Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Steffen Rempp

Multithreading für eine Bildverarbeitungs-Pipeline unter Linux auf einem ARM-basierten MPSoC

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Steffen Rempp

Multithreading für eine Bildverarbeitungs-Pipeline unter Linux auf einem ARM-basierten MPSoC

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof.Dr.-Ing. Bernd Schwarz

Zweitgutachter: Prof.Dr. Gunter Klemke

Eingereicht am: 17. Mai 2013

Steffen Rempp

Thema der Arbeit
Multithreading für eine Bildverarbeitungs-Pipeline unter Linux auf einem ARM-basierten MPSoC

Stichworte
MPSoC, ARM, OMAP, Zynq, Linux, SMP, Scheduling, Bildverarbeitung, Multithreading, parallele Software, Pipeline, Fahrerassistenzsystem, Hough Transformation, Kamera Kalibrierung, Fahrspurführungsalgorithmen

Kurzzusammenfassung
Diese Arbeit beschreibt den Entwurf und die Implementierung einer Fahrspurführung unter Linux auf einem ARM Cortex A9 DualCore Prozessor, welcher in eine MPSoC Plattform integriert ist. Der Videostream einer IP-Kamera wird von Co-Prozessoren verarbeitet. Durch Pixeldatenreduzierung und Bildverarbeitungsoperationen wird die Lage der Fahrspur erkannt und der Lenkwinkel berechnet. Die Verarbeitungskette wird parallelisiert und nach verschiedenen Multithreading Konzepten auf zwei CPU Kernen ausgeführt.

Title of the paper
Multithreading of a computervision pipeline on an ARM based MPSoC in Linux

Keywords
MPSoC, ARM, OMAP, Zynq, Linux, SMP, scheduling, computer vision, multithreading, parallel software, pipeline, driving assitance, Hough Transformation, camera calibration, lane following algorithms

Abstract
This work presents the design and implementation of a Linux based lane following application on an MPSoC with an ARM Cortex A9 DualCore. Handling the videostream by an IP camera is supported by Co-processors. The location of the lane will be detected by reducing the pixel data and a set of coputer vision operations to compute the steering angle. The pipeline will be parallelized on two CPUs following different multithreading patterns.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Die Taktfrequenz und die Instruction-Level Parallelität von Mikroprozessoren verdoppelte sich in den letzten Jahrzehnten alle 18-24 Monate und erreicht gegenwärtig ihre Grenzen [50]. Gleichzeitig erfordern Embedded Anwendungen wie z.B. Multimedia- oder biometrische Signalverarbeitungsanwendungen eine immer höhere Rechenkapazität von Single-Chip Systemen, um Echtzeitanforderungen bei geringem Stromverbrauch und geringen Kosten zu erfüllen. Gerade in Signalverarbeitungsapplikationen, in denen mehrere Datenströme parallel empfangen und verarbeitet werden, steigt der Bedarf an flexiblen Multi-Standard Lösungen [17]. Die technische Herausforderung [37] in der Dimensionierung der Größe des Systems liegt darin, alle Anwendungsanforderungen in allen Betriebszuständen zu erfüllen, ohne dabei Ressourcen zu verschwenden, um somit eine Plattform zu schaffen, die exakt für ihre Aufgaben zugeschnitten ist.

Multiprocessor Systems-on-Chip (MPSoC) dienen als Lösungsansatz für diese wachsenden Leistungsanforderungen, was sich auch in der rasanten Entwicklung von MPSoC Produkten im globalen Embedded Markt widerspiegelt [22], wie z.B. Entwicklungswerkzeuge, MPSoC-Plattformen oder Architekturen. Sie vereinen mehrere programmierbare oder funktionsspezifische Prozessorsysteme auf einem Chip und bieten somit die Plattform für die parallele Ausführung mehrerer Funktionen. Der Einsatz von FPGA-basierten Systemen für die Realisierung einer MPSoC Plattform durch die Integration von Beschleunigermodulen und Softcore IP Prozessoren innerhalb einer programmierbaren Logik bietet Entwicklern die meisten Freiheitsgrade in der parallelen Verarbeitung von Funktionen durch verschiedene Module auf dem FPGA. Hinzu kommt die Rekonfigurierbarkeit und die flexible Erweiterung um neue Modulbausteine. Die zeitgleiche Entwicklung von Hardware und Software Komponenten innerhalb eines Systems wird hierbei unter dem Begriff Hardware/Software Co-Design zusammengefasst. Dahinter verbirgt sich der gemeinsame, zeitgleiche Entwurf von Hardware und Software-Anteilen, wobei verschiedene Varianten und Optionen der Realisierung von Funktionen in Hardware oder Software

erkannt werden und eine differenzierte Entscheidung zur Zuteilung getroffen wird.

Eine Alternative zu MPSoC auf FPGA-Systemen sind feststehende MPSoC Plattformen mit einer fixen Anzahl an verschiedenen Prozessor-Subsystemen, die zusammen auf einem Chip aufgabenspezifisch und gemäß einer definierten Architektur integriert werden. Dabei reicht das Spektrum an programmierbaren Prozessorsystemen innerhalb des MPSoC von 8-Bit Mikrocontrollern bis zu 32-Bit (MultiCore) Applikationsprozessoren. Auch in der Art der Programmierung reicht das Spektrum von einem 8-Bit Befehlssatz bis hin zu 32-Bit Runtime Libraries von Hochsprachen wie z.B. C/C++ und Unterstützung durch High-Level (Echtzeit-)Betriebssysteme. Je nach Anforderung der zu realisierenden Applikation beginnt entweder die Suche nach einem geeigneten feststehenden Multiprozessorsystem, mit welchem alle funktionalen und systematischen Anforderungen erfüllt werden, oder es werden Funktionsbausteine in Hardware und Software definiert, die ebenso alle Anforderungen erfüllen und als applikationsspezifisches MPSoC auf einem FPGA realisiert werden.

Eine Verschmelzung beider Alternativen ist die Integration eines Applikationsprozessors mit einem FPGA-System. Mit der *Zynq 7000 Extensible Processing Platform* (EEP) von Xilinx [45] sowie dem *Cyclone-* und *Arria-* FPGA-basierten SoC von Altera [1] stehen nun Produkte diesen Designs nahezu zeitgleich auf dem Markt zur Verfügung. Sowohl Xilinx als auch Altera integrieren in ihren FPGA-Systemen einen feststehenden ARM Cortex-A9 DualCore Prozessor. Dieses Design vereint sowohl die Vorzüge eines hochfrequent getakteten DualCore Applikationsprozessors, auf welchem parallele Software auf zwei CPUs ausgeführt werden kann, als auch die Freiheitsgrade im HW/SW Co-Design durch die rekonfigurierbare Programmierlogik auf dem FPGA. Die Entscheidung im HW/SW Co-Design bei der Zuteilung von Funktionen entweder in Software auf dem Prozessor oder als Modulbaustein auf dem FPGA wird maßgeblich von der Leistungsfähigkeit des Prozessors beeinflusst, bzw. von der Anzahl der Funktionen, die der Prozessor unter Berücksichtigung aller anwendungsspezifischen Anforderungen innerhalb eines definierten Zyklus auszuführen vermag.

Im Forschungsprojekt FAUST (*Fahrerassistenz- und Autonome Systeme)* des Departements Informatik an der Hochschule für Angewandte Wissenschaften Hamburg werden prototypische Fahrerassistenzsysteme entwickelt, um die Entwicklungsprozesse im Bereich Embedded Systems Design zu durchdringen. Dabei liegen die Forschungsschwerpunkte

auf dem Einsatz von Sensorik, Telemetrie und Bildverarbeitung in Echtzeitsystemen auf unterschiedlichen Software- und Hardwarearchitekturen und Plattformen. Im Bereich Autonomes Fahren werden Modellfahrzeuge im Maßstab 1:10 entwickelt, auf welchen die Hardware- und Softwareplattformen integriert und durch Fahrerassistenzapplikationen gesteuert werden. Das Ziel der Projektgruppe HPEC (*High Performance Embedded Computing*) ist die Entwicklung von FPGA-basierten Fahrerassistenzsystemen. Das SAV (*System on Chip Autonomous Vehicle*) Projekt vereinigt hierbei die Arbeitsergebnisse vorangegangener Arbeiten [21], [36] für eine FPGA-basierte Fahrspurführung auf verschiedenen Hardware Plattformen mit den Schwerpunkten der Integration verschiedener Fahrerassistenzapplikationen in eine einheitliche Systemarchitektur und der Inbetriebnahme des SAV. Das Ziel des SAV Projektes ist die Portierung der Systemarchitektur auf die Zynq 7000 EEP von XILINX durch HW/SW Co-Design auf dem FPGA und dem ARM Cortex A9 DualCore Prozessor.

Ziele der vorliegenden Arbeit

Aus den genannten Teilkomponenten lassen sich die Ziele der vorliegenden Arbeit detaillierter ableiten:

- Das Hauptziel dieser Arbeit ist der Entwurf und die Implementierung einer rein Software-basierten Fahrspurführungsapplikation auf dem ARM Cortex A9 DualCore Prozessor.
- Die Basis hierfür ist die FPGA-basierte Fahrspurführung des SAV.
- Der ARM Cortex A9 DualCore Prozessor befindet sich auf einem Entwicklungsboard [30] innerhalb der *Open Multimedia Application Plattform* [43] (OMAP 4430) von Texas Instruments, die zur Vorbereitung auf die Zynq 7000 Plattform eingesetzt wird, bis diese verfügbar ist.
- Auf dem ARM Prozessor wird ein Linux-basiertes Betriebssystem zur Unterstützung der Software Applikation eingesetzt.
- Mit diesem Software Ansatz wird die Leistungsfähigkeit des ARM Cortex A9 Prozessors in Bezug auf die Applikationsanforderungen der Fahrspurführung analysiert, um damit die Grundlage zum Entscheidungsprozess innerhalb des HW/SW Co-Designs für die Portierung der SAV Systemarchitektur auf die Zynq Plattform zu liefern.

- Durchdringung und Realisierung von verschiedenen UNIX konformen Multithreading Technologien und Entwurfsmustern zur parallelen Programmierung auf dem ARM Cortex A9 DualCore Prozessor unter Linux.
- Zusätzlich werden Kamera- und bildverarbeitende Bibliotheken und Frameworks bei der Implementierung eingesetzt, sowie deren Unterstützung für die ARM Plattform und Tauglichkeit für die Fahrspurführungs-Pipeline überprüft.
- Es wird eine Entwicklungsumgebung eingerichtet, in der die Softwareentwicklung für den ARM Prozessor unabhängig von der Zielplattform durch eine Reihe von Entwicklungswerkzeugen unterstützt wird.

Gliederung der Arbeit

In **Kapitel 2** werden verschiedene Multiprozessor-Architekturen miteinander verglichen. Darauf aufbauend wird das SMP Modell des Linux Kernels zur Unterstützung von symmetrischen Multiprozessoren beschrieben.

Kapitel 3 liefert eine Übersicht zur Hardwareplattform und der OMAP Architektur. Neben dem ARM Prozessor werden weitere Co-Prozessren innerhalb zur Multimediaverarbeitung eingesetzt. Am Beispiel der OMAP Plattform wird die errichtete Entwicklungsumgebung für die Softwareentwicklung für ARM-basierte Zielplattformen beschreiben. Des Weiteren werden die Frameworks für Multimediaverarbeitung und Multithreading aufgeführt.

Kapitel 4 beschreibt den Entwurf und die Implementierung der Fahrspurführungsapplikation. Zuerst werden verschiedene Multithreading Entwurfsmuster für die parallele Ausführung der Verarbeitungskette vorgestellt. Es folgt die Beschreibung des theoretischen und mathematischen Funktionsprinzips der einzelnen Funktionsblöcke entlang der Pipeline sowie deren Implementierung. Darauf aufbauend wird die Softwarestruktur und der dynamische Ablauf der Applikation erläutert.

In **Kapitel 5** werden die Ergebnisse der Leistungsanalyse präsentiert, die eine Aussage über die Auslastung und den Ressourcenverbrauch des ARM DualCore Prozessors durch die Fahrspur-Pipeline treffen.

Den Abschluss dieser Arbeit bildet die Zusammenfassung in **Kapitel 6**.

2. Multiprozessorarchitekturen

Die Definition für MPSoC Architekturen, nach der unterschiedliche Prozessoren auf einem integrierten Schaltkreis integriert sind, bietet Spielraum für eine Vielzahl an verschiedenen MPSoC Architekturen. Die einzelnen Prozessorsysteme werden anhand ihrer Charakteristika wie z.b. den Befehlssatz, die Prozessorarchitektur, die Taktfrequenz oder die Cache Anordnung unterschieden. Abgesehen von den unterschiedlichen Eigenschaften der Prozessor-Subsysteme unterscheiden sich MPSoC Architekturen durch die Infrastruktur On-Chip, wie z.B. die integrierten Kommunikationsmechanismen zwischen den Prozessoren. Hierbei wird zwischen nachrichtengekoppelten und speichergekoppelten Multiprozssorsystemen unterschieden. Während in nachrichtengekoppelten Architekturen die Prozessoren über Nachrichtenbussysteme On-Chip miteinander kommunizieren, nutzen die speichergekoppelten Systeme einen gemeinsamen Speicher zur Kommunikation durch einen ebenfalls gemeinsam genutzten Speicherbus. Basierend auf den technischen Eigenschaften eines Multiprozessorsystems werden Architekturen vor allem nach der Zuteilung von Funktionen auf Prozessoren, bzw. nach der daraus resultierenden hierarchischen Struktur, unterschieden.

2.1. Heterogene Multiprozessorsysteme

Bei heterogenen MPSoC handelt es sich in den meisten Fällen um nachrichtengekoppelte Multiprozessorsysteme, in denen die Kommunikation durch Message Passing Interfaces über ein Bussystem On-Chip erfolgt, welches die Prozessor-Subsysteme untereinander verbindet. Beispiel hierfür ist das AMBA (*Advanced Microcontroller Bus Architecture*) On-Chip Bussystem für ARM-Prozessoren, welches auch auf der Zynq Plattform integriert ist. Jeder Prozessor wird als eigenständig betrachtet und verfügt über einen eigenen Speicherbereich mit eigenem Adressraum. Es gibt keinen gemeinsam genutzten Speicher On-Chip. Die Heterogenität und vollständige Nebenläufigkeit der Prozessorsysteme wird vor allem

in komponentenorientierten Applikationen mit funktionaler oder *Task-Level* Parallelität ausgenutzt, indem die Softwarekomponenten auf Prozessor-Subsysteme verteilt werden. Moderne Smartphones weisen bis zu acht Prozessorsysteme auf, darunter ein oder meh-

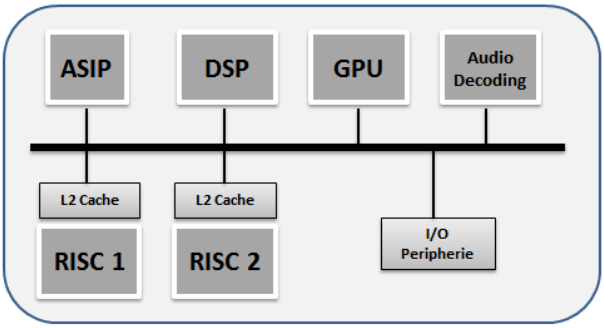

Abbildung 2.1.: Modell eines heterogenen MPSoC in einem Smartphone

rere RISC (Multicore-) Prozessoren zur Ausführung eines mobilen Betriebssystems oder für Kontrollfunktionen der I/O Peripherie sowie eine Reihe von funktionssspezifischen Prozessoreinheiten für die Multimediaverarbeitung. Darunter fällt z.B. das Abspielen von Audiosequenzen durch einen Audioprozessor oder das 3D Rendering durch eine *Graphical Processing Unit* (GPU). Ein Funktionsblock muss hierbei nicht auf ein Prozessorsystem beschränkt werden, sondern kann ebenso durch mehrere Prozessoren ausgeführt werden. Beispielsweise gliedert sich der MPEG-2 Encoding Prozess in eine Verarbeitungspipeline durch die drei Funktionsblöcke *Motion Estimating, Discrete Cosine Transformation* und *Huffman Coding*. Diese Blöcke können nebenläufig auf drei Prozessor-Subsystemen ausgeführt werden [18].

2.2. Symmetrische Multiprozessoren

Symmetrische Multiprozessorsysteme haben ihren Ursprung sowohl in Großrechenanlagen als auch in Form von Multicore-Prozessoren in PC Systemen und halten zunehmend Einzug in die Embedded Systeme, was die Entwicklung von von Mikroprozessoren mit mehreren CPU Kernen im Embedded Bereich widerspiegelt, wie z.B. der ARM11 MPCore oder der ARM Cortex A8 oder A9 Prozessor. Symmetrische Multiprozessorsysteme werden durch folgende Eigenschaften [38] definiert:

- Alle Prozessoren teilen sich denselben Speicherbereich und Adressraum (*Shared Memory*)
- Alle Prozessoren kommunizieren durch den gemeinsamen Speicher über einen gemeinsam genutzten Speicherbus, wobei die Speicherzugriffszeit über den Bus von jedem Prozessor aus identisch ist. Diese Prinzip wird auch als *Uniform Memory Access* (UMA) bezeichnet.
- Alle Prozessoren teilen sich den Zugriff auf gemeinsam genutzte I/O Geräte und Schnittstellen und können gleichberechtigt dieselben Funktionen ausführen.
- Alle Prozessoren werden von einem SMP-fähigen Betriebssystem kontrolliert, welches Interaktionen zwischen Prozessoren und ihren ausgeführten Prozessen gewährleistet.

Neben der Zahl der Kerne bei Multicore-Prozessoren ist die Anzahl der Cache Levels sowie deren Anordnung und Zugriffsart ein weiteres Vergleichskriterium von unterschiedlichen symmetrischen Multiprozessorarchitekturen. Sobald in Shared Memory Systemen ein Speicherbereich in den Cache eines Prozessors kopiert wird, ergibt sich somit das Problem der Cache Kohärenz. In vielen Prozessorarchitekturen werden deshalb zusätzliche Hardware Komponenten integriert um die Cache Kohärenz zu gewährleisten, wie z.B. die *Snoop Control Unit* (SCU) im ARM Cortex A9 MPCore, welche nach dem MESI Protokoll die Konsistenz der Daten zwischen den Caches der CPUs erhält.

