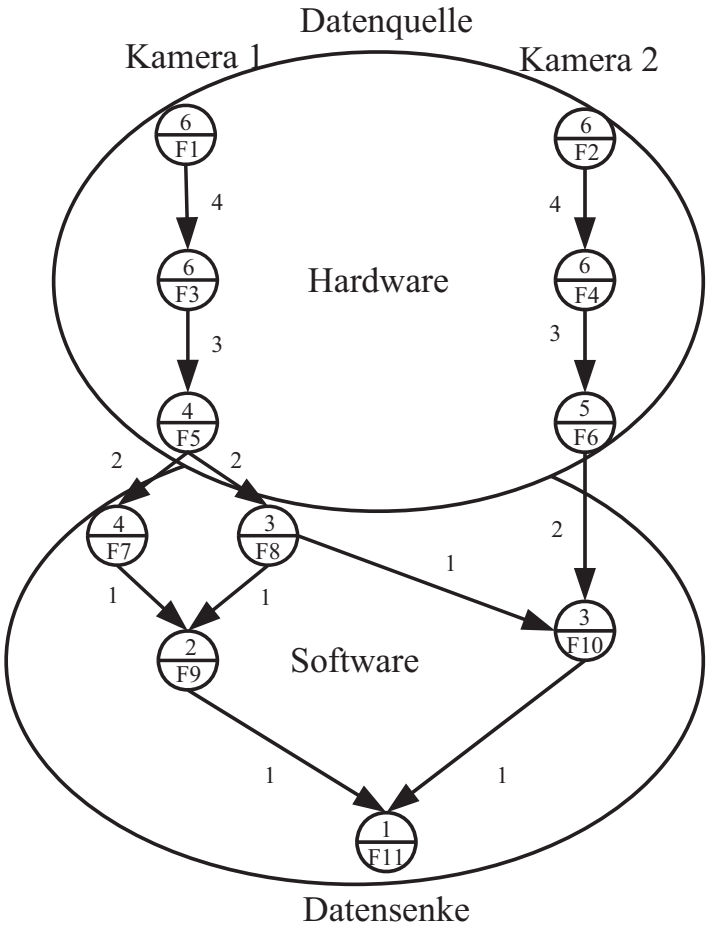

Abbildung 3.13: Partitionierter Beispielgraph $G(V, E)$

Als Startkonfiguration wird festgelegt, dass jede Softwarefunktion auf einem separaten Prozessor ausgeführt wird. Ziel des Sarkar-Algorithmus [94] ist die Minimierung der Graphenverarbeitungskosten C_G bei gleichzeitiger Verringerung der Kommunikationskosten C_C. Das lässt sich mit folgender Ungleichung ausdrücken:

$$C_G = \sum C_{P_i} + \sum C_{C_{ij}} \leq \sum C_{P_i} + C_C \tag{3.32}$$

Um das Ziel zu erreichen, werden die Kanten des Graphen nach ihren Kommunikationskosten geordnet. Die Kosten der Kante mit den höchsten Kommunikationsaufwendungen werden auf Null gesetzt und die durch die Kante verbundenen Funktionen zu einem *Cluster* zusammengefasst. Die Kante wird markiert und

für die weiteren Betrachtungen nicht mehr herangezogen. Im nächsten Schritt folgt die Kante mit den zweithöchsten Kosten. Die Verarbeitungsschritte werden so lange wiederholt, bis alle Kanten betrachtet wurden. Dabei können auch zwei *Cluster* zu einem *Cluster* zusammengefasst werden. Die Entnahme einer Funktion aus einem Cluster ist hingegen nicht möglich. Das Verfahren von Sakar ermöglicht eine nichtlineare Clusterung, d.h. unabhängige Funktionen z.B. F_7 und F_8 können auch in einem Cluster zusammengefasst werden.

Prozessorscheduling: Werden Funktionen in einem Cluster zusammengefasst, können sie nicht mehr parallel zueinander abgearbeitet werden. Stattdessen müssen sie seriell arbeiten. Dazu wird, gemäß dem List-Scheduling, der Funktion mit dem höchsten Level in einem Prozessor ein Verarbeitungszeitraum zugewiesen. Für die nun folgende Funktion muss überprüft werden, ob der Verarbeitungszeitraum und der Verarbeitungszeitpunkt nicht schon durch zuvor geschedulte Funktionen belegt ist. Sollte das der Fall sein, muss der Beginn der Verarbeitung für diese Funktion so lange verschoben werden, bis ein freier Zeitraum im Prozessor gefunden wurde. Da der Verarbeitungszeitraum einer Funktionen immer nur nach hinten verschoben werden kann, verändern sich die Verarbeitungszeiträume der darauf folgenden Funktion entsprechend. Als Folge ergibt sich eine insgesamt veränderte Verarbeitungszeit für den gesamten Algorithmus.

Die Einordnung der entsprechenden Verarbeitungszeiten auf dem Prozessor und die Verschiebung der folgenden Verarbeitungszeit erfolgt direkt während der Clusterung.

Im Graphen in Abbildung 3.14 ist der geclusterte Softwaregraph zu sehen. Es sind zwei Cluster mit zwei und drei eingebundenen Funktionen entstanden. Dabei gilt:

$$SW = \bigcup_{i=0}^{1} Cluster_i \tag{3.33}$$

$$\bigcap_{i=0}^{1} Cluster_i = \emptyset \tag{3.34}$$

Es befinden sich also alle Softwareelemente in einem entsprechenden Cluster

(Gleichunng 3.33), wobei sich kein Softwareelement in zwei Clustern befinden darf (Gleichunng 3.34).

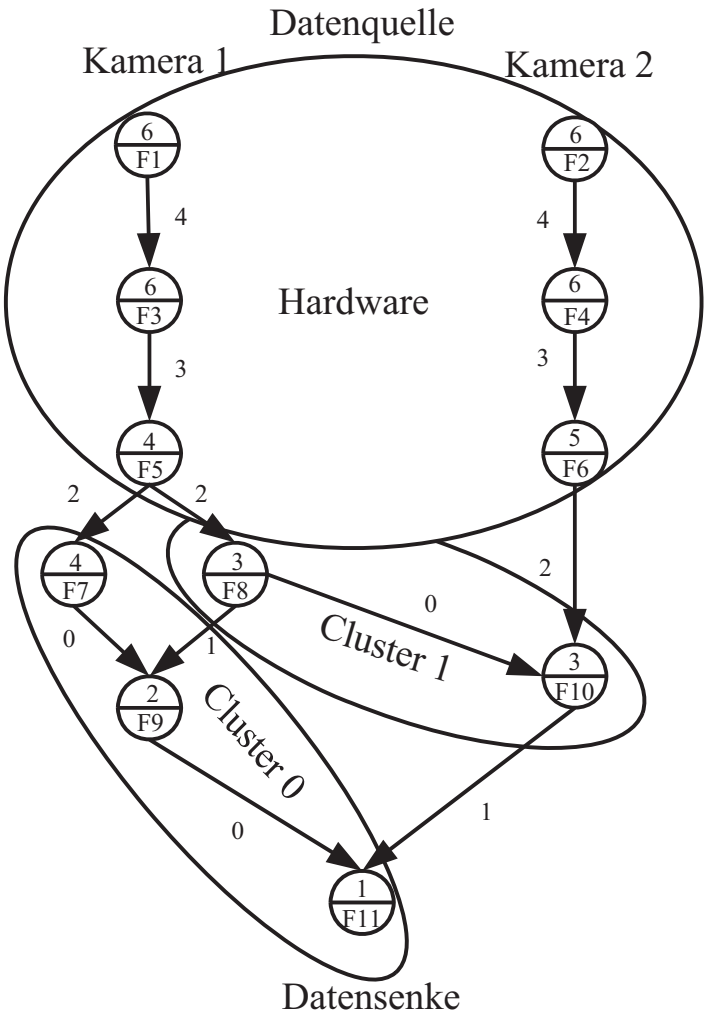

Abbildung 3.14: Partitionierter und geclusterter Beispielgraph $G(V, E)$

Da alle Funktionen mit Kanten verbunden sind, die als Kommunikationkosten den Wert eins besitzen, lässt sich hier keine Prioritätenliste erstellen. Es ist aber feststellbar, welches der kritische Pfad KP ist (Pfad mit der längsten Ausführungszeit). Existieren mehrere Pfade mit dem gleichen Länge KP wird ein Pfad ausgewählt. Im hier beschriebenen Fall ist es der Pfad $\{F_7, F_9, F_{11}\}$. Dabei werden zunächst die beiden Funktionen mit dem niedrigsten Level (siehe Tabelle 3.3) geclustert. Die Länge des Pfades verringert sich somit auf 8. Der neue kritische

Pfad ist $\{F_8, F_{10}, F_{11}\}$. Eine Clusterung der Funktion F_{10} in $Cluster_0$ würde eine Verlängerung des kritischen Pfades bedeuten, deshalb werden hier die Funktionen F_8 und F_{10} zu $Cluster_1$ zusammengefasst. KP ist jetzt 8. Eine weitere Verringerung des kritischen Pfades ist nicht mehr möglich, aber eine Verringerung der Kommunikationskosten, indem die Funktion F_4 in $Cluster_0$ mit einbezogen wird. Durch die Clusterung konnten die Kommunikationskosten $C_{C_{7,9}}$, $C_{C_{9,11}}$ $C_{C_{8,10}}$ eingespart werden. Gleichzeitig verringerte sich KP von 9 auf 8. In Tabelle 3.4 sind drei mögliche Clustervarianten dargestellt.

	KP	C_C
1 Proz.	13	0
2 Proz. = 2 Cluster	8	2
5 Proz. = 5 Cluster	9	5

Tabelle 3.4: Kosten für die Softwareimplementierung

Ein weiteres Entscheidungskriterium für die Clusterung ist die Prozessorausnutzung. Sind die maximalen Verarbeitungszeiten für einen Prozessor erreicht, kann keine weitere Funktion zu dem entsprechenden Cluster hinzugefügt werden. Die maximalen Verarbeitungszeiten für den konkreten Algorithmus werden den Nebenbedingungen, siehe Anhang C.1, entnommen. Allgemein lassen sich folgende Regeln für die Clusterung aufstellen:

- Verkleinerung von $KP \rightarrow$ Clusterung akzeptiert
- keine Verkleinerung von KP, Verringerung von $C_C \rightarrow$ Clusterung akzeptiert
- Vergrößerung von $KP \rightarrow$ Clusterung abgewiesen
- max. Verarbeitungskosten auf Proz. erreicht $\rightarrow$ Clusterung abgewiesen

Weitere Ausführungen zu einem eventuell entstehenden Pipelining der Prozessoren finden sich im folgenden Abschnitt.

3.4.6 Pipelining und Kommunikation

Durch die Partitionierung in Hardware und Software und durch die Clusterung der Softwarefunktionen auf verschiedene Prozessoren ist ein Multiprozessorsystem mit angeschlossener Hardware entstanden, dessen Verarbeitungsfunktionen miteinander kommunizieren müssen. Das partitionierte System wird in Pipelinestufen aufgeteilt [13]. Die Kommunikation erfolgt innerhalb einer Pipelinestufe nicht deterministisch, nach Abarbeitung der Funktion, durch das Feuern eines Tokens. Wurden alle Eingangstoken einer Funktion abgefeuert (alle Vorgängerfunktionen haben ihre Datenverarbeitung beendet), kann die Funktion selbst abgearbeitet werden, weil nun alle nötigen Eingangsdaten vorliegen. Zwischen zwei Pipelinestufen erfolgt die Kommunikation deterministisch zu genau festgelegten Zeiten.

Da die Kommunikation innerhalb des partitionierten Systems hauptsächlich über Punkt-zu-Punkt Verbindungen erfolgt, wird kein standardisiertes Bussystem, wie CAN [70], FlexRay [62] oder FlexRay [62], benötigt. Es kann auch auf eine eigene angepasste Lösung zurückgegriffen werden, die den Kommunikationsoverhead verringert und eine schnelle Kommunikation ermöglicht (globale und lokale Schnittstelle in Kapitel 3.2.2 und 3.2.3).

Die Verarbeitung innerhalb des partitionierten Systems ist synchron und basiert auf einem Basistakt. Von diesem Takt werden alle Prozessor-, Hardware- und Kommunikationstakte abgeleitet.

Zum Erzeugen der Pipelinestufen müssen die maximalen Verarbeitungszeiten der Elemente bekannt sein. Dadurch lassen sich die Zeitpunkte für die Datenübernahme zwischen zwei Funktionen genau bestimmen. Durch die Partitionierung und Clusterung und der dabei durchgeführten Schätzung der Kommunikations- und Verarbeitungszeiten liegen alle wesentlichen Daten für die Erstellung der Pipelinestufen vor.

Der für das Pipelining zu analysierende Graph ist ein verbundener Graph aus parallel und sequentiell angeordneten Knoten (Funktionen) mit den zugehörigen Verarbeitungszeiten. Für einen solchen Graphen gilt: *Ein durch Kanten verbunde-*

ner Graph kann nur so schnell abgearbeitet werden, wie sein langsamster Knoten. Der Knoten mit der längsten Verarbeitungszeit stellt die Referenz für die Erzeugung der Pipelinestufen dar. Dabei darf diese Zeit die maximale Verarbeitungszeit des Algorithmus nicht überschreiten. Die maximale Verarbeitungszeit wird in den Nebenbedingungen festgelegt. Es ist darauf zu achten, dass die Funktionen einer Pipelinestufe diese möglichst gut ausfüllen, sie jedoch keinesfalls überschreiten. Um keine Daten bei der Übergabe zu verlieren, sind deshalb entsprechende Sicherheitsaufschläge bei der Auslastung der Pipelinestufen zu berücksichtigen.

Mithilfe des aktuellen Levels (nach Partitionierung und Clusterung) und der Zuordnung zu den einzelnen Verarbeitungselementen, lässt sich, unter Anwendung des List-Scheduling die Position einer Funktion in der Pipeline ermitteln (siehe Tabelle 3.5) [90].

Funktion F_i	1	2	3	4	5	6	7	8	9	10	11
Level L_{F_i}	33	31	23	21	14	12	7	8	3	5	1
Co-Level CL_{F_i}	6	6	16	16	23	24	29	28	31	31	33
Verarb. Element	H0	H1	H2	H3	H4	H5	S0	S1	S0	S1	S0

Tabelle 3.5: Level und Co-Level für $G(V, E)$ mit zugeordnetem Verarbeitungselement

Das langsamste Verarbeitungselement ist $Prozessor_0$ mit $Cluster_0$. Demzufolge bestimmt $Cluster_0$ mit einer geschätzten Periodendauer von $T_s = 11$ die Gesamtverarbeitungsgeschwindigkeit. Der längste Pfad für F_{11} verläuft über die Elemente von $Cluster_1$. Da somit $Prozessor_0$ eine Zeiteinheit auf $Prozessor_1$ warten muss, ergibt sich $T_s = T_s + 1$ (nichtdeterministische Kommunikation mittels Tokens). In Abbildung 3.15 ist ein Zeitplan für den partitionierten und geclusterten Graphen aus Abbildung 3.14 zu sehen.

Die durch das Pipelining der Funktionen entstandene höhere Datenverarbeitungsrate ist im Zeitplan sehr gut zu erkennen.

Nachdem die Pipelinestufen für den Graphen ermittelt worden sind, wird eine zentrale Steuereinheit erzeugt, die jeden Prozessor und jedes einzelne Logikelement über einen Impuls startet. Die Wiederholdauer (T_s) des Impulses ist dabei

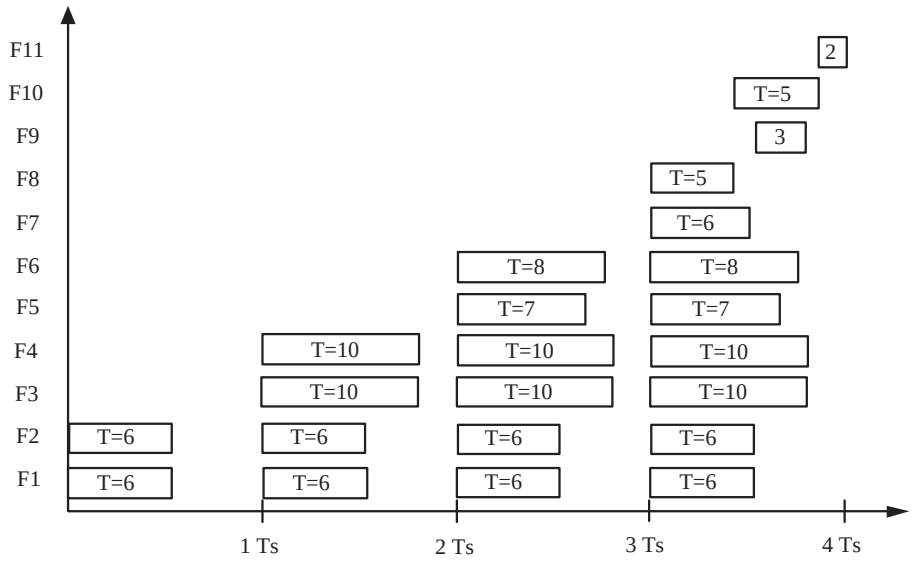

Abbildung 3.15: Pipelinestufen für den Graphen aus Abbildung 3.14

aus dem Systemtakt abgeleitet und immer größer als die Verarbeitungszeit der längsten Funktion. Die Eingangsfunktionen der Pipelinestufen warten nach der Abarbeitung ihrer Eingangswerte auf den nächsten Impuls, um dann die neuen Eingangsdaten zu verarbeiten.

Die hier vorgeschlagene Mischung aus synchroner und asynchroner Kommunikation stellt einen Kompromiss zwischen optimaler Ausnutzung der Rechenressourcen und der Zuverlässigkeit der Realisierung dar.

3.4.7 Kostenfunktion

Um die Ergebnisse der Partitionierung mithilfe des Metropolalgorithmus innerhalb des Simulated Annealing bewerten zu können, muss eine Kostenfunktion für das partitionierte System aufgestellt werden. Die Kostenfunktion ermittelt dabei anhand der Systemeigenschaften die Kosten jeder einzelnen Komponente und darauf basierend, die des gesamten Systems.

Für die Aufstellung einer Kostenfunktion gibt es verschiedene Ansätze, die in [72] erklärt sind. Die Kosten einer Partition können funktional mithilfe eines Algorithmus oder nichtfunktional durch Experten ermittelt werden. Da die Partitionie-

rung objektiv und unabhängig von Experten durchgeführt werden soll, wird hier die funktionale Variante verwendet. Die funktionale Methode basiert auf einem mathematischen Modell, das die Kosten C anhand der vorhandenen Kostenfaktoren x_i schätzt (Gleichung 3.35).

$$C = f(x0, x1, ..., x_{l-1}) \tag{3.35}$$

Die Kostenfaktoren bilden dabei ein l-Tupel, das die Kosten einer in Hardware bzw. in Software implementierten Funktion beschreibt. Im vorliegenden Fall wird eine Funktion durch folgende Kostenfaktoren beschrieben:

- Verarbeitungskosten in SW
- Verarbeitungskosten in HW
- Kommunikationskosten
- Hardwareaufwand
- Entwicklungskosten HW

Die Kosten aller an der Partitionierung beteiligten Funktionen werden addiert und ergeben ihre Gesamtkosten (Gleichung 3.36).

$$C_{gesamt} = \sum_{i=0}^{n-1} C_i \tag{3.36}$$

Die mathematischen Modelle zur Ermittlung der Kosten können linear, multiplikativ oder exponentiell sein. Das lineare Modell [76] ist der klassische Ansatz eine Kostenfunktion aufzustellen. Er ist in Gleichung 3.37 dargestellt. Die Koeffizienten a_i gewichten dabei die einzelnen Kostenfaktoren. Sie werden meist empirisch ermittelt.

$$C = a_0 + \sum_{i=0}^{l-1} (a_i \cdot x_i) \tag{3.37}$$

Das multiplikative Modell ist in Gleichung 3.38 zu sehen. Die Koeffizienten haben die gleiche Funktion wie im linearen Modell. Einige Arbeiten beschränken die Koeffizienten dabei auf die Werte $[-1, 0, +1]$ [116].

$$C = a_0 \prod_{i=0}^{l-1} a_i^{x_i} \tag{3.38}$$

Im COCOMO II Projekt [82] wird ein exponentielles Modell zur Schätzung von Softwarekosten genutzt (siehe Gleichung 3.39). Dabei ist W die Wichtung einzelner Kostenfaktoren und EM ein Erfolgsmultiplikator, von denen insgesamt 17 Stück vorhanden sind. S ist die Größe der Software, gemessen in Tausend LOCs.

$$C = 2,94 \prod_{i=1}^{17} EM_i S^{0,91+0,01 \cdot \sum W} \tag{3.39}$$

Die Funktionen auf dem NIOS II und dem PC unterliegen einem sehr ähnlichen Verarbeitungsmodell (ähnliche Funktionsprinzipien). Damit kann der Zusammenhang zwischen der Verarbeitungszeit einer Funktion auf einem PC und auf dem NIOS II als linear angenommen werden. Es ist deshalb ein Schätzverfahren nach Gleichung 3.37 für die Verarbeitungskosten zu bevorzugen. Ein exponentielles Modell nach COCOMO II wird nur zur Schätzung der Portierungskosten des C-Codes nach VHDL genutzt. Hier findet eine Transformation zwischen zwei unterschiedlichen Eingabesprachen statt, was der Entwicklung einer neuen Software gleichzusetzen ist. Für den Bereich der Schätzung von Softwarekosten wird der COCOMO II Ansatz nach [82] als geeignet angesehen und er soll darum auch hier zum Einsatz kommen.

Die Kostenfaktoren können nur durch eine Implementierung des partitionierten Systems auf die Zielhardware genau bestimmt werden. Diese Variante würde zu enormen Entwicklungszeiten führen. Deshalb wird versucht, die Kostenfaktoren so exakt wie möglich zu schätzen [34] und an Beispielimplementierungen zu validieren.

Schätzung der Verarbeitungskosten

Die Verarbeitungskosten sind entsprechen der Verarbeitungszeit in der Logik bzw. auf einem Prozessor. Insbesondere für die Verarbeitungszeit wurden verschiedene Methoden entwickelt. Taktgenaue Verfahren versprechen dabei die genauesten

Ergebnisse [68]. Sie erfordern jedoch auch besonders genaue Modelle für Hardware und Software. Zudem benötigen sie für die Ergebnisermittlung einen sehr hohen Rechenaufwand. Sie werden daher nur selten verwendet. Andere Verfahren versuchen die Optimierungen des Compilers bzw. des Hardwaresynthesetools zu schätzen und somit auf die Programmlaufzeit im fertigen HW-SW Co-Design zu schließen [105], [123]. Das erfordert wesentlich weniger Rechenaufwand und ist besser geeignet. Die damit verbundene mögliche Vergrößerung des Fehlers spielt zunächst eine nur untergeordnete Rolle [12]. Im vorgestellten HW-SW Partitionierungssystem soll deshalb zur Schätzung der Verarbeitungszeiten ein Verfahren eingesetzt werden, dass zwar eine schnelle Schätzung ermöglicht, dafür aber weniger Genauigkeit verspricht.

Die Schätzung der Verarbeitungszeit einer Funktion, die auf einem Prozessor realisiert wird, ist bereits im Kapitel 3.4.2 besprochen worden. Hierbei wird nicht der Datenfluss innerhalb einer Funktion analysiert, sondern die Ausführungszeiten der Funktion auf einem PC werden anhand eines Beispieldatensatzes ermittelt. Die Ergebnisse können nun durch Gleichung 3.40 auf den NIOS II Prozessor übertragen werden.

$$C_{P_{i_{SW}}} = a_i \cdot t_{i_{PC}} \tag{3.40}$$

Der Parameter a_i wird mithilfe eines Benchmarktestes und der mit der Analyse gefundenen arithmetischen Operationen für jede Basisfunktion ermittelt. Dabei werden die Speedfaktoren der arithmetischen Operationen (z.B. Addition $\rightarrow SF_{add}$) durch eine gewichtete Addition zu a_i nach Gleichung 3.41 zusammengefasst. Die Gewichte der Addition werden durch das Verhältnis der Anzahl der arithmetischen Operation (z.B. n_{add}) zur Anzahl aller arithmetischen Operationen n_{ges} bestimmt.

$$a_i = \frac{n_{i_{add}}}{n_{i_{ges}}} \cdot SF_{add} + ... + \frac{n_{i_{multfloat}}}{n_{i_{ges}}} \cdot SF_{multfloat} \tag{3.41}$$

Die für das vorgestellte System ermittelten Geschwindigkeitsfaktoren $SF_{...}$ sind im Anhang in Tabelle 5.1 angegeben.

Für die Schätzung der Verarbeitungszeit einer Hardwarefunktion ist dieses Verfahren nicht geeignet, weil hier Funktionsteile parallelisiert oder in einer Pipeline realisiert werden können. Da der Datenfluss während der Codeanalyse nur zwischen den Funktionen und nicht innerhalb der Funktionen analysiert wurden, ist schwer vorherzusagen, wie hoch der Geschwindigkeitsfaktor einer Funktionsrealisierung in Logik gegenüber einer sequentiellen Realisierung auf einem Prozessor ist. Es ist deshalb nur möglich, aufgrund von Erfahrungswerten und der vorhandenen Kennzahlen einer Funktion Parameter zu bestimmen und damit die Verarbeitungszeit der Funktion als Logikrealisierung zu schätzen. Die Schätzung erfolgt in diesem Fall nach dem Gesetz von Amdahl (Gleichung 2.2). Die Gesamtverarbeitungszeit entspricht hierbei der Verarbeitungszeit der langsamsten Logikoperation ($t_{par} = t_{OP}$) und einem sequentiellen Anteil t_{seq}.

Die Verarbeitungszeiten der Logikoperationen t_{OP} werden vorher durch Messung bestimmt. Der sequentielle Anteil an der Verarbeitungszeit t_{seq} wird nach Gleichung 3.43 aus dem sequentiellen Anteil f_S und t_{OP} bestimmt. Der sequentielle Anteil lässt sich hier durch die Metriken nach Halstead HD und McCabe CC ermitteln.

$$f_S = \frac{CC + HD}{LOC} \quad mit\,\{(CC + HV) < LOC\} \tag{3.42}$$

$$t_{seq} = \frac{f_S}{1 - f_S} \cdot t_{OP} \tag{3.43}$$

Aus den ermittelten Zeiten lässt sich wiederum der Geschwindigkeitsfaktor S_n nach Gleichung 2.4 bestimmen. Die Gesamtverarbeitungszeit der Hardware ergibt sich zu:

$$C_{P_{i_{HW}}} = t_{seq} + t_{OP} \tag{3.44}$$

Schätzung der Kommunikationskosten

Die Kommunikation erfolgt hier grundsätzlich über einen gemeinsamen Speicher (Shared Memory Architektur), der als DualPortRam ausgeführt ist. Es wird davon ausgegangen, dass die Datenwortbreite nicht größer als die Busbreite ist und somit ein Datenwort komplett übermittelt werden kann. Die Übertragung eines Datenwortes benötigt mit Adresserzeugung, Adressübermittlung und Rückübermittlung der Daten n Taktperioden T_T. Diese Operation wird für m_i zu übertragende Datenworte D ausgeführt. Daraus ergibt sich für die Kommunikationszeit folgende Gleichung:

$$C_{C_i} = (n \cdot T_T) \cdot m_i \tag{3.45}$$

n ist für eine Kommunikation zwischen zwei Logikelementen kleiner als für eine Kommunikation zwischen zwei Prozessoren bzw. von Logik zu Prozessor, da Prozessoren über komplexe Busse kommunizieren und entsprechend höhere Latenzzeiten zu berücksichtigen sind.

Weil durch die Simulation des Algorithmus auf dem PC anhand eines Beispieldatensatzes die Mengen für D sehr genau bestimmt werden können, ergibt sich auch für die ermittelten Kommunikationszeiten ein zuverlässiger Wert.

Die geschätzten Kommunikationzeiten der Kanten, die für einen Prozessor oder ein Logikelement einen Eingang darstellen, werden diesem zugeordnet. Existieren mehrere Eingänge für einen Prozessor, werden die Kommunikationszeiten der Kanten addiert. Bei Logikelementen wird die größte Kommunikationszeit für die weiteren Berechnungen herangezogen, weil hier die Kommunikation in Hardware erfolgt und parallel abgearbeitet werden kann (siehe Abbildung 3.16).

Aufgrund dieser Eigenschaft und der unterschiedlichen Latenzzeiten bei der Kommunikation ergeben sich entsprechend unterschiedliche Kommunikationskosten bei Logik bzw. bei Prozessorimplementierungen einer Funktion.

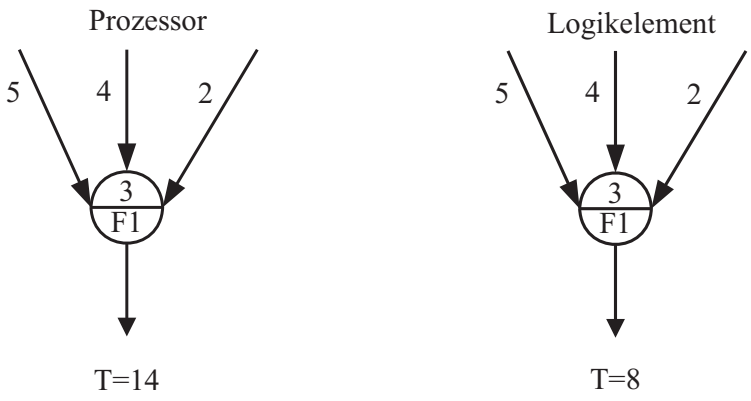

Abbildung 3.16: Max. Periodendauer für Proz. und Logikelement mit mehreren Eingängen

Schätzung des Hardwareaufwandes

Der Hardwareaufwand A_{HW} für eine Funktionsrealisierung als Logik kann sehr einfach durch eine Akkumulation aller vorhandenen Logikelemente (Addierer, Multiplizierer) nach der Art der dort verwendeten Daten (ganze oder reele Zahlen) ermittelt werden.

$$A_{HW} = k \cdot A_{add} + l \cdot A_{addfloat} + ... + n \cdot A_{multfloat} \tag{3.46}$$

Die verwendeten Prozessoren gehen hier mit jeweils 3.500 LC mit in die Berechnung ein.