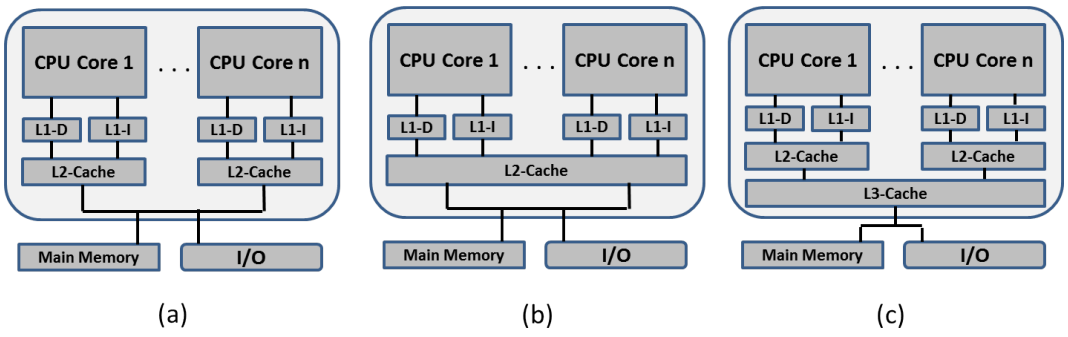

Abbildung 2.2.: Unterschiede in den Cache-Anordnungen zwischen Multicore Architekturen

Die unterschiedlichen Cache Anordnungen verschiedener Prozessorarchitekturen sind in

Abbildung 2.2 dargestellt. In 2.2(a) besitzt jeder Prozessorkern einen eigenen Level 1 Daten- und Instruktion-Cache sowie einen eigenen Level 2 Cache. Über einen gemeinsamen Speicherbus hat jeder Kern Zugang zum gemeinsam genutzten Hauptspeicher und zur Peripherie. Ein Beispiel für diese Cache Anordnung ohne einen gemeinsamen (*Shared*) Cache ist der AMD Opteron Multicore Prozessor. 2.2(b) zeigt eine Multicorearchitektur mit einem Shared Level 2 Cache. Beispiele hierfür sind der ARM Cortex A9 MPCore oder der Intel Core Duo Prozessor. Die Multicorearchitektur in 2.2(c) weist zusätzlich zu getrennten Level 1 und Level 2 Caches einen gemeinsamen Level 3 Cache auf, wie sie auf dem Intel Core i7 Prozessor vorhanden ist.

Die Nutzung eines gemeinsamen Caches hat mehrere Vorteile gegenüber getrennten Caches. Wenn ein Thread auf einem Prozessorkern einen Datenbereich aus dem Hauptspeicher in den Shared Cache holt und ein zweiter Thread auf einem anderen Kern denselben Bereich adressiert, ist dieser bereits auf dem Shared Cache vorhanden, was die Zugriffszeit reduziert. Außerdem müssen gemeinsam genutzte Daten in einem Shared Cache nicht repliziert werden. Die Latenzzeit bei Interprozess-Kommunikation über Shared Memory auf einem Shared Cache wird ebenfalls reduziert, da kein Umkopieren von Daten notwendig ist. Der Vorteil bei getrennten Level 2 Caches ist, dass Prozessorkerne noch schnellere Zugriffszeiten auf ihre privaten Level 2 Caches haben, was sich vor allem bei funktionaler Parallelität von Threads mit wenig Datenabhängigkeit und Synchronisation zwischen Prozessen bemerkbar macht.

2.3. Multicore Prozessoren in MPSoC Architekturen

Innerhalb einer MPSoC Architektur führt der Trend [31] zu einer Abnahme der Anzahl an Prozessor-Subsystemen bei gleichzeitiger Zunahme an CPU Kernen durch Multicore Prozessoren. Damit wird Zahl der dedizierten Softwarekomponenten für verschiedene Prozessorarchitekturen reduziert und der Softwareanteil auf den Multicore Prozessoren erhöht. Parallele Software wird somit auf mehreren Kernen nach dem Prinzip MIMD (*Multiple Instruction Multiple Data*) ausgeführt. Durch die Integration eines Multicore Prozessors innerhalb eines MPSoC wird somit ein zusätzlicher Grad an Parallelisierung erreicht.
Durch die Integration des ARM Cortex A9 DualCore Prozessors wird dieser zusätzliche Parallelisierungsgrad sowohl innerhalb der MPSoC Architektur der OMAP Plattform

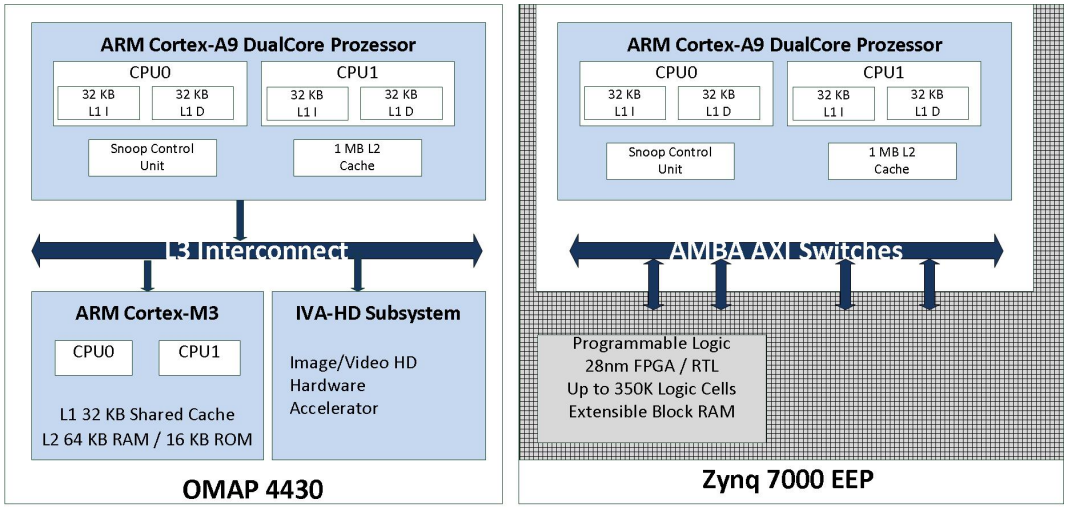

Abbildung 2.3.: Gewinn einer zusätzlichen Parallelisierungsebene innerhalb des MPSoC durch den DualCore

als auch der Zynq 7000 Plattform gewonnen. Der Unterschied zwischen beiden besteht darin, dass die Parallelisierung außerhalb des ARM Prozessors auf MPSoC Ebene durch den FPGA der Zynq Plattform frei gestaltbar ist, während, die Prozessorsubsysteme auf der OMAP Plattform fest vorgegeben sind.

2.4. SMP im Linux Kernel

UNIX basierte Betriebssysteme gehören mittlerweile zu den am meisten verbreiteten Standards im Bereich (Echtzeit-) Betriebssysteme für Embedded Systeme [46]. Unter dem Begriff *Symmetric Multiprocessing* verbirgt sich die Kernelunterstützung für Multicore-Prozessoren durch gleichmäßiges Verteilen von Prozessen auf mehrere CPU Kerne innerhalb des Schedulings. Die am meisten verwendete Datenstruktur im Linux Kernel ist der *Process Descriptor* [10] vom Typ `task_struct`, welcher einen Prozess mit eigenem Adressraum repräsentiert, beispielsweise einen Thread. Seine Felder fassen alle Informationen des Prozesses für dessen Verwaltung durch den Kernel zusammen, z.B. die `pid` für die Thread-ID oder seinen aktuellen Zustand im Feld `state`. Für jeden auszuführenden Prozess wird ein Process Descriptor erzeugt. Der Kernel weist jeder CPU ihre eigene *Runqueue* zu, ein Satz von insgesamt 140 doppelt verketteten Listen von Process Descriptoren, wobei jeder Runqueue eine Priorität zugeordnet wird. Bei

einem Contex Wechsel entzieht der Scheduler dem aktuellen Prozess die CPU Ressourcen und weist sie gemäß der konfigurierten `Scheduling Policy` dem nächsten Prozess in derselben Runqueue-Liste zu. Zusätzlich wird die Auslastung der Runqueues mit derselben Priorität auf den CPUs verglichen. Sobald diese nicht ausgewogen ist, werden Process Descriptoren der stärker ausgelasteten Runqueue auf die schwächer ausgelastete umverteilt, bis das Verhältnis ausgewogen ist.

Die Verarbeitungskette der Kernel-Funktionen, die maßgeblich an SMP beteiligt sind, ist in der folgenden Abbildung 2.4 dargestellt. Der Übersicht halber werden nur die wichtigsten Funktionen auf oberster funktionaler Ebene beschrieben. Diese bedienen sich einer großen Zahl von Hilfsfunktionen und Makros im Linux Kernel.

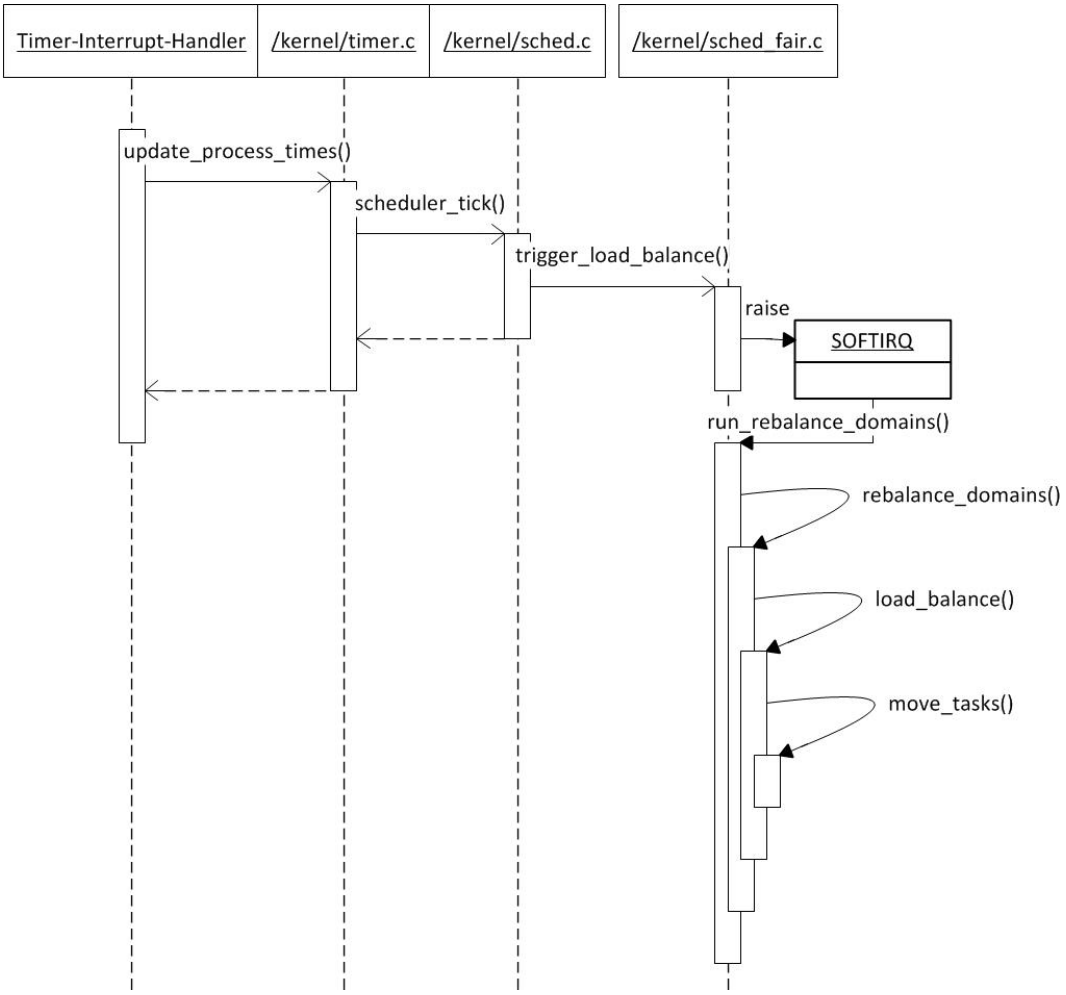

Abbildung 2.4.: Load Balancing der CPU Auslastung in SMP-Systemen durch den Linux Kernel

Die Funktion `scheduler_tick()` wird bei jedem Timer Interrupt durch die Interrupt Handler Funktion aufgerufen. Sie initiiert das Re-Scheduling und ruft am Ende die Funktion `trigger_load_balance()` auf. Diese bestimmt die Anzahl der Prozesse in den Runqueues aller CPUs und überprüft, ob deren Arbeitsauslastungen ausgeglichen sind. Falls nein wird ein Software-Interrupt ausgelöst, dem die Handler-Funktion `run_rebalance_domains()` bei der der Initialisierung des Schedulers zugewiesen wurde. Diese initiiert das Re-Balancing der Runqueues durch die Funktion `load_balancing()`. Sie sucht die am stärksten ausgelastete Runqueue und ermittelt die Anzahl Prozesse, die in die Runqueue derselben Priorität auf einer anderen CPU umverteilt werden, damit die Arbeitsauslastung wieder ausgeglichen ist. Die Migration von Prozessen wird von der Funktion `move_tasks()` durchgeführt.

3. Übersicht zur MPSoC Plattform der Fahrspurführung

Dieses Kapitel beschreibt zunächst die Hardwarestruktur der OMAP 4430 Plattform und diejenigen Prozessor-Subsysteme, die für die Ausführung der Funktionalität der Fahrspurführung genutzt werden. Eine komplette Übersicht zur Hardware-Plattform, sowie der Software-Architektur und der Entwicklungsumgebung ist in Abbildung 3.1 dargestellt.

3.1. OMAP Plattform

Die OMAP 4430 [43] integriert verschiedene Prozessor-Subsysteme auf einem Chip, die über ein Level 3 Interconnect Bussystem On-Chip miteinander verbunden sind. Der OMAP 4430 MPSoC-Chip ist auf dem Entwicklungsboard *PandaBoard* [30] integriert, welches dem OMAP Prozessorsystem eine Reihe von Peripherie-Geräten zur Verfügung stellt. Die Speicherhierarchie setzt sich aus einem 56 KB On-Chip SRAM und einem 1GB Off-Chip SDRAM Speicherblock auf dem PandaBoard zusammen. Durch ein SDMMC Interface steht zusätzlich der Speicherbereich einer SD Karte zur Verfügung, die die Bootloader Dateien, das Kernel-Image und das Linux Dateisystem enthält. Insgesamt kann die OMAP 4430 Architektur einen Adressraum von 4 GB verwalten. Zur externen Kommunikation enthält das PandaBoard Schnittstellen für USB, JTAG und RS-232, sowie einen Ethernet und WLAN Adapter.

3.1.1. ARM Cortex A9 Prozessor

Der ARM Cortex A9 DualCore Prozessor verfügt pro CPU jeweils über 32KB Level 1 Instruction- und Data-Caches sowie 1 MB geteilten Level 2 Cache. Die Cache-Kohärenz

wird durch eine Snoop Control Unit (SCU) auf Hardware Ebene hergestellt. Jeder CPU Kern verfügt über einen *Vector Floating Point* Co-Prozessor zur Auslagerung von Fließkommaberechnungen. Eine *Memory Management Unit* (MMU) verwaltet für beide Kerne den virtuellen Adressraum für den gemeinsamen Zugriff auf den Arbeitsspeicher.

3.1.2. Multimediaverarbeitung durch Co-Prozessoren

Das Fahrspurführungssystem implementiert ein Kamera-Interface, das einen Videodatenstrom entgegennimmt, aus welchem mit Bildverarbeitungsmethoden die Fahrspur ermittelt wird. In dieser Applikation wird eine IP-Kamera eingesetzt, die über Ethernet einen Videostream entweder im *Motion JPEG* (mjpg) Format oder im MPEG-4 Format liefert. Für die Dekodierung des MPEG-4 Streams steht auf der OMAP 4430 Plattform ein *Image Video Accelerator - High Dfinition* (IVA-HD) Subsystem für Video Encoding und Decoding zur Verfügung, welches neben MPEG-4 auch weitere Multimedia-Formate unterstützt. Eine Open-Source Lösung für die Bild- und Videoverarbeitung auf der OMAP 4430 Plattform bietet die SYSLINK Architektur [39] von OMAP. Es beinhaltet sowohl die Schnittstelle für die Inter-Prozessor-Kommunikation (IPC) zwischen den Prozessor-Subsystemen auf der OMAP Plattform, als auch die Firmware-Images für den ARM Cortex-M3 DualCore Prozessor und den IVA-HD Baustein inklusive der Multimediadecoder. Die Decodierung des MPEG-4 Streams auf dem IVA-HD Subsystem wird dabei durch den ARM Cortex-M3 verwaltet. Über das SYSLINK Interface nimmt die Firmware auf dem ARM Cortex-M3 Steuerungs– und Kontrollnachrichten vom ARM Cortex A9 Prozessor zur Steuerung des Dekodierprozesses entgegen [28]. Das IVA-HD Subsystem verarbeitet 1920 X 1080 Pixel bei 30 Frames pro Sekunde, was einer Gesamtauflösung von 20 Mb/s entspricht.