Schätzung der Entwicklungskosten der Hardwarelösung

Um festzustellen, wie aufwendig die Portierung einer Softwarefunktion in eine Hardwarelösung ist, soll der Portierungsaufwand in Personenmonaten PM_{HW} nach COCOMO II [82] ermittelt werden. Die eingestellten Parameter sind in Kapitel 5 dargestellt.

Aufgestellte Kostenfunktion des Gesamtsystems:

Die Kosten des partitionierten Systems ergeben sich aus der Summe der Einzel-

kosten der verwendeten Funktionen (Gleichung 3.47). Die Kosten einer Funktion werden wiederum aus der gewichteten Addition der im Vorfeld beschriebenen einzelnen Kostenfaktoren gebildet.

$$C_{Ges} = \sum_{i=0}^{N} \left(a_C \cdot (C_{P_i} + C_{C_i}) + b_C \cdot A_{HW_{F_i}} + r_i c_C \cdot PM_{HW_{F_i}} \right) \tag{3.47}$$

Mit $r \in \{0, 1\}$ wird angegeben, ob die Kosten einer Funktion in der Kostenfunktion als Hardwarerealisierung oder als Softwarerealisierung berücksichtigt werden.

3.5 Diskussion

Moderne FPGAs haben die Möglichkeit, aufgrund ihrer Größe und des Einsatzes von NIOS-Softcore Prozessoren, sehr flexibel auf die Anforderungen des Algorithmus zu reagieren. Eine optimale Anpassung des Algorithmusgraphen an den Hardwaregraphen mit Logikelementen und Prozessoren ist somit wesentlich leichter zu finden, als in bisherigen automatischen Partitionierungssystemen. Für die Partitionierung des Algorithmus in Hardware und Software wurde zunächst eine Analyse vorgenommen, um die Ausführungszeiten, die Ausführungshäufigkeiten, die Struktur und die Komplexität der Funktionen im Algorithmus zu bestimmen. Anhand dieser Daten wurde mithilfe des List-Scheduling iterativ bestimmt, welche Funktionen als Logik realisiert werden sollen. Die verbleibenden Funktionen wurden durch den Sarkar Algorithmus zu Clustern zusammengefasst und auf NIOS-Prozessoren alloziert. Für ein vollständig partitioniertes System wurde mithilfe des Simulated Annealing bestimmt, ob die aktuelle Partitionierung angenommen oder wieder verworfen wird. Das Verfahren wird so lange wiederholt, bis die Abbruchbedingungen erfüllt sind. Gewöhnlich ist die Abbruchbedingung die Echtzeitbedingung des Algorithmus, die in den Nebenbedingungen fixiert wurde.

Im folgenden Kapitel wird nun ein Beispielalgorithmus aus der Bildverarbeitung vorgestellt, dessen Realisierung durch die hier erläuterte automatische Partitionierung in einem Hardware-Software Co-Design in Kapitel 5 erläutert wird.

Kapitel 4

Assistenzsystem im KFZ

In diesem Kapitel wird der Algorithmus für ein bildverarbeitendes Assistenzsystem im Kraftfahrzeug vorgestellt. Das Assistenzsystem soll mithilfe einer Stereokameraanordnung Fahrzeuge im Rückraum des eigenen Fahrzeugs erkennen sowie deren Position und Geschwindigkeit schätzen. Die Erzeugung eines echtzeitfähigen Hardware-Software Co-Designs für das vorgestellte Assistenzsystem durch das im Kapitel 3 vorgestellte Partitionierungssystem ist in Kapitel 5 dargestellt.

Mit dem vorgestellten Algorithmus werden erhabene, sich bewegende Objekte und Straßenmarkierungen erkannt. Weiterhin wird ein Verfahren zur Weiterverfolgung der detektierten Objekte und deren Einordnung in eine virtuelle Straßenkarte beschrieben. Mithilfe einer hierarchischen Tiefenkarte wird eine Datenreduktion vorgenommen, um eine schnelle Abarbeitung zu gewährleisten. Das System wurde für Autobahnfahrten konzipiert, um die Position von Fahrzeugen sowie deren Geschwindigkeit in einer Entfernung zwischen $-150m$ und $-10m$ zu ermitteln und den Fahrer bei schnell herannahenden Fahrzeugen frühzeitig zu warnen. Dafür wurden zwei Kameras, im Normalfall der Stereophotogrammetrie, im Heck des Fahrzeugs angebracht, um den rückwärtigen Verkehr zu beobachten.

Damit in den übertragenen Bildern relevante Fahrzeuge zu erkennen sind, müssen sie folgende Eigenschaften besitzen:

- ungefähr senkrechte Fahrzeugkanten
- Erhabenheit gegenüber der Straße
- Relativgeschwindigkeit ist gleich Null oder kleiner gegenüber dem eigenen Fahrzeug.

4.1 Erstellung der Tiefenkarte

Zur Erzeugung der Tiefenkarte wird eine Kantendetektion samt Extraktion der Tiefeninformation in den Stereobildern mittels zeilenbasierter Kreuzkorrelation durchgeführt (Kapitel 4.1.1). Die Erhabenheit der gefundenen Kanten gegenüber der Straße wird mit einem Tiefen-Histogramm, das in Kapitel 4.2.2 beschrieben ist, geprüft. Die Relativgeschwindigkeit wird daraufhin mit einem Zeit-Histogramm getestet. Eine genauere Schätzung der Relativgeschwindigkeit erfolgt in Kapitel 4.4 durch ein Kalman-Filter.

Um die Tiefeninformation zu extrahieren, wird im linken Bild nach Kanten gesucht. Ist eine Kante gefunden, wird versucht im rechten Bild die gleiche Kante wiederzufinden. Dazu wird ein Referenzblock im linken Bild mit einem Suchblock, der im rechten Bild in der gleichen Zeile über das Bild geführt wird, verglichen. Der Referenzblock befindet sich dabei an der Position der Kante. Die Differenz in Pixel zwischen Referenz- und Suchblock an der Stelle maximaler Ähnlichkeit $Q = Q_{max}$ stellt die Disparität dar (siehe auch Kapitel 2.2.1). Die Disparität ist nach Gleichung 2.10 indirekt proportional zur Entfernung. Um den korrekten Block im rechten Bild zu finden, muss die Ähnlichkeit mit dem Referenzblock im linken Bild durch eine Korrelationsfunktion überprüft werden. Die Korrelationsfunktion wird im folgenden Abschnitt vorgestellt.

4.1.1 Ähnlichkeitskriterium

Das Ähnlichkeitskriterium Q wird mithilfe der quadrierten, mittelwertfreien Kreuzkorrelationsfunktion [11], [6] berechnet (Gleichung 4.1). Weitere Verfahren, wie die MAD-Funktion, wurden von Aschwanden verglichen. Die Ergebnisse sind in [11] dargestellt. Die normierte, mittelwertfreie Kreuzkorrelationsfunktion wurde gewählt, da sie unempfindlich gegenüber additiven und multiplikativen Störungen ist. Additive Störungen sind zum Beispiel gravierende Helligkeitsunterschiede zwischen den Bildern. Multiplikative Störungen sind beispielsweise Helligkeitsübergänge innerhalb der Bilder.

$$Q = \frac{\left(\sum_{j=0}^{n-1} \sum_{i=0}^{m-1} \overline{(F(i,j)} \cdot \overline{(P(\xi+i,\eta+j)} \right)^2}{\sum_{j=0}^{n-1} \sum_{i=0}^{m-1} \overline{(F(i,j)^2} \cdot \sum_{j=0}^{n-1} \sum_{i=0}^{m-1} \overline{(P(\xi+i,\eta+j)^2}} \tag{4.1}$$

$\overline{F(i,j)}$- mittelwertfreies Pixel des Suchblockes
$\overline{P(i,j)}$- mittelwertfreies Pixel des Referenzblockes
m, n - Größe des Suchfensters
ξ, η - Verschiebung in x,y-Richtung

Die additiven Störungen, die sich zwischen den beiden Bildern ergeben, werden durch die mittelwertfreie Betrachtung weitgehend ausgeschlossen. Multiplikative Störungen werden durch die Normierung stark reduziert. Als Ergebnis der normierten mittelwertfreien KKF ergeben sich Werte zwischen -1 und $+1$. Wird die Funktion Q auf eine Anzahl von Korrelationen zwischen den Suchblöcken und dem Referenzblock angewendet, ergibt das Maximum dieser Funktion die größte Ähnlichkeit Q_{max} zwischen Referenzblock und dem korrespondierenden Suchblock.

Durch die exakte Ausrichtung der Kameras im Normalfall der Stereophotogrammetrie, kann angenommen werden, dass sich die Objekte in beiden Kamerabildern auf gleicher Höhe befinden. Demzufolge kann die Korrespondenzsuche der KKFMF zeilenorientiert stattfinden [111]. Q muss für einen Referenzblock al-

so nur für eine geringe Anzahl von Suchblöcken berechnet werden, was zu einer Einsparung an Ressourcen und Rechenzeit führt.

In Abbildung 4.1 ist das Ergebnis der KKFMF anhand einer real aufgenommenen Straßenszene und der daraus generierten Tiefenkarte zu sehen.

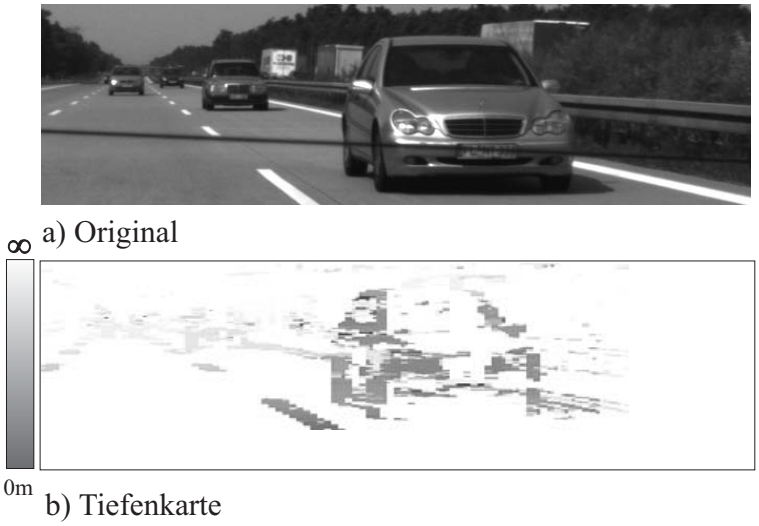

Abbildung 4.1: Originalbild und daraus generierte Tiefenkarte

Bei großen Entfernungen sind die Objekte (z.B. Autos) sehr klein (wenige Pixel) und auch die Disparität nimmt nur geringe Werte an. Im Gegensatz dazu sind nahe Objekte relativ groß, die Disparität nimmt große Werte an. Deshalb soll die Bildauflösung für die Lageberechnung naher Objekte reduziert werden. Die dadurch gleichzeitig verringerte Disparität bedingt die Berechnung einer wesentlich geringeren Zahl an Korrelationswerten bei voller Suche. Dazu werden Ebenen mit zugeordneten Tiefenbereichen verschiedener Auflösung eingeführt [111]. Die Erzeugung der einzelnen Ebenen ist in Anhang B.1 dargestellt.

Weiterführende Untersuchungen zur Merkmalsgewinnung und zur Erstellung der Tiefenkarte durch die KKFMF sind Bestandteil einer anderen Arbeit und werden daher nicht eingehender beschrieben.

4.1.2 Subpixelinterpolation

Nach der Ermittlung der Bildkoordinaten eines Referenzblockes und der Bestimmung der Disparität Δu, lassen sich aus diesen Werten die 3-d-Koordinaten eines Punktes berechnen. Zuvor wird jedoch für die berechneten Disparitäten eine Subpixelinterpolation durchgeführt. Durch die Subpixelinterpolation lässt sich nach [35] die Genauigkeit der Disparität maximal um den Faktor 10 verbessern. Um eine Interpolation durchführen zu können, werden nach Hoschek [46] $2n + 1$ Stützpunkte benötigt. Hier werden drei Stützstellen verwendet. Die erste Stützstelle ist der Ort des Korrelationsmaximums $Q_{max} = Q_0$. Die weiteren Stützpunkte befinden sich jeweils um ein Pixel links und rechts verschoben vom Maximum.

Die Subpixelinterpolation wird nach Gleichung 4.2 durchgeführt. In Anhang B.2 ist die Herleitung der Gleichung dargestellt.

$$\Delta u_s = \Delta u + \frac{\frac{1}{2}(Q_1 - Q_{-1})}{2Q_0 - Q_1 - Q_{-1}} \tag{4.2}$$

Δu_s ist die zu berechnende subpixelgenaue Disparität, Δu die zuvor ermittelte Disparität, Q_0 ist der Korrelationskoeffizient des Maximums, Q_{-1} und Q_1 sind die Korrelationskoeffizienten jeweils ein Pixel links bzw. rechts vom Maximum.

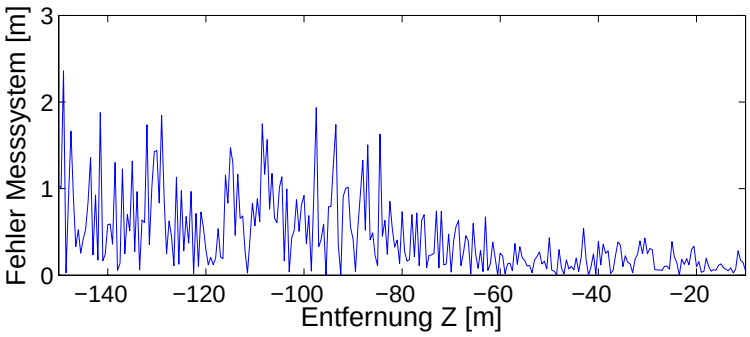

Abbildung 4.2: Fehler des Gesamtsystems durch Messung ermittelt

In Abbildung 4.2 ist der Fehler des Messsystems nach einer durchgeführten Subpixelinterpolation, wie er durch Messung ermittelt wurde, dargestellt. Es ist zu

erkennen, wie sich der Fehler mit sich verringernder Entfernung ebenfalls verringert.

4.2 Objekterkennung

In diesem Abschnitt wird die Detektion erhabener Objekte und deren Zusammenfassung zu Objekthypothesen beschrieben. Dabei wird gezeigt wie erhabene Objekte mithilfe eines Tiefen-Histogramms erkannt werden können und mit einem Zeit-Histogramm weiterverfolgt werden.

4.2.1 Clusterverfahren

Ziel der Clusterung ist es, gegenüber der Straße erhabene, beliebig geformte Objekte zu finden. Interessant sind dabei Objekte, die sich in gleichbleibender Entfernung zum Bezugsfahrzeug befinden oder sich dem Bezugsfahrzeug nähern. Das Bezugsfahrzeug bildet den Koordinatenursprung des zur Bildaufnahme genutzten Stereokamerasystems.

Um Fahrzeuge in einer Stereoszene, basierend auf den gefunden 3-d-Punkten, zu finden, lassen sich zwei Clusterverfahren unterscheiden. Zum einen die Segmentierung der Szene, basierend auf einem parametrisierten Objektmodell [61], und zum anderen ein stochastisches Clusterverfahren [104], [109]. Da zwischen den detektierten Bildkoordinaten $x, y, \Delta u$ und den realen Koordinaten X, Y, Z nach den Gleichungen 2.10-2.12 ein nichtlinearer Zusammenhang besteht, ist es für die Segmentierung nötig, alle Bildkoordinaten in reale Koordinaten zu überführen. Hier können nun die einzelnen 3-d-Punkte anhand der a-priori festgelegten Objektparameter aus der 3-d-Punktewolke zu verschiedenen Objekten zusammengefügt werden. Sind zwei 3-d-Punkte entsprechend der Objektparameter räumlich einander zugehörig, d.h. erfüllen sie die entsprechenden Entfernungskriterien, bilden sie einen Cluster. Zu einem Cluster können weitere Punkte gehören, die ebenfalls die Entfernungskriterien zu dem gerade erzeugten Cluster erfüllen. Jeder Punkt, der die Kriterien erfüllt, kann den Cluster erweitern. Somit können große

Cluster entstehen, die nicht mehr nur die 3-d-Punkte eines Fahrzeuges clustern, sondern auch fehlerhaft detektierte Umgebungspunkte mit einbeziehen. Da die zu erkennenden Fahrzeuge eine große Anzahl unterschiedlicher Objektparameter aufweisen (PKW, Transporter, LKW), ist eine zuverlässige Detektion nur mit großem algorithmischem Aufwand möglich.

Die stochastischen Verfahren sind für den vorgestellten Anwendungsfall geeignetere Verfahren. In [47] und [48] wird ein Kondensationsalgorithmus vorgestellt, den Suppes [104] zur Hindernisdetektion in fahrerlosen Transportfahrzeugen aufgegriffen hat. Mit diesem Algorithmus können alle signifikant über den Boden hinausragenden Objekte, unabhängig von ihrer Form, erkannt werden.

Der Kondensationsalgorithmus ermittelt aus einer Verteilung von 3-d-Punkten über der X, Z-Ebene und der Kenntnis der Unsicherheiten jedes einzelnen Punktes eine Funktion $\Pi(X, Z)$, die ein Maß für die Existenz eines Objektes an der Position (X, Z) angibt. Die Funktion $\Pi(X, Z)$ wird als Potentialmaßfunktion bezeichnet. Die Wahrscheinlichkeitsverteilung eines 3-d-Punktes wird dabei auf die X, Z-Ebene projiziert. Liegt dort bereits die Wahrscheinlichkeitsverteilung eines anderen 3-d-Punktes, wird der Wert der Verteilung an der entsprechenden Stelle um den Betrag der Wahrscheinlichkeit erhöht. Durch dieses *Aufkondensieren* der unterschiedlichen Verteilungen entstehen in der X, Z-Ebene eine Menge von Maxima. Das Potentialmaß gibt dabei an, mit welcher Wahrscheinlichkeit an einer bestimmten Position ein erhabenes Objekt zu vermuten ist.

Das Gesamtpotentialmaß an einer Position der X, Z-Ebene setzt sich nach Gleichung 4.3 additiv aus den Einzelpotentialmaßen der 3-d-Punkte zusammen.

$$\Pi(X, Z) = \sum_i \Pi_i(X, Z) \tag{4.3}$$

4.2.2 Histogrammgenerierung

Im hier vorgestellten System werden zwei Histogramme [53] generiert, in denen verschiedene Potentialmaße erzeugt werden. Das erste Histogramm resultiert aus der Tiefenkarte und wird Tiefen-Histogramm genannt. Das zweite Histogramm

dient der zeitlichen Weiterverfolgung erkannter Potentialmaßmaxima aus dem Tiefen-Histogramm. Hiermit können Objekte detektiert werden, die sich über die Zeit im gleichen Abstand zum eigenen Fahrzeug befinden oder sich dem Fahrzeug nähern.

Durch die endliche Ausdehnung eines Pixels in der Kamera, der Erzeugung des hierarchischen Tiefenbereiches und der Nutzung von 16x1 Blöcken für die Korrelation (siehe Abschnitt 4.1.1) ist der Flächeninhalt, über den die Wahrscheinlichkeit aufgetragen wird, nicht mehr infinitisimal klein (Quantisierung siehe Abbildung B.4 im Anhang). Durch die Größe der Fläche kann angenommen werden, dass die Wahrscheinlichkeit, dass ein gefundener 3-d-Punkt X_0, Z_0 innerhalb dieser Fläche liegt, gegen eins geht. Dadurch vereinfacht sich Gleichung 4.3 zu Gleichung 4.4.

$$\Pi(X, Z) = \begin{cases} 1 & \text{falls } X = X_0 \wedge Z = Z_0 \\ 0 & \text{sonst} \end{cases} \tag{4.4}$$

Tiefen-Histogramm

Im Tiefen-Histogramm werden die Potentialmaße der Tiefenkarte in der x, Δu-Ebene gebildet. Es ist nicht notwendig die Bildkoordinaten in reale 3-d-Koordinaten umzurechnen. Das Potentialmaß lautet also $\Pi(x, \Delta u)$. Dazu werden die detektierte x-Koordinate und die Disparität Δu (nicht subpixelgenau) mit der dazugehörigen Hierarchie-Ebene als Adressierung für eine Speicherzelle genutzt. Die Speicherzelle bildet einen Akkumulator, in dem die entsprechenden Potentialmaße nach Gleichung 4.3 additiv zusammengefasst werden. In einem mit dieser Speicherzelle assoziierten Speicherbereich werden die subpixelgenaue Disparität und die y-Koordinate geschrieben, wobei die beiden assoziierten Speicherzellen bei jedem neuen Eintrag überschrieben werden und somit später nur die subpixelgenaue Disparität und die y-Koordinate des letzten Eintrags Verwendung finden. In Abbildung 4.3 ist das Tiefen-Histogramm für die Tiefenkarte aus Abbildung 4.1 zu sehen.

Für den Algorithmus interessante Objekte besitzen ein Potentialmaß, das sich über einem fest vorgegebenen Schwellwert $\Pi_{Schwelle}$ befindet. Der Schwellwert

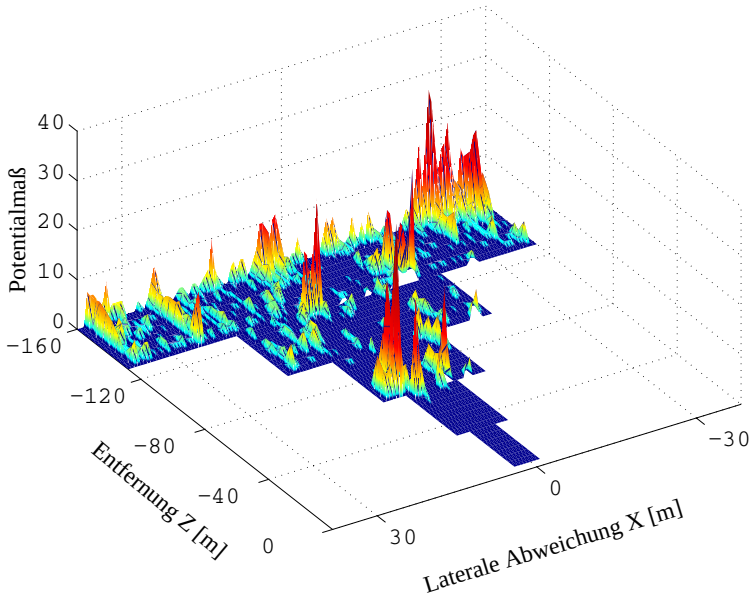

Abbildung 4.3: Aus der Tiefenkarte (Abb. 4.1) generiertes Tiefen-Histogramm

ist über das gesamte Bild konstant. Da sich nähernde Fahrzeuge im Bild größer werden und damit auch mehr Zeilen auf dem Bildsensor beanspruchen, vergrößert sich das Potentialmaß für ein Fahrzeug je näher es sich am Kamerasystem befindet. Um über das gesamte Bild vergleichbare Größenverhältnisse der Fahrzeuge zu erreichen, wird das Potentialmaß auf die Hierarchie-Ebene, in der es sich befindet, angepasst. Gültige Potentialmaße erfüllen folgende Ungleichung 4.5:

$$\Pi_{Schwelle} \leq \frac{\Pi(x, \Delta u)}{2^{Ebene}} \tag{4.5}$$

In welcher Ebene sich ein Potentialmaß befindet hängt von seiner Disparität hab. Die Erzeugung der Ebenen ist in Anhang B.1 dargestellt.

Zeit-Histogramm

Das Zeit-Histogramm dient der Weiterverfolgung der Objekte und wird wie das Tiefen-Histogramm adressiert. Hier wird das Potentialmaß jedoch nicht mehr aus der Summe der Einzelpotentiale gebildet, sondern aus dem zeitlichen Verhalten der gültigen Potentialmaße in aufeinander folgenden Bildern. Die Posi-

tionen aktuell gültiger Potentialmaße des Tiefen-Histogramms werden auf das Zeit-Histogramm übertragen. Befindet sich in einer Suchmaske um die aktuelle Position ein Zeit-Potentialmaß aus den vorhergehenden Bildern, wird der Wert dieses Potentialmaßes entsprechend erhöht und an die jetzt aktuelle Position geschrieben.

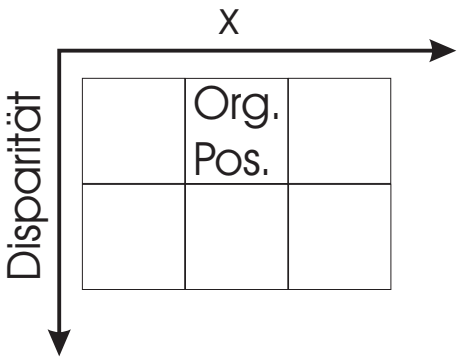

Abbildung 4.4: Suchmaske für Suche im Zeit-Histogramm

Das Potentialmaß im Zeit-Histogramm zeigt das Alter eines Punktes an. Das Alter gibt dabei an, wie oft ein Objekt in den vorangegangenen Bildern erkannt wurde. Eine Suchmaske, wie sie hier verwendet wird, ist in Abbildung 4.4 zu sehen. Mit dieser Suchmaske werden nur Objekte detektiert, die sich über mehrere Bilder im gleichen Abstand zum eigenen Fahrzeug befinden oder sich ihm nähern. Besaß das Potentialmaß bereits im vorhergehenden Bild eine Clusternummer, wird diese Clusternummer auch in das neue Bild übernommen.

Werden innerhalb der Suchmaske mehrere mögliche Potentialmaße gefunden, wird das höchste verwendet. Alle Potentialmaße im Zeit-Histogramm ohne assoziierten Eintrag im Tiefen-Histogramm werden dekrementiert, da das dort vermutete Objekt nicht mehr existiert. Erreicht ein Zeit-Potentialmaß einen Schwellwert, gilt dieses Potentialmaß als zu einem erhabenen sich bewegenden Objekt zugehörig. Hierbei tritt eine Verzögerung bei der Detektion eines erhabenen, sich bewegenden Objektes auf. Ist das Objekt detektiert und bewegt es sich kontinuierlich, zeigt das Zeit-Histogramm immer die aktuelle Position des Objektes an. Mithilfe des Zeit-Histogramms können sich entfernende Objekte (langsame Fahrzeuge, Brücken, Schilder) oder nur sporadisch auftauchende Objekte (Fehldetektionen und Rauschen) gefiltert werden. Ein aus dem Tiefen-Histogramm aus

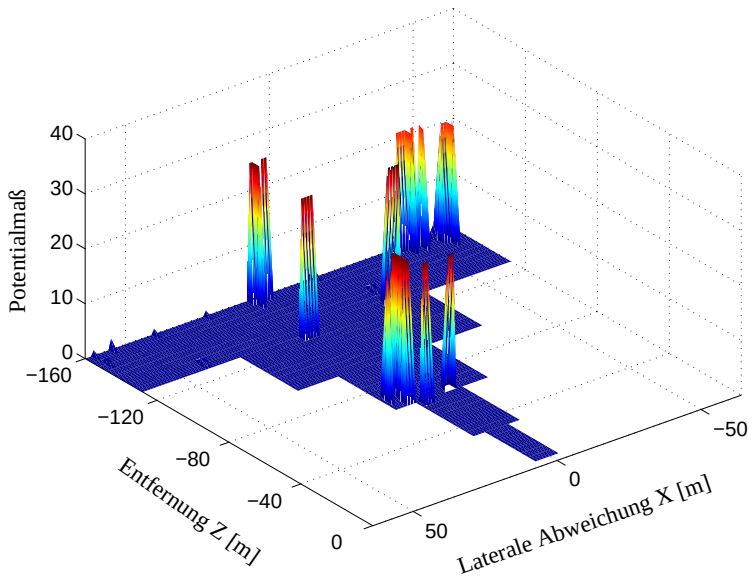

Abbildung 4.5: Zeit-Histogramm aus Tiefen-Histogramm generiert

Abbildung 4.3 generiertes Zeit-Histogramm ist in Abbildung 4.5 zu sehen.

4.2.3 Clusterung

Die Clusterung [54] erfolgt auf Basis des Zeit-Histogramms. Dabei werden die Potentialmaße des Zeit-Histogramms, die oberhalb des Schwellwertes liegen, zu einzelnen Fahrzeugclustern zusammengeführt. Fahrzeuge erscheinen im Bildsensor größer, je näher sie sich an der Kameraanordnung befinden. Das heißt die Abstände zwischen der linken und der rechten Kante eines Fahrzeuges vergrößern sich im Bild. Mit der hierarchischen Aufteilung des Tiefenbereichs und der damit verbundenen Hierarchie-Ebenengenerierung ist es möglich, die nichtlineare Beziehung zwischen Bildkoordinaten und 3-d-Koordinaten (siehe Gleichungen 2.11 und 2.10) nahezu auszugleichen und somit die Darstellungsbreite eines Fahrzeuges im Bild in einem konstanten Bereich zu halten. In Abbildung 4.6 ist der Bereich dargestellt, den ein $1,80m$ breites Fahrzeug im Histogramm einnimmt. Es ist zu erkennen, dass die laterale Ausdehnung des Fahrzeuges über den gesamten Tiefenbereich nahezu konstant bleibt. Durch die Reduzierung der Auflösung befinden

sich die Kanten eines Fahrzeuges immer sehr nah beieinander und können somit leicht detektiert werden [97].

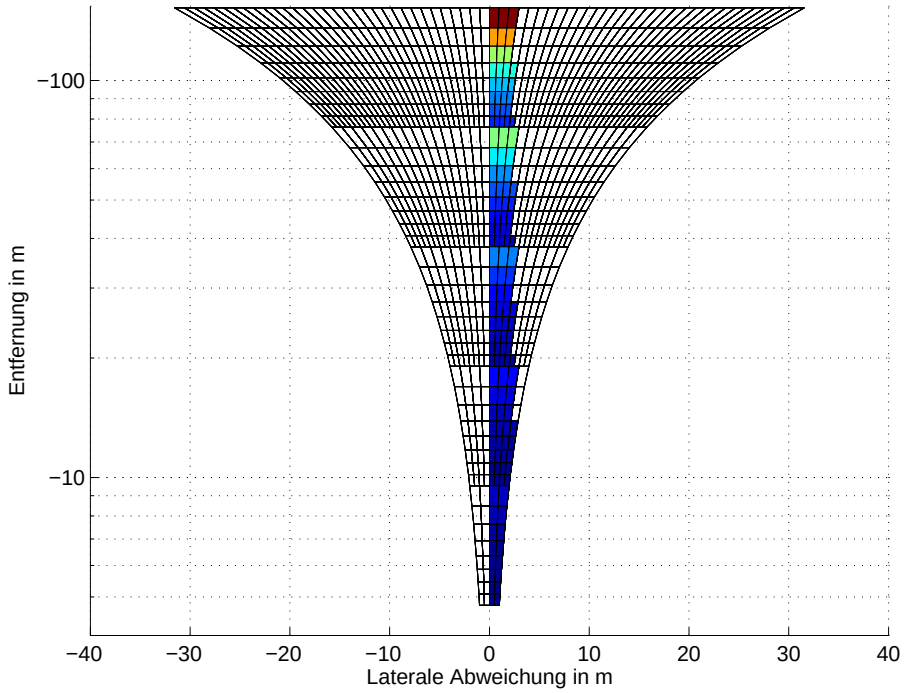

Abbildung 4.6: $1,80\ m$ breites Fahrzeug im Histogramm

Geclustert wird grundsätzlich direkt mit den Koordinaten aus dem Zeit-Histogramm, wobei eine Anpassung bei Überschreitung einer Ebenengrenze vorgenommen werden muss. Die Disparitäten und x-Koordinaten im Zeit-Histogramm sind ebenenabhängig und müssen in korrekte Werte umgerechnet werden, um daraus wiederum die 3-d-Punkte genau zu berechnen. Dazu werden die Gleichungen 4.6 und 4.7 verwendet.

$$x = \left(\frac{m}{2} + (x_{Ebene} \cdot s)\right) \cdot 2^{Ebene} \tag{4.6}$$

$$\Delta u = (\Delta u_{min} + \Delta u_{Ebene}) \cdot 2^{Ebene} \tag{4.7}$$

Wegen der Größe des Such- und Referenzblockes müssen $OFFSET$-Werte für die Umrechnung benutzt werden. In Gleichung 4.6 ist der $OFFSET = \frac{m}{2}$ und stellt

die Mitte des Referenzblockes dar. m ist dabei die Länge des Referenzblockes. Für einen Block mit $16x1$ Pixeln ist $\frac{m}{2}$ also 8. s ist die Differenz zwischen den Mittelpunkten zweier benachbarter Referenzblöcke. Δu_{min} in Gleichung 4.7 repräsentiert die minimale Disparität. Da hier nur Fahrzeuge bis $-150m$ erkannt werden sollen, können alle Disparitäten, die kleiner als 8 sind, aus den Histogrammen ausgeblendet werden. Δu_{min} ist demzufolge 8. Die Clusterung startet bei den hohen Disparitäten und wandert dann zu den niedrigen Disparitätswerten. Dabei können auch verschiedene Fahrzeuge zu einem Cluster zusammengefasst werden, wenn die Distanz zwischen ihnen sehr gering ist. Beim Überschreiten einer typischen Fahrzeugbreite wird automatisch eine Trennung der Cluster vorgenommen. Ein Ergebnisbild der Clusterung, basierend auf der Tiefenkarte aus Abbildung 4.1 und den Histogrammen der Abbildungen 4.3 und 4.5, ist in Abbildung 4.7 zu sehen. Die Clusternummer wird auf Basis des ältesten Punktes eines Clusters vergeben. Diese Nummer wird für alle Punkte eines Clusters in das Zeit-Histogramm eingetragen.

Abbildung 4.7: Ergebnis der Clusterung

Die 3-d-Berechnung erfolgt hier nur für den Mittelpunkt des Clusters. Der Mittelpunkt wird durch Mittelwertbildung aller am Cluster partizipierenden Potentialmaße und ihrer subpixelgenauen Disparitäten berechnet.

4.3 Spurerkennung und virtuelle Straßenkarte

Die Spurerkennung dient der Zuordnung der Fahrzeuge auf eine bestimmte Straßenspur, um zu detektieren, welche Fahrzeuge sich auf der Überholspur befinden.

4.3.1 Verfahren zur Spurerkennung

Zur automatischen Spurerkennung existieren bereits mehrere verschiedene Ansätze. Einige davon werden hier kurz vorgestellt.

Das ARCADE System [59] ist ein robustes System zur Detektion des Straßenverlaufes. Dabei werden die Punkte der detektierten Kanten genutzt. Dieses System arbeitet noch mit einem Rauschen der Punkte von 50%. Eine andere Möglichkeit ist in [60] beschrieben. Hier werden parabolische Kurven als Modell zugrunde gelegt, die dann die Straßengrenzen auf flachem Grund repräsentieren. Ein Ansatz, der die Hough-Transformation mit einem Curve-Fitting Algorithmus verbindet, wird in [117] vorgestellt. Hiermit lässt sich auch eine Spurerkennung in Kurven durchführen. Dabei wird das Bild in verschiedene Entfernungsbereiche aufgeteilt. In jedem Bereich wird eine Spurerkennung mittels Hough-Transformation durchgeführt. Die einzelnen erkannten Fahrspuren werden dann zu einer Fahrspur zusammengeführt. Eine weitere Möglichkeit der Spurerkennung mit Hilfe der Kantendetektion wird in [15] beschrieben.

4.3.2 Hough-Transformation mit Tiefenkarte

Im vorgestellten System soll die Spurerkennung aus den Daten der Tiefenkarte bzw. aus den Daten des Tiefen-Histogramms erfolgen. Da die Spurerkennung zunächst nur für Autobahnen angewendet werden soll und Autobahnspuren eine geringe Kurvigkeit aufweisen, können sie im Nahbereich als gerade angenommen werden. Somit brauchen komplexe Ansätze, wie in [59] oder [60] beschrieben, nicht verwendet zu werden. Hier ist es einfacher, insbesondere um die Echtzeitbedingungen einzuhalten, eine Hough-Transformation [106] anzuwenden. Mit der Hough-Transformation können Punkte detektiert werden, die auf einer Geraden liegen.

Die Hough-Transformation erfolgt mit den Daten des Tiefen-Histogramms. Wie bereits im Kapitel 4.2.3 beschrieben, kann die nichtlineare Beziehung zwischen Bildkoordinaten und 3-d-Koordinaten durch die Hierarchie-Ebenen nahezu ausgeglichen werden. Daher erscheinen die Geraden im Tiefen-Histogramm ebenfalls als

Gerade. Es ist also nicht notwendig die Bildkoordinaten in 3-d-Koordinaten umzurechnen. Die Potentialmaße der Straßenmarkierungen sind im Tiefen-Histogramm eindeutig in den Hierarchie-Ebenen 1,2 und 3 zu erkennen, also zwischen den Disparitäten Δu 16 und 120 (entspricht einer Entfernung von $Z = -76m$ bis $-10m$), deshalb wird nur dieser Entfernungsbereich für die Hough-Transformation benutzt. Um die flachen Objekte im Tiefen-Histogramm zu erkennen, wird hier ein neuer Schwellwert für das Potentialmaß eingeführt, der aber wesentlich niedriger als der Schwellwert für erhabene Objekte ist.

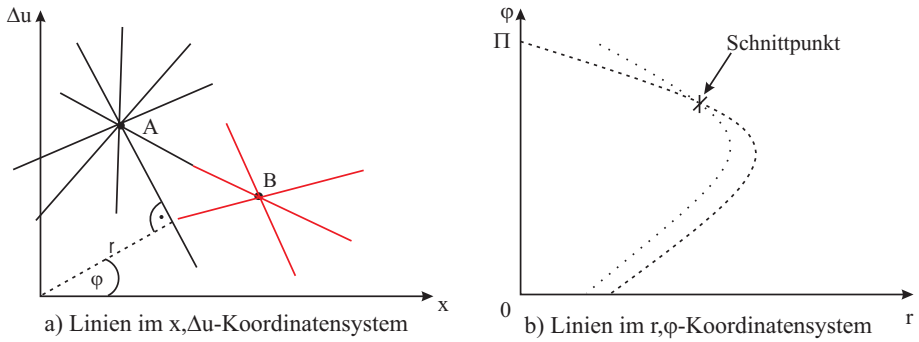

Abbildung 4.8: Hough-Transformation

Um festzustellen, welche Punkte auf einer Geraden liegen, wird für jeden als flach erkannten Punkt ein Geradenbüschel erzeugt (Abbildung 4.8.a). Als Basis für die Geradenbeschreibung im $x, \Delta u$- Koordinatensystem dient die Hesse'sche Normalform (Gleichung 4.8).

$$r = x \cdot cos\varphi + \Delta u \cdot sin\varphi \tag{4.8}$$

Dabei dient der Abstand r der Geraden zum Koordinatenursprung und der daraus resultierende Winkel φ der Beschreibung dieser Geraden. Trägt man r und φ in ein entsprechendes r, φ-Koordinatensystem ein, so erhält man genau einen Punkt darin.

Falls nur ein Punkt A (Abbildung 4.8.a) erkannt wurde, ist der Anstieg der Geraden durch diesen nicht bekannt. Es ist daher notwendig, die Transformation vom $x, \Delta u$- Koordinatensystem in den r, φ-Raum für eine größere Anzahl von Geraden mit einem gemeinsamen Punkt A, jedoch verschiedenen Anstiegen, vorzunehmen.

Wenn man dieses Geradenbüschel in den r, φ-Raum transformiert, ergibt sich eine Kurve wie in Abbildung 4.8.b dargestellt. Generiert man auch für Punkt B ein Geradenbüschel, ergibt sich eine gemeinsame Gerade im $x, \Delta u$-Raum. Für alle kollinearen Punkte ergibt sich genau diese Gerade mit dem gleichen Radius r und dem Winkel φ. Dadurch bildet sich im r, φ-Raum für alle Geradenbüschel von allen kollinearen Punkten ein Schnittpunkt. Mit Hilfe eines Akkumulators (Abbildung 4.9) kann die Anzahl der Kurvenschnitte im r, φ-Raum detektiert werden. Ein hoher Eintrag im Akkumulator bedeutet eine hohe Anzahl von kollinearen Punkten und damit eine Gerade im $x, \Delta u$-Raum. Wird ein bestimmter Schwellwert überschritten, charakterisiert die Position des Akkumulatoreintrages im r, φ-Raum die Koordinaten einer Geraden. Führt man für diese Koordinaten nun eine Rücktransformation in das $x, \Delta u$-Koordinatensystem durch, kann man entlang dieser Linie flache Punkte finden, die zu einer Geraden gehören und als flache Cluster markiert sind.

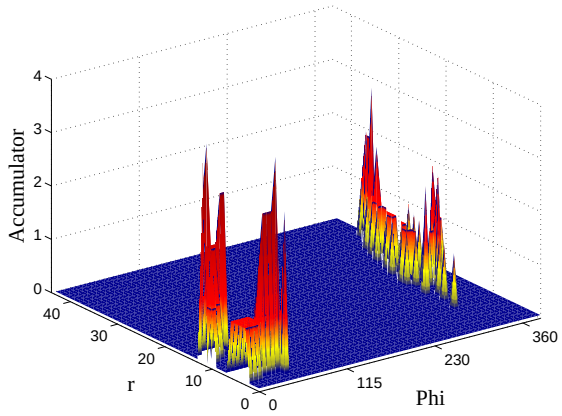

Abbildung 4.9: Akkumulator für die Hough-Transformation

Straßenmarkierungen sind auf Autobahnen längs zur Fahrtrichtung angeordnet, deshalb kann die Generierung der Geradenbüschel auf Winkel zwischen 0° und 50° für die Straßenmarkierungen links des Kamerasystems und auf Winkel zwischen 310° und 360° für Straßenmarkierungen rechts vom Kamerasystem beschränkt werden. Die Auflösung der Geradenwinkel wird auf 5,625° festgelegt (entspricht einer $\frac{1}{64}$-Teilung des Einheitskreises), um den Speicheraufwand gering zu halten und den Einfluss von Winkelabweichungen zu verringern. Aufgrund dieser Be-

schränkungen ergibt sich ein Akkumulatorbild wie in Abbildung 4.9. Dabei sind die Einträge auf zwei Gebiete konzentriert und Lücken gegenüber Abbildung 4.8.b treten auf.

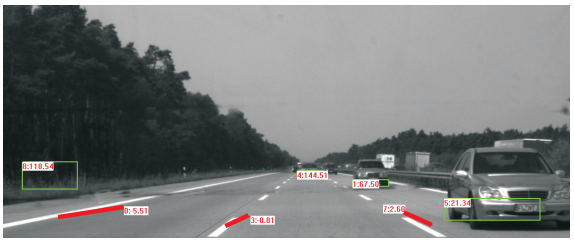

Abbildung 4.10: Spurerkennung mit der Hough-Transformation

Die gefundenen Straßenmarkierungen im $x, \Delta u$-Raum werden in 3-d-Koordinaten umgerechnet und stellen die Eingangsdaten für die virtuelle Straßenkarte im folgenden Abschnitt dar. Das Ergebnisbild der Hough-Transformation ist in Abbildung 4.10 zu sehen.

4.3.3 Virtuelle Straßenkarte

Relevante Fahrzeuge sind jene, mit denen eine Kollisionsgefahr bei einem Spurwechsel besteht. Diese Fahrzeuge weisen eine sehr hohe Relativgeschwindigkeit auf oder sie befinden sich auf der Nachbarspur. Deshalb ist eine spurgenaue Lokalisierung von erkannten Fahrzeugen notwendig.

Der zurückgelegte Weg des eigenen Fahrzeuges sowie der erkannten Fahrzeuge im Rückraum werden in eine lokale Karte eingetragen. Mit der relativen Lage der Fahrzeuge zur zurückgelegten Wegstrecke wird ihre Spurzugehörigkeit ermittelt. Unter Nutzung der eigenen Fahrzeugdaten ist eine ausreichend genaue Ermittlung der zurückgelegten Wegstrecke möglich.

Zur Beschreibung des gefahrenen Weges müssen die zurückliegenden Positionsänderungen des eigenen Fahrzeuges bekannt sein. Durch Aneinanderreihen von Positionsänderungen lässt sich die Wegstrecke rekonstruieren (Abbildung 4.11). Dieser Vorgang wird als Dead-Reckon-Methode [61] bezeichnet. Eine Positionsänderung lässt sich durch die Verschiebung um eine gerade Wegstrecke d und eine

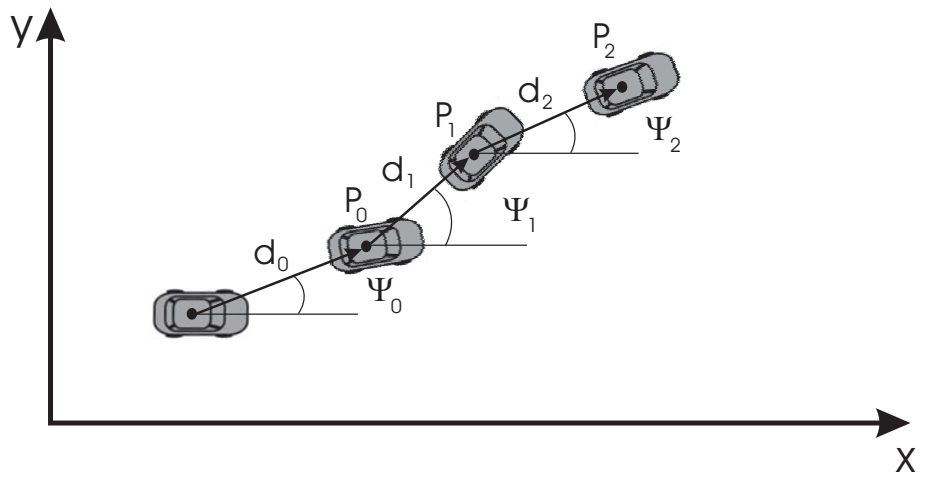

Abbildung 4.11: Fahrzeugposition mit Dead-Reckon-Methode [61]

Fahrzeugdrehung ψ ermitteln. d und ψ werden aus der Differenz zwischen den entsprechenden Messwerten zweier aufeinander folgender Messpunkte gebildet. Das Messintervall stellt die Bildfrequenz des Aufnahmesystems dar (hier $25Hz$). Die Wegstrecke resultiert aus der Eigengeschwindigkeit des Fahrzeuges, der Winkel ψ ist durch ein am Fahrzeug vorhandenes ESP gegeben. Beide Werte werden z.B. über den CAN-Bus bereitgestellt. In Kombination des eigenen zurückgelegten Weges mit der Lageänderung der detektierten Straßenmarkierungen kann die Form und Lage der Straße ermittelt werden. Damit lässt sich eine virtuelle Straßenkarte generieren, in der die erkannten Fahrzeuge eingetragen sind.

Da der zu überwachende Entfernungsbereich im vorliegenden Fall -150 bis $-10m$ beträgt, wurde die virtuelle Straßenkarte auch nur für diesen Bereich erzeugt. Der Entferungsbereich wurde in Halbmeterintervalle eingeteilt, um die Straße für Eigengeschwindigkeiten ab $50km/h$ korrekt abtasten zu können (bei $50km/h$ und 25 $Bilder/s$ legt ein Fahrzeug $0,54$ $m/Bild$ zurück). Die bereits eingetragenen Straßenmarkierungen werden je Bild um die gefahrene Wegstrecke nach hinten verschoben. Innerhalb der gefahrenen Wegstrecke werden die im aktuellen Bild gefundenen Straßenmarkierungen eingetragen.

Die verschobenen Straßenmarkierungen werden um die Fahrzeugdrehung ψ lateral nach links bzw. rechts nach Gleichung 4.9 um ΔX gedreht.

$$\Delta X_i = Z_i \cdot tan\psi \tag{4.9}$$

ΔX_i ist dabei die laterale Verschiebung einer bereits erkannten Straßenmarkierung um den Winkel ψ an der Stelle Z_i.

4.4 Kalman-Filter

Um das dynamische Verhalten der Fahrzeuge im Rückraum analysieren zu können ist es nötig, Störeinflüsse aus der Umgebung, statistische Schwankungen und das Rauschen des Bildaufnahmesystems zu minimieren [78]. Ziel ist es, die Position des Fahrzeuges im nächsten Bild zu schätzen, um Fehldetektionen zu verhindern und Messfehler zu glätten. Weiterhin soll die Geschwindigkeit der Fahrzeuge zuverlässig ermittelt werden, wodurch Gefahrensituationen rechtzeitig erkannt werden können.

Das Kalman-Filter [71] ist ein Algorithmus, der diesen Anforderungen genügt. Mit ihm lassen sich die Geschwindigkeiten der Objekthypothesen (Fahrzeuge) ohne Verzögerung ermitteln.

4.4.1 Aufbau und Eigenschaften

Ein Kalman-Filter ist nach [74] ein optimaler, rekursiver Datenverarbeitungsalgorithmus. Seine Eigenschaften sind dadurch gekennzeichnet, dass es für eine breite Klasse von Problemen das optimale Filter ist und trotzdem eine lineare Struktur aufweist. Es verwendet alle Messwerte entsprechend ihrer Genauigkeit, um die gesuchte Größe zu ermitteln. Dafür werden gewisse a-priori Kenntnisse über das dynamische Verhalten des Systems und der Messfehler, über das statistische Verhalten der Messstörungen und der unbekannten Systemstörgrößen sowie alle verfügbaren Kenntnisse über die Anfangswerte benötigt. Der wesent-

liche Vorteil dieses Verfahrens besteht darin, dass der Schätzwert pro Zeitpunkt nur anhand des zu diesem Zeitpunkt vorliegenden Messwertes und des jeweils vorherigen Schätzwertes sowie dessen Varianzen berechnet wird. Also muss auch nur der vorherige Zustand gespeichert werden. Aufgrund der rekursiven Struktur des Algorithmus geht jedoch nicht nur der aktuelle Wert in das Schätzergebnis ein, sondern es werden auch alle zurückliegenden Werte wirksam. Da nur der vorhergehende geschätzte Zustand herangezogen wird, ist die Berechnung sehr effizient. So eignet sich das Kalman-Filter besonders für die hier geforderte Echtzeitanwendung. Zur Kennzeichnung der statistischen Eigenschaften der verwendeten Rauschprozesse reichen die ersten beiden Momente, Erwartungswert und Kovarianz, aus, wenn sie von weißem gaußverteiltem Rauschen *getrieben* werden. Der Vorteil eines Kalman-Filters gegenüber einem Wiener-Filter besteht in der Nichtstationarität. Aufgrund die Nichtstationarität besitzt das Kalman-Filter eine zeitvariante Struktur. Kalman-Filter werden im Gegensatz zu Wiener-Filtern im Zustandsraum durch rekursive Gleichungen beschrieben und sind in ihrer zeitdiskreten Form auf Digitalrechnern zu implementieren. Das Systemmodell des zeitdiskreten Kalman-Filter-Algorithmus lässt sich durch die Gleichungen 4.10 bis 4.14 beschreiben. Die Herleitung der Gleichungen wurde in [78] dargestellt und soll hier nicht weiter betrachtet werden.

$$K_k = P_k \cdot C_k^T (C_k P_k C_k^T + R_k)^{-1} \tag{4.10}$$

$$x_k^{\#} = x_k^* + K_k(y_k - C_k x_k^*) \tag{4.11}$$

$$P_k^* = P_k - K_k C_k P_k) \tag{4.12}$$

$$x_{k+1}^* = A_k x_k^{\#} + B_k u_k) \tag{4.13}$$

$$P_{k+1} = A_k P_k^* A_k^T + Q_k \tag{4.14}$$

x_k ist ein Vektor von Zuständen und u_k ein Vektor deterministischer Eingangsgrößen (fahrzeugeigener Bewegungsgrößen). A_k stellt die Zustandsübergangsfunktion dar und B_k repräsentiert die Eingangsmatrix. y_k ist der Vektor der Beobachtungen (Größen aus der Clusterung), C_k ist die Beobachtungsmatrix. Die interne Struktur des Kalman-Filters nach [96] ist in Abbildung 4.12 dargestellt. Mit der Korrekturmatrix K_k werden die geschätzten Zustandsgrößen verbessert.

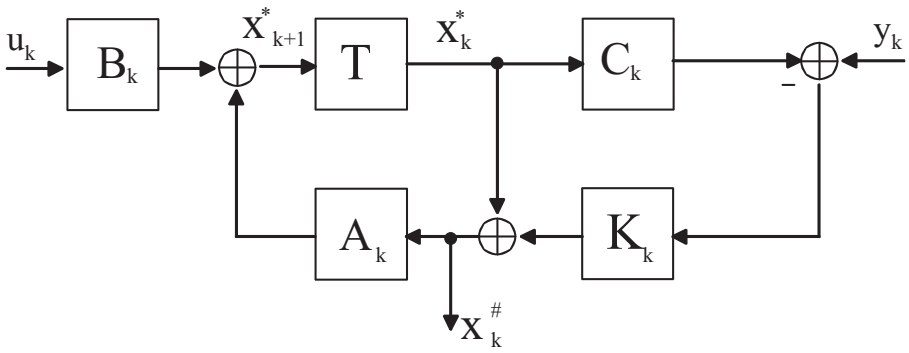

Abbildung 4.12: Interne Struktur eines Kalman-Filters

Über die Korrekturmatrix K_k wird die Differenz zwischen tatsächlichem Messwert yk und dem aus x^*_{k+1} abgeleiteten Schätzwert zu x^*_k addiert, was dann $x^\#_k$ ergibt. Der a posteriori Schätzwert $x^\#_k$ besitzt eine geringere Varianz als der a-priori Schätzwert x^*_k und wird deshalb für eine weitere Verarbeitung genutzt.

Zur Bestimmung der Messfehlervarianz R_k werden die Standardabweichungen in $Z-$ und $X-$Richtung des Stereokamerasystems herangezogen. P_k sind die Schätzfehlerkovarianzmatrizen des Systems. Die Kovarianzmatrix Q_k bestimmt die Trägheit des Systems. Niedrige Kovarianzen führen zu einem trägen Filterverhalten, d.h. die prädizierten Werte folgen den realen Eingangswerten nur langsam, während bei hohen Kovarianzen schnelle Anpassungen möglich sind.

Für jedes gefundene Objekt wird ein eigenes Kalman-Filter gestartet. Die Zuordnung bereits bestehender Kalman-Filter Objekte zu aktuell gefunden Objekten wird in Anhang B.3 beschrieben. Die Dimensionierung der Kalman-Filter ist in Anhang B.4 dargestellt.

4.4.2 Ergebnisse aus der Verwendung der Kalman-Filter

Der vorgestellte Kalman-Filter erzeugt, mit der in Anhang B.4 ermittelten Dimensionierung der Kalman-Filter-Matrizen, Signalverläufe wie in Abbildung 4.13 dargestellt.

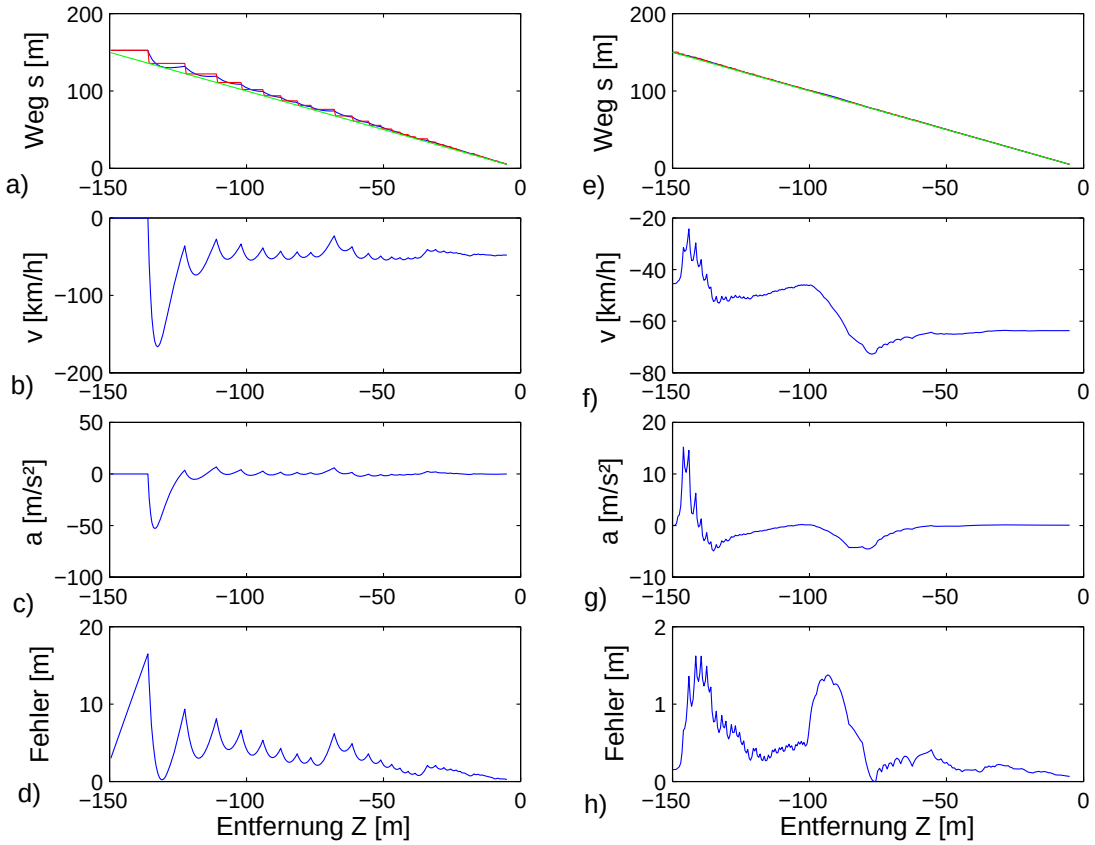

Abbildung 4.13: Vergleich der nicht optimierten (a-d) und optimierten (e-h) Ermittlung der Objektbewegungsparameter

Die Abbildungen 4.13 a-d gehören zu einem nicht optimierten Kalman-Filter-System ohne subpixelinterpolierte Eingangsentfernungswerte. Die Kovarianzmatrix Q_k wurde nicht adaptiv ausgelegt. Weiterhin wurden weder die Schätzfehlerkovarianz P_k noch die Geschwindigkeit des am nächsten am Kamerasystem befindlichen Fahrzeuges als Startwerte für das neue KF-Objekt übernommen. In den Abbildung 4.13 e-f sind die Signalverläufe eines optimierten Systems mit subpixelinterpolierten Eingangswerten sowie adaptiver Kovarianzmatrix und Schätz-

fehlerkovarianz zu sehen.

In beiden Systemen (optimiert und nicht optimiert) wurde als Geschwindigkeit des KF-Objektes $45,5km/h$ gewählt. Bei einer Entfernung von $-100m$ wurde das KF-Objekt im optimierten System auf $63,6km/h$ beschleunigt. Für das optimierte KF-Objekt ist zur Initialisierung der Endwert der Schätzfehlerkovarianz P_k des nicht optimierten Objektes und dessen prädizierte Geschwindigkeit gewählt worden. Die Bildfrequenz beträgt in beiden Fällen $25Hz$.