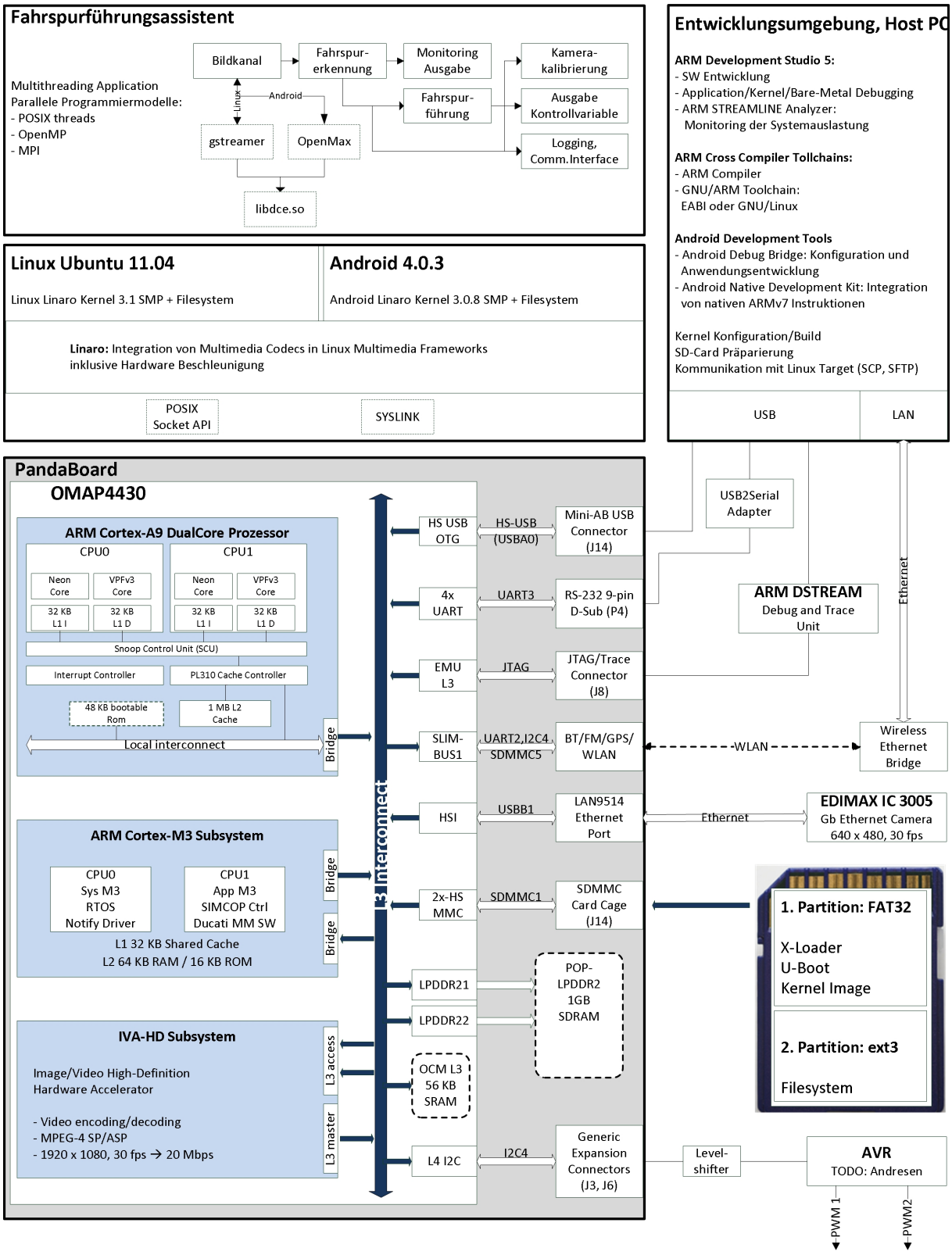

Abbildung 3.1.: Übersicht zur Hardware Plattform, der Softwarestruktur und Entwicklungsumgebung

3.2. Softwarekomponenten auf dem ARM Prozessor

Die Softwarehierarchie auf dem ARM Prozessor besteht aus der Betriebssystemebene und der Fahrspurführung auf Applikationsebene. Hinzu kommen Frameworks und Bibliotheken zur Unterstützung der Applikation bei der Video- und Bildverarbeitung, sowie Linux APIs zur Nutzung von Kernelfunktionen, etwa für das Multithreading.

3.2.1. Betriebssystem

Auf dem ARM Prozesor wird das Betriebssystem *Linaro* [23] eingesetzt, das von der gleichnamigen Non-Profit Organisation entwickelt wurde, die aus ca. 120 Entwicklern besteht und es sich zum Ziel gemacht hat, die Open Source Softwareentwicklung für ARM Linux voranzutreiben. Es basiert auf der Linux Kernelverison 3.1.5 SMP und wurde in eine Ubuntu-Umgebung integriert. Der Kernel wurde für die Unterstützung der OMAP Hardware um das SYSLINK Framework erweitert, was die Firmware für den ARM Cortex M3 und die Multimediadecoder für das IVA-HD Subsystem einschließt. Die API für die SYSLINK Funktionalität wird durch die Bibliothek `libdce.so` (*Distributed Codec Engine*) der Applikationsebene zur Verfügung gestellt.

3.2.2. Bibliotheken für Bildverarbeitung und Multithreading

Die Fahrspurführung nutzt folgende Bibliotheken und Frameworks zur Unterstützung der Kamera- und Bildverarbeitung, sowie für Multithreading:

- *GStreamer* [13] für das Kamera Handling
- *OpenCV* [16] zur Bildverarbeitung
- *POSIX Threads Library* für Multithreading unter Unix
- *OpenMP* [12] Framework für paralleles Programmieren auf Shared Memory basierten Multicore Prozessoren

GStreamer ist ein Open Source Multimedia Framework zur Unterstützung der Verarbeitung von Videostreams in Softwareapplikationen und wird in diversen Mediaplayern verwendet. Die Grundfunktionen werden in der fixen *Core Library* zusammengefasst, die auf dem GLib 2.0 Object Modell für objektorientiertes Design unter Unix C basiert.

Die Grundfunktionalität besteht im Handling diverser Containerformate, Codecs oder Streamingprotokollen. Die zweite Säule neben der Core Library ist die Plugin Architektur, durch die eine Reihe diverser Plugins zu den Grundfunktionen integriert werden. Für die Fahrspurführung wird das `ducatimpeg4dec` Plugin genutzt. Dabei handelt es sich um eine Erweiterung für die hardwareunterstützte Videoverarbeitung auf der OMAP Plattform durch Nutzen der SYSLINK-Schnittstelle.

OpenCV ist ebenfalls ein Open Source Framework von Bildverarbeitungsfunktionen. Die Funktionalität erstreckt sich dabei über viele Bereiche der Bildverarbeitung, wie z.B. Transformationen, Mustererkennung, geometrische Abstrahierung oder Kamerakalibrierung. Die Fahrspurführung nutzt OpenCV Funktionen zur Kamerakalibrierung und für Bildtransformationen. Zugleich werden OpenCV Datenstrukturen und Formate für die Speicherung von Bildmengen oder geometrischen Objekten wie Punkte, Vektoren oder Matrizen genutzt. Hierfür werden zahlreiche Hilfsfunktionen aus OpenCV für algebraische Operationen wie z.B. Matrizenmultiplikationen verwendet.

3.3. Entwicklungsumgebung

Die Fahrspurführung wurde auf einem Host-PC mit dem Programmierwerkzeug ARM Development Studio (DS) 5 [5] entwickelt, das zusammen mit weiteren Hilfswerkzeugen wie einem SCP Client, einem seriellen Terminal oder den standardmäßigen Build-Utilities unter Linux die Entwicklungsumgebung für die Zielplattform darstellt (vgl. Abb. 3.2). ARM DS-5 kann für die Applikationsentwicklung zwar plattformunabhängig auf Windows oder Linux eingerichtet werden, jedoch setzt das Erstellen des Linaro-Images auf einer bootfähigen SD Karte Linux-Werkzeuge voraus. In dieser Arbeit wurde die Entwicklungsumgebung auf einer virtuellen Linux Maschine auf einem Host-PC errichtet [33].

In ARM DS-5 kann zwischen der ARM Compiler Toolchain [4] und dem GNU ARM Linux Compiler [25] gewählt werden. Beide Compiler sind kompatibel zueinander, da beide dasselbe *Application Binary Interface* (ABI) [3] der ARM Architektur implementieren. Lange Zeit bestand der Vorteil des ARM Compilers gegenüber dem GNU Compiler in der Erweiterung um den Befehlssatz für den VFP Co-Prozessor für hardwareunterstützte Fließkommaberechnungen. Der GNU Compiler `arm-linux-gnueabi-gcc` für ARM Linux unterstützte bisher nur die Integerarithmetik für ARM Mikrocontroller. Mittlerweile

wurde die Nachfolgegeneration des GNU ARM Compilers `arm-linxu-gnueabihf-gcc` veröffentlicht, die ebenfalls den VFP Co-Prozessor unterstützt (erkennbar am Postfix `-hf-` für *Hardware-Floating* und ab der ARM DS Version 5.9 angeboten wird. Da in der Fahrspurführungsapplikation abgesehen von den trigonometrischen Berechnungen der Hough Transformation größtenteils nur Operationen auf ganzzahlbasierten Pixelmengen durchgeführt werden und Punktkoordinaten ebenfalls nur in Integerdatentypen gespeichert werden, bringt die hardwareunterstützte Fließkommaberechnung für die Applikation keine Vorteile. ARM DS-5 ist ein auf Eclipse basierendes Entwicklungs-

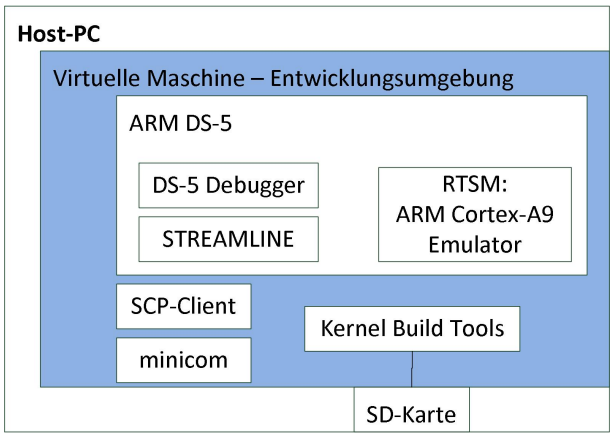

Abbildung 3.2.: Übersicht zu den Komponenten der Entwicklungsumgebung

umgebungswerkzeug (IDE) zur Erstellung von Applikationen und Libraries speziell für ARM Prozessoren. Unterstützt werden die Sprachen C und C++. Neben den Standard Features von Eclipse, wie z.B. dem Code Editor oder dem Project Manager bietet ARM DS-5 folgende Erweiterungen für die Entwicklung von ARM Software:

- Der *DS-5 Debugger* [6] unterstützt für das Cross-Debugging von Applikationen auf Zielplattformen folgende drei Debug-Modi: *Bare-Metal* (ohne Betriebssystem), *Kernel Debugging* für Linux, Symbian OS und Android, sowie *Application Debugging* für Software auf Applikationsebene. Daneben bietet der DS-5 Debugger standardisierte Debug Konfigurationen für alle Prozessoren der Cortex-A Serie von ARM. Über integrierte Netzwerk- und Kommunikationsmechanismen wie dem *Remote System Explorer* (RSE) sowie SSH und Telnet Terminals wird eine direkte Verbindung zum Linux Filesystem auf dem ARM Cortex-A9 Prozessor der Zielplattform hergestellt.

- Der *STREAMLINE Performance Analyzer* überwacht die Systemauslastung des ARM Prozessors durch zeitgetriggerte Auswertung von CPU Statistiken während der Ausführung oder des Debuggings der entwickelten Software.
- Mit dem *Real-Time System Model* (RTSM) Simulator können eine Reihe von ARM Prozessoren, darunter auch die Cortex-A Serie auf dem Host-PC emuliert werden.

Das Cross-Debugging von Applikationen auf einer Zielplattform von einem Host-PC aus wird durch entferntes Starten eines GDB-Server Prozesses auf der Zielplattform durch den DS-5 Debugger auf dem Host-PC realisiert. Der GDB-Server ist die Gegenstelle zum DS-5 Debugger, verwaltet die Debug Session auf der Zielplattform und leitet die Ausgaben über Ethernet an den DS-5 Debugger weiter.

Die Gegenstelle für den STREAMLINE Analyzer ist das Kernelmodul *gator* auf der Zielplattform. Bei jedem Timer-Interrupt Handling sammelt es die CPU Statistiken wie z.B. die CPU Auslastung, die Verteilung der ausgeführten Prozesse auf die CPU Kerne, sowie die Cache- und Speicherbelegung, Interrupts oder Netzwerkaktivitäten. Diese werden während einer Streaming-Session pro Sekunde über eine Ethernet Schnittstelle an den STREAMLINE Analyzer geschickt, der sie in ARM DS-5 grafisch in Form von Diagrammen ausgibt. Der STREAMLINE Analyzer wird für die Leistungsanalyse des ARM Prozessors bezüglich der Auslastung durch die Fahrspurführung genutzt, vgl. Kap. 5. Eine Übersicht zu den Kommunikationsmechanismen zwischen der Entwicklungsumgebung und der Zielplattform ist in der folgenden Abbildung 3.3 dargestellt.

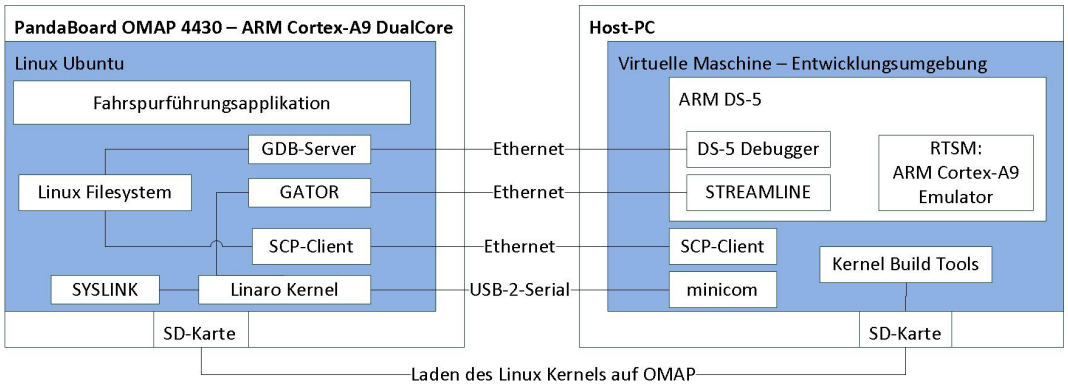

Abbildung 3.3.: Kommunikation zwischen der Zielplattform und der Entwicklungsumgebung auf dem Host-PC

4. Entwurf und Implementierung der Fahrspurführung

Die Fahrspurführungsapplikation wurde in dieser Arbeit in C unter Linux zur Ausführung auf dem ARM Cortex A9 Prozessor der OMAP Plattform entwickelt. Ihre funktionalen Eigenschaften basieren auf bestehenden Fahrspurführungsapplikationen, die im Rahmen des HPEC Projekts auf verschiedenen Hardwareplattformen entwickelt wurden. Dieses Kapitel beschreibt die Softwarestruktur und den dynamischen Ablauf der Applikation, sowie die Funktionskonzepte der einzelnen Verarbeitungsstufen und deren Implementierung in C.

4.1. Entwurfsmuster und Multiprocessing Konzepte

Zunächst werden verschiedene Entwurfskonzepte für Multithreading-Applikationen zur Programmierung des ARM Cortex-A9 DualCore Prozessors vorgestellt. Nach diesen Programmiermodellen wird das Fahrspurführungssystem auf dem ARM Cortex-A9 implementiert. Darauf aufbauend wird das *Pipeline* Entwurfsmuster für die sequentielle Datenverarbeitungskette durch mehrere simultan ausgeführte Threads vorgestellt. Anschließend werden mit der funktionalen und der datenabhängigen Parallelität zwei Konzepte der Multithreading-Programmierung erläutert. Beide Implementierungen werden bei gleicher Funktionalität im Hinblick auf die Leistungskennzahlen CPU-Auslastung und Latenzzeit miteinander verglichen.

4.1.1. Pipeline Pattern

Im Fahrspurführungssystem wird die Fahrspur von einer IP-Kamera aufgenommen und als MPEG-4 Stream über das Netzwerkprotokoll RTP übermittelt. Der erste Schritt auf

Software-Ebene ist das Zerlegen des dekodierten Streams in einzelne Bild-Frames. Somit ergibt sich eine Bildsequenz der Fahrspur in zeitlicher Reihenfolge, wobei jedes Frame gemäß der Bildreihenfolge eine Kette (vgl. Abb. 4.1) von folgenden Verarbeitungsstufen durchläuft, bis die Ausgabe des Lenkwinkels für das Modellfahrzeug am Ende der Verarbeitungskette erfolgt.

1. *Imaging*: Aufspalten des Kamerastroms in einzelne Bilder, vgl. Kap.
2. *Binarizing*: Binarisieren des Graustufenbildes in weiße und schwarze Pixelmengen
3. *Erosionsfilter*: Reduktion der weißen Pixelmenge zur Identifikation von Kanten im Bild
4. *Dynmamic ROI*: Verschiebung und Anpassung einer *Region Of Interest* um die Fahrspurmarkierung, gemäß ihrer Lagenverschiebung.
5. *Hough Transformation*: Extrahieren der Fahrspur aus dem Bild als geometrisches Objekt in Form einer Geraden
6. *Lane Following*: Berechnung des Lenkwinkels anhand der Lage der Fahrspurgeraden
7. *PWM Output*: Ausgabe des Lenkwinkels

Jede Verarbeitungsstufe wird von einem Funktionsblock innerhalb der Fahrspurführungsanwendung [21] realisiert. Die einzelnen Funktionsblöcke entlang der Verarbeitungskette sind in Abbildung 4.1 dargestellt. Dieses Entwurfskonzept einer Verarbeitungskette von Datensequenzen durch mehrere Funktionsblöcke entspricht dem Entwurfsmuster *Pipeline*, welches in [15] durch Multithreading erweitert wird, indem einzelne Verarbeitungsblöcke durch Threads implementiert werden, die auf mehreren Prozessorkernen ausgeführt werden. Durch diese simultane Ausführung von Threads auf mehreren CPUs wird zwar nicht die Verarbeitungszeit pro Thread erhöht, dafür jedoch der Verarbeitungsdurchsatz, da die Stufen gleichzeitig unabhängige Arbeitsschritte auf unterschiedliche Frames ausführen können. Das Funktionsprinzip dieses Entwurfsmusters [32] bleibt auch mit der Parallelisierung dasselbe:

- Zwischen einem Thread und seinem Vorgänger, bzw. Nachfolger besteht eine Ein-/Ausgabe-Beziehung.
- Nachdem Thread_n das Frame_n verarbeitet hat, stellt er es seinem Nachfolger Thread_{n+1} bereit und nimmt Frame_{n+1} von seinem Vorgänger Thread_{n-1} entgegen.

- Durch die simultane Ausführung von Threads auf mehreren CPUs wird ein Element auf einer Stufe bearbeitet, während das nächste Element bereits von der vorherigen Stufe bearbeitet wird.

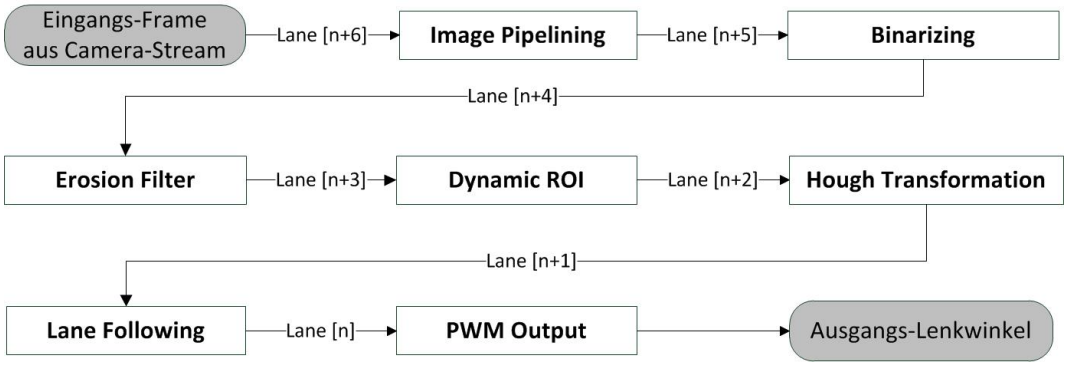

Abbildung 4.1.: Verarbeitungsstufen der Fahrspurführungsapplikation für die einzelnen Bilddatensequenzen

4.1.2. Entwurfsmuster für die parallele Fahrspur Pipeline

Innerhalb der Applikation übersteigt die Zahl der Pipeline-Stufen (vgl. Abb. 4.1) deutlich die Zahl von 2 CPU-Kernen des ARM Cortex-A9 Prozessors der OMAP 4430 Plattform. Das Pipeline Konzept, wonach jede Verarbeitungsstufe in einem eigenen Thread ausgeführt wird, ist in dieser Arbeit nicht berücksichtigt, da der Synchronisationsaufwand zwischen Threads sonst nicht Verhältnis zum funktionalen Aufwand der einzelnen Pipelinestufen steht. Stattdessen werden mehrere Pipelinestufen in einem Thread zusammengefasst. Im Folgenden werden zwei verschiedene Entwurfsmuster für eine Multithreading-Implementierung der Fahrspurführungsapplikation auf den zwei CPU-Kernen vorgestellt.

1. Insgesamt zwei Threads, ein Thread pro CPU, mehrere Pipeline-Stufen pro Thread (vgl. Abb. 4.2)

2. Ein Thread insgesamt, sequentielle Verarbeitung aller Pipeline-Stufen, datenabhängige Parallelisierung innerhalb jeder Stufe (vgl. Abb. 4.1.2) durch OpenMP. Das Load-Balancing zwischen CPUs wird dem Linux Scheduler überlassen.

Im ersten Entwurfsmuster wird die Pipeline mit zwei Threads realisiert, wobei jeder Thread simultan auf seiner eigenen CPU ausgeführt wird (siehe Abbildung 4.2). Es findet

somit eine explizite Zuweisung zwischen einem Thread und seiner CPU statt. Der Thread *ImageProcessing* implementiert den ersten Teil an Pipeline-Stufen der Verarbeitungskette, wobei die einzelnen Pipeline-Stufen gemäß der Fahrspurführungsanwendung sequentiell durchlaufen werden. Nach vollständigem Durchlaufen des ersten Fahrspur-Frames aller Pipeline-Stufen des ersten Teils wird dieses dem Thread *PathFollowing* bereitgestellt, der den zweiten Teil der Pipeline-Stufen bis zur Lenkwinkelausgabe ausführt. Zeitgleich

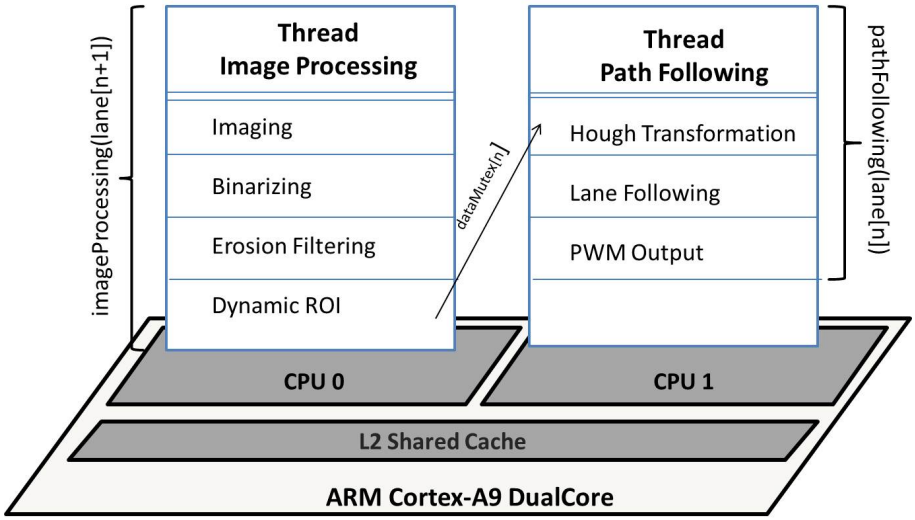

Abbildung 4.2.: Funktionale Parallelität mit zwei Threads

beginnt der Thread *ImageProcessing* mit der Verarbeitung des nächsten Fahrspur-Frames. Der Verarbeitungsdurchsatz reduziert sich damit im Vergleich zur komplett sequentiellen Verarbeitungskette um die Hälfte. Bevor die Verarbeitung der Frame-Sequenz vollständig simultan durch beide Threads erfolgt, muss der Thread *PathFollowing* im ersten Durchgang auf die Verarbeitung des ersten Frames durch den Thread *ImageProcessing* warten, bevor er aktiv wird.

Im zweiten Entwurfsmuster wird die gesamte Pipeline von einem Thread implementiert. Gemäß der standardmäßigen CPU Zuweisung in Linux wird der Thread wie die gesamte Applikation zunächst auf der CPU 0 gestartet (vgl. Abb. 4.1.2), sofern keine expliziten CPU-Zuweisungen per Kommandozeile erfolgen. Durch die datenabhängige Parallelisierung innerhalb der einzelnen Pipeline-Stufen wird eine Hälfte der parallelisierten Programmsequenzen auf die CPU 1 ausgelagert. Dies ähnelt in entfernter Weise dem Master/Slave Prinzip, mit dem Unterschied, dass der Master die Aufgabe dem Slave nicht komplett zuweist, sondern beide sich die Aufgabe teilen. Für die datenabhängige

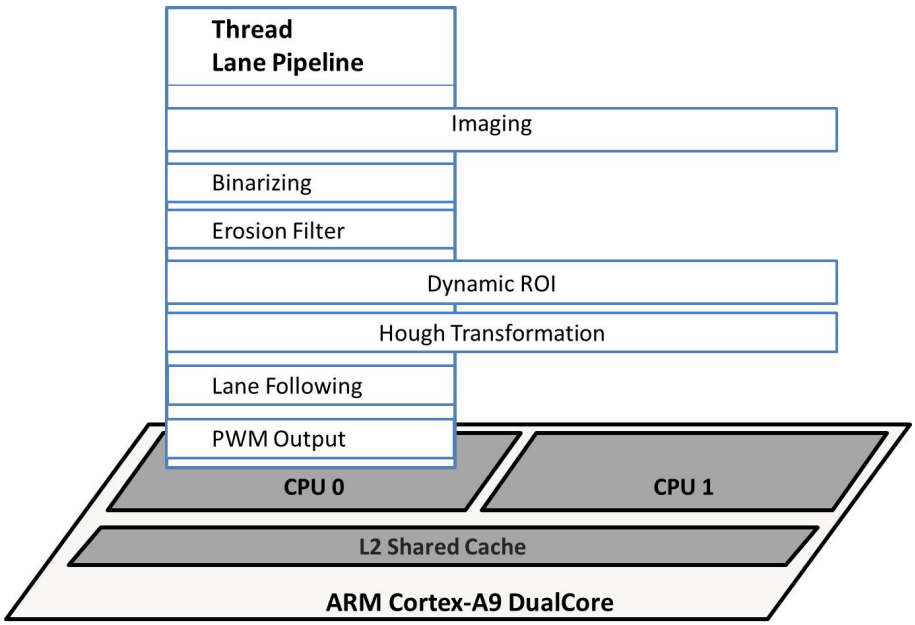

Abbildung 4.3.: Datenabhängige Parallelität mit Arbeitsteilung

Parallelisierung innerhalb der Pipeline-Stufen wurden Code-Sequenzen in den Pipeline-Funktionen *Imaging, Dynamic ROI* und *Hough Transformation* zur Parallelisierung durch OpenMP [12] identifiziert.

Beide Entwurfsmuster sind jeweils Bestandteil zweier zusätzlicher Synchronisationskonzepte (*Partizipation* und *Paketaufteilung*) zwischen der Fahrspurführungsapplikation und einer Laserscanner-basierten Applikation aus [19] zur Erkennung von Hindernissen auf der Fahrspur. Somit teilen sich beide Applikationen mit unterschiedlicher Verarbeitungssequenz die Ressourcen des ARM Cortex A9 Prozessors. Die Interprozesskommunikation beider Applikationen erfolgt über eine Shared Memory Schnittstelle, indem die Hinderniserkennung über Shared Memory ein Signal an die Fahrspurführung schickt und somit einen Ausweichvorgang auf die andere Fahrspur initiiert. Der einfachste Entwurf, wonach jede Applikation auf ihrer eigenen CPU ausgeführt wird, setzt voraus, dass beide Applikationen ihre Verarbeitungsketten sequentiell auf einer CPU innerhalb der von der jeweiligen Sensorik vorgegebenen Frequenz realisieren. Für den Fall, dass mindestens eine Applikation zeitweise beide CPUs in Anspruch nehmen muss, um unterhalb der Verarbeitungsfrequenz zu bleiben, wurden die beiden Multithreading Entwurfmuster für die Fahrspurführung jeweils um die simultane Ausführung zweier Applikationen auf zwei CPUs erweitert:

- Das Konzept der *Partizipation* basiert auf dem ersten Konzept der funktionalen Parallelität durch zwei Threads.
- Das Konzept der *Paketaufteilung* erweitert das zweite Konzept der datenabhängigen Parallelisierung innerhalb einiger Pipeline-Funktionen.

Das Konzept *Partizipation* sieht eine explizite Zuweisung der Hinderniserkennung an die CPU 0 vor, auf welcher auch der Kernel ausgeführt wird. Die Frequenz für die Umgebungserfassung durch den Laserscanner beträgt 10 Hz. Der Thread *Image Processing* innerhalb der Fahrspurführung wird ebenfalls explizit an die CPU 1 zugewiesen und um so viele Pipeline-Funktionen aus dem Thread *Path Following* ergänzt, sodass die Verarbeitungsfrequenz von 30 Bildern pro Sekunde nicht überschritten wird. Die übrigen Pipeline-Funktionen werden vom Thread *Path Following* auf der CPU 0 ausgeführt. Durch das Prinzip der Partizipation überlässt die Hinderniserkennung dem verschlankten Thread *Path Following* einen Teil ihrer CPU-Ressourcen.

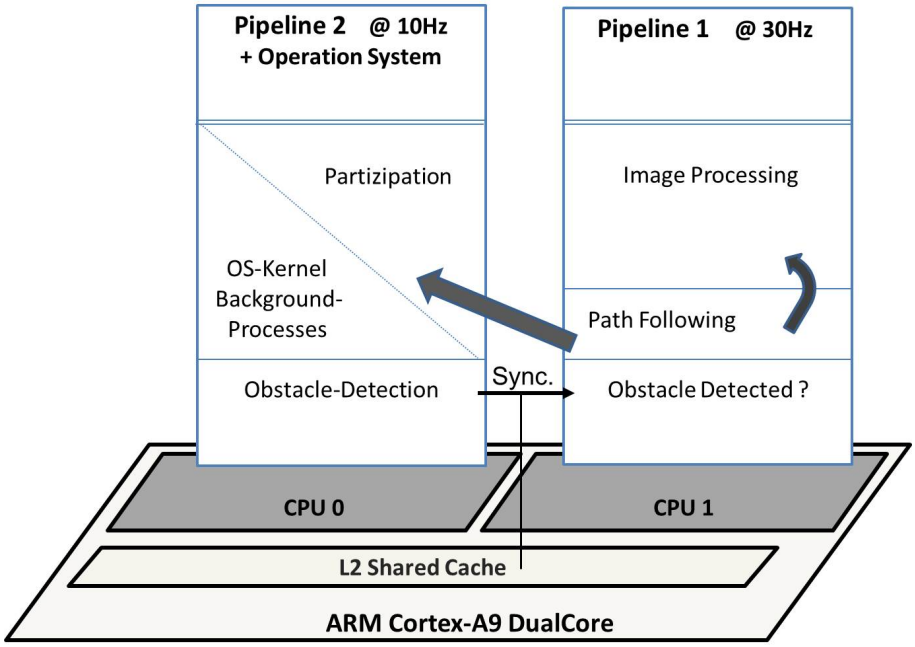

Abbildung 4.4.: Partizipationskonzept: Die Ressourcen der CPU 0, die die Hinderniserkennung nicht benötigt, werden dem verschlankten Thread *Path Following* aus der Fahrspurführung überlassen.

Das Konzept der Paketaufteilung sieht keine explizite Zuweisung eines Threads an eine CPU vor. In der Fahrspurführung werden einzelne Codesequenzen innerhalb einiger

Pipeline-Funktionen durch OpenMP parallelisiert, und auch innerhalb der Hinderniserkennung werden mehrere Threads parallel ausgeführt. Analog zum Konzept der datenabhängigen Parallelität wird das Load-Balancing zwischen CPUs dem Linux Scheduler überlassen.

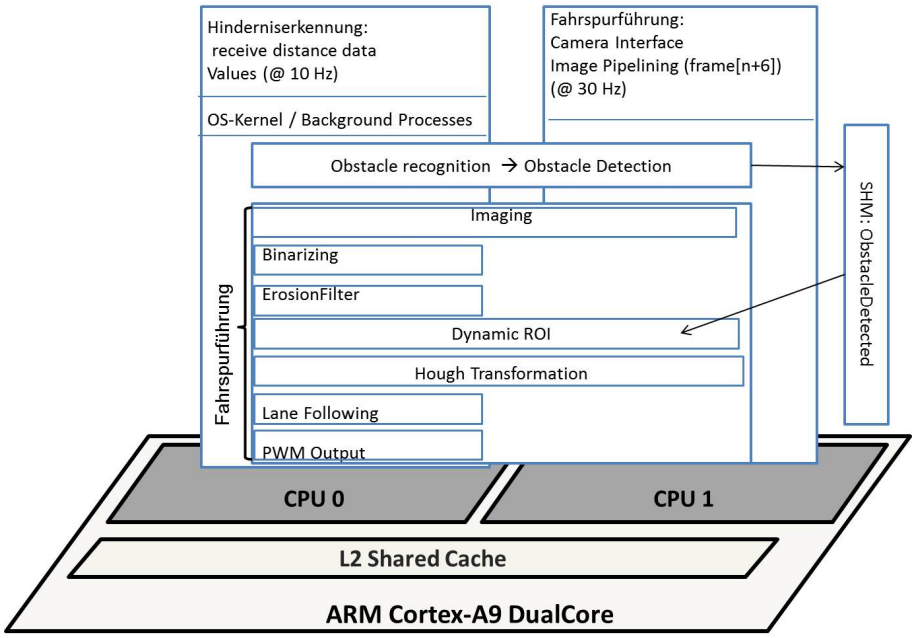

Abbildung 4.5.: Paketaufteilung: SMP Scheduling statt expliziter Thread Zuweisung

4.2. Grundfunktionen zur Initialisierung der Pipeline

Die Initialisierungsphase bei Programmstart dient vor allem dazu, alle Datenstrukturen zu initialisieren und den Speicherbereich zu allokieren, sowie die Konfigurationsparameter einzulesen und die Applikation soweit wie möglich zu konfigurieren um die Pipeline für die Fahrspurverfolgung starten zu können. Hierzu zählen die Kalibrierung der Kamera, die Initialisierung des ROI im Fahrspurbild und Initialisierung der Schnittstelle zur Laserscanner-basierten Hinderniserkennung über Shared Memory. Das Ziel während der Initialisierungsphase ist es, soviel Programmlogik wie möglich von den Pipeline-Funktionen in die Grundfunktionen zur Initialisierung zu verschieben, um die Pipeline-Funktionen so schlank wie möglich zu halten und somit Bearbeitungszeit zu sparen. In den folgenden Abschnitten werden diese Grundfunktionen sowie ihre einzelnen Arbeitsschritte im Detail erläutert.

4.2.1. Kalibrierung der Kamera

Durch die Kalibrierung der Kamera entsteht eine Relation zwischen dem Kamerabild und dem realen Fahrspuruntergrund in folgenden zwei Schritten:

- Projektive Transformation zwischen dem perspektivisch verzerrten Schrägbild aus der Kameraperspektive und dem entzerrten Kamerabild aus der Vogelperspektive auf die Fahrspur.
- Berechnung des Verhältnisses zwischen Pixeln im entzerrten Kamerabild und Zentimetern auf der realen Fahrspurebene.

Um im zweiten Schritt ein konstantes Verhältnis zwischen Entfernungen im Bild und auf der Fahrspurebene berechnen zu können, wird im ersten Schritt eine Transformation ermittelt, durch die einzelne Bildpunkte in Pixelkoordinaten im perspektivisch verzerrten Schrägbild der Kamera zu ihren jeweils korrespondierenden Bildpunkten im entzerrten Kamerabild aus der Vogelperspektive ebenfalls in Pixelkoordinaten zu transformieren. Die projektive Transformation zwischen einem Bildpunkt P'(x', y') in Pixelkoordinaten aus dem verzerrten Schrägbild und seinem korrespondierenden Bildpunkt $P_T(x_T,y_T)$ wird durch eine nicht singuläre 3x3 Homographie-Matrix H realisiert (vgl. Gleichung 4.1). Hierzu werden die korrespondierenden Bildpunkte P' und P_T um eine Dimension ergänzt und somit in dreidimensionalen homogenen Vektorkoordinaten dargestellt.

$$\vec{p_T} = H\vec{p'} \Leftrightarrow \begin{pmatrix} x_T \\ y_T \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \tag{4.1}$$

Die Homographie-Matrix H fasst die folgenden drei Grundfunktionen der Modelltransformation [26] zusammen:

- *Translation* der Blickposition der Kamera relativ zum Fahrzeugkoordinatensystem.
- *Rotation* der Kameraperspektive auf 90 Grad zur Fahrspurebene.
- *Skalierung* durch Multiplikation der homogenen Transformations-Matrix mit einem Skalierungsfaktor s, mit $s = 1$ in diesem Fall.