Der Vergleich zwischen optimiertem und nicht optimiertem KF-System zeigt eine deutliche Verbesserung des Schätzfehlers und der prädizierten Geschwindigkeit des KF-Objektes. Während der Fehler in Abbildung 4.13.d) zu Beginn der Prädiktion $17m$ beträgt, fällt er in 4.13.h) auf unter $1,5m$. Der Fehlerausschlag in $-100m$ Entfernung entsteht durch die Beschleunigung des KF-Objektes und das damit verbundene Herauslaufen der Prädiktion. Durch die adaptive Anpassung der Kovarianzen in der Kovarianzmatrix wird dieser Fehler aber sehr schnell zurückgeführt. Daraus ergibt sich wiederum eine schnelle Anpassung der KF-Prädiktion an die reale Geschwindigkeit des KF-Objektes. Diese schnelle Anpassung ist bereits beim Einschwingvorgang zu beobachten. Während des Einschwingens (10 Bilder) wurde σ^2_{Q0} verwendet, um dann auf σ^2_{Q1} zu wechseln. Beim Herauslaufen der prädizierten Werte wurde auf σ^2_{Q2} gewechselt, um dann, nachdem für das KF-Objekt eine Anpassung vorgenommen wurde, wieder σ^2_{Q1} zu verwenden. Weitere Details zur Bestimmung von σ^2 sind in Anhang B.4 zu finden.

Für die Ermittlung der korrekten Entfernung und Geschwindigkeit werden etwa 25 Entfernungsmessungen benötigt. Es vergeht also eine Sekunde bis das Kalman-Filter eine zuverlässige Schätzung des Bewegungsverlaufes eines detektierten Objektes liefert.

4.5 Diskussion

In diesem Kapitel wurden die implementierten Algorithmen vorgestellt. Es wurde ein Verfahren präsentiert, mit dem man aus Bildern von, im Normalfall der Stereo-

photogrammetrie, angeordneten Kameras eine Tiefenkarte mit Hilfe einer Kreuzkorrelationsfunktionsfunktion erzeugt. Dabei kam ein hierarchisches Verfahren zum Einsatz, das den Tiefenbereich in verschiedene Entfernungsebenen und unterschiedliche Auflösungen aufteilt und so zu einer Aufwandsreduktion beiträgt. Weiterhin wurde gezeigt, wie mit Hilfe eines statistischen Clusterverfahrens erhabene und sich bewegende Objekte erkannt werden können. Dazu wurden jeweils ein Tiefen- und ein Zeit-Histogramm erzeugt, in welchen mithilfe von Schwellwerten erhabene, sich bewegende Objekte erkannt wurden. Zu Weiterverfolgung der erkannten Objekte wurde ein adaptives Kalman-Filter eingesetzt. Somit konnten Sprünge bei der Entfernungsmessung ausgeglichen und die Geschwindigkeiten der einzelnen Objekte geschätzt werden. Die Erkennung der Straßenmarkierungen wurde durch eine Hough-Transformation realisiert, mit der eine spurgenaue Zuordnung der erkannten Fahrzeuge möglich war. Eine Übersichtsdarstellung des vorgestellten Assistenzsystems ist in Abbildung 4.14 zu sehen. Hier wurde auch die Einordnung in das Schichtenmodell der Bildverarbeitung nach Abbildung 3.1 kenntlich gemacht.

Die Erstellung der Hierarchieebenen wurde dabei der Vorverarbeitung zugeordnet, während die Korrelationsfunktion selbst schon Teil der Merkmalsextraktion war. Weitere Elemente der Merkmalsextraktion waren die Subpixelinterpolation sowie die Erstellung des Tiefen- und des Zeit-Histogramms. Die Clusterung erhabener Objekte wurde der Klassifikation zugeordnet. Die Interpretation beinhaltete die Objektverfolgung mittels Kalman-Filter ebenso, wie die Aktualisierung der Straßenkarte. Die Ergebnisausgabe erfolgte am Schluss der Verarbeitungskette.

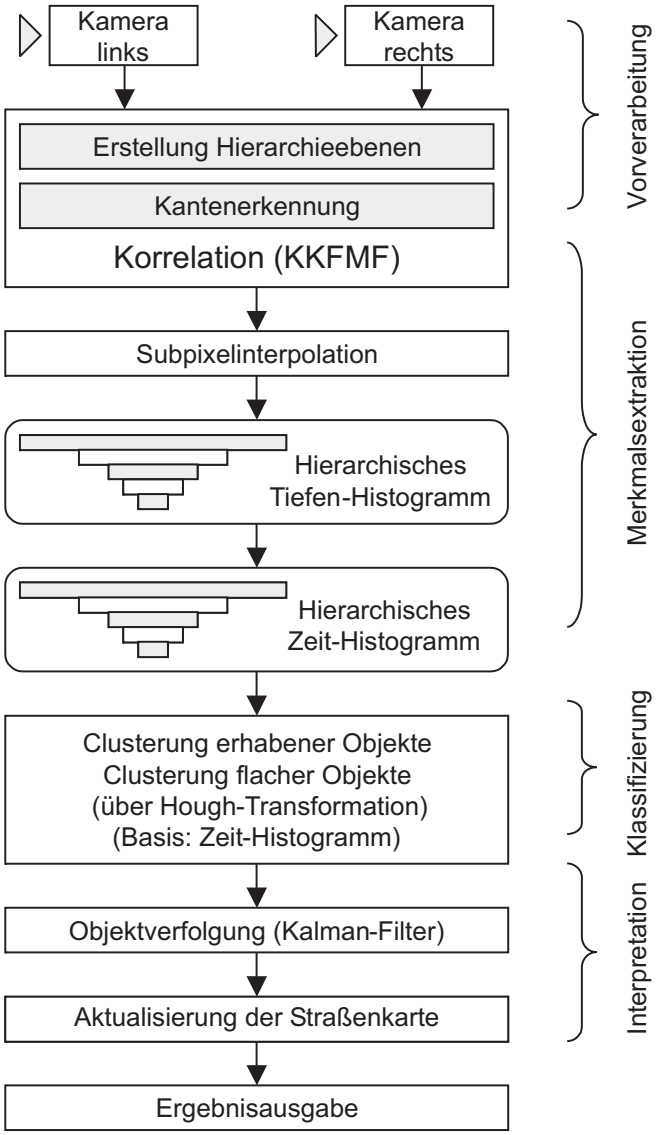

Abbildung 4.14: Einordnung des Algorithmus in das Schichtenmodell der Bildverarbeitung

Kapitel 5

HW-SW Partitionierung des Assistenzsystems

Nachdem in Kapitel 3 das System zur Partitionierung eines Bildverarbeitungsalgorithmus und in Kapitel 4 der zu partitionierende Beispielalgorithmus vorgestellt wurden, sollen nun die erklärten Schritte zur Partitionierung auf den Algorithmus angewendet werden. Die Ergebnisse der Partitionierung werden anschließend auf der entwickelten Basishardware überprüft.

5.1 Basishardware

Das in Abbildung 5.1 vorgestellte Design stellt die für die Implementierung des partitionierten System entworfene Hardware dar. Der Versuchsaufbau wurde im Rahmen dieser Arbeit erstellt und besteht aus zwei Altera Stratix FPGAs mit jeweils 60.000 Logikzellen. Die Kameras werden mittels zweier *CameraLink*-Schnittstellen an den Versuchsaufbau angeschlossen. Zwei weitere *CameraLink*-Ausgänge ermöglichen eine Darstellung der aufgenommenen bzw. manipulierten Bilder auf einem PC via Framegrabber. Weitere Schnittstellen können über entsprechende Steckverbinder auf der Rückseite der Platine angeschlossen werden.

Die Basishardware besitzt eine auf $5V$ Eingangsspannung ausgelegte Stromver-

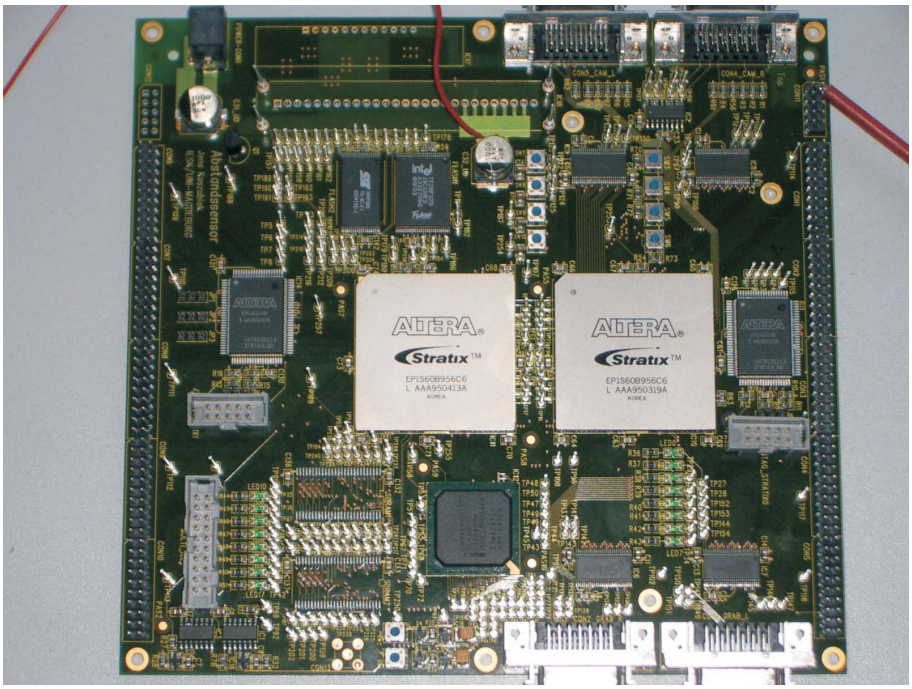

Abbildung 5.1: Entworfene Platine

sorgung und stellt auf dem Board die $3,3V$ und $1,5V$ Betriebsspannung der verschiedenen Schaltkreise zur Verfügung. Der Systemtakt wird zentral über einen Quarz erzeugt und auf der Platine verteilt. Über PLLs in den FPGAs können andere Takte aus dem Systemtakt abgeleitet werden. Das Board ermöglicht eine Steuerung der Kameras aus dem FPGA heraus, d.h. das FPGA stellt die Belichtungszeit der Kameras ein. Die Steuerregister in den Kameras können über eine RS232-Schnittstelle vom FPGA aus verändert werden. Die Taktversorgung beider Kameras erfolgt ebenfalls über das FPGA, so dass beide Kameras mit einem Takt gleicher Frequenz und Phase versorgt werden. Die Bildaufnahme ist demzufolge pixelsynchron. Neben dem internen Speicher auf den FPGAs ist für jedes von ihnen ein Flashspeicher zur Systemkonfiguration der programmierbaren Hardware sowie weitere Flashspeicher für die Software der implementierten NIOS II Prozessoren enthalten. Die $4MByte$ SDRAMs können als globaler Speicher dienen, falls der interne Speicher nicht ausreicht. Die Programmierung der FPGAs und das Beschreiben der FLASH-Speicherbausteine erfolgt über ein $JTAG$-Interface vom PC aus.

5.2 PC-Referenzsystem

Die Laufzeiten der durch die Analyse ermittelten Funktionen F_i wurden auf einem PC gemessen, da eine Emulation der Programme auf einem NIOS II Emulator nicht möglich war. Der PC hatte folgende Leistungsmerkmale:

- Pentium M 725 (Dothan Core), 1600 MHz
- 512 MB RAM, 2MB L2 Cache
- 1524 DMIPS

Ein Pentium M 725 Standard PC mit 1600 MHz Taktfrequenz, 512 MB RAM und 2MB L2-Cache kann zuvor in den Hauptspeicher geladene Kamerabilder mit einer Verarbeitungszeit von $100ms$ abarbeiten, was einer maximalen Bildrate von $10Hz$ entspricht. Problematisch ist das Nachladen der Bilder von der Festplatte, bzw. von Livebildern aus einem Framegrabber, da durch die Kameras kontinuierlich $50MByte/s$ Daten erzeugt werden. Diese Daten müssen vom PC eingelesen und verarbeitet werden. Wie bereits erwähnt, ist der Algorithmus datengetrieben, d.h. die Kameras liefern ohne Unterbrechung Daten über den Bus an den Prozessor. Wenn der DMA Betrieb aktiviert ist, können die Daten über den Bus in den Hauptspeicher gelesen werden. Der DMA Betrieb blockiert dabei den gesamten Bus, so dass der Prozessor nicht zum selben Zeitpunkt auf die gerade eingelesenen Daten zugreifen kann. Nach dem Einlesen der Daten werden sofort, nach der Integrationszeit der Kameras (hier $4ms$), neue Daten mittels DMA-Betrieb an den Hauptspeicher gesendet. Diese Zwischenzeit reicht also nicht aus, um den Algorithmus mit $100ms$ Verarbeitungszeit abzuarbeiten. Die Gesamtverarbeitungszeit auf dem PC beträgt also $140ms$.

5.3 Datenflussgraph des Assistenzsystems

Bevor eine Partitionierung des Assistenzsystems durchgeführt werden kann, muss der entsprechende Datenflussgraph erstellt werden.

Der Datenflussgraph des Assistenzsystems aus Kapitel 4 wurde mit Hilfe der Daten der statischen und dynamischen Analyse (siehe Kapitel 3.4.2) als Teil der Partitionierung aufgestellt und ist in Abbildung 5.2 zu sehen. Die Richtung des Datenflusses ist durch Pfeile gekennzeichnet. Die Basisfunktionen, die den eigentlichen Algorithmus enthalten, werden mit ihren Funktionsnummern (siehe auch Tabellen C.5 bis C.8 in Anhang C.3) durch Kreise repräsentiert. Hüllfunktionen, die den Datenfluss auf die einzelnen Basisfunktionen verteilen, werden nur während der Analyse und der Aufstellung der Datenflussgraphen berücksichtigt. Für die weitere Partitionierung sind sie nicht mehr erforderlich. Die in den Funktionsblöcken abgearbeiteten Algorithmen sind im Bild rechts dargestellt und können mit Abbildung 4.14 aus Kapitel 4.5 verglichen werden.

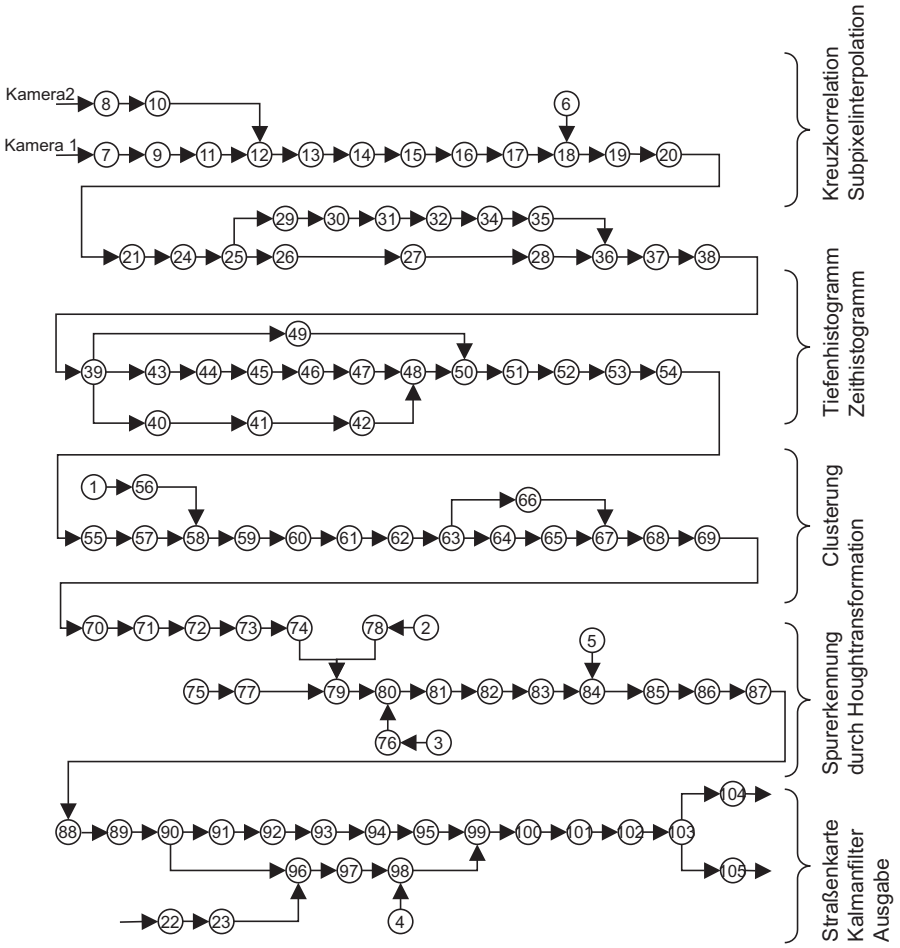

Abbildung 5.2: Datenflussgraph des Assistenzsystems basierend auf Abb. 4.14

Die Analyse erfolgt automatisch durch ein im Rahmen dieser Arbeit entwickeltes Analyseprogramm. Dabei werden die Namen der Funktionen, ihre Aufrufhäufigkeiten, die Verarbeitungszeiten, die Komplexitäten der verwendeten arithmetischen Operationen, die lokalen und globalen Variablen und die übergebenen Variablen ermittelt. Die Ergebnisse sind in den Tabellen C.5 bis C.8 in Anhang C.3 dargestellt. Für jeden Funktionsaufruf im Datenfluss wird eine eindeutige Nummer F_i vergeben, so dass der Datenfluss eindeutig identifizierbar ist.

Die Funktionen F_7 und F_8 bilden den Eingang des Systems. Sie nehmen die Daten der Kameras an und erzeugen die einzelnen Entfernungslevel, die dann in den Funktionen F_9 und F_{10} zu einer Zeile zusammengefügt werden. Die parallele Verarbeitung der Kameradaten von linker und rechter Kamera endet bereits bei Funktion F_{12} (Beginn der Kreuzkorrelationsfunktion). Die folgenden Funktionen hängen durch ihre verwendeten Variablen als Pipeline zusammen. Der Datenfluss ist sehr linear, ohne viele Verzweigungen aufgebaut. Die Vorverarbeitung endet mit dem Zusammenführen der beiden Datenströme in Funktion F_{12}. Dort schließt sich die Kreuzkorrelationsfunktion an (F_{12} bis F_{19}). Die Ergebnisse der Kreuzkorrelationsfunktion werden in F_{20} (GenDataSet) aufbereitet und in Funktion F_{21} (History) in das Tiefen-Histogramm eingetragen. Bis zu diesem Zeitpunkt ist die Verarbeitung zeilenbasiert. Alle weiteren Funktionen nutzen die Daten eines kompletten Bildes. Nach der Erstellung des Tiefen-Histogramms wird in Funktion F_{24} festgestellt, welche Einträge als erhaben und welche als flach gelten. Die Ergebnisse werden dann mit dem Zeit-Histogramm des Vorgängerbildes verglichen. Die durch F_{24} gefundenen erhabenen Objekte werden dabei durch die Funktionen F_{25} bis F_{54} verglichen und in ein Zeit-Histogramm eingetragen. Durch den Vergleich wird festgestellt, ob sich ein Objekt in konstantem oder sich verringerndem Abstand zum eigenen Fahrzeug befindet und somit ein erhabenes, sich bewegendes Objekt darstellt. Die Merkmalsextraktion endet mit dem fertigen Aufstellen des Zeit-Histogramms in Funktion F_{55} (SwapHist). Die Klassifizierung der Objekte durch die Clusterung findet innerhalb der Funktionen F_{56} (Cluster_reset) bis F_{74} (Cluster_remove_old) statt. Hier werden die einzelnen erhabenen und sich bewegenden Objekte zu Clustern zusammengefasst und eindeutige Clusternummern vergeben. Die Cluster erhalten die Clusternummer des ältesten an diesem Clu-

ster beteiligten Histogrammeintrages. Ist bisher keine Clusternummer vergeben worden, wird eine neue Nummer erzeugt. Die Klassifikation der Fahrspurmarkierungen findet durch die Funktionen F_{74} (Hough_init) bis F_{83} (rphi_count) statt. Dazu werden die in Funktion F_{24} gefundenen flachen Histogrammeinträge für die Hough-Transformation herangezogen. Im Anschluss erfolgt die Berechnung der 3-d-Koordinaten für die Mittelpunkte jedes gefundenen erhabenen und flachen Clusters in Funktion F_{84} und daran anschließend die Interpretation der Daten durch die Ermittlung der Geschwindigkeit mithilfe der Kalman-Filter. Hierbei werden die gefundenen Cluster durch die Funktionen F_{85} (Cluster_searcha) bis F_{93} den Kalman-Filter-Objekten zugeordnet. Daraus werden die Filterwerte in Funktion F_{94} neu berechnet. Für im System nicht vorhandene Cluster werden in Funktion F_{95} neue Kalman-Filter-Objekte erzeugt. Im Anschluss wird die Straßenkarte durch die Funktionen (F_{96} bis F_{103}) aktualisiert. Die Funktionen F_{104} und F_{105} dienen zuletzt der Ausgabe der Daten an das Benutzerinterface.

5.4 Evaluierung der Schätzverfahren für die automatische Partitionierung

Um eine Clusterung und Partitionierung durchzuführen, müssen die in Kapitel 3.4.7 angegebenen Kostenfunktionen und Schätzverfahren an Beispielen evaluiert werden. Grundsätzlich sollte eine Schätzung immer die für die Implementierung ungünstigeren Größen als die gemessenen Größen ergeben, um bei der Realisierung unterhalb der angegebenen Kosten zu bleiben.

Die Messung der Verarbeitungszeiten auf dem PC erfolgt, während der Abarbeitung einer Beispielszene für jeden einzelnen Aufruf der Funktion stattfindet, durch vorheriges Einfügen entsprechender Quellcodes (siehe Kapitel 3.4.2). Zur Auswertung werden die maximal gemessenen Verarbeitungszeiten ausgewählt, wobei ein Offset von $1\mu s$ für das Starten und Beenden des Zählers zu berücksichtigen ist. Die gemessenen PC-Zeiten für die einzelnen Funktionen sind in Anhang C.3 dargestellt.

Die Portierung der Software erfolgte auf ein mit $60MHz$ getaktetes NIOS II Mul-

tiprozessorsystem (mit 70 DMIPS - Dhrystone Million Instructions Per Second) und auf die mit $20MHz$ arbeitende Logik.

Durch den unterschiedlichen Aufbau der Prozessorarchitekturen im Pentium M und im NIOS II - unterschiedliche Cachegröße, 5-stufiges Pipelining im NIOS II gegenüber 15-stufigem Pipelining im Pentium, Avalon Bus gegenüber PCI-Bus, RISC gegenüber CISC - reicht es nicht einfach aus, anhand der Taktraten abzuschätzen, wie schnell der Code auf dem NIOS II ausgeführt wird. Es ist daher notwendig mit einem Benchmarktest die Rechenleistungen der beiden Prozessoren zu vergleichen. Ein möglicher Test ist der Dhrystone-Benchmarktest [118]. Dieser Benchmarktest verwendet Integer- und String-Operationen. Es werden jedoch keine Gleitkommaoperationen getestet. Die Resultate des Tests werden in DMIPS (Dhrystone Million Instructions Per Second) angegeben.

Da ein Verhältnis zwischen PC und NIOS II bei den DMIPS von $21,77$ und den Prozessortakten von $26,6$ besteht, ist zu sehen, dass der NIOS II Prozessor, gegenüber dem Pentium M, für den Dhrystone-Testvektor eine leicht bessere Effektivität aufweist. Da mit dem Dhrystontest keine Gleitkommaoperationen getestet werden und die Verarbeitungszeit von Textstrings im vorliegenden Anwendungsfall nur eine untergeordnete Rolle spielt, ist der Test nur bedingt für den Vergleich zwischen dem PC-System und dem NIOS II System geeignet. Für einen spezifischen Vergleich von Integer- und Gleitkommaoperationen auf den beiden Systemen wurde ein eigener Test entwickelt. Die ermittelten Geschwindigkeitsfaktoren SF sind in Tabelle 5.1 dargestellt.

SF Integeraddition	10.4
SF Integersubtraktion	12.4
SF Integerdivision	23.1
SF Integermultiplikation	12.4
SF Gleitkommaaddition	744
SF Gleitkommasubtraktion	824
SF Gleitkommadivision	813
SF Gleitkommamultiplikation	570

Tabelle 5.1: Geschwindigkeitsfaktoren mit Benchmarktest

Es ist zu erkennen, dass der selbst entwickelte Test für die Integeroperationen bessere Werte ergibt, als es der Vergleich der DMIPS vermuten lässt. Der Unterschied entsteht, weil hier keine Stringoperationen Anwendung finden, sondern nur arithmetische Operationen innerhalb von Schleifen getestet werden. Der NIOS II Prozessor weist für diese Art von mathematischen Operationen eine sehr hohe Effektivität auf, da das System nicht durch ein Betriebssystem ausgebremst wird und die Operationen entsprechend der RISC-Architektur in Hardware ausgeführt werden. Bei den Gleitkommaoperationen zeigt sich ein anderes Bild. Hier sind die Geschwindigkeitsfaktoren besonders hoch, was für eine geringe Effektivität in diesem Bereich spricht, weil standardmäßig keine Hardwareerweiterungen für Gleitkommaoperationen innerhalb des NIOS II vorhanden sind. Dem Entwickler bleibt es aber selbst überlassen, an dieser Stelle Erweiterungen der Prozessorstruktur vorzunehmen und somit die Geschwindigkeitsfaktoren zu verbessern.

Zur Überprüfung der Schätzverfahren aus Kapitel 3.4.7 wurden die gemessenen Werte von Beispielfunktionen mit ihren geschätzten Werten verglichen. Die Ergebnisse des Vergleichs sind in den folgenden Abschnitten dargestellt.

5.4.1 Schätzung der Verarbeitungskosten auf dem NIOS II

Die verwendeten Beispielfunktionen gehören in den Bereich der Merkmalsextraktion (F_{24} bis F_{55}) und der Klassifikation (F_{56} bis F_{74}). Die Schätzung der Verarbeitungszeit erfolgt durch Berechnung eines Transferparameters a_i nach Gleichung 5.1. Für die Funktion F_{63} - Cluster1 wurden durch die Analyse insgesamt 32 Zuweisungsoperationen, 14 Logikoperationen, 15 Additionen und 5 Subtraktionen gefunden. Also ist $n_{ges} = 66$

$$a_{63} = \frac{n_{63_{add}}}{n_{63_{ges}}} \cdot SF_{add} + ... + \frac{n_{63_{multfloat}}}{n_{63_{ges}}} \cdot SF_{multfloat} \tag{5.1}$$

Gleichung 5.1 ergibt für die Funktion F_{63}: $a_{63} = 10,55$. Mit Gleichung 5.2 wird daraus dann die Verarbeitungszeit geschätzt.

$$C_{P_{i_{SW}}} = 10,55 \cdot t_{i_{PC}} \tag{5.2}$$

Da die Verarbeitungszeit auf dem PC $2,2\mu s$ beträgt, wird sie auf dem Prozessor mit $C_{P_{i_{SW}}} = 23,2\mu s$ geschätzt.

Nr.	Funktion	geschätzte Kosten $C_{P_{i_{SW}}}$ $[\mu s]$	gemessene Kosten $C_{P_{i_{real}}}$ $[\mu s]$
24	Freq2statusErhaben	860	895
25	TakeFrom_imax	22,9	31,1
26	Clear_histtmp	16,3	13,5
30	maxi	19,9	15,3
37	dilev	15,9	18
39	newhiststat	17,2	25,3
55	SwapHist	16,3	10,8
56	CLUSTER_reset	42,5	63,5
57	strtoldest	22,9	23
58	OldestSearch	27,7	33,5
59	CLUSTER_new	25,2	35,6
61	ClusterStart	11,8	8,9
63	Cluster1	23,3	18,6
64	Left_over	15,2	22,3
65	WriteCluster	15,5	13,1
74	CLUSTER_remove_old	26,3	22,7

Tabelle 5.2: Vergleich geschätzter und realer NIOS II Verarbeitungszeiten

In Tabelle 5.2 sind die durch den Partitionierungsalgorithmus geschätzten und die gemessenen Verarbeitungszeiten für einzelne Beispielfunktionen auf dem NIOS II Prozessor zu sehen. Gemessen wurde der Einzelaufruf der Beispielfunktion mit einem über Interrupts angesteuerten Hardware Counter auf dem FPGA. Es ist ein gute Annäherung der Schätzwerte an die realen Messwerte zu erkennen. Der mittlere Fehler beträgt $8,4\mu s$. Seine Varianz ist $71\mu s^2$.

5.4.2 Schätzung der Hardwarekosten

Die Entwicklung der Logikfunktionen erfolgte im Rahmen einer anderen Arbeit, so dass hier die geschätzten und realen Größen verglichen werden können.

Zu den Hardwarekosten zählen die Verarbeitungszeit des Algorithmus in der Logik, die Entwicklungszeit der Logik und der Verbrauch von Logikzellen. Die entsprechenden Werte werden beispielhaft für die Funktion F_{15} (MWK-KKF) bestimmt.

Verarbeitungskosten $C_{P_{i_{HW}}}$: Die Funktion F_{15} stellt die eigentliche Kreuzkorrelationsfunktion dar. Hier werden 11 Zuweisungsoperationen, eine Integeraddition, vier Gleitkommadditionen, zwei Gleitkommamultiplikationen und eine Gleitkommadivision durchgeführt. Die längste Verarbeitungszeit innerhalb der Funktion F_{15} ergibt sich aus der Gleitkommadivision. T_{OP} wird daher auf $16,5\mu s$ gesetzt. Zusammen mit den verwendeten Variablen ergibt sich eine Schwierigkeit HD von $3,46$. Die Komplexität nach McCabe CC beträgt 4 und der Code besitzt insgesamt $LOC = 53$ Codezeilen. Daraus ergibt sich ein sequentieller Anteil f_S von 14%. Die Berechnung von f_S aus den Komplexitäten und der Zeilenanzahl ist in Gleichung 5.3 noch einmal dargestellt. Weitere Erläuterungen finden sich im Anhang C.4.

$$f_S = \frac{CC + HD}{LOC} \quad mit\,\{(CC + HV) < LOC\} \tag{5.3}$$

$$t_{seq} = \frac{f_S}{1 - f_S} \cdot T_{OP} \tag{5.4}$$

Der sequentielle Anteil an den Gesamtverarbeitungskosten ergibt sich somit aus Gleichung 5.4 zu $2,7\mu s$. Die geschätzten Verarbeitungskosten der Funktion F_{15} in Logik betragen also $C_{P_{i_{HW}}} = 19,2\mu s$. Die geschätzten Hardwareverarbeitungskosten weiterer Funktionen sind in Tabelle 5.3 vergleichend mit den realen Verarbeitungszeiten dargestellt.

Hardwareaufwand: Der Logikzellenverbrauch entsteht aus der Summe aller verwendeten Logikoperationen. Die Zuweisungsoperationen werden nicht berücksichtigt, weil Ergebnisse einer Logik direkt zum Eingang der darauf folgenden Logik geführt werden und somit Zuweisungen in dieser Form nicht existieren. Der gesamte Logikzellenverbrauch lässt sich mit den im Anhang C.3 ermittelten Logikzellenverbräuchen der einzelnen Operationen durch Aufakkumulieren ermitteln. Für die Funktion F_{15} ergibt sich so ein geschätzter Logikzellenverbrauch von 14244 Logikzellen (LC).

Entwicklungskosten: Für die Vorhersage der Entwicklungskosten der Hardwarelösung wurde der COCOMO II Ansatz [82] gewählt (Gleichung 5.5), weil die

Portierung von C/C++ in VHDL mit der Kostenschätzung eines Softwareprojektes vergleichbar ist und für diesen Bereich der COCOMO II Ansatz als besonders geeignet angesehen wird [18]. Die bereits eingetragenen Faktoren sind durch Böhm [18] ermittelt und basieren auf Erfahrungswerten. Es wird sich dabei auf Projekte mittlerer Komplexität bezogen. Einfache Projekte erhalten einen Faktor kleiner als $2,94$. Sehr komplexe Projekte erhalten dagegen einen Faktor größer als $2,94$. Die Wichtungsfaktoren W liegen üblicherweise zwischen 0, für nicht berücksichtigt und 4, für sehr wichtig. Je kritischer ein Erfolgsmultiplikator EM gesehen wird, um so höher wird er eingestellt. $1,5$ wird dabei als Obergrenze angesehen und $1,0$ als $Defaultwert$. Die gewählten Wichtungsfaktoren und die Erfolgsmultiplikatoren sind in Anhang C.4 dargestellt. Bis auf EM8 sind alle Faktoren mit heuristisch ermittelten Werten belegt. EM8 (Komplexität des Algorithmus) wird mit der ermittelten Schwierigkeit der Funktion HD nach Halstead beschrieben (siehe auch Gleichung 3.23). Die Größe des Codes S wird nicht, wie im originalen COCOMO II Ansatz, mit $KiloLinesofCode$, sondern nur mit $100 LinesofCode$ angegeben. Das ist der Tatsache geschuldet, dass Funktionen gleicher Funktionalität in Hardware wesentlich schwieriger zu realisieren sind und so der Faktor 10 gerechtfertigt ist. In Gleichung 5.5 ist die Berechnung der benötigten Entwicklungskosten auf Basis des COCOMO II Ansatzes noch einmal dargestellt.

$$PM_{HW} = 2,94 \prod_{i=1}^{17} EM_i S^{0,91+0,01 \cdot \sum W} \tag{5.5}$$

Daraus ergeben sich für die Funktion F_{15} mit den angenommenen Werten Entwicklungskosten von 7 Monaten. Der Vergleich von realer und geschätzter Entwicklungszeit in Tabelle 5.3 bestätigt die Vorgehensweise für die untersuchten Funktionen F_7 bis F_{21}. Die Funktionen F_0 bis F_6 werden nicht überprüft, da es sich dabei um Initialisierungsfunktionen handelt.

Trotz unterschiedlicher MoCs (Models of Computation), d.h. Computing in Time (Prozessor) und Computing in Space (Logik), konnten für die meisten portierten Funktionen gute Näherungen in den Verarbeitungszeiten, den Entwicklungszeiten in Personenmonaten (PM) und im Verbrauch der Logikzellen (LC) gefunden werden. Da jedoch die Funktionen nicht nur auf einen anderen Prozessortyp por-

Nr.	Funktion	HW-Zeit-gesch. [μs]	HW Zeit-gem. [μs]	Entw.-Zeit gesch.[PM]	Entw.-Zeit real [PM]	LC gesch.	LC Real
7	createLevelLeft	0,31	0,8	2,03	1,5	656	500
8	createLevelRight	0,31	0,8	2,03	1,5	656	500
9	CalcLinesLeft	0,44	0,1	1,5	1	328	57
10	CalcLinesRight	0,73	0,1	1,5	1	328	57
11	CalcMaxDisp	0,24	0,05	0,41	1	128	30
12	calcab	0,65	0,85	0,83	1	3617	677
13	CalcMeana-aa	2,4	0,3	2,3	2	7385	2618
14	vara	0,23	0,05	0,4	0,5	160	18
15	MWF-KKF	1,92	3	7,0	5	14244	15258
16	ifmaxk	0,3	0,1	0,3	0,5	96	16
17	subpix	2,6	0,8	2,3	1	8743	32
18	LineStore	0,24	0,8	0,5	0,5	128	32
19	CalcUpLev	0,38	1,6	1,3	2	919	650
20	GenDataSet	0,31	0,8	1,5	1	1239	124
21	history	0,31	0,8	2,4	3	776	356

Tabelle 5.3: Vergleich von realen und geschätzten HW-Eigenschaften der Funktionsblöcke

tiert werden, sondern komplett neu entwickelt werden müssen, sind die Fehler wesentlich größer als bei der Schätzung der Verarbeitungskosten auf einem NIOS II. Insbesondere die Funktion F_{17} ($subpix$) fällt durch große Abweichungen auf. Die fehlerhafte Schätzung kommt zustande, weil in der Software verschiedene Gleitkommaarithmetiken verwendet werden, die in der Hardware durch Festkommaarithmetiken ersetzt worden sind. Diese Arithmetiken können natürlich auch in der Software realisiert und somit in der Analyse der Funktion berücksichtigt werden. Hier ist aber zu verdeutlichen, dass die Qualität des Hardware-Software Co-Designs direkt mit der Qualität der Quellcodes bzw. der Erfahrung des Quellcodeentwicklers zusammenhängt. Trotz des hier vorgestellten automatischen Partitionierungssystems spielt die Erfahrung eines Entwicklers also noch eine große Rolle.

Nachdem die vorgestellten Schätzverfahren evaluiert worden sind, soll die Partitionierung eines Bildverarbeitungsalgorithmus nach Kapitel 3, anhand des in Kapitel 4 vorgestellten Assistenzsystems, erfolgen.

5.5 Hardware-Software Partitionierung

Der erste Schritt der Partitionierung stellt die Clusterung der Funktionen zu einem Multirechnersystem dar. In Abbildung 5.3 ist das geclusterte System zu sehen. Die Cluster entsprechen den farbigen Flächen. Jede Fläche entspricht einem Prozessor im Multiprozessorsystem. Die Prozessornummer P_i ist jeweils rechts oben neben bzw. in jedem Cluster zu sehen.

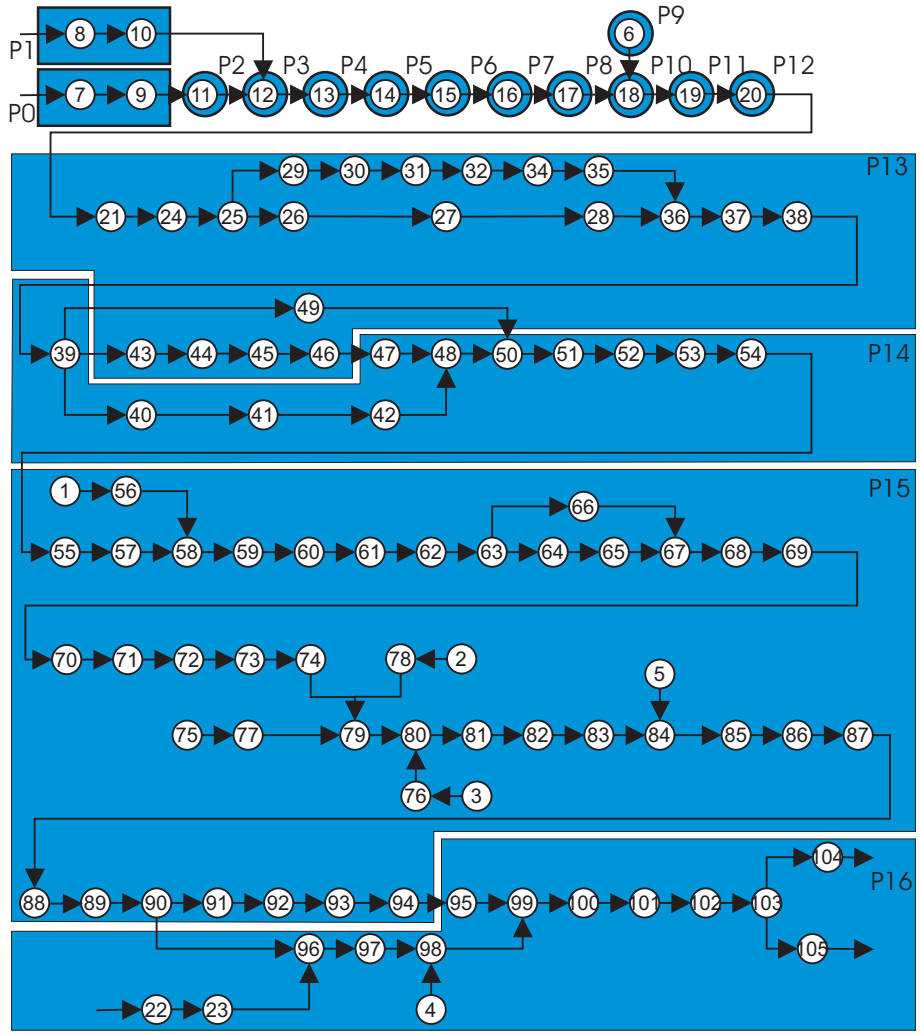

Abbildung 5.3: Datenflussgraph des geclusterten Assistenzsystems

Die Clusterung erfolgt nach dem Ansatz von Sarkar [94] (siehe auch Kapitel 3.4.5). Dabei werden die Funktionen, ihren Kommunikationskosten entsprechend, zu Clustern zusammengefügt. Als Randbedingung für das Assistenzsystem gilt die

maximale Verarbeitungszeit für ein Pixel ($50ns$), eine Zeile ($51\mu s$) bzw. ein Bild ($40ms$). Da der Algorithmus des Assistenzsystems durch die Kamera datengetrieben ist, gelten diese Randbedingungen gleichzeitig als Echtzeitbedingungen für die Partitionierung. Das Schichtenmodell der Bildverarbeitung hat gezeigt, dass die Bearbeitung der Daten von Punktoperatoren über lokale bis zu globalen Operatoren erfolgt. Der in Kapitel 4 beschriebene Algorithmus verhält sich dem Schichtenmodell entsprechend. Im konkreten Fall werden keine Punktoperationen durchgeführt. Die Kantenerkennung, die Kreuzkorrelation und die Histogrammerzeugung werden innerhalb einer Zeile ausgeführt und die folgenden Algorithmen wie Clusterung, Kalman-Filter und Straßenkarte werden über den Zeitraum eines Bildes abgearbeitet. Durch die Linearität des Datenflusses kann in den Nebenbedingungen festgelegt werden, dass alle Funktionen bis zur *History* (F_{21}) innerhalb einer Zeile und alle folgenden Funktionen innerhalb eines Bildes abgearbeitet werden. Für die entsprechenden Funktionen gelten die dafür ermittelten Echtzeitbedingungen.

Die Verarbeitungszeiten eines Prozessors $t_{i_{SW}}$ werden aus den Verarbeitungskosten der darin enthaltenen Cluster bestimmt. Um die Verarbeitungszeiten für die Prozessoren innerhalb einer Zeile zu bestimmen, wird die geschätzte Verarbeitungszeit durch die Anzahl der ausgewerteten Kamerazeilen ZA dividiert (siehe Gleichung 5.6).

$$t_{i_{SW_{Zeile}}} = \frac{t_{i_{SW}}}{ZA} \tag{5.6}$$

Es ist zu erkennen, dass neben den großen Clustern gerade zu Beginn viele Cluster gefunden werden, die nur eine Funktion beinhalten. Diese Funktionen ($F_{11} - F_{20}$) erfüllen entweder gerade noch die Echtzeitbedingungen oder verstoßen gegen sie und können daher nicht mit anderen Funktionen geclustert werden. In Tabelle 5.4 sind die geschätzten Verarbeitungszeiten der einzelnen Prozessoren dargestellt.

Weil der Datenfluss sehr linear ist und somit fast alle Funktionen hintereinander auf unterschiedlichen Prozessoren abgearbeitet werden, entsteht ein Pipelining. Die Prozessoren P_0 und P_1 können parallel Daten verarbeiten und gehören deshalb zu einer Pipelinestufe. Prozessor P_8 mit Funktion F_{17} und Prozessor P_9 mit

Prozessornummer	geschätzte Verarbeitungszeiten $t_{i_{SW}}$ [ms]	enthaltene Funktionen
0	31,47 (0,063 je Zeile)	F_7, F_9
1	37,55 (0,075 je Zeile)	F_8, F_{10}
2	317,24(0,634 je Zeile)	F_{11}
3	347,47(0,694 je Zeile)	F_{12}
4	20505 (41,01 je Zeile)	F_{13}
5	401,8 (0,803 je Zeile)	F_{14}
6	27394 (54,78 je Zeile)	F_{15}
7	789,6 (1,579 je Zeile)	F_{16}
8	716 (1,432 je Zeile)	F_{17}
9	0,0011	F_6
10	326,1 (0,652 je Zeile)	F_{18}
11	1512,8(3,024 je Zeile)	F_{19}
12	31,4 (0,063 je Zeile)	F_{20}
13	30,5	$F_{21}...F_{38}$und $F_{43}...F_{46}, F_{49}$
14	36,6	F_{39},$F_{40}...F_{42}$ und $F_{47}...F_{54}$
15	35,2	$F_1...F_3$, F_5 und $F_{55}...F_{94}$
16	16,3	$F_4, F_{22}, F_{23}, F_{95}...F_{105}$

Tabelle 5.4: Verarbeitungszeiten des Multirechnersystems

Funktion F_6 können ebenfalls in einer Pipelinestufe verarbeitet werden, da die Daten erst für Prozessor P_{10} bereitstehen müssen. Bis Prozessor P_{12} werden die Daten also in 11 Pipelinestufen abgearbeitet. Die Funktionen in den folgenden vier Prozessoren werden ebenfalls seriell abgearbeitet und bilden vier Pipelinestufen. Durch die Aufteilung der Funktionen auf das Multiprozessorsystem entsteht damit ein 15-stufiges Pipelining.

Innerhalb einer Pipeline können die Daten nur so schnell abgearbeitet werden, wie es die langsamste Pipelinestufe erlaubt. Prozessor P_6 mit der MWF-KKF (F_{15}) ist im vorliegenden Fall der Prozessor mit der längsten Verarbeitungszeit t_{SW}. Für die 500 Zeilen eines kompletten Bildes benötigt er $27,3s$. Also kann erst nach $27,3s$ die erste Zeile des neuen Bildes eingelesen werden. Um die Analysedaten eines Bildes zu erhalten, müssen die Verarbeitungszeiten aller Pipelinestufen aufsummiert werden, was zu einer Latenzzeit des Gesamtsystems von $52,5s$ führt. Die niedrige Verarbeitungsgeschwindigkeit ergibt sich aus den während des Benchmarktestes ermittelten Geschwindigkeitsfaktoren (siehe Tabelle 5.1). Durch den Einsatz von Gleitkommaoperationen ergibt sich z.B. für die Funktion MWF-KKF (F_{15}) eine Verarbeitungszeit auf dem NIOS II von $107\mu s$ zu $1,0\mu s$ auf

dem PC. Zusätzlich zu der erhöhten Verarbeitungszeit wird diese Funktion auch besonders häufig aufgerufen ($AH_{F_{15}} = 116000$). Die so berechneten Zeiten verletzen die aufgestellten Echtzeitbedingungen aus Anhang C.1. Deshalb muss eine Aufteilung des Systems in ein Hardware-Software System vorgenommen werden.

Partitionierungsverlauf

Für die Partitionierung sind insbesondere zwei Maße interessant. Zum einen die Verarbeitungszeit des Systems und zum anderen die entstehenden Kosten einer Partitionierung. Beide Maße sind während der Partitionierung nach jeder Iteration geschätzt worden und sind in den Diagrammen in Abbildung 5.4 dargestellt. Als Iteration wird ein Durchlauf von Phase zwei des Partitionierungsprozesses nach Abbildung 3.7 bezeichnet. Innerhalb einer Iteration wird jeweils eine Funktion von der Software- in die Logikdomain oder von der Logik- in die Softwaredomain überführt.

Ausgangspunkt ist das gelcusterte System aus Abbildung 5.3. Die dort ermittelten Kosten stellen die Referenz für die weiteren Partitionierungsschritte dar. Ziel ist eine echtzeitfähige Partition des Algorithmus.

Als Verlauf für den Abkühlungsprozess im Simulated Annealing ist der geometrische Ansatz (Gleichung 5.7) gewählt worden, weil er nach Wiangtong [119] weniger Rechenressourcen benötigt als der Ansatz nach Lundy und Mess, jedoch ein vergleichbar gutes Ergebnis liefert. In der Literatur werden für den Kontrollparameter Startwerte von 220 vorgeschlagen. Aufgrund der schnellen Annäherung des Partitionierungsverlaufes an ein Ergebnis mit minimalen Kosten (siehe Abbildung 5.4) wurde der Kontrollparameter mit $CP_{Start} = CP_{alt} = 50$ und α mit $0,9$ gewählt.

$$CP_{neu} = \alpha \cdot CP_{alt} \tag{5.7}$$

Wird eine Partition angenommen, so wird der Kontrollparameter neu berechnet. Eine Partition wird angenommen, wenn die Kosten C der neu berechneten Partition unter den Kosten der bisher niedrigsten Partition liegen $\Delta C = C_k - C_{k-1}$.

Wenn ΔC positiv ist, wird eine Annahmewahrscheinlichkeit P nach Gleichung 5.8 berechnet, die auf den Werten von CP und ΔC basiert.

$$P = e^{-\frac{\Delta C}{CP}} \tag{5.8}$$

P wird mit einer Zufallswahrscheinlichkeit P_x verglichen und muss größer sein als diese, um ein gültige Partition zu erhalten. Wird eine Partition abgelehnt, behält der Kontrollparameter seinen alten Wert bei. Die Auswahl der Funktion für die neu zu ermittelnde Partition, erfolgt über ihre Priorität nach Gleichung 5.9. Der Geschwindigkeitsgewinn und das Level bzw. Co-Level einer Funktion sind die wichtigsten Größen zur Bestimmung der Priorität. Das Level gibt an, wie weit eine Funktion von der Datensenke entfernt ist und das Co-Level gibt an, wie nah sich eine Funktion an der Datenquelle befindet. Aufgrund des Schichtenmodells der Bildverarbeitung ist bekannt, dass die Funktionen in der Nähe der Datenquelle (Vorverarbeitung und Merkmalsextraktion) besonders gut für eine Realisierung in Logik geeignet sind. Diese Funktionen haben ein niedriges Co-Level. Im Datenfluss trifft das aber auch für Initialisierungsfunktionen zu, die erst wesentlich später benötigt werden. Die Funktionen mit einem hohen Level liefern wiederum die Daten für eine große Zahl nachfolgender Funktionen. Die Priorität der zu portierenden Funktionen ist daher proportional zu ihrem Level (Gleichung 5.9).

$$P_{L_{i_{SW}}} = (1 - StF) \cdot (0,5 \cdot L_{F_i} + 0,4 \cdot (C_{P_{i_{SW}}} - C_{P_{i_{HW}}}) - 0,1 \cdot CC) \tag{5.9}$$

Der Straffaktor ist zu Beginn Null. Wird die durch den Transfer der Funktion entstandene Partition abgewiesen, wird $StF = 0,9$ gesetzt und bei jeder Iteration mit α multipliziert. Mit der sukzessiven Verringerung von StF wird sichergestellt, dass sich die Priorität der abgewiesenen Funktion verringert und zunächst andere Funktionen für die Partitionierung verwendet werden. Es wird aber gleichzeitig dafür gesorgt, dass die Funktion zu einem späteren Zeitpunkt noch einmal portiert werden kann. Für eine portierte Funktion wird die Priorität mit dem Co-Level (Gleichung 5.10) bestimmt, so dass die Funktion nicht komplett von der

Prioritätsbestimmung ausgeschlossen und so eine Rückportierung aus der Logik in eine Softwarelösung möglich ist.

$$P_{CL_{i_{HW}}} = (1 - StF) \cdot (0,5 \cdot CL_{F_i} + 0,4 \cdot (C_{P_{i_{SW}}} - C_{P_{i_{HW}}}) - 0,1 \cdot CC) \quad (5.10)$$

Die Kosten jeder einzelnen Partition sind mit folgender Funktion ermittelt worden:

$$C_{Ges} = \sum_{i=0}^{N} \left(0,5 \cdot (C_{P_i} + C_{C_i}) + 0,25 \cdot A_{HW_{F_i}} + 0,25 r_i \cdot PM_{HW_{F_i}}\right) \quad (5.11)$$

Die in den Gleichungen 5.9, 5.10 und 5.11 verwendeten Faktoren, dienen zur Gewichtung der einzelnen Prioritäts- bzw. Kostenmaße und sind heuristisch bestimmt worden.

Der Verlauf der Partitionierung ist in Abbildung 5.4 für 30 Iterationsschritte dargestellt. Abbildung 5.4.a) zeigt den Hardwareaufwand für das Hardware-Software System. Dazu zählen die Logikzellen für die implementierten Logikfunktionen und die NIOS II Prozessoren. In Abbildung 5.4.b) sind die Entwicklungskosten für die in die Logikdomain transferierten Funktionen zu sehen. Die geschätzte Verarbeitungszeit des gesamten Hardware-Software Systems ist in Abbildung 5.4.c) dargestellt. Die Ordinate ist aufgrund des großen Wertebereichs logarithmisch aufgeteilt worden. Abbildung 5.4.d) zeigt die Partitionierungskosten des Gesamtsystems nach Gleichung 5.11.

Nach 15 Iterationsschritten sind die aufgestellten Echtzeitbedingungen mit einer geschätzten Gesamtverarbeitungszeit von $117ms$ erreicht (Verlauf in Abbildung 5.4.c)). Die langsamste Pipelinestufe ist nun Prozessor P_1 mit $36,6ms$. Die Partitionierung wird daraufhin abgebrochen. Das somit entstandene echtzeitfähige Hardware-Software Systems ist in Abbildung 5.5 dargestellt. Aus den Verläufen in Abbildung 5.4.a) ist zu erkennen, dass der Hardwareaufwand zunächst fällt, da die Logikrealisierungen der portierten Funktionen weniger Logikzellen verbrauchen als für die Implementierung eines kompletten NIOS II Prozessors benötigt

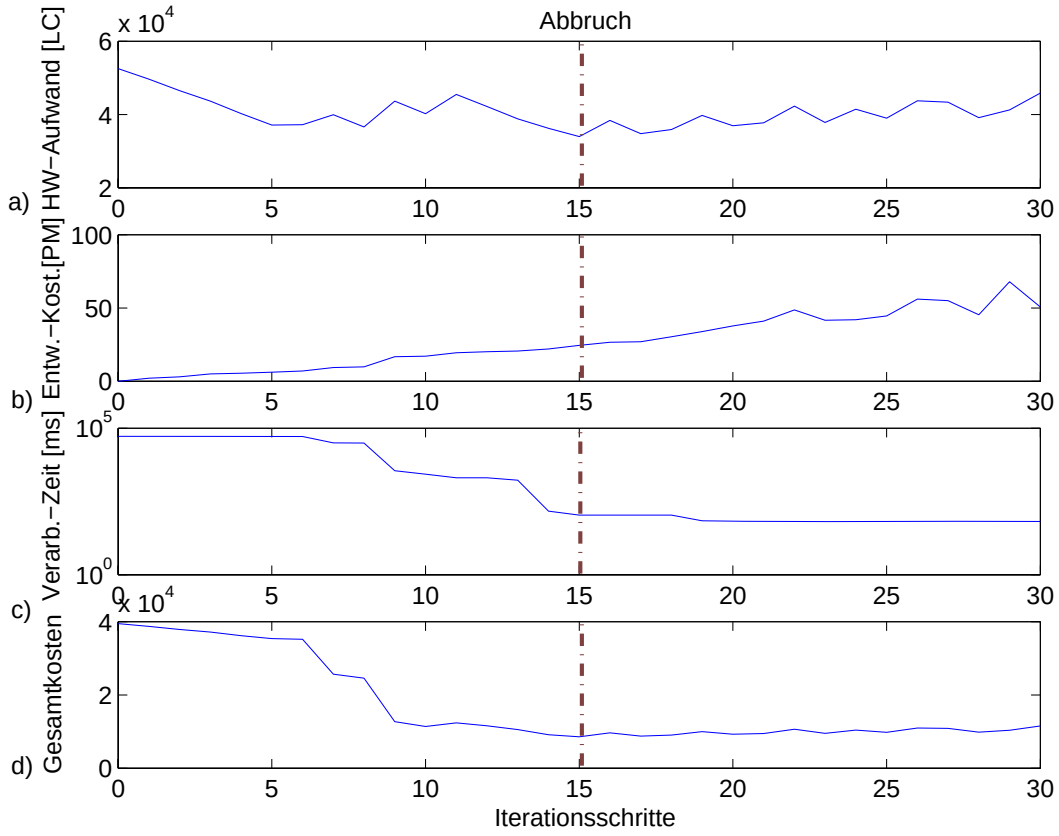

Abbildung 5.4: Verlauf der Partitionierung a) Hardwareaufwand b) Entwicklungskosten c) Verarbeitungszeit des Gesamtsystems d) Partitionierungskosten

werden. Mit der Portierung der Funktionen F_{13}, F_{15} und F_{17} steigt der Hardwareaufwand und sinkt dann wieder bis zum Iterationsschritt 15 auf ein Minimum ab. Anschließend ist wieder eine leichte Steigerung des Hardwareaufwandes zu beobachten. Durch die Portierung weiterer Funktionen in die Logikdomain steigt die Entwicklungszeit für die benötigte Logik ständig an, während die Verarbeitungszeit nur bis zum 15. Iterationsschritt signifikant fällt und anschließend auf einem konstant niedrigen Niveau verbleibt. Die Kosten der Partitionierung verlaufen, aufgrund der verwendeten Gewichte, ähnlich wie die Verarbeitungszeit. Die Partition bei Iterationsschritt 12 ist trotz erhöhter Kosten angenommen worden, da $P > P_x$. Das in Iteration 11 entstandene lokale Minimum (Transfer von Funktion F_{17} in die Logikdomain) kann also wieder verlassen werden. Bei Iterationsschritt 15 wird das globale Minimum erreicht. Weitere lokale Minima werden

bei den Iterationsschritten 17, 20, 23, 25 und 28 erreicht. Da der Kontrollparameter CP noch groß genug ist, um das Verlassen von Minima zu gestatten, werden auch weiterhin Funktionen in die Logikdomain überführt. Es ist deshalb nötig, die Partition mit dem globalen Minimum zu speichern und diese am Ende der Partitionierung als die optimale Partition auszugeben, falls die Abbruchbedingung nicht erreicht wird. Diese optimale Partition erfüllt nicht unbedingt alle Nebenbedingungen, weil eventuelle Echtzeitkriterien verletzt werden. Hier muss der Entwickler entweder überprüfen, ob die gefundene optimale Partition real bessere Verarbeitungszeiten erreicht, als die mit diesem System geschätzten, oder es muss auf eine schnellere Basishardware zurückgegriffen werden.

5.6 Resultate für das partitionierte System

Das Hardware-Software Partitionierungssystem besteht aus zwei unterschiedlichen Programmen - der Analyse und der automatischen Partitionierung. Die statische Analyse benötigt für die 4000 Zeilen Quellcode des Assistenzsystems etwa $15s$, während die dynamische Analyse, und hier insbesondere die Ermittlung der Auftrittshäufigkeiten der Funktionen, bis zu $24h$ dauern kann. Die automatische Partitionierung selbst benötigt etwa 2 Minuten für die 30 vorgestellten Iterationsschritte.

In Abbildung 5.5 ist das in Hardware und Software partitionierte Assistenzsystem zu sehen. Die Werte, auf denen die Paritionierung beruht, befinden sich im Anhang in den Tabellen C.5 bis C.8. Die Partitionierung ist nach 15 Iterationen mit Erreichen der Echtzeitbedingungen abgebrochen worden. Die Funktionen F_7 bis F_{20} sind als Logik in Hardware realisiert und mit HW gekennzeichnet. Die verbleibenden Funktionen sind vom System auf insgesamt 4 Prozessoren (P_0 bis P_3) verteilt worden.