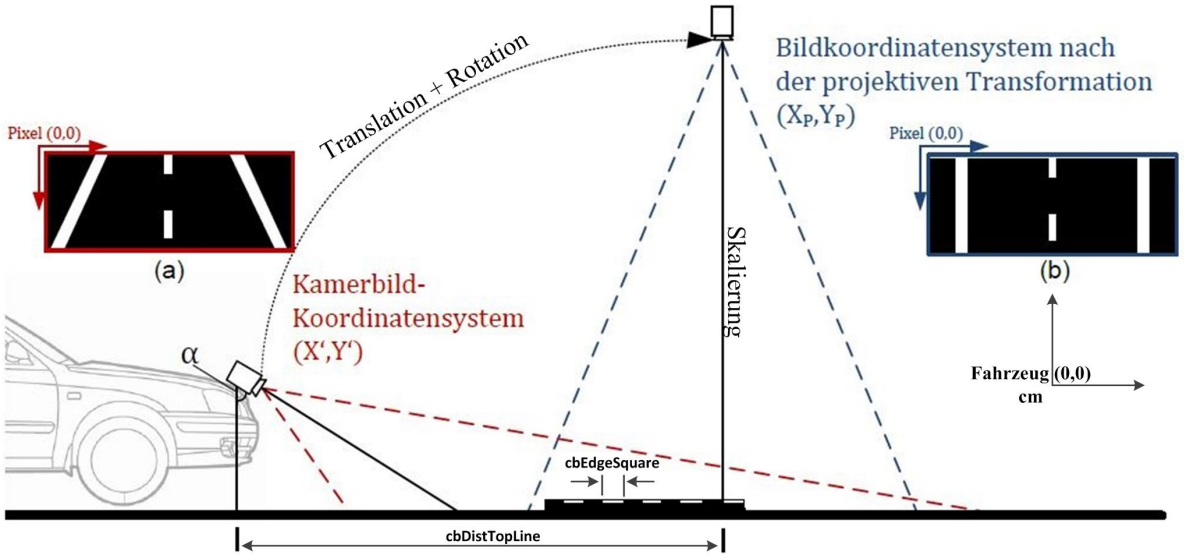

Abbildung 4.6.: Projektive Transformation zwischen dem räumlich verzerrten Schrägbild der Kamera (a) und dem entzerrten Kamerabild aus der Vogelperspektive (b). Das entzerrte Kamerabild wird somit auf die Fahrzeugebene gelegt und projiziert.

Die Homographie-Matrix H kann auf zwei verschiedenen Wegen ermittelt werden. In der ersten Möglichkeit werden ihre Koeffizienten aus intrinsischen und extrinsischen Kameraeigenschaften [14] berechnet. Zu den intrinsischen Eigenschaften zählen z.B. die Brennweite und Linsenverzerrung. Die extrinsischen Eigenschaften beinhalten die geometrische Lage der Kamera, wie den Neigungs– und Richtungswinkel zum Fahrzeugkoordinatensystem und den Abstand zur Fahrzeugebene. In der zweiten Möglichkeit wird die Gleichung 4.1 als lineares Gleichungssystem gelöst, indem vier korrespondierende

Punkte-Paare ermittelt und in das Gleichungssystem eingesetzt werden. Somit können die Matrixkoeffizienten auf umgekehrtem Wege berechnet werden. In dieser Arbeit wird die zweite Möglichkeit vorgezogen, da OpenCV für die Kalibrierung eine Reihe diverser Hilfsfunktionen zur Verfügung stellt, welche für die Kalibrierung aufgerufen werden. Dabei benutzt OpenCV ein Schachbrettmuster für die Kalibrierung, welches auf die Fahrzeugebene gelegt wird (vgl. Abb 4.6), indem die inneren Ecken des Schachbretts sucht und deren Pixelkoordinaten im Bild ermittelt werden. Eine innere Ecke wird von vier Feldern umschlossen, wobei die jeweils diagonal gegenüberliegenden die gleiche Farbe besitzen.

Der gesamte Kalibriervorgang von der Ermittlung der Homographie-Matrix H bis zur Berechnung des Verhältnisses *pixel:cm* wird von der Funktion `calibrateCamera()` im Modul `cameraInterface` realisiert. Vor jedem Kalibriervorgang wird das Schachbrettmuster ungefähr in der Bildmitte gelegt und parallel zu den Achsen im Fahrzeugkoordinatensystem ausgerichtet. Die Kantenlänge eines Feldes auf dem Schachbrett (vgl. Abb 4.6) wird gemessen und im Konfigurationsparameter `cbEdgeSquare` gespeichert. Zusätzlich wird der Abstand zwischen der Kamerahalterung auf dem Fahrzeug und der obersten Linie der inneren Ecken des Schachbretts (nicht zu verwechseln mit der obersten Kante des Musters) vermessen und im Konfigurationsparameter `cbDistTopLine` gespeichert. Für die Kalibrierung in der Funktion `calibrateCamera()` werden folgende Arbeitsschritte ausgeführt:

1. Einlesen eines Kamerabilds mit dem ausgerichteten Schachbrettmuster durch Aufruf von `imaging()`.
2. Erkennung des Schachbrettmusters und allen 49 Ecken eines regulären Schachbrettes im verzerrten Schrägbild durch die OpenCV-Funktion `cvFindChessboardCorners()`
3. Ermittlung der Pixel-Koordinaten der inneren Ecken durch die OpenCV-Funktion `cvFindCornerSubPix()`. Die gefundenen Ecken werden in einem Array gespeichert.
4. Ermittlung der äußeren Ecken `Img_Pt1..4` im Array, die das Quadrat der inneren Ecken aufspannen.
5. Aufsteigendes Sortieren der vier äußeren Ecken anhand ihrer Pixel-Koordinaten, vgl. Abb. 4.7, links.

6. Ermittlung der Distanz `diffx_p12` zwischen `Img_Pt1` und `Img_Pt2` durch Subtraktion der x-Koordinaten (rote Linie). Diese Achse durch die zwei Bildpunkte wird als Rotationsachse für die projektive Transformation festgelegt.

7. Definition der vier korrespondierenden Objektpunkte `Obj_Pt1..4` im entzerrten Kamerabild, vgl. Abb. 4.7, rechts. Die y-Koordinaten der beiden oberen Objektpunkte sind identisch zu ihren korrespondierenden Bildpunkten, da die Rotationsachse auf der gleichen Höhe bleibt und deren Länge `diffx_p12` in beiden Koordinatensystemen ebenfalls identisch ist. Die x-Koordinaten der oberen beiden Objektpunkte haben die gleiche Differenz `diffx_p12`, werden aber um die Bildmitte in x-Richtung gelegt. Befindet sich die Kamera auf dem Fahrzeug in x-Richtung versetzt, wird dies somit durch eine Rotation um die Höhenachse in der perspektivischen Transformatin ausgeglichen. Die restlichen y-Koordinaten ergeben sich durch Aufspannen eines Quadrats mit der Kantenlänge `diffx_p12`.

8. Berechnung der Homographie-Matrix H durch die OpenCV-Funktion `cvGetPerspectiveTransform()`. Ihr werden zwei Arrays übergeben, die mit den vier korrespondierenden Punktepaaren mit gleicher Indizierung gefüllt sind. Die Berechnung erfolgt durch das Lösen das linearen Gleichungssystems in 4.1.

9. Speichern der berechneten Matrix im Feld `homographyMatrix` innerhalb der Datenstruktur vom Typ `PipelineData`.

10. Projektive Transformation des Kamerabildes durch die OpenCV-Funktion `cvWarpPerspective()`. Ihr wird jeweils ein Pointer auf zwei Bildpuffer, sowie die berechnete Matrix H übergeben.

11. Das Verhältnis *pixel:cm* entspricht dem Verhältnis von `diffx_p12` und sechs Kantenlängen eines Schachbrettfeldes:

$$\frac{1Pixel}{1cm} = \frac{\texttt{diffx\_p12}}{6 \cdot \texttt{cbEdgeSquare}} \tag{4.2}$$

    Das errechnete Verhältnis wird im Feld `ppcm` innerhalb der Datenstruktur vom Typ `PipelineData` gespeichert.

12. Für die Umrechnung in y-Richtung zwischen Fahrzeug- und Bildkoordinaten wird der Abstand in y-Richtung zwischen den Ursprüngen der jeweiligen Koordinatensysteme berechnet und in Pixelkoordinaten im Feld `yDistPx` der Datenstruktur `PipelineData` gespeichert, vgl. Abb. 4.7, rechts. Der Abstand vom Fahrzeug zur

Rotationsachse ist durch den Konfigurationsparameter `cbDistTopLine` bekannt und wird zur y-Koordinate eines der oberen Objektpunkte addiert:

$$\texttt{yDistPx} = \texttt{Obj\_Pt2.y} + \texttt{cbDistTopLine} \cdot \texttt{ppcm} \tag{4.3}$$

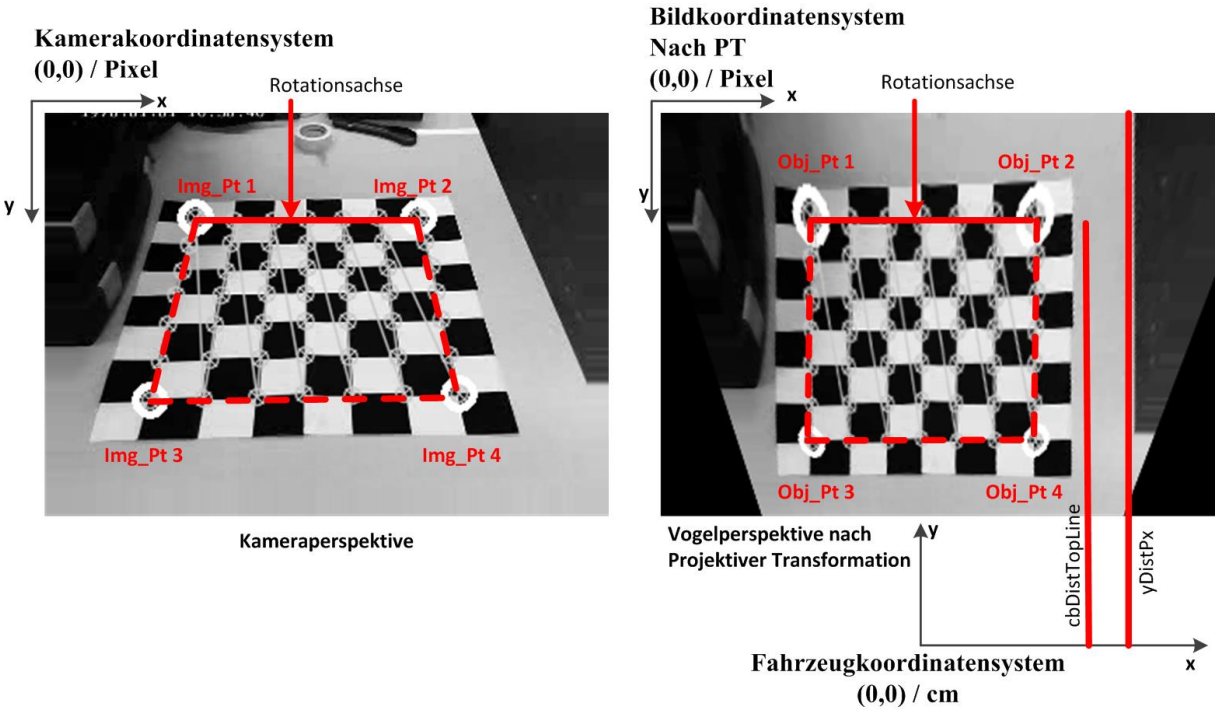

Abbildung 4.7.: Kamerakalibrierung: Die beiden Koordinatensysteme der Kamera- und Vogelperspektive sind in Ursprung, Orientierung und Dimensionierung identisch, ebenso der Abstand `diffx_p12` zwischen den beiden oberen Punktepaaren. Das Fahrzeugkoordinatensystem wird in x-Richtung in die Bildmitte gelegt.

Die Funktion `calibrateCamera()` liefert für die Fahrspurführung folgende drei Hilfswerkzeuge um die physikalisch reale Fahrbahn durch eine Kameraaufnahme abbilden zu können:

- Die Homographie-Matrix H zur Transformation eines Bildpunktes im verzerrten Schrägbild zu einem Objektpunkt im Kamerabild aus der Vogelperspektive auf die Fahrbahnebene.
- Das Verhältnis *pixel:cm* um einen Bildpunkt in einen realen Punkt auf der Fahrbahnebene in cm umzurechnen.

- Den Abstand der Ursprünge zwischen dem Bildkoordinatensystem und dem Fahrzeugkoordinatensystem, um zu einem Bildpunkt den Abstand in y-Richtung vom Fahrzeug in cm umrechnen zu können.

Diese Hilfsmittel werden von der Funktion `transformBetweenPixelAndCM()` im Modul `pipelineFunctions` benutzt, um Bildpunkte in reale Fahrbahnpunkte umzurechnen. Die Funktion bekommt folgende Parameter übergeben:

```
void transformBetweenPixelAndCM
(
    const CvPoint2D32f * src,  /* Vorhandener Quellpunkt    */
    CvPoint2D32f * dst,        /* Zu berechnender Zielpunkt */
    PipelineData * pd, /* Datenstruktur, die o.g. Hilfmittel
                          speichert */
    int distorted, /* Gibt an, ob uebergebener Bildpunkt
                      aus dem verzerrten Schraegbild oder aus dem
                      entzerrten Kamerabild aus der Vogelperspektive
                      entnommen ist. */
    int fromPixelToObj /* Gibt an, ob von Bildpunkt nach
                      Fahrbahnpunkt berechnet werden soll,
                      oder umgekehrt */
)
```

Listing 4.1: Projektive Transformation eines Punktes statt eines Bildes

Diese Funktion wird während der Fahrspurpipeline von der Pipeline-Funktion `dynamicROI()` aufgerufen, um einzelne Bildpunkte zu transformieren. Im Gegensatz zur FPGA-basierten SAV-Applikation wird die projektive Transformation nicht auf den gesamten Bildbereich angewendet, um somit Berechnungszyklen einzusparen. Aus diesem Grund wird die projektive Transformation in dieser Arbeit auch nicht als Pipeline-Funktion eingestuft, sondern lediglich als Hilfsfunktion, um einzelne, für die Fahrspurverfolgung relevante Punkte innerhalb der Pipeline zu transformieren.

4.2.2. Shared Memory Schnittstelle zur Hindernisserkennung

Unix Betriebssysteme stellen eine Reihe von Techniken [44] zur Programmierung von Inter-Prozess-Kommunikation (IPC)an. Über den Shared Memory Mechanismus greifen mehrere Prozesse mit eigenem Adressraum lesend oder schreibend auf einen gemeinsamen

Datenspeicher zu. Dieser wird vom Kernel verwaltet und ist die schnellste Form der IPC, da kein Kopieren von Erzeugern zu Verbrauchern nötig ist, weil der Speicherbereich gemeinsam verwendet wird. Im Gegensatz zu anderen IPC Mechanismen, wie z.B. Pipes, Message Queues oder Sockets, wird der Shared Memory nicht automatisch synchronisiert. Ein gleichzeitiges Schreiben durch mehrere Prozesse setzt eine selbstständige Synchronisationskontrolle voraus.

Die Fahrspurführung legt für die Kommunikation mit der Hinderniserkennung einen Shared Memory Bereich an, der systemweit über seinen Identifier String `"syncLaneSystem"` bekannt ist. Der folgende Quellcodeabschnitt beschreibt das Öffnen und Schreiben des Shared Memory Bereichs.

```
1  int shmFd = shm_open("syncLaneSystem",
2                       O_CREAT | O_RDWR | O_EXCL,
3                       (mode_t)0400);
4
5  ftruncate(shmFd, (off_t)sizeBytes);
6
7  char * shmPtr = (char *)mmap(NULL, sizeBytes,
8                               PROT_READ | PROT_WRITE,
9                               MAP_SHARED, shmFd, (off_t)0
10                             );
11
12 memcpy(shmPtr, data, sizeBytes);
```

Listing 4.2: Schreibvorgang in den gemeinsamen Speicher

Die Unix API Funktion `shm_open()` liefert einen Filedeskriptor, mit dem aus der Applikation auf den Bereich zugegriffen wird. Mit der API Funktion `mmap()` wird der gemeinsame Speicher in den Adressbereich der aufrufenden Applikation eingefügt. Die API Funktion `ftruncate()` legt die Größe des gemeinsamen Adressraums fest, um Speicherzugriffsfehler zu vermeiden. Das Schreiben in den gemeinsamen Speicher erfolgt durch `memcpy()`. Die Schnittstelle zwischen beiden Applikationen erfolgt dadurch, dass die Hinderniserkennung ein 4 Byte großes Wort an die erste Adresse des Speichers schreibt sobald ein Ausweichvorgang gestartet werden soll, während die Fahrspurführung den Speicherbereich zyklisch ausliest. Mit diesem Mechanismus durch nur einen Erzeuger und einen Verbraucher ist derzeit keine Synchronisation nötig.

4.2.3. Initialisierung des ROI

Die Suche nach den weißen Vordergrundpixeln der Fahrspur im Kamerabild wird auf einen Bildbereich *Region Of Interest* reduziert, welcher um die Fahrspur gelegt wird, um Schleifenzyklen zu vermeiden. Der ROI wird durch seinen Ursprung sowie ein konstantes Offset in x-Richtung, bzw. in y-Richtung definiert und spannt somit ein Viereck innerhalb des Kamerabildes auf (vgl. Abb. 4.8). Die y-Koordinate des ROI Ursprungs ist in cm durch den Konfigurationsparameter `roi_y_cm` definiert und wird in Pixel-Koordinaten transformiert. In der Funktion `initROI()` im Modul `pipelineFunctions` wird der ROI

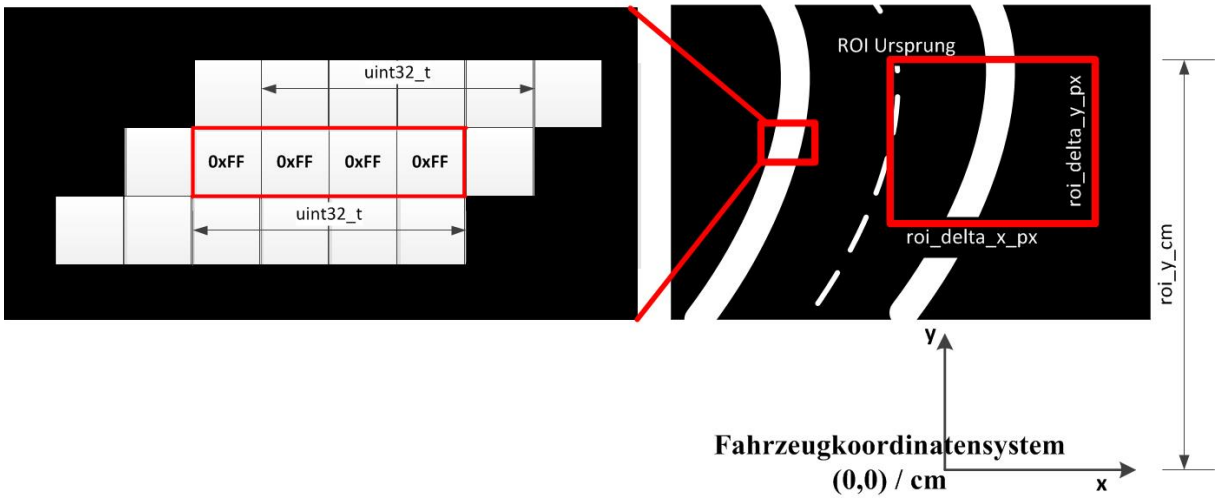

Abbildung 4.8.: Der ROI wird um die rechte Fahrbahnmarkierung gelegt, sodass der erste Block von vier weissen Vordergrundpixeln ein Offset in x-Richtung von der Hälfte der Kantenlänge `roi_delta_x_px` zum ROI Ursrprung erhält.

Ursprung in x-Richtung durch Korrektur der x-Koordinate initial an die Lage der Fahrspur angepasst. Über den Konfigurationsparameter `countLanesOnArea` ist die Anzahl der Fahrspurmarkierungen bekannt, wobei der ROI am Anfang standardmäßig um den rechten Fahrspurrand gelegt wird (vgl. Abb. 4.8). Das Finden der weißen Randmarkierungen im Kamerabild erfolgt durch eine Iteration über den Bildpuffer auf einer Pixelzeile (in x-Richtung), sowie auf der Höhe des ROI Ursprungs. Das OpenCV Bildformat `IplImage` enthält als Lastdaten einen festen Speicherbereich, welcher mit dem Kamerabild gefüllt wird, wobei pro Byte der Graustufenwert eines Pixels gespeichert wird, z.B. `0xFF` für Weiß und `0x00` für Schwarz. Eine pixelweise Iteration über den Speicherbereich entspricht somit einer Byte-weisen Iteration durch den Datentyp `uint8_t`. Um Schleifenzyklen zu

reduzieren, wird über eine Pixel-Zeile in 4-Byte-Blöcken durch den Datentyp `uint32_t` iteriert. Als 32-Bit Prozessor benötigt der ARM Cotex A9 Prozessor für eine 32-Bit Integer Operation eine Instruktion. Da es in ARM C Compilern effizienzsteigernde Speicherzugriffsmechanismen für Operationen auf 8-Bit oder 16-Bit Datenlängen durch 32-Bit Prozessoren gibt [48], werden durch die Vergrößerung der Indizierung auf vier Bytes zwar keine CPU-Instruktionen eingespart, dafür jedoch Schleifenzyklen während der Iteration über eine Pixelzeile.

Sobald ein Block von von vier benachbarten Bytes mit dem Graustufenwert `0xFFFFFFFF` für weiß gefunden wurde, wird der Adress–Offset (Index) im Speicherbereich des Kamerabildes in Pixel-Koordinaten als Punkt mit Weite x und Höhe y umgerechnet.

0 0/0	1 1/0	2 2/0	3 3/0	4 4/0	5 5/0	... ...	383 383/0
384 0/1	385 1/1	386 2/1	387 3/1	388 4/1	389 5/1	... ...	767 383/1
. .	. .	. .	. .	. .	. .	. .	. .
104064 0/271	104065 1/271	104066 2/271	104067 3/271	104068 4/271	104069 5/271	... ...	104447 383/271

Abbildung 4.9.: Das Kamerabild mit einer Auflösung von 384x272 Pixeln (Bytes) als Array von Graustufenwerten dargestellt. Jeder Address-Offset (Array-Index) kann zu Pixel-Koordinaten (Punkten) umgerechnet werden.

Ein Bildpunkt in Pixel-Koordinaten mit Weite x und Höhe y wird mit folgender Gleichung in den Index *i8* des Arrays der Größe *(xWidth x yHeight)* in Bytes umgerechnet:

$$i8 = y \cdot xWidth + x \tag{4.4}$$

Umgekehrt wird der Index *i8* folgendermaßen in Pixelkoordinaten umgerechnet:

$$x = i8 \quad \% \quad xWidth \tag{4.5}$$

$$y = \frac{i8}{xWidth} \tag{4.6}$$

Die in 4.6 errechnete y-Koordinate wird anschließend in einen Integer Datentyp konvertiert, um Nachkommastellen zu eliminieren. Die Umrechnung des Array-Index *i8* eines

Arrays vom Typ `uint8_t` in den Array-Index *i32* eines Arrays vom Typ `uint32_t` erfolgt in:

$$i32 = \frac{i8 \quad \% \quad 4}{4} \tag{4.7}$$

$$i8 = \{i32 \cdot 4 + 0, \quad i32 \cdot 4 + 1, \quad i32 \cdot 4 + 2, \quad i32 \cdot 4 + 3\} \tag{4.8}$$

Nachdem in der Pixelzeile vier benachbarte weiße Pixel einer Fahrbahnmarkierung gefunden wurden, wird aus dem zugehörigen Index *i32* durch (4.8) der Byte-Index *i8* berechnet. Daraus werden mit (4.5) und (4.6) die Pixelkoordinaten des zugehörigen Bildpunktes berechnet, und dieser in einem Array gespeichert. Die x-Koordinate des ROI Ursprungs ergibt sich, indem von der x-Koordinate des ermittelten Bildpunktes die Hälfte des ROI Bereichs `roi_delta_x_px` abgezogen wird. Somit liegt eine Fahrspurmarkierung in der Mitte der oberen ROI Kante. Liefert die Shared Memory Schnittstelle durch Aufrufen der Funktion `shm_ObjectDeteced()` das Signal zum Fahrspurwechsel auf die linke Fahrbahn, so wird die x-Koordinate des ROI Ursprungs aus dem zuerst gefundenen Bildpunkt berechnet, welcher an der ersten Stelle im Array gespeichert wird. Somit wird der ROI Bereich um die linke Fahrspurmarkierung gelegt. Falls kein Hindernis erkannt wurde wird der zuletzt gefundene Bildpunkt im Array verwendet, welcher die rechte Fahrbahnmarkierung repräsentiert.

4.3. Pipelinefunktionen zur Fahrspurverfolgung

Die Pipeline-Funktionen sind entweder durch OpenCV oder im Modul `pipelineFunctions` implementiert und werden aus einer Thread-Funktion heraus aufgerufen. Die Verarbeitung des Kameradatenstroms ist im Modul `cameraInterface` implementiert, um alle kameraspezifischen Programmabschnitte inklusive der Schnittstelle zum GStreamer-Framework zu kapseln. Aus Gründen der Übersicht und Wartbarkeit wird jede selbst implementierte Pipeline-Funktion als logisch zusammenhängende Sequenz von Aktionsschritten innerhalb ihrer eigenen C Funktion vollständig implementiert (vgl. Tab.4.1). Somit erfolgt das Ausführen einer Pipeline-Funktion innerhalb eines Threads durch einen einzigen Funktionsaufruf. In diesem Abschnitt wird der Zweck jeder Pipeline-Funktion zur Fahrspurverfolgung sowie die auszuführenden Arbeitsschritte erläutert.

Pipeline-Funktion	C Funktion	Modul
Verarbeitung des Kameradatenstroms	`imaging()`	`cameraInterface`
Binarisierung des Fahrspurbildes	`cvErode()`	`OpenCV`
Erosionsfilter	`cvErode()`	`OpenCV`
Dynamic ROI	`dynamicROI()`	`pipelineFunctions`
Hough Transformation	`houghTransformation()`	`pipelineFunctions`
Fahrspurführung	`laneFollowing()`	`pipelineFunctions`

Tabelle 4.1.: Übersicht zur Zuordnung von Pipeline- und C-Funktion

4.3.1. Verarbeitung des Kameradatenstroms

Diese Pipeline-Funktion realisiert die Aufteilung des MPEG-4 Kamerastroms in einzelne Bilder. Die Verwaltung des Kamerastroms erfolgt durch Funktionen und Datenstrukturen des GStreamer-Frameworks. Die GStreamer-Funktion `gst_app_sink_pull_buffer()` liefert einen Pointer auf den vom Kamerabild gefüllten Speicherbereich. Hierzu stellt die GStreamer-Funktion `gst_caps_get_structure()` die zugehörigen Kopfdaten wie z.B. Höhe und Weite des Kamerabildes bereit, womit die Größe des Kamerabildes bestimmt wird. Die Höhe und Breite entsprechen der Kameraauflösung, welche zwar schon bei Programmstart durch die Konfigurationsparameter `cameraFrameWidth` und `cameraFrameHeight` bekannt ist, aber zur Sicherheit nochmals mit den von GStreamer gelieferten Werten verglichen wird. Das Kopieren der Bilddaten ist in der folgenden

Quellcodesequenz implementiert. Es wird über die Höhe und Weite des Kamerapuffers iteriert und der Inhalt in die OpenCV-Datenstruktur vom Typ `IplImage` kopiert. Ein Pointer auf diesen zweiten Speicherbereich wird in einer Datenstruktur vom Typ `Lane` gespeichert.

```
1  for(y=0; y < height; y++)
2  {
3    const uint8_t *src = buf->data + iBeginFrame*width;
4    char *dst = (*frame)->imageData + y*(*frame)->widthStep;
5
6    for(x=0; x<width; x+=8)
7    {
8      *(uint64_t *)(dst+x) = *(uint64_t *)(src+x);
9    }
10 }
```

Listing 4.3: Übergang von GStreamer Videostream zu OpenCV Image

Um die Anzahl der Schleifeniterationen minimal zu halten, wird in Richtung der Bildweite mit dem größtmöglichen primitive Datentyp in Linux von 8 Bytes (`uint64_t`) iteriert und somit jeweils 8 Bytes kopiert.

4.3.2. Binarisierung des Fahrspurbildes

Die erste Pipeline-Funktion (`imaging()`) liefert ein `IplImage`-Frame als Momentaufnahme der Fahrspur. Das Bild wurde innerhalb der GStreamer-Pipeline bereits in Graustufen skaliert (vgl. Abbildung 4.10, links), wobei es stets einen deutlichen Kontrast zwischen den eher hellen Fahrspurmarkierungen und der eher dunklen Fahrbahn gibt. Diese Pipeline-Funktion binarisiert nun das Graustufenbild und hebt somit die weißen Fahrspurmarkierungen auf schwarzer Fahrbahn ab. Für die folgenden bildverarbeitenden Pipeline-Funktionen ermöglicht die Binarisierung des Fahrspurbildes eine einfachere Verarbeitung, da es kein differenziertes Graustufenspektrum zu beachten gibt sondern stattdessen pro Pixel nur zwei Werte in Frage kommen, schwarz (0) oder weiß (255). Dies wird insbesondere bei der Überprüfung von Bereichen ausgenutzt, indem beispielsweise 4 benachbarte schwarze Pixel zusammen auf den Wert 0 liefern.

Für die Binarisierung eines Bildes bietet die OpenCV Library die Funktion `cvThreshold()` an. Ihr Aufruf erfolgt im Anschluss an die Funktion `imaging()`.

```
cvThreshold(
    laneRunner->frame,
    laneRunner->frame,
    thresholdCV,
    255,
    CV_THRESH_BINARY);
```

Listing 4.4: Binarisierung der Fahrspur durch OpenCV

Die Funktion bekommt das aktuelle `frame` sowohl als Quelle und Ziel übergeben, sowie den Schwellwert und den Maximalwert von 255 für die Binarisierung. Der Schwellwert ist im Konfigurationsparameter `threshold` gespeichert.

Ein niedriger Schwellwert von 127 (Mitte) verursacht eine höhere Anfälligkeit für

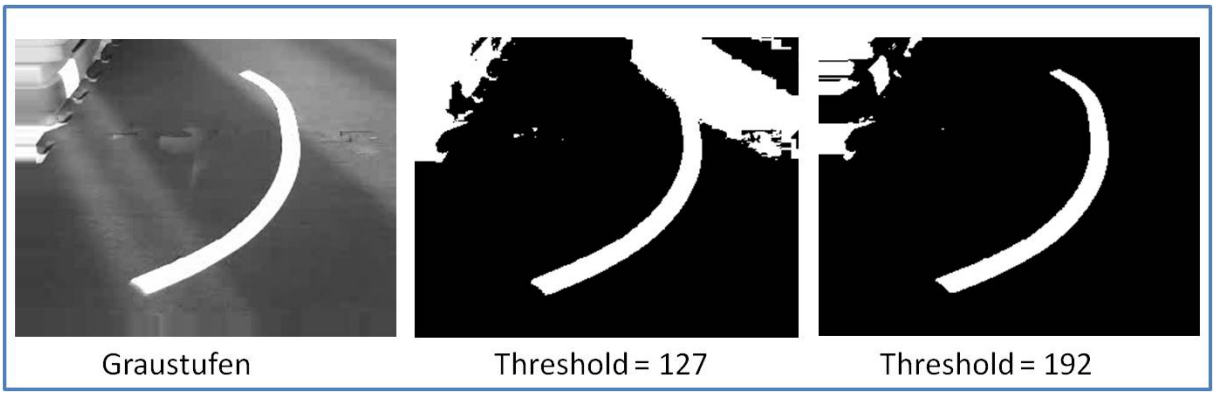

Abbildung 4.10.: Graustufenbild und unterschiedliche Schwellwerte

Verfälschungen durch störende Lichtverhältnisse, wie z.B. einfallende Sonnenstrahlen im linken Bild. Ein Schwellwert von 192 (rechts) reichte in diesem Lichtverhältnis aus, um die Störungen durch die Sonnenstrahlen zu eliminieren.

4.3.3. Erosionsfilter

Nachdem ein Bild in der zweiten Pipeline-Funktion binarisiert wurde kann die Pixelmenge in exakt zwei Teilmengen differenziert werden, die weißen Vordergrund-Pixel der Fahrspur zum einen und die schwarzen Hintergrund-Pixel des Fahrbahnuntergrunds zum anderen. Welche Teilmenge hierbei als Vorder- bzw. Hintergrundmenge bezeichnet wird,

entscheidet der prozentuale Anteil, wobei die Teilmenge mit der geringeren Anzahl als Vordergrundmenge bezeichnet wird. Die Pipeline-Funktion *Erosion* erfüllt die Aufgabe, die Teilmenge der weißen Vordergrund-Pixel im Bild zu reduzieren um die weiße Kante der Fahrspur zu schärfen, damit sie durch die Hough Transformation als geometrisches Objekt eindeutiger identifiziert werden kann, (vgl. Abschnitt 4.3.5). Die Erosion gehört zu den Basisoperationen innerhalb der morphologischen Bildverarbeitung [26]. Im Allgemeinen handelt es sich hierbei um die Analyse eines Bildes auf Formen und Strukturen durch verschiedene Mengenoperationen. In diesem Fall ist die Mengenoperation eine Faltung des Originalbildes als Pixelmenge A mit einer zweiten Pixelmenge B, die als kleinstes Strukturelement der zu suchenden Form im Bild interpretiert werden kann.

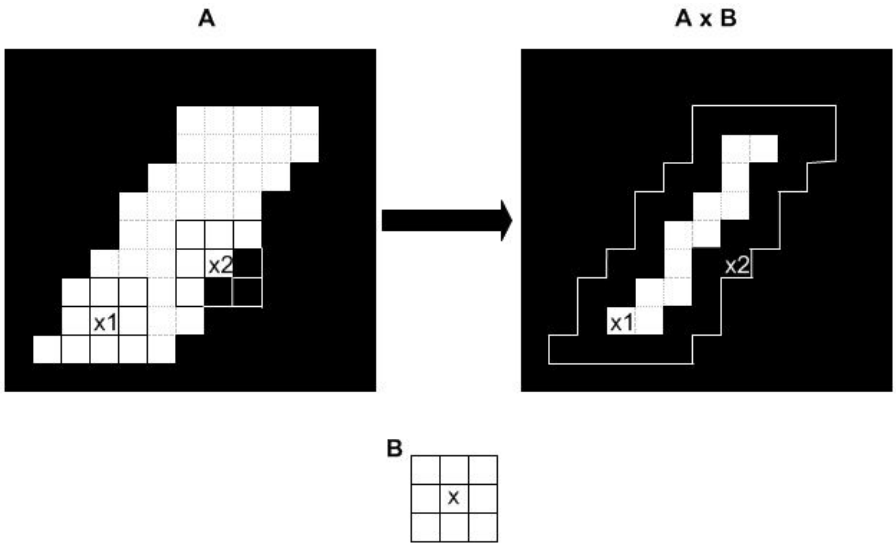

Abbildung 4.11.: Erosion als Faltung mit Strukturelement B

Für das Identifizieren der Fahrspur wurde als Strukturelement eine Menge B von 9 Pixeln in Form einer 3x3-Matrix mit einem Bezugspunkt in der Mitte gewählt, wie sie in Abbildung 4.11 unten dargestellt ist. Das linke Bild zeigt einen Ausschnitt eines Fahrspurbildes als Pixelmenge A. Die Ergebnismenge der Erosion $A\ x\ B$ (rechts) entsteht durch die Faltung der Menge A mit dem Strukturelement B, indem der Bezugspunkt in B auf jedes einzelne Pixel in der Menge A gelegt wird. Hinter der Faltungsoperation verbirgt sich die Überprüfung durch logische Vergleichsoperationen, ob das gesamte Strukturelement B auf einer Pixelposition in die Vordergrundmenge der weißen Fahrspur-Pixel passt. Ist dies wie in Abbildung 4.11 bei Pixel `x1` der Fall, ist das Ergebnis für `x1` weiterhin ein weißes Vordergrundpixel. Liegt das Strukturelement B auf Pixel `x2`, so passt die Menge

B nicht vollständig in die weiße Fläche, und das Ergebnis für Pixel `x2` ist ein schwarzes Hintergrundpixel.

Die Erosion erfüllt den Zweck, dass die Menge der Vordergrundpixel reduziert wird und die zu suchende Form geschärft wird. Daneben dient es auch dazu, einzelne verirrte Vordergrundpixel (Rauschen) zu eliminieren. Der Rechenaufwand für die Erosion hängt damit von der Pixelauflösung des Bildes ab, da jedes Pixel gefaltet wird.

OpenCV bietet für eine Reihe morphologischer Operationen, darunter auch die Erosion, entsprechende Funktionen an [14]. Die Pipeline-Funktion *Erosion* wird von der Funktion `cvErode()` aus der OpenCV Library ausgeführt und folgendermaßen aus dem Thread `threadImageProcessing()` aufgerufen:

```
1 cvErode(
2     lane->frame,       /* IplImage * src    */
3     lane->frame,       /* IplImage * dst    */
4     NULL,                    /* IplConvKernel * B */
5     1                        /* int iterations    */
6 );
```

Listing 4.5: Erosion der Fahrspur durch OpenCV

Die Funktion erwartet im ersten Eingabeparameter das binarisierte Originalbild der Fahrspur als Pixelmenge A und im zweiten Eingabeparameter ein Zielbild, in das die Ergebnismenge der Erosion $A\ x\ B$ gespeichert wird. Der dritte Eingabeparameter ist das Strukturelement B. Ist dieses nicht definiert (`NULL`) wird hierfür die in Abbildung 4.11 dargestellte 3x3-Matrix und dem Bezugspunkt in der Mitte verwendet. Der vierte Eingabeparameter legt die Anzahl fest, wie oft die Erosion auf das Bild angewendet werden soll. Da sich bei jedem Durchgang die Anzahl der Vordergrundpixel reduziert, bedeutet dies im Falle der Fahrspurführung, dass die Fahrspur immer schmaler wird, wie Abbildung 4.12 zeigt.