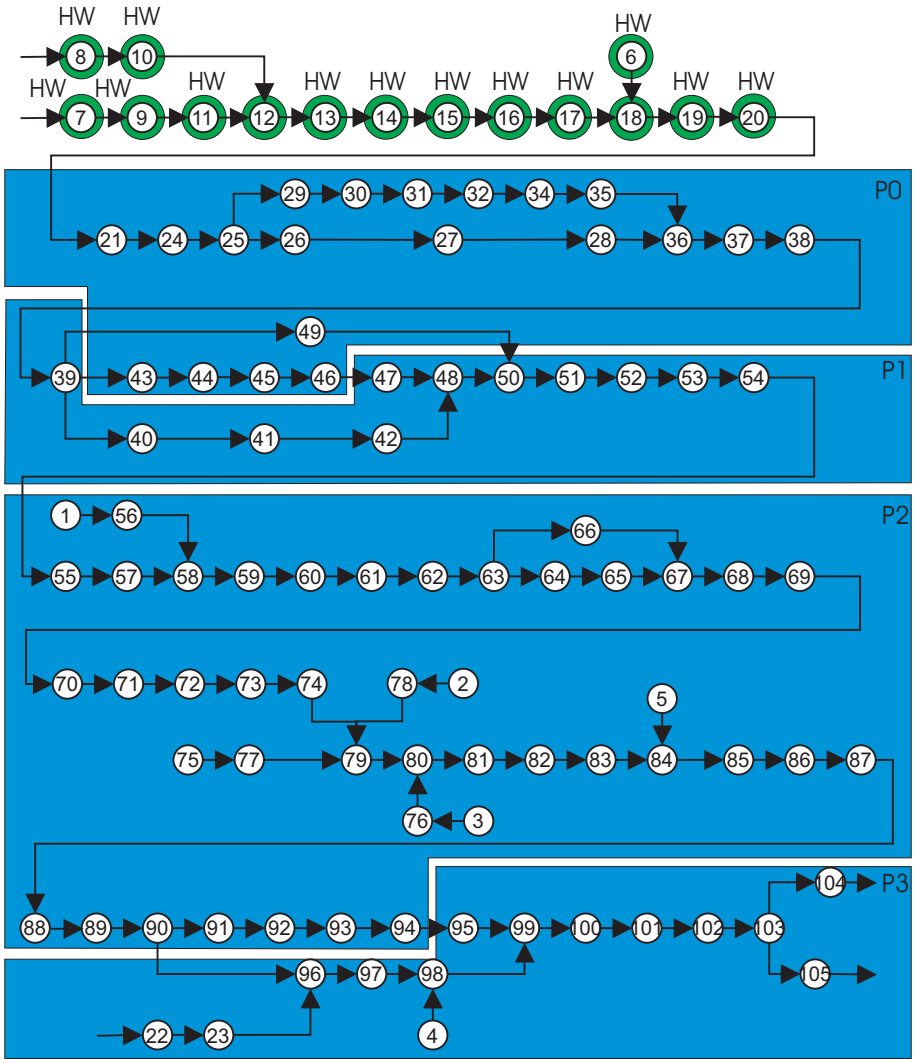

Abbildung 5.5: Datenflussgraph des Assistenzsystems

Wie zu erkennen ist, sind die besonders rechenintensiven Algorithmen der Merkmalsextraktion für eine Logikrealisierung vorgesehen worden. Wie bereits anhand des Schichtenmodells der Bildverarbeitung erläutert, sind das auch die Funktionen, die besonders für eine Logikrealisierung geeignet sind.

Das partitionierte System erreicht die in Tabelle 5.5 vorgestellten geschätzten und gemessenen Verarbeitungszeiten:

	$Hardware$ [ms]	P_0 [ms]	P_1 [ms]	P_2 [ms]	P_3 [ms]
Schätzung	0,008	30,5	36,6	35,2	16,3
Messung	0,007	28,5	38,7	33,9	15,9

Tabelle 5.5: Verarbeitungszeiten der einzelnen Elemente

Dieses System kann, im Gegensatz zur PC Lösung, explizit auf die angeschlossene Hardware, in dem Fall die Kameras, angepasst werden. Durch das Pipelining kann die erzeugte Logik mit der Verarbeitung nach dem Einlesen einer Kamerazeile beginnen. Mit einer Gesamtverzögerung von $70\mu s$ liegen die Daten für die weitere Verarbeitung auf den Prozessoren nahezu zeitgleich mit dem Ende der letzten Zeile am Eingang des Hardware-Software Systems an. Die folgenden Prozessoren können die Daten dann innerhalb der Bildauslesezeit von $40ms$ abarbeiten, da die Kommunikation in diesem Fall über zwei Speicherbausteine erfolgt. Ein Speicher wird von Prozessor P_0 ausgelesen, während der andere Speicher von der Logik mit den Daten des folgenden Bildes beschrieben wird. Mit dem Bildwechsel wechseln dann auch die Speicherzugriffe von Logik und Prozessor P_0.

Mithilfe des in dieser Arbeit vorgestellten Hardware-Software Partitionierungssystems ist automatisiert eine Hardware-Software Partitionierung gefunden worden, die die geforderten Echtzeitanforderungen erfüllt und dabei mit 24 Personenmonaten Portierungszeit einen optimalen Zeitaufwand liefert (Vergleich: Nach 30 Iterationsschritten - Zeitaufwand 48 PM bei halbierter Gesamtverarbeitungszeit von $65,7ms$). Gleichzeitig ist die gute Eignung des zugrunde liegenden Schichtenmodells der Bildverarbeitung bestätigt worden. Die Portierung der für eine Logikrealisierung vorgesehenen Software erfolgt im Rahmen einer parallel laufenden Forschungsarbeit. Der Zeitaufwand konnte für einen versierten VHDL-Entwickler bestätigt werden. Für weniger geübte Entwickler sind entsprechend längere Portierungszeiten vorzusehen, das kann beim Einstellen der Erfolgsmultiplikatoren und Wichtungsfaktoren im COCOMO II berücksichtigt werden.

Durch Realisierung des vorgeschlagenen Hardware-Software Systems ist die Qualität der Schätzung bestimmt worden. Ein Vergleich zeigt, dass die realen Verarbeitungszeiten sehr gut mit den berechneten Zeiten übereinstimmen. Für die Schätzung des Hardwareaufwandes und der Entwicklungszeiten lassen sich ebenfalls sehr gut Näherungen durch das System finden.

	Schätzung	Messung
Latenzzeit t_{ges} [ms]	118,6	116,9
Hardwareaufwand [LC]	38800	30485
Entwicklungskosten [PM]	24	22,5

Tabelle 5.6: Vergleich gemessener und geschätzter Systemgrößen

Die Leistungsaufnahme des Systems beträgt mit der verwendeten Basishardware $5W$, wohingegen die Leistungsaufnahme des PC-Systems bei $100W$ liegt. Der Platzbedarf des eingebetteten System ist ebenfalls wesentlich geringer. Mit den erfüllten Echtzeitbedingungen und dem niedrigen Platz- und Energiebedarf lässt sich das vorgestellte Hardware-Software Co-Design auch im vorgesehenen Anwendungsbereich, nämlich im Automobil, als eingebettetes System betreiben.

5.7 Diskussion

Das vorgestellte Hardware-Software Partitionierungssystem hat für den Beispielbildverarbeitungsalgorithmus aus Kapitel 4 eine Partition mit minimalen Portierungskosten gefunden, die auch die geforderten Echtzeitbedingungen einhält. Für die Schätzung der Verarbeitungskosten auf dem NIOS II sind gute Näherungen mit minimalen Abweichungen gefunden worden. Die Basis für die Schätzung der Verarbeitungskosten stellt ein eigens entwickelter Benchmarktest dar.

Die Schätzwerte für die Entwicklungskosten der Softwarefunktionen in die Logikdomain konnte mithilfe des COCOMO II Ansatzes ebenfalls in guter Näherung erfolgen. Hier sind jedoch Verbesserungen durch feinere Anpassungen der Erfolgsmultiplikatoren und Wichtungsfaktoren, aufgrund von Erfahrungswerten aus der

Entwicklung weiterer Hardware-Software Systeme, möglich.

Die Schätzung der Verarbeitungskosten in Hardware und der Hardwareaufwand der portierten Funktionen konnten nur in grober Näherung erreicht werden. Hier wurden teilweise Abweichungen um den Faktor 7 zwischen geschätzten und gemessenen Größen ermittelt. Die großen Abweichungen entstehen aufgrund der unterschiedlichen MoCs, d.h. Computing in Time (Prozessor) und Computing in Space (Logik). Im Vergleich zu den Verarbeitungszeiten auf den Prozessoren (insgesamt 117 ms) sind die Verarbeitungszeiten auf der Hardware ($0,007$ ms) sehr gering. Daher wirkt sich der relativ große Fehler bei der Schätzung der Verarbeitungszeit der Hardware nur minimal auf das Gesamtsystem aus und ist deshalb zu tolerieren. Verbesserte Schätzungen erfordern eine detailliertere Analyse des Bildverarbeitungssystems, insbesondere der Datenabhängigkeiten innerhalb einer Basisfunktion sowie ein größeres a-priori Wissen über die Realisierungsmöglichkeiten der Funktionen in der Logikdomain. Dazu zählt zum Beispiel die mögliche Umwandlung von Gleitkommaoperationen in Festkommaoperationen und die damit verbundene Einsparung an Logikzellen.

Durch den Ausbau einer Elementebibliothek mit standardisierten Funktionen als Software und Logik kann die Schätzung der Kosten wesentlich verbessert werden. Gleichzeitig würden sich die Entwicklungszeiten des Hardware-Software Systems stark verringern.

Das Partitionierungssystem wurde speziell für Algorithmen entwickelt, die dem Schichtenmodell der Bildverarbeitung entsprechen. Inwiefern das Partitionierungssystem auch für andere Klassen von Algorithmen geeignet ist, bleibt zu überprüfen.

Kapitel 6

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Entwicklung eines automatischen Partitionierungssystems für Bildverarbeitungsalgorithmen zur echtzeitfähigen Implementierung auf eingebetteten Systemen.

Als Ausgangspunkt liegt der Bildverarbeitungsalgorithmus in einer C/C++ Realisierung vor. Das Partitionierungssytem basiert auf drei Modellen. Das Schichtenmodell der Bildverarbeitung zeigt eine allgemeine Aufteilung von Bildverarbeitungsalgorithmen in Vorverarbeitung, Merkmalsextraktion und Klassifikation/Interpretation. Die Datenrate von der Vorverarbeitung zur Interpretation nimmt dabei stark ab, während die Komplexität der Verarbeitung zunimmt. Das verwendete Softwaremodell unterstellt eine Aufteilung des Algorithmus in Basis- und Hüllfunktionen. Basisfunktionen enthalten die eigentliche Funktionalität und beinhalten keine weiteren Funktionsaufrufe, während in Hüllfunktionen nur Funktionsaufrufe und Kontrollflussanweisungen (Schleifen und Verzweigungen) vorhanden sind. Als Hardwaremodell wird ein FPGA verwendet, um die Basisfunktionen optimal und flexibel zwischen der Logikdomain und der Softwaredomain aufteilen zu können. Dabei ist es möglich neben der Logik auch mehrere Softcore-Prozessoren zu verwenden.

Die Partitionierung des Bildverarbeitungsalgorithmus wird in zwei Phasen durchgeführt. In der ersten Phase erfolgt die Analyse des C/C++ Codes bezüglich Verarbeitungszeit, Aufrufhäufigkeit, Komplexität und Konnektivität. Darauf ba-

sierend erfolgt das Scheduling. Hierfür werden die Adjazenzmatrix und die inverse Adjazenzmatrix des Algorithmus gebildet sowie die Level und Co-Level der Funktionen berechnet. Nach einer Clusterung der Basisfunktionen zu einem Multiprozessorsystem nach Sakar, werden die zuvor aufgestellten Echtzeitbedingungen überprüft. Sind sie nicht erfüllt, wird zur eigentlichen Partitionierung in Phase zwei übergegangen.

Die Partitionierung erfolgt auf Basis des Simulated Annealing, so dass auch ein Verlassen von lokalen Minima möglich ist. Dabei wird das Multiprozessorsystem wieder aufgelöst und den einzelnen Funktionen eine Priorität auf Basis ihrer Level zugeordnet. Die Funktion mit der höchsten Priorität wird in Logik überführt. Die als Software verbleibenden Funktionen werden wieder zu einem Multiprozessorsystem zusammengeführt. Anschließend werden die Kosten der gesamten Partition durch eine Kostenfunktion ermittelt. Die Kostenfunktion ist eine gewichtete Addition der einzelnen Kostenfaktoren. Die Kostenfaktoren Verarbeitungskosten, Kommunikationskosten, Hardwareaufwand und Entwicklungszeit werden mithilfe der in der Analyse ermittelten Daten geschätzt. Die Partition wird durch den Metropolalgorithmus, auf Basis der geschätzten Kosten, angenommen bzw. abgelehnt. Phase zwei wird solange durchgeführt, bis die Echtzeitbedingungen erfüllt sind oder der Kontrollparameter des Simulated Annealing seine Abbruchbedingung erreicht hat.

Als Anwendungsbeispiel für das Partitionierungsystem dient ein Assistenzsystem mit zwei im Fahrzeug angebrachten Kameras, die im Normalfall der Stereophotogrammetrie ausgerichtet sind und den Rückraum des Fahrzeuges bei Autobahnfahrten überwachen. Das Assistenzsystem soll den Fahrer bei Überholmanövern vor Gefahren durch schnell herannahende Fahrzeuge warnen. Um dem Fahrer ausreichend Reaktionszeit zu gewähren, ergibt sich als Echtzeitbedingung des Systems eine maximale Latenzzeit von $200ms$, bei einer Bildfrequenz der Kameras von $25Hz$.

Es ist gezeigt worden, dass das Partitionierungssystem nach 15 Iterationsschritten eine Partition mit minimalen Kosten findet. Durch die reale Implementierung dieser Partition konnte die Echtzeitfähigkeit bewiesen werden. Das Assistenzsystem wurde dabei als Logik und als Software - verteilt auf vier NIOS II Prozesso-

ren - realisiert. In dieser Form wurde eine Gesamtverarbeitungszeit von $116,9ms$ bei einer maximalen Verarbeitungszeit einer Pipelinestufe von $38,7ms$ erreicht. Anhand der durch die Implementierung vorliegenden Werte für den Hardwareaufwand, der Entwicklungszeit für die Logik sowie der Verarbeitungszeiten auf den Prozessoren und in der Logik, konnten die durch Schätzung während der automatischen Partitionierung ermittelten Werte in guter Näherung bestätigt werden.

Literaturverzeichnis

[1] T. Adam and J. D. K.M. Chandy. *A comparision of list schedules for parallel processing systems. ACM*, 17:685–690, 1974.

[2] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. *Pyramid methods in image processing. RCA Engineer*, 29-6:33–41, 1984.

[3] A. Agarwal and M. Levy. *Going multicore presents challenges and opportunities. Embedded System Design Europe*, 4:28–33, 2007.

[4] A. Aiken and A. Nicolau. *Optimal Loop Parallelization. ACM-Sigplan*, pages 308–317, 1988.

[5] J. Albertz and W. Kreiling. *Photogrammetric Guide.* Herbert Wichmann Verlag, 4. edition, 1989.

[6] P. Albrecht and B. Michaelis. *Improvement of Spatial Resolution of an Optical 3-D Measurement Procedure. IEEE Instrumentation and Measurement*, 47:158–162, 1998.

[7] Altera. *Simultaneous Multi-Mastering with the Avalon Bus*, 1 edition, 2002.

[8] Altera. *Stratix Device Handbook*, 1 edition, 2004.

[9] G. Amdahl. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS Conference Proceedings*, 30:483–485, 1967.

[10] H. Arnold. *Konfigurierbare Prozessoren für Basisstationen. Markt&Technik*, 32:28–30, 2005.

[11] P. F. Aschwanden. *Experimenteller Vergleich von Korrelationskriterien in der Bildanalyse.* PhD thesis, ETH Zurich, 1993.

[12] A. Baghdadi, N. E. Zergainoh, W. O. Cesario, and A. A. Jerraya. *Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems. IEEE Trans. on Software Engineering*, 28(9):822–831, 2002.

[13] S. Bakshi and D. Gajski. *Hardware/Software Partitioning and Pipelining. ACM IEEE Proc. of the 34th conference on Design automation*, pages 713 – 716, 1997.

[14] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabarra. *Hardware-Software Co-Design of Embedded Systems.* Kluwer Academic Publishers, 1997.

[15] M. Bertozzi and A. Broggi. *GOLD: A parallel Real-Time Stereo Vision System for Generic Obstacle and Lane Detection. IEEE Trans. Image Processing*, pages 62–81, 1998.

[16] M. Bertozzi, A. Broggi, and A. Fascioli. *Vision-based intelligent vehicles: state of the art and perspectives. Robotics and Autonomous Systems*, 32:1–16, 2000.

[17] M. Betke and H. Nguyen. *Highway Scene Analysis from a Moving Vehicle under Reduced Visibility Conditions. Proceedings of the International Conference on Intelligent Vehicles*, pages 131–136, 1998.

[18] B. Boehm. *Safe and simple software cost analysis. IEEE Software*, 17:14–17, 2000.

[19] C. Boeres and A. Lima. *Hybrid Task Scheduling: Integrating Static and Dynamic Heuristics. Proc. of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 199–206, 2003.

[20] M. Borg, R. Mentzer, and K. Singh. *Digital imaging using CMOS sensors. International IC China*, pages 37–47, 2001.

[21] G. Bosmann. *A Survey of Co-Design Ideas and Methodolgies.* vrije Unversität, 2000.

[22] P. Boulet, A. Darte, G. Silber, and F. Vivien. *Loop parallelization algorithms: From parallelism extraction to code generation. Parallel Computing*, 24:421–333, 1998.

[23] P.-J. Burt and E.-H. Adelson. *The Laplacian Pyramid as a Compact Image Code. IEEE Transactions on Communications*, 31:532–540, 1983.

[24] W.-S. C. *The transputer. SIGARCH Comput. Archit. News*, 13(3):292–300, 1985.

[25] T. Camus, D. Coombs, M. Herman, and T. Hong. *Real-Time Single Workstation Obstacle Avoidance Using Only Wide-Field Flow Divergence. VIDRE: Journal of Computer Vision Research*, 1(3):30–58, 1999.

[26] E. Clarke. *FPGAs-programmierbare Systeme auf einem Chip. Elektronik*, 5:36–39, 2004.

[27] B. Dalay. *Accelerating system performance using SOPC Builder. System-on-Chip 2003*, pages 3–5, 2003.

[28] R. Diestel. *Graphentheorie.* Springer Verlag, 2 edition, 2000.

[29] K. C. J. Dietmeyer, J. Sparbert, and D. Streller. *Model Based Object Klassifikation and Object Tracking in Traffic Scenes from Range Images. Proc. of IV01, IEEE Intelligent Vehicles Symposium*, pages 2–1, 2001.

[30] G. Doblinger. *Signalprozessoren.* J. Schlembach Fachverlag, 2 edition, 2004.

[31] R. Dutta and C. Weems. *Parallel Dense Depth from Motion on the Image Understanding Architecture. Proc. of the IEEE CVPR*, pages 154–159, 1993.

[32] C. Ebert and R. Dumke. *Software-Metriken in der Praxis.* Springer Verlag, 1996.

[33] H. El-Remini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[34] F. Fahid and D. D. Gajski. *Incremental Hardware Estimation during Hardware/Software Functional Partitioning. Trans. on VLSI Systems*, 3(3):459–464, 1995.

[35] U. Franke, D. Gavrila, S. Gorzig, F. Lindner, F. Paetzold, and C. Wohler. *Autonomous Driving Goes Downtown. IEEE Intelligent Systems*, 13(6):40–48, 1998.

[36] G. Färber. *Arbeitstiere für Echtzeitsysteme.* TU München, 1 edition, 2004.

[37] L. Gaafar and S. Masoud. *Genetic algorithms and simulated annealing for scheduling in agile manufacturing. Int. Journal of Production Research*, 43(14):3069–3085, 2005.

[38] A. Gerasoulis and T. Yang. *A comparision of clustering heuristics for scheduling DAGs on multiprocessors. Journal of distributed computing*, pages 138–153, 1992.

[39] G. Govindu, R. Scrofano, and V.-K. Prasanna. *A Library of Parameterizable Floating Point Cores for FPGAs and their Application to Scientific Computing. Proc. Int. Conf. Eng. Reconfigurable Systems and Algorithms*, 2005.

[40] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* Kluwer Academic Publishers, 1995.

[41] R. K. Gupta and G. D. Micheli. *Hardware/Software Cosynthesis for Digital Systems. IEEE Design and Test of Computers*, pages 29–41, 1993.

[42] S. Habata, M. Yokokawa, and S. Kitawaki. *The Earth Simulator System. NEC Res. & Development*, 44(1):21–26, 2003.

[43] M. Halstead. *Natural Laws Controlling Algorithm Structure. ACM SIGPLAN Notices*, 7(2):19–26, 1972.

[44] H.-U. Heiss. *Prozessorzuteilung in Parallelrechnern.* Wissenschaftsverlag, 1998.

[45] J. Henkel. *Automatisierte Hardware/Software-Partitionierung im Entwurf integrierter Echtzeitsysteme.* Shaker Verlag, 1996.

[46] L. Hoschek. *Grundlagen der geometrischen Datenverarbeitung.* Teubner Verlag, 1992.

[47] M. Isard and A. Blake. *Contour tracking by stochastic propagation of conditional density. Proc. European Conf on Computer Vision ECCV*, pages 343–356, 1996.

[48] M. Isard and A. Blake. *Condensation - condensational density propagation for visual tracking. International Journal of Computer Vision*, 29:5–28, 1998.

[49] A. Jaenicke and W. Luk. *Parameterised Floating-Point Arithmetic on FPGAs. Proc. of the IEEE conf. on Acoustics, Speech, and Signal Processing*, 2:897–900, 2001.

[50] B. Jähne. *Digitale Bildverarbeitung.* Springer Verlag, 5 edition, 2002.

[51] A. Kalavade and E.-A. Lee. *Hardware/Software Co-Design Using Ptolemy - A Case Study. Univerity of California at Berkeley*, pages 1–18, 1992.

[52] C. Kanz. *Produkthaftung und Produktsicherheit: Welche Rolle können Risiko-Nutzen-Analyse und Code of Practice spielen. 3. Workshop Fahrerassitenzsysteme FAS2005*, pages 86–91, 2005.

[53] J. Kaszubiak, M. Tornow, R. Kuhn, and B. Michaelis. *Real-time,3-D-multi object position estimation and tracking. 17. Int. Conference on Pattern Recognition*, 1:785–788, 2004.

[54] J. Kaszubiak, M. Tornow, R. Kuhn, B. Michaelis, and C. Knöppel. *Real-Time Vehicle and Lane Detection with Embedded Hardware. IEEE Intellingent Vehicle Symposium*, pages 618–623, 2005.

[55] B. Kerninghan and S. Lin. *An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal*, 220, 1970.

[56] S.-M. Kim, J.-H. Park, S.-M. Park, B.-T. Koo, K.-S. Shin, K.-B. Suh, I.-K. Kim, N.-W. Eum, and K.-S. Kim. *Hardware-software implementation of MPEG-4 video codec. Electronics and Telecommunications Research Institute journal*, 25(6):489–502, 2003.

[57] S. Kirkpatrick, C. Gelatt, and M. Vecchi. *Optimization by simulated annealing. Science*, pages 671–680, 1983.

[58] C. Klauck and C. Maas. *Graphentheorie und Operations Resarch für studierende der Informatik.* Wißner Verlag, 1999.

[59] K. Kluge. *Extracting Road Curvature and Orientation from Image Edge Points without Perceptual Grouping into Features. Proc. IEEE Intelligent Vehicle Symposium*, pages 109–114, 1994.

[60] K. Kluge and S. Lakshmanan. *A Deformable Template Approach to Lane Detection. Proc. IEEE Intelligent Vehicle Symposium*, pages 54–59, 1995.

[61] C. Knoeppel. *Stereobasierte und spurgenaue Erkennung von Strassenfahrzeugen im Rueckraum eines Strassenfahrzeuges.* PhD thesis, Universität Magdeburg, 2001.

[62] H. Kopetz. *Research Report: A Comparision of TTP/C and FlexRay.* TU Wien, 10/2001 edition, 2001.

[63] K. Kraus. *Photogrammetrie - Band 1 Grundlagen und Standardverfahren.* Dümmler Verlag, 1990.

[64] J. Krieger. *Stoffzusammenfassung zur Bild- und Signalverarbeitung.* Universität Heidelberg, 2006.

[65] P. J. M. Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications.* Kluwer Academic Publishers, 1989.

[66] E. Lagnese and D. Thomas. *Architectural partitioning for system level synthesis of integrated circuits APARTY. IEEE Transactions on Computer Aided Design of ICs and Systems*, 10:847–860, 1991.

[67] J. Lahtinen and P. Silander. *Empirical Comparison of Stochastic Algorithms. Proc. of the Second Nordic Workshop on Genetic Algorithms and their Applications*, 1996.

[68] M. Lajolo, M. Lazarescu, and A. S. Vincentelli. *A Compilation-based Software Estimation Scheme for Hardware/Software Co-Simulation. CODES*, pages 85–89, 1999.

[69] J. Langheim, A. Buchanan, U. Lages, and M. Wahl. *CARSENSE- New environment sensing for advanced driver assistance systems. IEEE Intellingent Vehicle Symposium*, pages 89–94, 2001.

[70] W. Lawrenz. *CAN-Controller Area Network.* Huetig, 1999.

[71] S. Lee and Y. Kay. *A Kalman filter approach for accurate 3D motion estimation from a sequence of stereo images. 10th International Conference on Pattern Recognition*, pages 104–108, 1990.

[72] H. Leung and Z. Fan. *Software Cost Estimation. Handbook Software Engineering and Knowledge Engineering*, 2:1–14, 2002.

[73] H. Liebig. *Rechnerorganisation.* Springer Verlag, 2002.

[74] O. Loffeld. *Estimationstheorie, Band I und II.* Oldenburg Verlag, 1990.

[75] T. Mainikas and J. Cain. *Genetic Algorithms vs.. Simulated Annealing A Comparison of Approaches for Solving the Circuit Partitioning Problem. Techincal Report*, 96(101):1–15, 1996.

[76] S. Matsuda. *Optimal Hopfiel Network for Combinational Optimization with Linear Cost Funktion. Transactions on Neural Networks*, 9(6):1319–1330, 1998.

[77] T. J. McCabe and A. H. Watson. *Software Complexity. Journal of Defense Engineering 7*, 12:5–9, 1994.

[78] R. Mecke. *Grauwertbasierte Bewegungsschätzung in monokularen Bildsequenzen unter besonderer Berücksichtigung bildspezifischerim Störungen.* PhD thesis, Universität Magdeburg, 1999.

[79] M. Mehrwein. *Ein wiederverwendungsorientiertes Hardware/Software-Codesign-System für Mikrocontrollerbasierte Systeme.* VDI Verlag, 2002.

[80] G. Meynants, B. Dierickx, D. Scheffer, and J. Vlummens. *A wide dynamic range CMOS stereo camera. Advanced Microsystems for Automotive Applications*, 1998.

[81] H. Müller. *Mikroprotessortechnik.* Vogel Fachbuch, 2002.

[82] P. Musilek, W. Pedrycz, N-Sun, and G. Succi. *On the Sensitivity of COCOMO II Software Cost Model. 8. IEEE Symposium on Software Metrics*, page 13, 2002.

[83] M. Nölle. *Konzepte zur Entwicklung paralleler Algrithmen der digitalen Bildverarbeitung.* VDI Verlag, 1996.

[84] W. Oberschelp and G. Vossen. *Rechneraufbau und Rechnerstrukturen.* Oldenbourg Verlag, 2003.

[85] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization (Algorithms and Complexity).* Prentice Hall, 1982.

[86] M. Parkinson, P. Taylor, and S. Paramerswaran. *C to VHDL Converter in a Codesign Environment. Proceedings of VHDL international Users Forum*, pages 100–108, 1994.

[87] P.Chou, R. Ortega, and G. Borriello. *The Chinook Hardware/Software Co-Synthesis System. Int. Symp. on Systems and Synthesis*, pages 22–27, 1995.

[88] M. Platzner. *Hardware/Software Codesign.* ETH Zürich, 2002.

[89] P.Mengel, C.Doemens, and L. Listl. *Fast range imaging by CMOS sensor array through multiple double short time integration (MDSI). International Conference on Image Processing (ICIP)*, 2:169–172, 2002.

[90] R.-B-Jones and V.-H. Allan. *Software Pipelining: A Comparision and Improvement. Proc. of the 23rd symposium on Microprogramming and microarchitecture*, pages 46–56, 1990.

[91] R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Pub., 1993.

[92] R.-M. Russell. *The CRAY-1 computer system. Commun. ACM*, 21(1):63–72, 1978.

[93] S. Salleh and A. Y. Zomaya. *Schedulling in Parallel Computing Systems.* Kluwer Academic Publishers, 1999.

[94] V. Sarkar. *Partitioning and Scheduling Parallel Programms for Execution on Multi-Processors.* MIT Press, 1989.

[95] M. Schamberger. *Fahrerassistenzsysteme - wo stehen wir heute, was sind die Potentiale von morgen. 3. Workshop Fahrerassistenzsysteme FAS2005*, pages 86–91, 2005.

[96] H. Schlitt. *Systemtheorie für stochastische Prozesse.* Springer Verlag, 1992.

[97] P. Schroeter and J. Biguen. *Hierarchical Image Segmentation by Multi-Dimensional Clustering and Orientation-Adaptive Boundary Refinement. Pattern Recognition*, 28:695–709, 1995.

[98] L. Semeria and A. Gosh. *Methodology for Hardware/Software Co-verification in C/C++. Proc.of the conf. on Asia South Pacific design automation*, pages 405–408, 2000.

[99] C. Siemers. *PLDs und Mikrocontroller/Mikroprozessoren. Entwicklerforum Programmierbare Logik/München*, 4:118–192, 1997.

[100] C. Siemers. *Die Welt der rekonfigurierbaren Prozessoren. Elektronik*, 21:42–48, 2005.

[101] A. Sikora. *Programmierbare Logikbauelemente.* Carl Hanser Verlag, 2001.

[102] A. Sikora and R. Drechsler. *Software-Engineering und Hardware-Design.* Carl Hanser Verlag, 2001.

[103] J. Song, S. Na, H.-G. Kim, H. Kim, and C.-S. Lin. *Depth Measurement Associated with the Mono Camera System with a Rotating Mirror. Proceedings of the Third IEEE Pacific Rim Conference*, pages 1145 – 1152, 2002.