4.3.4. Dynamic ROI

Die Pipeline Funktion `dynamicROI()` im Modul `pipelineFunctions` erfüllt zwei Hauptaufgaben. Zum einen werden innerhalb des ROI weiße Bildpunkte der Fahrspurmarkierung

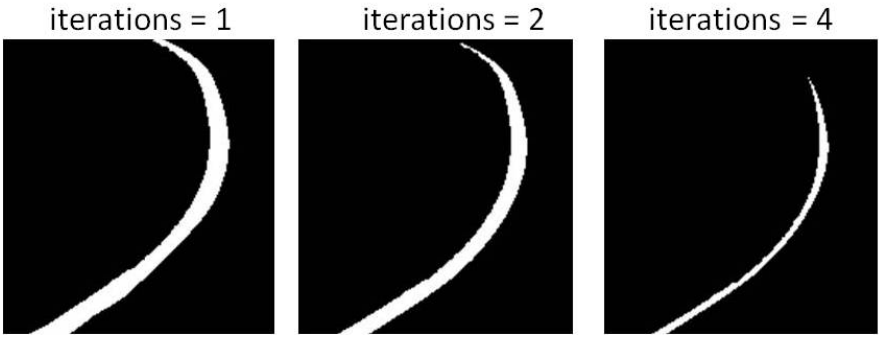

Abbildung 4.12.: Ergebnisbild nach der Erosion mit 1, 2 und 4 Durchgängen

identifiziert und gespeichert, zum anderen wird der Ursprung des ROI Bereichs in x-Richtung an die veränderte Lage der Fahrspurlinie angepasst. Dies erfolgt nach demselben Verfahren wie in der Funktion `initROI()`, indem der oberste gefundene Bildpunkt der weißen Markierung, welcher auf der Höhe der oberen ROI Kante liegt, in deren Mitte liegt. Damit wird der Wertebereich des Radius für die Hough Transformation (vgl. Kap. 4.3.5) eingeschränkt, was wiederum Schleifendurchgänge und damit Rechenzeit reduziert. Das Ergebnis der Funktion `dynamicROI()` für die weiteren Berechnungsschritte der Fahrspur-Pipeline ist ein Array von weißen Bildpunkten, durch die die Fahrspur innerhalb des ROI Bereichs verläuft. Dazu werden folgende Arbeitsschritte ausgeführt:

1. Überprüfung, ob ein Fahrspurwechsel stattfinden soll durch Aufrufen der Funktion `shm_ObjectDetected()`, vgl. Kap 4.2.2.

2. Liefert die Shared Memory Schnittstelle einen Fahrspurwechsel, wird der ROI Bereich neu initiiert durch Aufrufen der Funktion `initROI` (vgl. Kap. 4.2.3). Dadurch wird der ROI um die betreffende Fahrbahnmarkierung gelegt, zu welcher der zu berechnende Lenkwinkel führen soll.

3. Ermittlung des Vorgängers der aktuellen Datenstruktur vom Typ `Lane` sowie dessen ROI. Handelt es sich um das erste Element und wird die Pipeline-Funktion zum ersten mal ausgeführt, so ist ebenfalls im ersten Element er initial erzeugte ROI aus der Funktion `initROI()` gespeichert.

4. Iteration durch den Bilddatenspeicher innerhalb des ROI Bereichs des Vorgängers: In y-Richtung wird über einen Zeilenabstand iteriert, welcher über den Konfigurationsparameter `distRowsRoi` festgelegt wird. Dadurch wird die Anzahl Pixelzeilen in y-Richtung und somit die Anzahl der Schleifendurchgänge beeinflusst.

5. Pro festgelegt Pixelzeile wird in x-Richtung innerhalb des ROI über einen Block von vier Pixeln iteriert, welcher über den Datentyp `uint32_t` angesprochen wird. Diese Iteration in x-Richtung erfolgt analog zur Iteration innerhalb der Funktion `initROI()`, vgl. Abb. 4.8.

6. Der Index zum gefundenen Block von vier weißen Pixeln wird in einen Bildpunkt in Pixelkoordinaten umgerechnet und im Array `pointsWhiteLane_px[]` innerhalb der aktuellen `Lane` Struktur gespeichert.

7. Nach der Iteration in y-Richtung sind die gefundenen weißen Bildpunkte pro iterierte Pixelzeile im Array gespeichert. Jeder Bildpunkt wird in einen Objektpunkt auf der Fahrbahnebene in cm durch die Funktion `transformBetweenPixelAndCM()` perspektivisch transformiert.

8. Das erste Element im Array der gefundenen weißen Bildpunkte liegt auf der oberen Kante des ROI, da dessen y-Wert und der y-Wert des ROI Ursprungs identisch sind. Der ROI Ursprung des aktuellen `Lane` Elements wird in x-Richtung wieder korrigiert, indem der oberste Bildpunkt die obere ROI Kante wieder in der Mitte teilt.

Diese Pipeline-Funktion markiert den Übergang vom Bildbereich in den physikalisch realen Bereich des Fahrspurkoordinatensystems, indem sie eine Anzahl an Punkten in cm aus den Bilddaten extrahiert, durch die die weiße Fahrbahnmarkierung verläuft. Bisher führten alle vorangegangenen Pipeline-Funktionen Operationen auf den Bilddatenspeicher aus. Die Hough Transformation und die Fahrspurführungsalgorithmen berechnen dagegen ausschließlich auf Punkte und Abstände im Fahrzeugkoordinatensystem. In diesen Übergang fließt auch die perspektivische Transformation mit ein, jedoch nur von einzelnen Bildpunkten statt des gesamten Bildbereichs. Zuerst werden die weißen Bildpunkte im perspektivisch verzerrten Schrägbild gesucht und anschließend in Fahrzeugkoordinaten transformiert.

Die Funktion `dynamicROIparallel()` beinhaltet dieselbe Implementierung wie `dynamicROI()`. Hinzu kommt die Parallelisierung der Schleifeniteration in y-Richtung durch die folgende OpenMP Direktive:

```
1 #pragma omp parallel default(none) \
2 private(j, iStart32, iStop32, i32, prevIndex32, pixelrow) \
3 shared(iIndexFoundWhiteLanes, pd, iLaneIndex_ip,
4 xVorgaenger, yVorgaenger) \
```

```
5  num_threads(2)
6  for(j=0; j< roi_delta_y_px; j=j+distRowsRoi)
7  {
8    iStart32 = /* Startpunkt aus ROI und j berechnen */
9    iStop32 = /* Stoppunkt aus ROI und j berechnen */
10   // in x-Richtung innerhalb ROI iterieren
11   for(i32 = iStart32; i32 <= iStop32; i32++)
12   {
13     // 4 weisse Pixel hintereinander gefunden
14       if(pixelrow[i32] == 0xffffffff)
15     { /* ... */
16       // Kritischen Bereich betreten
17       #pragma omp critical
18       {
19         /* gefundenen Punkt speichern */
20       }
21     }
22   }
```

Listing 4.6: OpenMP Direktive für dynamicROI() zur Parallelisierung der Iteration in y-Richtung

Innerhalb der `#pragma` Anweisung werden alle im parallelisierten Block verwendeten Variablen entweder als private oder geteilte Variablen festgelegt. Durch `num_threads()` wird die Anzahl an parallelen Threads festgelegt, die der Anzahl der CPUs entspricht. Die privaten Felder werden in den Stack der Threads kopiert. Sie sind jeweils nur im eigenen Thread sichtbar und werden zwischen den Threads nicht synchronisiert. Beispielsweise wird der Iterationsbereich der Laufvariablen `j` durch OpenMP auf beide Threads verteilt, sodass jeder Thread die Hälfte der Schleifendurchgänge durchführt. Sie muss während der Ausführung nicht zwischen den Threads synchronisiert werden, ebenso die Variablen `iStart32` und `iStop32` für den Start und Stop der Iteration in x-Richtung, die aus dem ROI in Abhängigkeit der Zeilennummer durch `j` berechnet werden. Die geteilten Variablen werden von allen Thread gemeinsam genutzt und durch OpenMP nicht synchronisiert. Bei gleichzeitigem Schreiben derselben Variablen aus verschiedenen Threads wird die Konsistenz durch die `#pragma` Anweisung `critical` sichergestellt, die um den Schreibzugriff gelegt wird. Jeder gefundene weiße Bildpunkt wird in Form seines Index des Bildbereichs von beiden parallelen Threads in ein gemeinsames Array

gespeichert. Der Laufindex `iIndexFoundWhiteLanes` wird während des Schreibvorgangs in das Array von beiden Threads inkrementiert und muss somit synchronisiert werden.

4.3.5. Hough Transformation

Das geometrische Referenzobjekt der Fahrbahnmarkierung ist eine Gerade. Nachdem in der vorangegangenen Pipeline-Funktion eine Reihe von weißen Bildpunkten ermittelt wurden, welche auf der Fahrbahnmarkierung liegen, wird nun nach dem Verfahren der Hough Transformation eine Gerade ermittelt, auf der die meisten gefundenen Punkte liegen. Die Hough Transformation ist ein Verfahren zum Erkennen von geometrischen Objekten anhand von Referenzobjekten innerhalb eines Bildes. Mit der Verwendung von Geraden als Referenzobjekte eignet sich die Hough Transformation zur Kantendetektion in Bildern. Durch Binarisierung der Bilddaten werden Kanten bereits hervorgehoben. was der Hough Transformation bei der Kantendetektion die Berücksichtigung auf eventuelle Schwellwerte beim Verfolgen eines Kantenübergangs von verschiedenen Graustufen erspart. In die Hough Transformation lassen sich beliebige geometrische Objekte Referenzobjekte in beliebigen Darstellungen integrieren, wobei das Verfahren für alle geometrischen Darstellungen gleich ist. Eine Gerade kann z.B. durch folgende Geradengleichung mit der Steigung m und dem Achsenabschnitt b eindeutig dargestellt werden:

$$y = m \cdot x + b \tag{4.9}$$

Jedoch eignet sich diese Darstellungsform nur bedingt für die Fahrspurerkennung, da deren Steigung m im Falle einer geradeaus verlaufenden Fahrspur gegen Unendlich konvergiert. Stattdessen wird die Gerade in der Hesseschen Normalform durch die Ursprungslotlänge r zum ROI Ursprung und den Winkel ϕ zwischen dem Urprungslot und der oberen ROI Kante (in x-Richtung) dargestellt:

$$r = x \cdot \cos(\phi) + y \cdot \sin(\phi) \tag{4.10}$$

Ein Punkt mit den Koordinaten x und y entspricht in der Hough Ebene einer sinusoidalen Funktion (vgl. Abb. 4.13, rechts). Diese wird gebildet, indem die Geradenparameter r und ϕ aller Geraden durch die Hessesche Normalform ermittelt werden, die durch den Punkt verlaufen. Eine Übertragung von weiteren Punkten erzeugt in der Hough Ebene weitere sinusoidale Funktionen. Eine Gerade, die durch zwei Punkte verläuft,

entspricht in der Hough Ebene dem Schnittpunkt der Sinusoiden der beiden Punkte. Die Gerade, auf der die meisten weißen Punkte der Fahrspur liegen entspricht in der Hough Ebene dem Schnittpunkt, durch den die meisten Sinusiode verlaufen. Die Anzahl

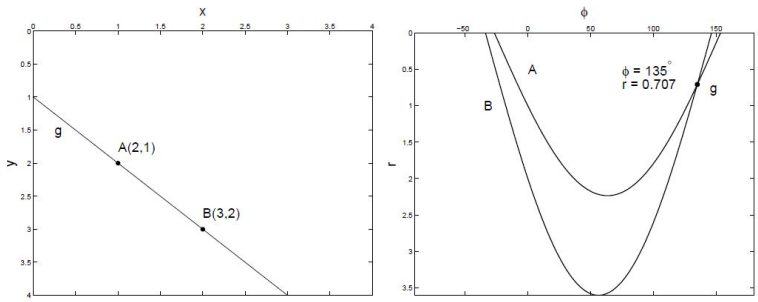

Abbildung 4.13.: Ein Bildpunkt A entspricht einer sinusoidalen Funktion in der Hough Ebene. Eine Gerade g, die durch die Bildpunkte A und B verläuft, entspricht dem Schnittpunkt der beiden Funktionen von A und B

der Berechnungsschritte durch die Hough Transformation wird maßgeblich durch die Einschränkung der beiden Wertemengen von r und ϕ beeinflusst. Um den Rechenaufwand zu begrenzen wird der Winkel $\phi_i \in \{\phi_{min}, \phi_{max}\}$, mit $i \in Z$ auf das Intervall zwischen -45° und +45° begrenzt. Zusätzlich wird die Winkelschrittweite $\Delta\phi$ nach [21] auf 2° gesetzt.

Der erste gefundene weiße Punkt auf der Fahrbahn befindet sich auf der oberen Kante des ROI in x-Richtung (`roi_delta_x_px`. Damit ist die Menge aller möglichen Fahrspurgeraden durch die Grenzwerte ϕ_{min}, ϕ_{max} und der Winkelschrittweite $\Delta\phi$ eindeutig definiert, vgl. Abb. 4.16. Mit dem Winkel $\phi = 0°$ zur x-Achse ergibt sich somit der obere Grenzwert r_{max} der Ursprungslotlänge als Hälfte des der oberen ROI-Kante in x-Richtung:

$$r_{max} = \frac{\texttt{roi\_delta\_x\_px}}{2} \tag{4.11}$$

Mit einem maximalen Winkel $\phi = 45°$ spannt die die minimale Ursprungslotlänge r_{min} zusammen mit der zugehörigen Geraden und den ROI Kanten ein rechtwinkliges, gleichschenkliges Dreieck auf, wodurch sich die minimale Ursprungslotlänge r_{min} mit Hilfe des Satzes von Phytagoras ergibt durch:

$$r_{min} = \frac{\texttt{roi\_delta\_x\_px}}{\sqrt{2}} = \frac{r_{max}}{\sqrt{2}} \tag{4.12}$$

Aufgrund der Achsensymmetrie zur oberen ROI Kante in x-Richtung ergibt sich für den minimalen Winkel ϕ = -45° dieselbe minimale Ursprungslotlänge r_{min}.

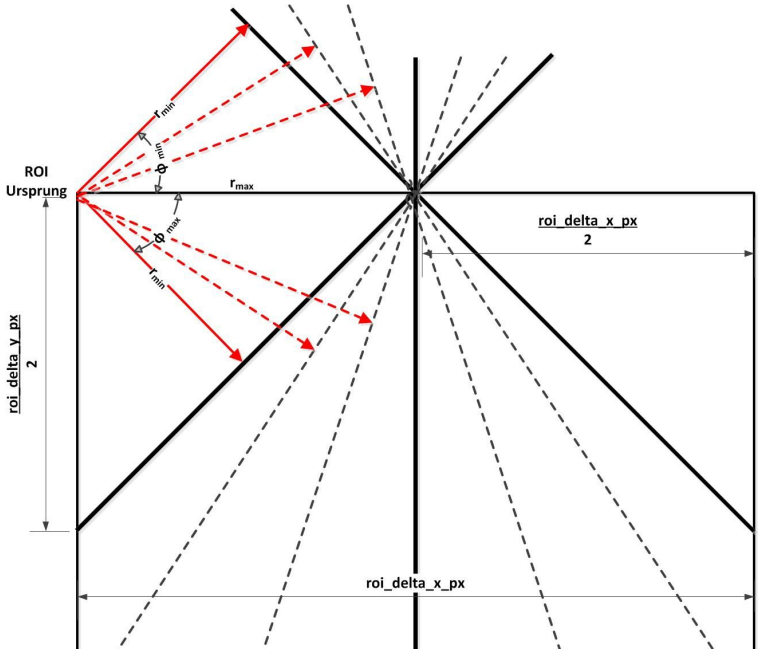

Abbildung 4.14.: Die Gerade der Fahrbahnmarkierung verläuft durch die Mitte der oberen ROI Kante. Daraus wird die Wertemenge von r ermittelt.

Die Grenzwerte für r und ϕ hängen also nur von einem Konfigurationsparameter ab und werden bei Programmstart einmalig initialisiert. Die Hough Ebene ist in der Fahrspurführung durch ein eindimensionales Array implementiert, dessen Größe ebenfalls durch die Grenzwerte aufgespannt wird:

$$sizeOfHoughPlane = \frac{2 \cdot \phi_{max}}{\Delta\phi} \cdot (r_{max} - r_{min} + 1) \tag{4.13}$$

Die Navigation innerhalb des Arrays wird durch eine Zuordnung zwischen dem Adressoffset (Index) und den zugehörigen Geradenparametern r_i und ϕ_i in beiden Richtungen realisiert.
Der Array-Index wird aus den Koordinatenpaar (r_i/ϕ_i) berechnet durch:

$$i = \frac{\phi_i - \phi_{min}}{\Delta\phi} \cdot (r_{max} - r_{min} + 1) + r_i - r_{min} \tag{4.14}$$

Der Geradenparameter r_i wird aus dem zugehörigen Array Index i wie folgt ermittelt:

$$r_i = i \ \% \ (r_{max} - r_{min} + 1) + r_{min} \tag{4.15}$$

ϕ	r_{min}	r_{min} +1	...	r_{max}
-45 °	_	_	...	_
-43 °	_	_	...	_
:	:	:	:	:
+45 °	_	_	...	_

Abbildung 4.15.: Aufspannen der Hough Ebene als eindimensionales Array durch die Geradenparameter r und ϕ.

Aus demselben Array-Index i wird die zweite Koordinate ϕ_i berechnet durch:

$$\phi_i = \phi_{min} + \frac{i \cdot \Delta\phi}{r_{max} - r_{min} + 1} \tag{4.16}$$

Die Hough Transformation wird in der Fahrspurführung durch die Funktion `houghTransformation()` im Modul `pipelineFunctions` implementiert. Es werden alle von der vorangegangenen Funktion `dynamicROI()` extrahierten Punkte der weißen Fahrbahn innerhalb des ROI in die Hough Ebene abgebildet. Die Funktion liefert als Ergebnis die ermittelten Geradenparameter r und ϕ des Schnittpunktmaximums. Diese dienen als Berechnungsgrundlage für die nachfolgenden Fahrspurführungsalgorithmen (vgl. Kap. 4.3.6). Dazu werden folgende Arbeitsschritte ausgeführt:

1. Rücksetzen jedes Elements des Arrays, welches die Hough Ebene umspannt, auf den Wert Null.
2. Neuberechnung der Koordinaten aller gefundenen Fahrbahnpunkte als Abstand relativ zum ROI Ursprung.
3. Iteration über alle gefundenen Fahrbahnpunkte
4. Iteration pro Punkt über den Wertebereich des Winkels ϕ
5. Pro Winkel wird der zugehörige Wert für r nach der Hesseschen Normalform in Gl. (4.10) berechnet.
6. Berechnung des Array-Index i aus den aktuellen Werten für r und ϕ durch Gl. (4.14).
7. Inkrementieren des Wertes des Elements im Array an der errechneten Position i.
8. Vergleich des neuen Wertes an der errechneten Position mit dem temporären Maximum.