[104] A. Suppes. *Stochastische Hindernisdetektion aus stereoskopischen Videosequenzen fuer fahrerlose Transportfahrzeuge.* VDI Verlag, 2004.

[105] K. Suzuki and A. S. Vincentelli. *Efficient software performance estimation methods for hardware/software codesign. Proceedings of the Design Automation Conference*, pages 605 – 610, 1996.

[106] I. D. Svalbe. *Natural Representations for Straight Lines and the Hough Transform on Discrete Arrays. IEEE Trans. Pattern Anal. Mach. Intell.*, 11:941–950, 1989.

[107] E. Tell, O. Seger, and D. Liu. *A Converged Hardware Solution for FFT, DCT and Walsh Transform. Proc. of the 8th. Int. Symposium on Signal Processing an its Applications*, pages 609–612, 2003.

[108] G. R. Thomas Rauber. *Parallele und verteilte Programmierung.* Springer, 2000.

[109] K.-D. Toennies. *Grundlagen der Bildverarbeitung.* Person Verlag, 2005.

[110] M. O. Tohki, M. A. Hossain, and M. H. Shaheed. *Parallel Computing for Real-Time Signal Processing and Control.* Springer Verlag, 2003.

[111] M. Tornow, B. Michaelis, R. Kuhn, R. Calow, and R. Mecke. *Hierarchical method for stereophotogrammetric multi-object-position Measurement. Pattern Recognition, DAGM Symposium*, pages 164–171, 2003.

[112] J. Tyrrel. *Low-cost sensor puts 3D-cameras in the picture. OptoandLaser Europe The European magazine for photonics professionals*, pages 20–21, 2004.

[113] B. Ulmer. *VITA - An Autonomous Road Vehicle for Collision Avoidance in Traffic. Intelligent Vehicles*, pages 36–41, 1992.

[114] P. Venhovens, K. Naab, and B. Adiprasito. *Stop and Go Cruise Control. Int. Journal of Automotive Technology*, 1:61–69, 2000.

[115] D. Verkest, J. Kunkel, and F. Schirrmeister. *System level design using C++. Proc. of the conf. on Design, automation and test in Europe*, pages 74–83, 2000.

[116] C. E. Walston and C. P. Felix. *A method of programming measurement and estimation. IBM Systems Journal*, 16(1):54–73, 1977.

[117] Y. Wang, E. K. Teoh, and D. Shen. *Lane detection and tracking using B-Snake. Image and Vision Computing*, 22:269–280, 2004.

[118] R. P. Weicker. *Dhrystone: a synthetic systems programming benchmark. Commun. ACM*, 27(10):1013–1030, 1984.

[119] T. Wiangtong, P.-Y. Cheung, and W. Luk. *Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware-Software Codesign. Design Automation for Embedded Systems*, 6:435–449, 2002.

[120] W. Wolf. *A Decade of Hardware/Software Codesign. IEEE Int. Symposium on Multimedia Software Engineering*, pages 38–42, 2003.

[121] T. Yang and A. Gerasoulis. *A dominant sequence clustering heuristic algorithm for scheduling DAGs on multiprocessors.* Technical Report, 1991.

[122] S. Zammattio. *Komplexe FPGAs und adäquate Tools für die SOC-Entwicklung. Elektronik*, 10:20–22, 2005.

[123] V. Zivojnovic and H. Meyr. *Compiled hw/sw co-simulation. Proceedings of the Design Automation Conference*, 1996.

Anhang A

Graphentypen

Hier sollen nun die häufig verwendeten Graphentypen vorgestellt werden. Beim **vollständigen Graphen** ist jeder Knoten direkt mit jedem anderen Knoten des Graphen verbunden. Mit dem Durchmesser $\delta(G) = 1$. Bei n Knoten gilt daher ein *Grad* $g(G) = n - 1$. Dies gilt auch für die Knoten- und Kantenkonnektivität.

Das **Ring-Netzwerk** ist ein erweitertes lineares Feld. Hierbei werden die Knoten in Form eines Ringes angeordnet, wobei der erste und der letzte Knoten ebenfalls miteinander verbunden sind. Der Durchmesser beträgt $\delta(G) = n/2$. Grad und Konnektivität beträgt 2.

Ein **d-dimensionales Gitter** besteht aus $n = n_1 \cdot n_2 \cdot ... \cdot n_d$ Knoten, die quadratisch angeordnet sind. Solch ein Gitter hat Nachbarschaftsbeziehungen in alle Himmelsrichtungen (von den Randknoten abgesehen).

Ein **d-dimensionaler Torus** ist ein d-dimensionales Gitter mit zyklisch geschlossenen Rändern.

Ein Graph mit konstantem Knotengrad bei logarithmisch ansteigendem Durchmesser ist der **k-dimesionale DeBrujin-Graph**. Ein DeBrujin-Graph der Dimension d besitzt $n = 2^d$ Knoten und einen Durchmesser von d. Der Knotengrad $g(G) = 4$.

Der **k-dimensionale Hyperwürfel** hat $n = 2^k$ Knoten, zwischen denen Kanten entsprechend eines rekursiven Aufbaus aus niedrigerdimensionalen Würfeln exi-

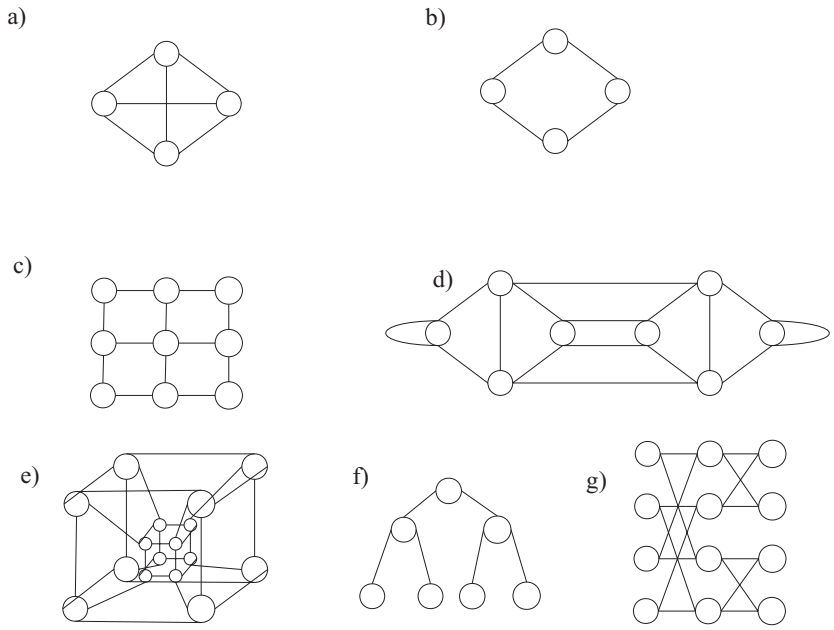

Abbildung A.1: Graphentypen a) vollständiger Graph, b) Ring c) 2-dimensionales Gitter, d) DeBrujin Graph, f) 4-dimensionaler Würfel, g) Binärer Baum h) Butterfly Graph

stieren. Er besitzt einen sehr günstigen Durchmesser und ist deshalb gut für eine nichtlokale Kommunikation geeignet. Hier wächst der Knotengrad mit steigender Dimension des Würfels.

Der **k-dimensionale Butterfly Graph** bietet hierfür ein Lösung. Er besitzt, genau so wie der DeBrujin Graph, einen logarithmisch anwachsenden Durchmesser bei konstantem Knotengrad.

Der Durchmesser des **k-stufigen binären Baumes** wächst linear mit der Höhe h des Baumes und damit logarithmisch zur Knotenzahl. Die Konnektiviät beträgt dabei maximal 3. Beim Abtrennen eines Knotens wird auch der jeweilige Zweig abgetrennt. In diesem Fall erfolgt der gesamte Datenverkehr zwischen linkem und rechtem Zweig über die Wurzel, was zu einer hohen Belastung führen kann. Durch Querverbindungen ist es möglich, diesen Mangel teilweise zu kompensieren.

In Tabelle A.1 sind die Parameter der vorgestellten Graphen zusammengefasst.

Graphen können in andere Graphen eingebettet werden. Dias ist nötig, wenn

Graph G mit n Knoten	Grad $g(G)$	Durchmesser $\delta(G)$	Kantenkonektivität $ec(G)$
Vollständiger Graph	$n-1$	1	$n-1$
Ring	2	$n/2$	2
d-dimensionales Gitter $n = d^d$	$2d$	$d(\sqrt[d]{n-1})$	d
d-dimensionaler Torus $n = d^d$	$2d$	$d(\frac{\sqrt[d]{n}}{2})$	$2d$
k-dimensionaler Hyperwürfel $n = 2^k$	$logn$	$logn$	$logn$
k-dimensionaler DeBrujin Graph $n = 2^k$	4	k	4
k-dimensionaler Butterfly Graph $n = k \cdot 2^k$	4	$k + (k/2)$	4
k-stufiger Binärer Baum $n = 2^k - 1$	3	$2log\frac{n+1}{2}$	1

Tabelle A.1: Eigenschaften ausgewählter Graphen

Softwaregraphen (Programm) auf Hardwaregraphen (Prozessoren, Prozessorelemente) abgebildet werden müssen. Der Idealfall wäre gegeben, wenn der Prozessorgraph die gleiche Struktur wie der Softwaregraph besitzt. Im Normalfall ist die Struktur des Softwaregraphen wesentlich komplexer als die Struktur des Hardwaregraphen. Deshalb müssen eine Anzahl von Softwareknoten auf einen Hardwareknoten implementiert werden. Dabei sollte die Anzahl der Softwarekanten aus dem Hardwareknoten klein sein, um den entstehenden Kommunikationsoverhead zu minimieren.

Anhang B

Ausführungen zum Assistenzsystem im KFZ

B.1 Hierarchische Aufteilung des Messbereiches

Der Messbereich des Systems soll sich in Z-Richtung von $-150m$ bis $-10m$ erstrecken. Bei großen Entfernungen sind die Objekte (z.B. Autos) sehr klein (wenige Pixel) und auch die Disparität nimmt nur geringe Werte an. Im Gegensatz dazu sind nahe Objekte relativ groß, die Disparität nimmt große Werte an. Die hohe Auflösung naher Objekte ist meist nicht erforderlich und kann sogar störend wirken. Deshalb besteht die Grundidee der Methode darin, die Bildauflösung für die Lageberechnung naher Objekte zu reduzieren. Die dadurch gleichzeitig verringerte Disparität (in Pixeln der aktuellen Auflösung) bedingt die Berechnung einer wesentlich geringeren Zahl an Korrelationswerten bei der vollen Suche.

Dafür werden Ebenen mit zugeordneten Tiefenbereichen verschiedener Auflösung eingeführt [111]. Die Erzeugung der einzelnen Ebenen erfolgt durch die Verringerung der Auflösung der Zeilen um einen festen Faktor für jede weitere Ebene. In Abbildung B.1.a ist ein Beispiel (typische Lösung) für den Faktor 1/2 dargestellt. Dies geschieht, indem mehrere benachbarte Pixel durch einen ersetzt werden, hier im Beispiel im Verhältnis 2:1. Die Auflösungsreduzierung erfolgt durch Mittelwertbildung (typische Lösung). Die einzelnen Ebenen werden jeweils mit dem

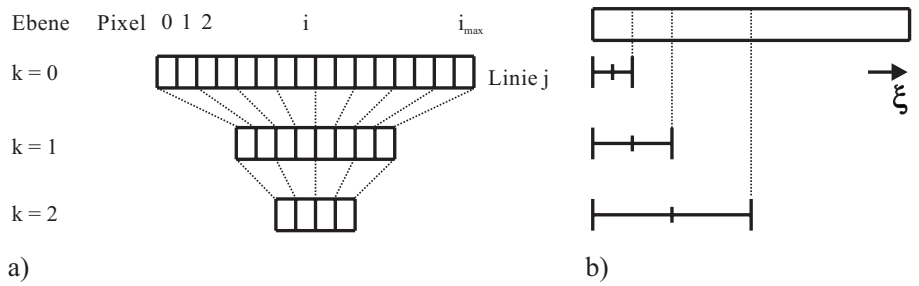

Abbildung B.1: a) Ebenengenerierung b) Suchbereich je Ebene

gleichen Algorithmus aus der vorhergehenden berechnet. Die maximale Verschiebung des Suchblockes beträgt jeweils 16 Pixel, wobei die unteren 8 Pixel bereits von der vorhergehenden Ebene abgedeckt werden (siehe Abbildung B.1.b). Trotz der festen maximalen Verschiebung, ergibt sich durch die Ebenengenerierung eine Verdopplung des Suchbereiches pro Ebene. Die Zeilen der Bilder in den verschiedenen Auflösungen werden dann zur Bestimmung der Disparität miteinander, wie in Kapitel 4.1.1 beschrieben, korreliert.

Zur Ermittlung der Disparität, werden die Pixel auf der Epipolarlinie der beiden Kameras korreliert. Die Such- und Referenzblöcke, zur Durchführung der Korrelation, haben eine Dimension von 16x1 Pixeln. Der Suchblock wird Pixel für Pixel über den Suchbereich (16 Pixel) des rechten Bildes geführt, beginnend an der Position des Referenzblockes im linken Bild. Der Korrelationswert wird für jede Position und für jede Ebene berechnet. Um die Korrelationswerte der ver-

Ebene k	Auflösung	Suchbereich [Pixel]	Anzahl der Korrelationen	Messbereich [m]
0	1/1	16	16	∞ -112
1	1/2	32	8	112-56
2	1/4	64	8	56-28
3	1/8	128	8	28-14
4	1/16	256	8	14-7

Tabelle B.1: Ebenen mit entsprechenden Tiefenbereichen

schiedenen Ebenen vergleichen zu können, wird ein Wichtungsfaktor benutzt. Die Position des maximalen Korrelationswertes stellt die Disparität dar.

Durch die Generierung der Ebenen ergibt sich eine Aufteilung des Systemmessbereiches, wie in Tabelle B.1 erläutert und in Abbildung B.2 dargestellt. Beim hierarchischen Ansatz repräsentiert jede Ebene einen bestimmten Tiefenbereich. Die Verteilung der verschiedenen Tiefenbereiche auf die einzelnen Ebenen hängt vom Stereosystem ab, insbesondere von der Kamerakonstante c und der Basisbreite B. Die konkrete Auflösung der Kameras spielt für die Größe der einzelnen Messbereiche keine Rolle, aber für die erreichbare Auflösung.

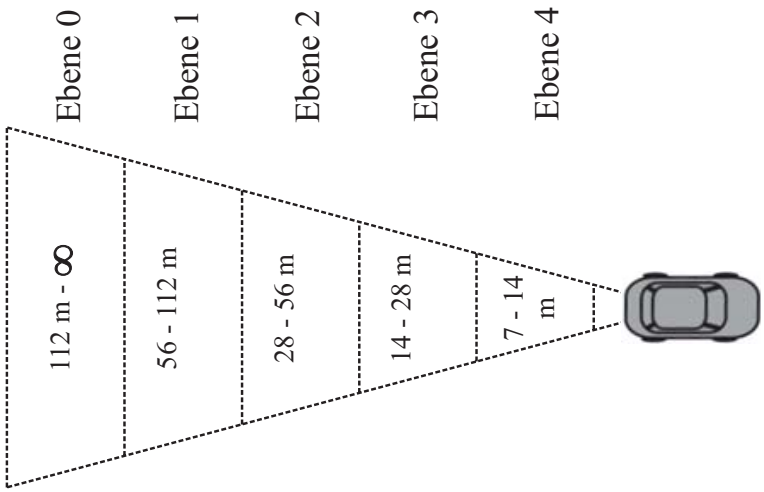

Abbildung B.2: Detektionsbereich des Systems mit Ebenen

Die Bildpyramide

Verfahren mit adaptiver Auflösung sind bereits seit längerem bekannt und werden häufig in der Bildkompression eingesetzt. Die Laplace-Pyramide [23] stellt hierbei ein reversibles Verfahren dar. Im vorgestellten Algorithmus wird eine Grauwert-Pyramide [2] verwendet, die zu den irreversiblen Verfahren zählt, da die Mittelwertbildung zweier benachbarter Pixel zur Ermittlung des Pixelgrauwertes in der nächsthöheren Ebene ein Tiefpassverhalten des Pyramidenalgorithmus erzeugt.

Durch das Tiefpassverhalten wird eine Kante im Bild um ein Pixel je Ebene verschliffen (siehe Gleichung B.1). Je höher die Ebene, um so stärker ist eine

Kante verschliffen. k ist dabei die Ebene, in die die Kante projiziert wird und W_k die Breite der Kante in Pixeln in der entsprechenden Ebene.

$$W_k = W_0 + k \tag{B.1}$$

Ein Fahrzeug besitzt mindestens eine linke und eine rechte Kante, die jeweils verschliffen wird. Kann ein Fahrzeug in einer bestimmten Entfernung, breiter als $2W_k$ Pixel auf dem Kamerachip, abgebildet werden, ist es möglich das Fahrzeug trotz der verschliffenen Kanten zu erkennen und durch den Korrelationsalgorithmus zu detektieren.

Die genauen Untersuchungen der Merkmalsgewinnung und die Erstellung der Tiefenkarte durch die KKFMF sind Bestandteil einer anderen Arbeit und werden daher nicht eingehender beschrieben.

B.2 Subpixelinterpolation

In Abbildung B.3 ist die graphische Darstellung der hier verwendeten Subpixelinterpolation mit drei Stützstellen zu sehen. Als erste Stützstelle wird der Ort des Korrelationsmaximums gewählt, der entsprechende Stützpunkt ist das Korrelationsmaximum. Weitere Parameterpaare liegen jeweils links und rechts des Maximums.

Für $(n+1)$ Parameterpaare (Δu_i) gibt es ein Polynom n-ten Grades.

$$Q_i = \sum_{j=0}^{n} A_j \cdot (\Delta u_i)^j \tag{B.2}$$

In diesem Fall soll ein Polynom zweiten Grades entwickelt werden, d.h. Q_i lässt sich wie folgt berechnen.

$$Q_i = A_0 + A_1 \cdot \Delta u_i + A_2 \cdot (\Delta u_i)^2 \tag{B.3}$$

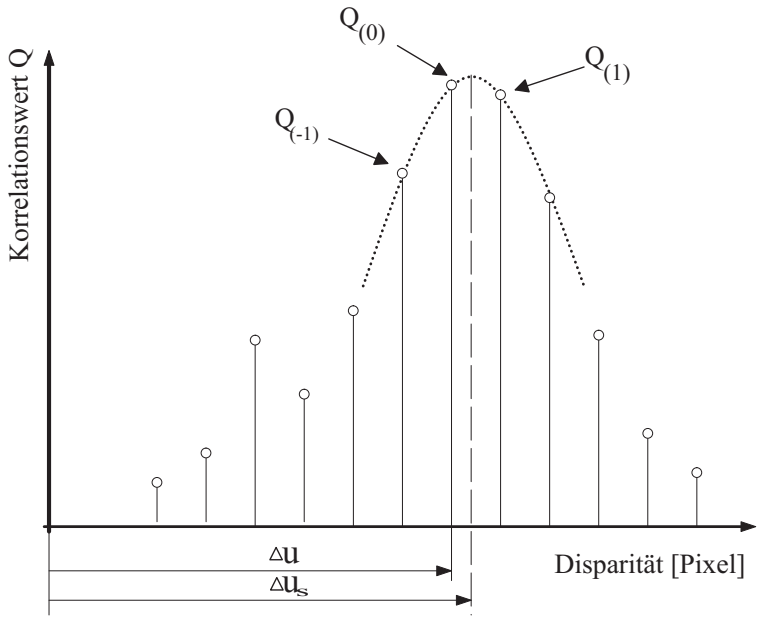

Abbildung B.3: Subpixelgenaue Ermittlung des Korrelationsmaximums

Wenn nun das Polynom um das Korrelationsmaximum herum für den Vektor $\Delta u_i = -1, 0, 1$ entwickelt wird, ergibt sich durch Umstellung nach A Gleichung B.4.

$$\begin{pmatrix} A_2 \\ A_1 \\ A_0 \end{pmatrix} = \begin{pmatrix} 0,5 & -1 & 0,5 \\ -0,5 & 0 & 0,5 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} Q_1 \\ Q_0 \\ Q_{-1} \end{pmatrix} \tag{B.4}$$

Mit der ersten Ableitung der Polynomfunktion wird das Extremum berechnet. Setzt man die 1. Ableitung Null lässt sich nach Umstellen und Einsetzen von Gleichung B.4 die subpixelgenaue Disparität bestimmen.

$$\Delta u_s = \Delta u + \frac{\frac{1}{2}(Q_1 - Q_{-1})}{2Q_0 - Q_1 - Q_{-1}} \tag{B.5}$$

Hierbei ist Δu_s die berechnete subpixelgenaue Disparität, Δu die ermittelte Disparität, Q_0 ist der Korrelationskoeffizient des Maximums, Q_{-1} und Q_1 sind die Korrelationskoeffizienten jeweils ein Pixel links bzw. rechts vom Maximum.

In Abbildung B.4 ist der Fehler des Systems für die Entfernungsmessung mit und ohne Subpixelinterpolation dargestellt. In den Diagrammen a) und c) ist jeweils

die gemessene Entfernung über der realen Entfernung eingetragen. Weiterhin ist die zu der realen Entfernung korrespondierende Entfernung durch die Ebenengenerierung dargestellt. In Abbildung B.4.a) ist die dadurch erzeugte Stufung sehr gut zu erkennen. Die Abbildungen b) und d) zeigen den jeweiligen Fehler zwischen realem Entfernungswert und den durch die Ebenenerzeugung ermittelten Wert. Es ist zu erkennen, dass der Fehler zum Koordinatenursprung hin abnimmt, was ein Verbesserung der Messgenauigkeit bedeutet. Ein Vergleich der Fehler in Abbildung b) und d) zeigt, dass durch die Subpixelinterpolation die Abweichung zwischen realem Wert und Messwert stark reduziert werden kann.

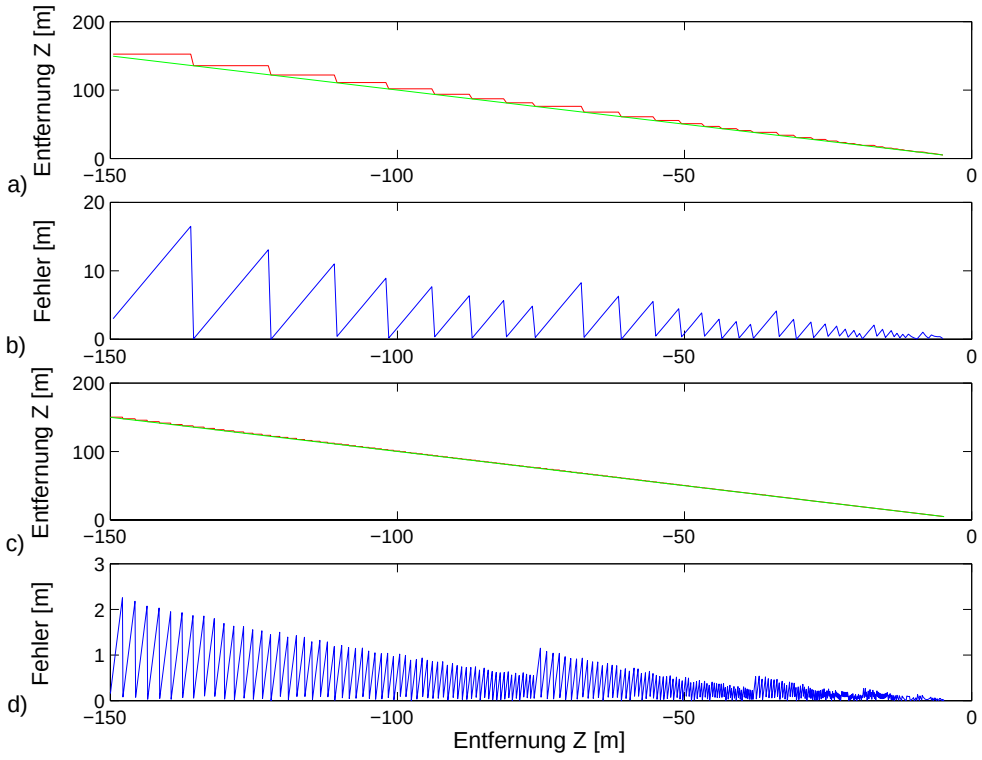

Abbildung B.4: a) Messkurve ohne Interpolation b) Fehler ohne Interpolation c) Messkurve mit Interpolation d) Fehler mit Interpolation

B.3 Weiterverfolgung

Ziel der Weiterverfolgung ist es, die Position und die Geschwindigkeit der erkannten oder kurzzeitig verdeckten Fahrzeuge im System zu schätzen und sie bei einer erneuten Detektion im folgenden Bild wieder richtig zuordnen zu können.

In der Rückraumszene eines Fahrzeuges können mehrere Fahrzeuge auftauchen. Diese müssen entsprechenden Kalman-Filtern über mehrere Bilder zugewiesen werden.

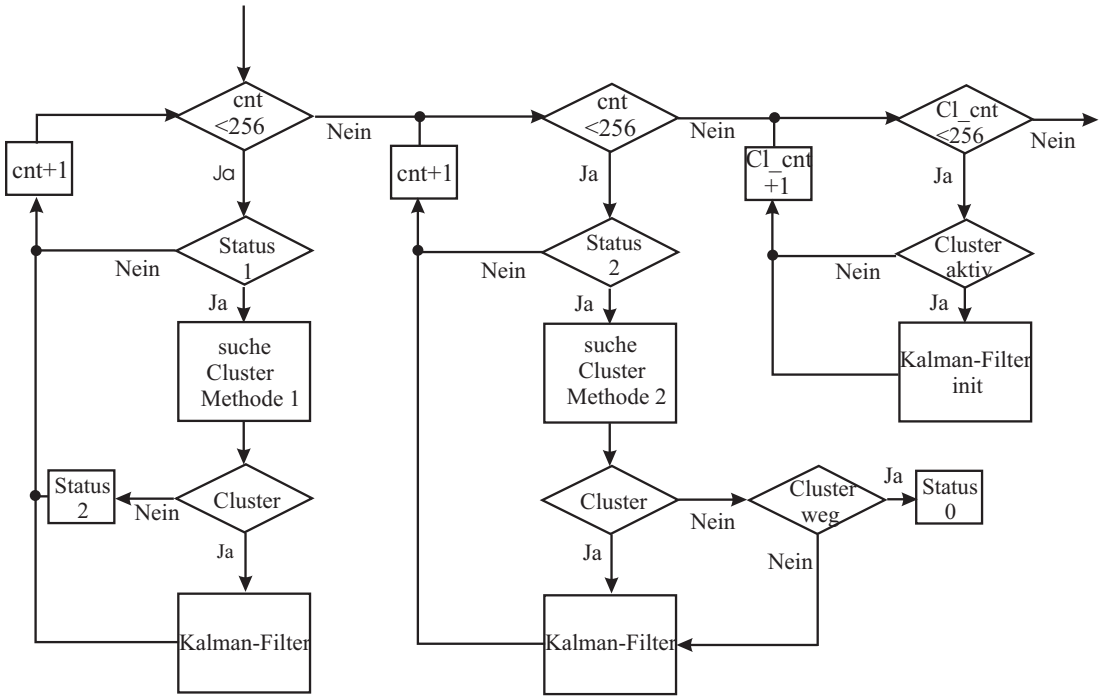

Abbildung B.5: Signallaufplan für die Weiterverfolgung von Objekten

Da die Möglichkeit besteht, dass Fahrzeuge im Rückraum neu erscheinen oder erkannte Fahrzeuge verschwinden, wird hier zwischen drei Fällen unterschieden

1. Zu einem KF-Objekt lässt sich kein Clusterobjekt finden. Das geschieht zum Beispiel durch Messaussetzer. Das Fahrzeug wird verdeckt oder das Fahrzeug hat bereits überholt.

2. Zu einem KF-Objekt lässt sich ein Clusterobjekt finden.

3. Zu einem Clusterobjekt lässt sich kein plausibles KF-Objekt finden. Das geschieht, wenn ein neues Fahrzeug oder ein anderes erhabenes Umgebungsobjekt, wie z.B. Brückenpfeiler, neu in der Rückraumszene erscheinen.

In Abbildung B.5 ist der Ablaufplan des Trackingalgorithmus dargestellt. Bei der Clusterung wird zunächst überprüft, ob es einen Cluster gibt, dessen Nummer mit dem Cluster aus dem vorhergehenden Bild übereinstimmt. Existiert ein solcher Cluster, wird das entsprechende KF-Objekt mit den Werten dieses Clusters gespeist. Gibt es keinen solchen Cluster, wird Anhand der prädizierten 3-d-Clusterwerte des KF-Objektes nach einem entsprechenden Clustermittelpunkt in der unmittelbaren Umgebung dieses Wertes gesucht. Die entsprechende Clusternummer wird beim Auffinden eines Clusters im KF-Objekt gespeichert. Ist auch diese Suche erfolglos, wird das KF-Objekt mit seinen eigenen prädizierten Werten gespeist. Taucht nach einer Anzahl von Bildern kein Cluster auf wird dann das KF-Objekt gelöscht. Für neu entstandene Cluster, für die kein KF-Objekt existiert, wird ein neues KF-Objekt generiert. Die Initialisierungswerte für die Schätzfehlerkovarianzmatrix P_k und die geschätzte Geschwindigkeit $x_k^{\#}(2)$ orientieren sich an dem KF-Objekt, das sich am dichtesten am eigenen Fahrzeug befindet. Somit entsprechen die Startwerte des KF-Objektes den Daten des aktuellen Verkehrsflusses.

B.4 Dimensionierung der Kalman-Filtermatrizen

Wie bereits in Kapitel 4.4.1 erläutert, hängt die Qualität der Kalman-Filter Ergebnisse von den a-priori Kenntnissen über das dynamische Verhalten des Systems und der Messfehler ab. Die Ermittlung der entsprechenden Daten wird im Folgenden beschrieben.

Zustandsübergangsfunktion zwischen z und z_k

Durch die Ebenengenerierung (siehe Kapitel B.1) ergibt sich eine Übertragungsfunktion des realen Entfernungswertes Z zum diskreten KF-Eingangswert - der

Entfernung z_k (in der allgemeinen Kalman-Gleichung als y_k bezeichnet). Aus der Übertragungsfunktion entsteht eine Stufenform, wie in Abbildung B.4.a dargestellt. Der resultierende Diskretisierungsfehler wird mithilfe einer Subpixelinterpolation verringert.

$$y_k = z_k = \frac{C}{int\left(\frac{z \cdot 8}{2^{int\left(lb\left(\frac{C}{z}\right)-3\right)}}\right) \cdot \frac{2^{int\left(lb\left(\frac{C}{z}\right)-3\right)}}{8}} \tag{B.6}$$

$$int(a) = max\left\{x \mid x \in Z\ ;\ x < a\right\} \tag{B.7}$$

$int(a)$ stellt dabei eine Rundungsoperation dar und entspricht einer Rundung des realen Zahlenwertes a auf die Vorkommastellen.

A_k, B_k , C_k und K_k

Die Zustandsübergangsfunktion A_k und die Eingangsmatrix B_k repräsentieren das Bewegungsmodell des eigenen Fahrzeuges und des erkannten KF-Objektes. Beide Matrizen werden aus einem kontinuierlichen Differentialgleichungsmodell abgeleitet und in ein zeitdiskretes Modell umgewandelt.

$$A_k = \begin{pmatrix} 1 & \Delta T & \frac{1}{2}\Delta T^2 \\ 0 & 1 & \Delta T \\ 0 & 0 & 1 \end{pmatrix} \tag{B.8}$$

$$B_k = \begin{pmatrix} -\Delta T & -\Delta T - \frac{1}{2}\Delta T^2 & -\Delta T - \frac{1}{2}\Delta T^2 - \frac{1}{6}\Delta T^3 \\ 0 & -\Delta T & -\Delta T - \frac{1}{2}\Delta T^2 \\ 0 & 0 & -\Delta T \end{pmatrix} \tag{B.9}$$

Die Herleitung der Gleichungen B.8 und B.9 findet sich in [61] und soll hier nicht weiter besprochen werden. ΔT ist der äquidistante Zeitschritt für die diskretisierten Matrizen und wird mit $\Delta T = 1$ angenommen.

Mit der Beobachtungsmatrix C_k wird der prädizierte Entfernungswert aus dem

Kalman-Filter vom gemessenen Entfernungswert subtrahiert. Die Beobachtungsmatrix lautet daher:

$$C_k = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \tag{B.10}$$

Die Korrekturmatrix K_k wird im System berechnet und ändert sich mit jedem Zeitschritt, während die bereits vorgestellten Matrizen konstant bleiben.

Varianzen R_k und Q_k

Die Varianzen des Systems bestimmen die Dynamik des Kalman-Filters. Eine hohe Dynamik bedeutet eine schnelle Nachführung der KF-Größen zur Eingangsgröße y_k (Hochpassverhalten). Eine niedrige Dynamik ergibt eine langsame Nachführung und damit auch ein langsames Einschwingen auf die Eingangsgröße (Tiefpassverhalten). Ziel ist es, die Dynamik des Kalman-Filters so zu wählen, dass die Eingangsgröße geglättet wird, damit die Abweichungen zwischen realen und ermittelten Größen, unter Berücksichtigung aller Störeinflüsse, minimal werden. Gleichzeitig soll das Kalman-Filter schnell auf die Bewegungsgrößen des KF-Objektes einschwingen und bei Geschwindigkeitsänderungen die eigenen Zustandsgrößen entsprechend schnell nachführen.

$$\sigma_Z = \left| -\frac{c \cdot B}{\Delta u^2} \right| \cdot \sigma_{\Delta u} \tag{B.11}$$

Die Messfehlervarianz R_k besteht aus zwei Komponenten (Gleichung B.12). Die erste Komponente ist der Fehler des Kamerasystems auf Basis des Fehlerfortpflanzungsgesetzes nach Gleichung 2.10 (siehe Gleichung B.11). Der zweite Term ist ein zufälliger Fehler, der durch Messung bestimmt werden muss.

$$R = \sigma_z^2 + \sigma_{zuf}^2 \tag{B.12}$$

Da die Messfehlervarianz nicht durch systeminterne Einstellungen verändert werden kann, sondern externen Faktoren unterliegt, bleibt noch die Kovarianzmatrix Q_k (Gleichung B.13) zur Anpassung des Systems an die Fahrzeugdynamik des detektierten KF-Objektes.

$$Q_k = \begin{pmatrix} \sigma_Q^2 & 0 & 0 \\ 0 & \sigma_Q^2 & 0 \\ 0 & 0 & \sigma_Q^2 \end{pmatrix} \tag{B.13}$$

Die Kovarianz σ_Q^2 wird nur in die Hauptdiagonale der Kovarianzmatrix eingetragen.

Eine hohe Messfehlervarianz verursacht ein langsames Einschwingen des Kalman-Filters, wogegen eine hohe Kovarianz ein schnelles Einschwingen des Kalman-Filters ermöglicht. Ziel ist es, nun einen Kompromiss zwischen guter Glättung der Eingangsgrößen und schnellem Einschwingverhalten des Systems zu finden. Zu diesem Zweck wird eine adaptive Kovarianz verwendet. Nachdem ein neues KF-Objekt erkannt wurde, wird zum schnellen Einschwingen eine hohe Kovarianz verwendet.

Entfernungsbereich in m	maximaler Prädiktionsfehler (Schwellwert)in m
125-150	1,5
100-125	1
75-100	0,5
65-75	1
20-65	0,5
<20	0,3

Tabelle B.2: Schwellwertbereiche für Kovarianzen

Nach 10 Bildern wird die Kovarianz verringert. Im eingeschwungenen Zustand überschreitet der Fehler zwischen y_k und $x_k^*(1)$ die Werte aus Abbildung 4.2 in Kapitel 4.4 nicht. Tritt dieser Fall trotzdem auf, führt das Kalman-Filter seine Bewegungsgrößen nicht schnell genug nach. Das entspricht einer Beschleunigung oder Verzögerung des KF-Objektes. Nimmt der Prädiktionsfehler also einen Wert an, der über einem bestimmen Schwellwert liegt, so muss die Kovarianz entsprechend verändert werden. Wie in Abbildung B.4 in Kapitel B.2 zu erkennen, verringert sich der Schwellwert mit der Distanz zwischen Kamerasystem und

KF-Objekt. Daher werden verschiedene Schwellwerte für verschiedene Distanzen benötigt. In Tabelle B.2 sind die entsprechenden Werte dargestellt.

σ^2_{Q0}	0,1
σ^2_{Q1}	0,0000001
σ^2_{Q2}	0,001

Tabelle B.3: Kovarianzen σ^2_Q

Wie zu erkennen ist, werden drei Kovarianzen benötigt. In Tabelle B.3 sind die hier verwendeten und empirisch ermittelten Kovarianzen dargestellt. σ^2_{Q0} ist die Kovarianz zum Einschwingen der KF-Objektes, σ^2_{Q1} für ein eingeschwungenes KF-Objekt und σ^2_{Q2} wird verwendet, wenn der Prädiktionsfehler den Schwellwert aus Tabelle B.2 überschreitet.

Anhang C

Daten für die Partitionierung

C.1 Nebenbedingungen

C.1.1 Echtzeitbedingung

Das für die Aufgabe geforderte Echtzeitsystem muss in der Ausführungszeit ein Fahrzeug erkennen und dessen Geschwindigkeit und Spur zuordnen können. Da das System für die Autobahnfahrt ausgelegt sein soll, können Geschwindigkeitsdifferenzen von bis zu $250km/h$ auftreten, wenn das eigene Fahrzeug steht und das sich nähernde Fahrzeug die maximal mögliche Höchstgeschwindigkeit aufweist. Aus Tabelle C.1 lässt sich nach Gleichung C.1 ermitteln, um wie viel sich ein Fahrzeug in einer gewissen Zeitspanne und mit einer gewissen Relativgeschwindigkeit dem eigenen Fahrzeug nähert.

$$v = s \cdot t \tag{C.1}$$

Wenn also eine menschliche Reaktionszeit von $800 - 1000ms$ angenommen wird, beträgt der Weg des Fahrzeugs bei einer Differenzgeschwindigkeit von $250km/h$ $69m$. Bei einer harten Echtzeitbedingung (Relativgeschwindigkeit $250km/h$) sollte das Bildverarbeitungssystem also nicht länger als $1200ms$ benötigen, um eine Warnung auszugeben, da sonst das in einer Entfernung von $150m$ als gefährlich

Zeit [ms]	bei 50km/h	bei 100km/h	bei 150km/h	bei 250km/h
20	0,28	0,55	0,83	1,38
40	0,55	1,11	1,65	2,75
60	0,83	1,66	2,49	4,15
80	1,11	2,22	3,33	5,55
100	1,38	2,76	4,14	6,90
200	2,76	5,52	8,28	13,80
400	5,52	11,04	16,56	27,60
800	11,04	22,08	33,12	55,20
1000	13,8	27,60	41,40	69,00

Tabelle C.1: Zurückgelegter Weg in Abhängigkeit von Zeit und Geschwindigkeit

erkannte Fahrzeug schon im Gefahrenbereich des eigenen Fahrzeugs sein könnte. Diese $1200ms$ entsprechen wiederum 30 Messwerten bei einer Kamerabildaufnahmezeit von $40ms$. Da das Kalman-Filter etwa 25 Messwerte benötigt, um verwertbare Ergebnisse zu liefern, bleiben also maximal 5 Bilder oder $200ms$ für die Bearbeitung eines Einzelbildes, wobei durch den kontinuierlichen Datenfluss aus den Kameras die Bildaufnahmezeit von $40ms$ als maximale Periodendauer T_S einer Pipelinestufe anzunehmen ist.

Verschiedene Teile des Algorithmus können pixelweise oder auch zeilenweise abgearbeitet werden und benötigen nicht die Daten eines ganzen Bildes. Für diese Funktionen ergeben sich geringere Latenzzeiten (Siehe Tabelle C.2).

Nebenbegingung	Wert [ms]
$T_{S_{Pixel}}$ je Pipelinestufe	0,00005
$T_{S_{Zeile}}$ je Pipelinestufe	0,051
$T_{S_{Bild}}$ je Pipelinestufe	40
$T_{S_{ges}}$ Latenzzeit - Gesamtsystem	200

Tabelle C.2: Nebenbedingungen

C.1.2 Zeit und Platzbedarf der Logikelemente

Der Zeit- und Platzbedarf von mathematischen Operationen in Logikblöcken wurde bereits in verschiedenen Arbeiten untersucht [49], [39]. Da die Architektur der FPGAs unterschiedlich ist, müssen die entsprechenden Werte neu ermittelt werden. Es sind dabei Fließkomma-Arithmetiken (FP) und Festkomma-Arithmetiken (Int) evaluiert worden.

Funktion	Latenzzeit [Takte]	Max. Freq. [MHz]	Logikelemente[LE]	NIOS-II [Takte]	mit FP [Takte]
FP Add.	8	88,45	1.163	387	14
FP Sub.	8	88,45	1.163	387	14
FP Multipl.	5	91,55	1.703	260	12
FP Division	33	130	3.041	450	32
Int Addition	ansynch.	91,44	32	3	3
Int Substrakt.	ansynch.	91,44	32	4	4
Int Multipl.	ansynch.	47,14	1.210	4	4
Int Division	ansynch.	5,35	1.125	8	8

Tabelle C.3: Zeit- und Flächenbedarf der Funktionsbasisblöcke

Die in Tabelle C.3 vorgestellten Funktionsblöcke wurden für einen Stratix Chip erstellt, simuliert und implementiert. Die verwendete Busbreite beträgt 32 Bit, da sich die genutzten Datenwortbreiten des Algorithmus ebenfalls auf 32 Bit belaufen. Eventuell kleinere Busbreiten können hier nicht berücksichtigt werden, weil hierfür eine wesentlich exaktere Analyse des Algorithmus nötig wäre, diese aber nicht mehr nur auf Basis der C/C++ Funktionen durchgeführt werden kann. Der Bedarf an Logikelementen eines Funktionsblockes lässt sich durch spezielle DSP Elemente verringern, wobei darauf zu achten ist, dass diese Elemente nur in sehr begrenzter Stückzahl zur Verfügung stehen.

Die Werte für den NIOS-II sind durch entsprechende Testroutinen ermittelt worden. Es ist zu erkennen, dass insbesondere die Fließkommaoperationen sehr viel Zeit benötigen. Durch den Einsatz eines veränderten Befehlssatzes, der speziell

für Fließkommaoperationen entworfen wurde, kann die Verarbeitungszeit dieser Funktionen enorm verringert werden. Der Platzbedarf eines NIOS-II Prozessors im FPGA vergrößert sich jedoch dabei von 3.000 LE (Logikelemente) auf 5.500 LE.

C.2 Schnittstellenspezifikation

Globale Schnittstelle

Der Avalon Bus (siehe Kapitel 3.2.2) ist die globale Schnittstelle zwischen den Prozessoren und den Logikelementen sowie zwischen den einzelnen Prozessoren. Zur Kommunikation wird eine Adresse vom Prozessor an den DPRAM gesendet. Dort wird die Adresse verarbeitet und die Daten werden an den Prozessor übergeben. Die Latenzzeit des Busses beträgt für Lese und Schreibzugriffe einen Takt. Durch die Adressauswertung innerhalb der Logik werden im vorgegeben Fall aber 6 Takte für die Schreib und Lesezugriffe benötigt.

Lokale Schnittstelle

Durch die direkte Kopplung der Logikelemente an einen Speicherbaustein, führt die lokale Schnittstelle einen einfachen Schreib-/Lesezugriff auf den Speicher mit einer Latenzzeit von einem Taktzyklus aus.

Um Datentransfers zwischen einem Master und einem Slave durchführen zu können, brauchen nicht alle in Tabelle C.4 beschriebenen Signale verwendet zu werden.

Signalname	Breite	Richtung	Beschreibung
clk	1	in	Globaler Takt. Alle Bustransfers sind synchron zum Takt
reset	1	in	Globales Reset Signal
chipselect	1	in	Der Slave ignoriert alle Eingänge, bis das Signal aktiv ist
address	1-32	in	Adressdaten vom Avalon Bus
begintransfer	1	in	Wird während jedes ersten Bustransferzyklus gesetzt
byteenable	0,2,4	in	Signalisiert, welche Bytes eines Datenwortes aktiviert sind
read	1	in	Request Signal, wenn Daten gelesen werden sollen
readdata	1-32	out	Datenleitung von der Peripherie zum Avalon Bus
write	1	in	Request Signal, wenn Daten gelesen werden sollen
writedata	1-32	in	Datenleitungen vom Avalon Bus zur Peripherie

Tabelle C.4: Avalon Bussignale

C.3 Funktionseigenschaften im Datenfluss

In den folgenden Tabellen sind die Eigenschaften der analysierten Funktionen dargestellt. Es werden Angaben zu den Lines of Code (LOC), zum Halstead Volumen (HV), zur Schwierigkeit nach Halstead (HD), zur Komplexität nach Mc Cabe (CC), zur maximalen Auftrittshäufigkeit der Funktion je Bild (AH), zur Verarbeitungszeit der Funktion auf dem PC und auf dem NIOS II Prozessor gemacht.

Aus Gründen der Übersichtlichkeit wurde auf die Berechnung der Portierungsko-

sten, des Flächenbedarfs und des Speicherbedarfs verzichtet. Die entsprechenden Werte lassen sich aus den angegebenen Beträgen nach den in Kapitel 3.4.7 dargestellten Formeln berechnen.

Nr.	Funktionsname	LOC	Halstead HV	Halstead HD	Mc Cabe CC	Häufigkeit je Bild AH	Zeit PC [μs]	Zeit Proz. [μs]
0	History_init	33	182	0,73	2	1	0,1	1
1	Cluster_reset	11	27	1,5	2	256	7	4,2
2	Phi_Calc	11	33	1	2	1	0,1	1,1
3	RPhi_init	8	5	1	2	1	0,1	1,2
4	RoadMap_init	11	11	1,25	2	1	0,1	1,1
5	Kalman_init	10	19	1,25	2	1	0,1	1,0
6	Store_init	11	30	0,75	2	1	0,1	1,0
7	createLevelLeft	15	75	1,5	2	1500	5,2	2,9
8	createLevelRight	15	75	1,5	2	1500	5,1	5,7
9	calcLinesLeft	7	42	1,5	2	2000	0,2	2,3
10	calcLinesRight	5	28	1,5	2	2000	0,2	2,3
11	CalcMaxDisp	9	18	0,5	3	116000	0,2	2,4
12	calcab	8	47	1,14	2	116000	0,2	2,2
13	CalcMeana_aa	17	66	1,5	2	116000	1,2	176
14	vara	10	16	1,5	3	116000	0,2	2,3
15	MWF_KKF	53	158	1,5	4	252368	1,0	107
16	ifmaxk	7	16	0,5	3	252368	0,2	2,5
17	subpix	17	111	1,5	3	16320	0,2	37
18	LineStore	7	45	0,78	2	116000	0,2	2,2
19	CalcUpLev	12	63	1,25	3	192000	0,1	7,2
20	GenDataSet	18	148	1,5	2	3786	0,2	8
21	History	18	122	1,5	3	3786	1,3	3,2
22	CreateCanDataforDSP	19	57	1,5	2	1	0,1	1
23	CAN_COM	17	62	1,5	3	1	0,1	1,1
24	Freq2StatusErhaben	34	51	1,12	4	37	7,68	860

Tabelle C.5: Eigenschaften der Funktionsblöcke - Teil 1

Nr.	Funktionsname	LOC	Halstead HV	Halstead HD	Mc Cabe CC	Häufigkeit je Bild AH	Zeit PC [μs]	Zeit Proz. [μs]
25	TakeFrom_imax	20	51	1,3	4	37	2,0	22,9
26	Clear_histtemp	5	6	1,5	2	128	1,4	16,3
27	Clear_histtemp	5	6	1,5	2	1	1,4	16,3
28	Clear_histtemp	5	6	1,5	2	1	1,4	16,3
29	maxi	6	6	0,5	3	5	1,4	19,9
30	maxi	6	6	0,5	3	5	1,4	19,9
31	maxi	6	6	0,5	3	5	1,4	19,9
32	maxi	6	6	0,5	3	5	1,4	19,9
33	maxi	6	6	0,5	3	5	1,4	19,9
34	maxi	6	6	0,5	3	5	1,4	19,9
35	maxi	6	6	0,5	3	5	1,4	19,9
36	TakeFrom_imax	20	51	1,3	4	5	2,0	22,9
37	dilev	20	105	1,0	5	36	1,4	15,9
38	Clear_histtemp	5	6	1,5	2	32	1,4	16,3
39	newhiststat	4	8	0,75	2	12	1,4	17,2
40	Clear_histtemp	5	6	1,5	2	110	1,4	16,3
41	Clear_histtemp	5	6	1,5	2	4	1,4	16,3
42	Clear_histtemp	5	6	1,5	2	4	1,4	16,3
43	maxi	6	6	0,5	3	68	1,4	19,9
44	maxi	6	6	0,5	3	68	1,4	19,9
45	maxi	6	6	0,5	3	68	1,4	19,9
46	maxi	6	6	0,5	3	68	1,4	19,9
47	maxi	6	6	0,5	3	68	1,4	19,9
48	TakeFrom_imax	20	51	1,3	4	68	2,0	22,9
49	dilev	20	105	1,0	5	183	1,4	15,9
50	Clear_histtemp	5	6	1,5	2	32	1,4	16,3
51	newhiststat	4	8	0,75	2	51	1,4	17,2
52	Clear_histtemp	5	6	1,5	2	1626	1,4	16,3
53	Clear_histtemp	5	6	1,5	2	35	1,4	16,3
54	Clear_histtemp	5	6	1,5	2	35	1,4	16,3
55	SwapHist	5	14	0,66	2	1	1,4	16,3

Tabelle C.6: Eigenschaften der Funktionsblöcke - Teil 2

Nr.	Funktionsname	LOC	Halstead HV	Halstead HD	Mc Cabe CC	Häufigkeit je Bild AH	Zeit PC [μs]	Zeit Proz. [μs]
56	CLUSTER_reset	11	27	1,5	2	256	7,0	42,5
57	strtoldest	15	28	0,6	5	19	1,7	22,9
58	OldestSearch	106	493	0,62	14	4	2,5	27,7
59	CLUSTER_new	28	36	1,5	4	4	2,2	25,2
60	CLUSTER_new	28	36	1,5	4	1	2,2	25,2
61	ClusterStart	16	121	0,65	3	2	1,1	11,8
62	CLUSTER_new	28	36	1,5	4	2	2,3	25,2
63	Cluster1	88	505	0,68	10	17	2,2	24
64	Left_over	36	188	0,65	4	23	1,4	15,2
65	WriteCluster	30	118	0,68	4	23	1,7	15,5
66	OldestSearch	106	493	0,62	14	1	2,5	27,7
67	CLUSTER_new	28	36	1,5	4	1	2,2	25,2
68	CLUSTER_new	28	36	1,5	4	1	2,2	25,2
69	ClusterStart	16	121	0,65	3	2	1,1	11,8
70	CLUSTER_new	28	36	1,5	4	2	2,2	25,2
71	Cluster1	88	505	0,68	10	17	2,5	24
72	Left_over	36	188	0,65	4	23	1,4	15,2
73	WriteCluster	30	118	0,68	4	23	1,7	15,5
74	Cluster_remove_old	11	7	1,0	3	1	2,2	26,3
75	Hough_init	7	4	0,5	2	1	0,1	42,1
76	RPhi_reinit	12	4	0,5	4	1	2,5	35,6
77	Genangle	39	127	1,5	9	480	1,7	52,6
78	Check_Direction	18	144	1,18	3	7	12	130
79	puttogether	35	194	1,5	3	4	1,4	16,3
80	ScanOldHough	45	134	0,7	7	4	2,0	22,6
81	Cluster_flat	104	506	1,5	10	7	3,6	39,2
82	Cluster_flat	104	506	1,5	10	7	3,6	39,2
83	rphicount	15	7	1	6	7	1,1	15,6

Tabelle C.7: Eigenschaften der Funktionsblöcke - Teil 3

Nr.	Funktionsname	LOC	Halstead HV	Halstead HD	Mc Cabe CC	Häufigkeit je Bild AH	Zeit PC [μs]	Zeit Proz. [μs]
84	DreiD	24	188	1,5	3	11	1,1	12,4
85	Cluster_searcha	17	53	0,75	4	9	1,1	12,7
86	kalman	128	2653	1,5	2	5	3,4	479
87	Cluster_Filter	35	140	1,5	4	5	1,1	11,8
88	M_Kalm1	14	24	0,75	4	5	1,1	12,5
89	Cluster_searchb	51	200	1,5	11	10	4,2	49,6
90	kalman	128	2653	1,5	2	5	3,4	47,9
91	Cluster_Filter	35	140	1,5	4	5	1,1	11,8
92	M_Kalm2	17	50	1,16	4	5	1,1	12,1
93	Hist_search	70	251	1,5	10	3	1,4	16
94	kalman	128	2653	1,5	2	6	3,4	479
95	Start_new_Kalman	100	279	0,83	7	1	4,7	50,5
96	RoadMap_reinit	7	26	1,5	2	1	2,5	595
97	calcdz	4	14	0,66	2	1	1,1	12,8
98	Rotate_Map	17	76	1,5	4	280	2,2	82
99	LaneSort	20	79	0,68	3	45	1,1	12,3
100	NewLane	16	37	0,625	3	3	1,1	1,2
101	Disapplane	26	51	0,75	6	15	1,1	12,7
102	Reorganise	17	87	1,07	3	1	2,2	19,7
103	Delete_Cluster	34	84	1,5	7	1	2,2	101
104	WerteuebertragungLdrei	25	84	1,1	1	5	6,1	63
105	Werteuebertragungflach	25	51	1,1	1	5	6,1	63

Tabelle C.8: Eigenschaften der Funktionsblöcke - Teil 4

C.4 Erfolgsfaktoren für COCOMO II

In der folgenden Tabelle sind die 17 Erfolgsmultiplikatoren und die vier Wichtungsfaktoren für den COCOMO II Ansatz dargestellt. Der COCOMO II Ansatz wird hier zur Schätzung der Entwicklungszeit von VHDL-Code für eine als C/C++ Code existierende Funktion angewendet.

Erfolgsmultiplikatoren	Wert
EM0 Erfahrung	1.1
EM1 Begabung	1.2
EM2 Begabung als Analyst	1.0
EM3 Personalkontinuität	1.0
EM4 Plattformerfahrung	1.0
EM5 Erfahrung mit der Sprache	1.25
EM6 Benötigte Zuverlässigkeit	1.25
EM7 Größe der Datenbasis	1.0
EM8 Komplexität	HD
EM9 Wiederverwendbarkeit	1.0
EM10 Dokumentation	1.0
EM11 Verarbeitungszeit	1.15
EM12 Hauptspeichernutzung	1.0
EM13 Flüchtigkeit der Plattform	1.0
EM14 Benutzte Softwaretools	1.0
EM15 benötigter Entwicklerzeitplan	1.0
EM16 Verteilte Entwicklung	1.0
Wichtungsfaktoren	
W0 Flexibilität	2.0
W1 Risikomanagment	2.0
W2 Teamzusammenhalt	0.0
W3 Prozesserfahrung	1.0

Tabelle C.9: Wichtungsfaktoren und Erfolgsmultiplikatoren für COCOMO II

Veröffentlichungen

Martin Streitenberger, **Jens Kaszubiak**, Wolfgang Mathies: *A novel approach to high speed digital pulse-formers based on ring oszillators for pwm and class-D Applications*, 112. AES Convention, 2002

Jens Kaszubiak, Michael Tornow, Robert W. Kuhn, Bernd Michaelis: *Real-time, 3-D-multi object position estimation and tracking*, ICPR'04 Cambridge/UK, Vol. 1, S. 785-788, 2004

Jens Kaszubiak, Michael Tornow, Robert W. Kuhn, Bernd Michaelis, Helmut Bresch: *Hardware-software co-design for an optical, real-time object detection and tracking system*, GSPx'04 Santa Clara/CA, 2004

Michael Tornow, **Jens Kaszubiak**, Robert W. Kuhn, Bernd Michaelis, Gerald Krell: *Stereophotogrammetric Real-Time 3-D-Machine Vision*, PRIA-7-2004, St. Petersburg/RU, Vol. 3, S. 940-943, 2004

Michael Tornow, Bernd Michaelis, Robert W. Kuhn, **Jens Kaszubiak**, Carsten Knöppel: *Echtzeit Objekterfassung und Positionsvermessung*, VDI Berichte Nr. 1876, S.137-158, 2005

Robert W. Kuhn, **Jens Kaszubiak**, Michael Tornow, Bernd Michaelis: *Echtzeit-fähiger Multipositionssensor*, Oldenburger 3D-Tage, S.22-29, 2005

Robert W. Kuhn, **Jens Kaszubiak**, Michael Tornow, Bernd Michaelis: *Realtime Estimation of Depth maps for Machine Vision*, EOS Conference On Industrial Imaging and Machine Vision 2005, München/D, S.98-100, 2005

Jens Kaszubiak, Michael Tornow, Robert W. Kuhn, Bernd Michaelis, Carsten Knöppel: *Real-Time Vehicle and Lane Detection with Embedded Hardware*, IEEE Intelligent Vehicle Symposium 2005 Las Vegas/USA, S.618-623, 2005

Michael Tornow, **Jens Kaszubiak**, Robert W. Kuhn, Bernd Michaelis, Thomas Schindler: *Hardware Approach for Realtime machine stereo vision*, WMSCI2005, Orlando/FL, Vol. 5, S. 111-116, 2005

Michael Tornow, **Jens Kaszubiak**, Robert W. Kuhn, Bernd Michaelis, Thomas Schindler: *Hardware Approach for Realtime machine stereo vision*, Journal of Systemics Cybernetics and Informatics, Vol. 4, S. 24-34, 2006

Jens Kaszubiak, Michael Tornow, Bernd Michaelis, Thomas Sczepanski: *Co-Design for Speed and Space Optimization of Chips for Image Processing Application*, IASTED MSO 2006, Gaborone, Botswana, S. 19-24, 2006

Michael Tornow, **Jens Kaszubiak**, Robert W. Kuhn, Bernd Michaelis, Gerald Krell: *Stereophotogrammetric Real-Time 3D Machine Vision*, Pattern Recognition and Image Analysis, Vol. 16, S. 100-103, 2006

Jens Kaszubiak, Robert W. Kuhn, Michael Tornow, Bernd Michaelis: *Real-Time 3-D Environment Capture Systems. Scene Reconstruction Pose Estimation and Tracking*, ARS Publishing, S.357-381, 2007

Lebenslauf

Name:	Jens Kaszubiak
Geburtsdatum:	10. Oktober 1977
Geschlecht:	männlich
Staatsangehörigkeit:	deutsch
Familienstand:	ledig

Berufstätigkeit:	2003 - dato	Wissenschaftlicher Mitarbeiter am IESK/FEIT der O-v-G Universität
	2001	Ingenieurpraktikum bei der Photonfocus AG/Schweiz
	1997	Fernmeldemonteur TCT Thale
Militärdienst:	1996 - 1997	Grundwehrdienst bei der Bundeswehr
Studium:	1997 - 2002	Studium der Elektrotechnik an der O-v-G Universität Magdeburg, Fachrichtung: Informationselektronik Abschluss: Diplomingenieur
Schulbildung:	1991 - 1996	Gymnasium „Am Thie“ in Blankenburg/H
	1984 - 1991	Polytechnische Oberschule Timmenrode

Magdeburg, den 30. Oktober 2007