Die Array-Position des Elements mit dem höchsten Wert entspricht der Geraden, die durch die meisten gefundenen Punkte führt. Zu dieser Position im Array werden die Geradenparameter r und ϕ nach den Gleichungen (4.15) und (4.16) berechnet. Tritt der Fall auf, dass das Schnittpunktmaximum an mehreren Positionen vorkommt, bedeutet dies, dass unterschiedliche Geraden durch die gleiche Anzahl an gefundenen Punkten gelegt werden können. Hierbei werden zu allen relevanten Positionen ebenfalls die Geradenparameter ermittelt und der Durchschnitt aus den unterschiedlichen Werten berechnet.

Analog zur Parallelisierung der Funktion `dynamicROI()` implementiert die Funktion `houghTransformationParallel()` die OpenMP Direktive zur Parallelisierung der Iteration über die gefundenen weißen Punkte, die in die Hough-Ebene abbgebildet werden:

```
#pragma omp parallel default(none) \
  private(iPhi_runner, iR_runner, i) \
  shared(iValue_max, iIndex_r_phi pd, lane) \
  num_threads(2)
  {
    iValue_max = 1;

    #pragma omp for
    // ueber die gefundenen weissen Punkte iterieren
    for(i = 0; i < lane->iCountWhiteHits; i++)
    {
      for( /* ueber phi iterieren */ )
      {
        /* r aus phi berechnen */

        #pragma omp critical
        {
          // Geraden-Punkt in Hough-Ebene erhoehen
          pd->aiHoughPlane[iIndex_r_phi] ++;
        }
      }
    }
```

Listing 4.7: OpenMP Direktive für die Hough Transformation zur Parallelisierung der Iteration über gefundenen weißen Punkte

Die Laufvariable `i` wird durch OpenMP auf die zwei Threads verteilt und ist somit ebenso privat wie z.B. der aktuell berechnete Radius `iR_runner`. Beide Threads schreiben auf dasselbe Array, welches die Hough-Ebene verwaltet. Der Laufindex `iIndex_r_phi` für dieses Array wird von beiden Threads geteilt. Ein Schreibvorgang in eine Stelle des Arrays über den Laufindex wird ebenfalls durch einen kritischen Abschnitt synchronisiert.

4.3.6. Fahrspurführungsalgorithmen

Für die Berechnung des Lenkwinkels α wird ein zielpunktbasierter Ansatz gewählt, indem ein Ansteuerungspunkt (*Look Ahead Point* - LAP) anhand der Lage der Fahrspur relativ zum Fahrzeug berechnet wird, auf den der Lenkwinkel das Fahrzeug zuführen soll. Die Berechnungsgrundlage für das Ansteuern des LAP durch den Lenkwinkel α sind die von der Hough Transformation berechneten Geradenparameter r und ϕ der als Gerade approximierten Fahrspur im ROI. Die Geradenparameter beziehen sich auf den Ursprung des ROI. Dieser entscheidet, zu welcher Fahrbahnmarkierung der LAP ausgerichtet werden soll. Der ROI wird in der Pipeline-Funktion `dynamicROI()` festgelegt. Über die Shared Memory Schnittstelle wird definiert, um welche Fahrspur der ROI gelegt wird. Dementsprechend wird in der Fahrspurführung entweder der linke oder der rechte LAP berechnet. Der Abstand d des LAP zur Fahrbahnmarkierung ist über den Konfigurationsparameter `offset_d` vorgegeben, ebenso die die *Look Ahead Distance* - LAD über den Konfigurationsparameter `lad_cm`. Die LAD ist der Abstand des LAP zum Ursprung der Fahrzeugkoordinatensystems.

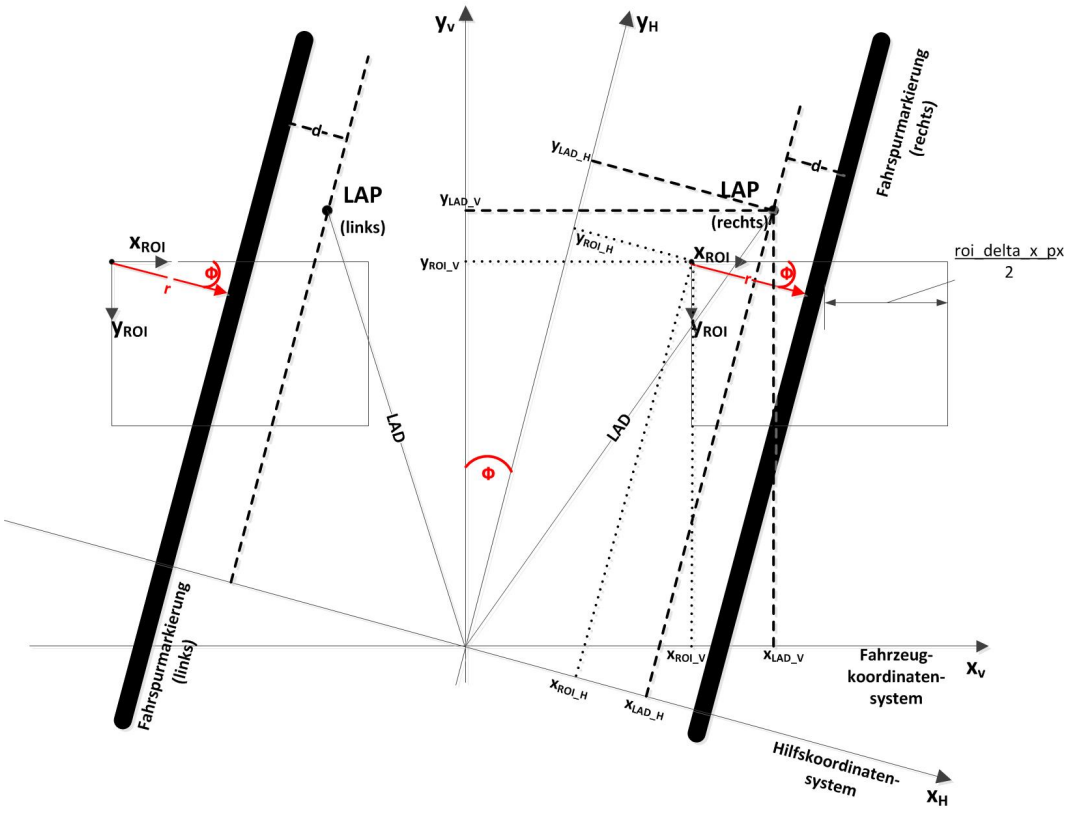

Abbildung 4.16.: Der ROI Ursprung entscheidet, zu welcher Fahrbahnmarkierung der LAP ausgerichtet ist. Der Abstand d des LAP zur Markierung und die LAD sind vorgegeben.

Die Berechnung des LAP erfolgt in folgenden Schritten:

1. Transformation des ROI Ursprungs von Fahrzeugkoordinaten (x_V, y_V) in ein Hilfskoordinatensystem (x_H, y_H) durch Rotation um den Winkel ϕ um den Koordinatenursprung:

$$\begin{pmatrix} x_{ROI_H} \\ y_{ROI_H} \end{pmatrix} = \begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix} \cdot \begin{pmatrix} x_{ROI_V} \\ y_{ROI_V} \end{pmatrix} \tag{4.17}$$

2. Berechnung der x-Koordinate des LAP im Hilfskoordinatensystem in Abhängigkeit zur Ursprungslotgeraden r, dem Abstand d und der x-Koordinate des ROI Ursprungs im Hilfskoordinatensystem. Wird die rechte Fahrspur angesteuert, befindet sich der LAP zu ihrer linken. Somit wird d von r subtrahiert:

$$x_{LAP_H} = x_{ROI_H} + r - d \tag{4.18}$$

Analog wird der Abstand d zum LAP, der sich rechts von der linken Fahrspur befindet, zu r addiert:

$$x_{LAP_H} = x_{ROI_H} + r + d \tag{4.19}$$

3. Berechnung der y-Koordinate des LAP im Hilfskoordinatensystem in Abhängigkeit der LAD und seiner zuvor berechneten x-Koordinaten:

$$y_{LAP_H} = \sqrt{LAD^2 - x_{LAP_H}^2} \tag{4.20}$$

4. Rücktransformation des LAP vom Hilfskoordinatensystem in das Fahrzeugkoordinatensystem durch die inverse Rotation durch den Winkel ϕ um den Koordinatenursprung:

$$\begin{pmatrix} x_{LAP_V} \\ y_{LAP_V} \end{pmatrix} = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix} \cdot \begin{pmatrix} x_{LAP_H} \\ y_{LAP_H} \end{pmatrix} \tag{4.21}$$

Für die Berechnung des Lenkwinkels α aus den Koordinaten des LAP stehen in der der Applikation zwei zielpunktbasierte Fahrspurführungsalgorithmen zur Auswahl:

- *Follow-The-Carrot*
- *Pure-Pursuit*

Durch *Follow-The-Carrot* wird der ermittelte LAP direkt angesteuert indem der Richtungsvektor zum LAP mit dem Betrag der LAD direkt um den Lenkwinkel α relativ zum Fahrzeugkoorinatensystem gedreht wird, vgl. Abb. 4.17, (a). Der Lenkwinkel α wird durch

$$\alpha = \arctan(\frac{x_{LAP_V}}{x_{LAP_V}}) \tag{4.22}$$

berechnet und das Fahrzeug damit direkt auf den LAP ausgerichtet. Dem Vorteil seiner einfachen Berechnung steht der Nachteil des verstärkt auftretenden Aufschwing- und Auspendelverhaltens [36] bei höheren Geschwindigkeiten und engen Kurven entgegen.
Mit *Pure-Pursuit* wird der LAP auf einer Kreisbahn mit der Krümmung γ angesteuert. Diese Kreisbahn verläuft sowohl durch den Ursprung des Fahrzeugkoordinatensystems als auch durch den LAP. Der Lenkwinkel α entspricht dem Winkel zwischen der Tangente zur Kreisbahn am Ursprung des Fahrzeugkoordinatensystems und dessen y-Achse. Die Kreiskrümmung γ wird nach [21] durch den Radius r_z des Kreises um das imaginäre

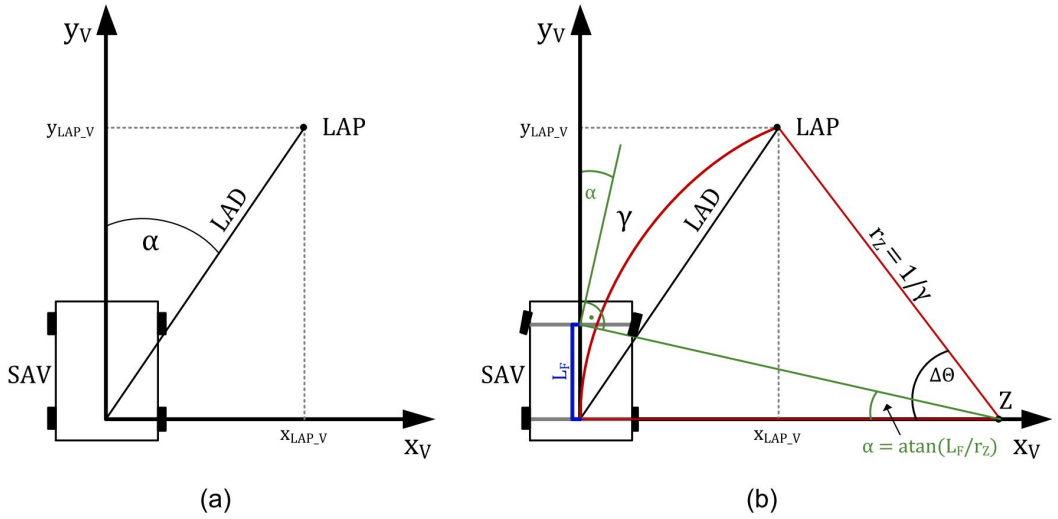

Abbildung 4.17.: Fahrspurführungsalgorithmen Follow-The-Carrot (a) und Pure-Pursuit (b) [21]. Bei FTC wird der LAP durch α direkt angesteuert, während durch PP der LAP auf einer berechneten Kreisbahn γ angesteuert wird.

Rotationszentrum Z (vgl. Abb. 4.17 (b)) in Abhängigkeit des Achsabstands L_F, dem LAP und der LAD berechnet durch

$$\gamma = \frac{1}{r_z} = \frac{2 \cdot x_{LAP_V} \cdot L_F}{LAD^2} \tag{4.23}$$

Nach dem Verfahren von *Pure-Pursuit* wird daraus der Lenkwinkel α ermittelt durch:

$$\alpha = \arctan(\frac{L_F}{r_z}) = \arctan(\frac{2 \cdot x_{LAP_V} \cdot L_F}{LAD^2}) \tag{4.24}$$

Dieses Verfahren berücksichtigt die kinematischen Eigenschaften des Fahrzeugs aufgrund der lenkbaren Vorderachse und der festen Hinterachse. Damit werden starkes Überschwingen, bzw. lange Einschwingphasen vermieden [36].

Die Berechnung des LAP sowie des Lenkwinkels α wird in der Funktion `laneFollowing()` im Modul `pipelineFunctions()` durchgeführt. Durch den Konfigurationsparameter `followTheCarrot` wird festgelegt, ob der Lenkwinkel α durch FTC oder durch PP berechnet werden soll. Zusätzlich kann durch den Konfigurationsparameter `fastLaneSwitchFTC` gesteuert werden, ob im Fall eines Fahrspurwechsels von der rechten auf die linke Spur aufgrund eines Ausweichvorgangs um ein erkanntes Hindernis der Algorithmus FTC verwendet werden soll oder nicht. Dieser Algorithmus eignet sich für einen abrupten

Ausweichvorgang besser, da der LAP bezüglich der linken Fahrspur direkt angesteuert wird, was im übertragenen Sinne einem Herumreißen des Lenkrades entspricht. Die Wahl des linken oder rechten LAD hat für die Berechnung durch PP keinen Einfluss. Hier werden die Symmetrieeigenschaften der *Arcustangenz*-Funktion ausgenutzt. Für den konkreten Fall, dass sich die die x-Koordinaten des linken und rechten LAP bei gleichem Betrag nur in ihrem Vorzeichen unterscheiden, ändert sich in Gl. 4.24 durch die x-Koordinate des LAP das Vorzeichen innerhalb des *Arcustangenz*. Da dieser eine punktsymmetrische Funktion ist würden sich die Lenkwinkel für beide LAP ebenfalls nur in ihrem Vorzeichen unterscheiden und sich somit an der y-Achse des Fahrzeugkoordinatensystems spiegeln. Somit würden sich ebenfalls die Kreisbögen zu den jeweiligen LAP an der y-Achse spiegeln. Dies entspricht im Falle eines Fahrspurwechsels einem weicher ablaufenden Ausweichvorgang.

4.3.7. Ausgabe des Lenkwinkels

Sobald die Hardware Plattform auf dem Modellfahrzeug zu dessen autonomer Steuerung integriert ist wird der berechnete Lenkwinkel über einen Pulsweitenmodulator (PWM) an die Fahrzeugelektronik übergeben und auf die drehbare Vorderachse übertragen. Da die I/O Peripherie des Pandaboards selbst keinen PWM besitzt, wurde das Pandaboard in [2] mit einem AVR Mikrocontroller verbunden, über dessen PWM Ausgang der Lenkwinkel übertragen wird. Die Ausgabe des Lenkwinkels von der Fahrspurapplikation auf dem ARM Cortex A9 über eine Schnittstelle auf ein PWM Modul wird in zukünftigen Projektschritten in die Pipeline-Funktion `pwmOutput()` im Modul `pipelineFunctions` implementiert. Derzeit führt diese Funktion keine funktionale Logik bezüglich der Ausgabe des Lenkwinkels durch. Stattdessen werden alle Felder innerhalb der `PipelineData` Struktur und des aktuellen `Lane` Elements zurückgesetzt, die innerhalb des aktuellen Pipeline-Durchgangs berechnet wurden, wie z.B. das Array der Hough-Ebene oder die Arrays der gefundenen weißen Punkte der Fahrbahn innerhalb des ROI auf dem Kamerabild.

4.4. Programmaufbau der Fahrspur Applikation

Die Implementierung der Pipeline-Funktionen bildet das zentrale Element der Fahrspurführung. Jedes Fahrspurbild ist in seiner eigenen Datenstruktur integriert, die daneben

alle weiteren Information zur aktuellen Fahrspuraufnahme kapselt. Hinzu kommen weitere Datenstrukturen und Hilfsfunktionen für die Verwaltung der Pipeline und des Programmflusses. Im Folgenden werden die Datenstrukturen für die Abbildung der Pipeline beschrieben, sowie die modulare Organisation der Applikation, welche die gesamte Infrastruktur um die Pipeline-Funktionen herum beinhaltet.

4.4.1. Datenstrukturen

Die zentrale Datenstruktur vom Typ `Lane` speichert sämtliche Informationen und Daten zu einem Fahrspurbild. Bei Programmstart werden insgesamt 30 `Lane` Strukturen deklariert und in einem Array verwaltet. Dies entspricht der Bildrate der Kamera von 30 Bildern pro Sekunde. Im Folgenden werden die wichtigsten Felder der `Lane` Struktur kategorisch zusammengefasst:

- Fahrspurbilder im OpenCV Format `IplImage`. Das Kamerabild ist im Feld `frame` gespeichert, das binarisierte Bild im Feld `frameBinarized` und das erodierte Bild im Feld `frameEroded`.
- Die Geradenparameter als Ergebnis der Hough Transformation werden in den Feldern `r` und `phi` gespeichert.
- Der ROI Ursprung ist in Pixelkoordinaten im Feld `roi_px` im OpenCV Format `CvPoint2D32f` gespeichert. Daneben speichern die Felder `roi_cm` und `roi_H` den ROI Ursprung in Fahrzeug- bzw. Hilfskoordinaten umgerechnet.
- Der LAP ist in Hilfskoordinaten im Feld `lap_H`, bzw. in Fahrzeugkoordinaten im Feld `lap_v` gespeichert.
- Die Arrays `pointsWhiteLane_px[]` und `pointsWhiteLane_cm[]` speichern die durch `dynamicROI()` gefundenen Punkte der weißen Fahrspur in Pixel- und Fahrzeugkoordinaten mit der Größe `iCountWhiteHits`.
- Der berechnete Lenkwinkel ist im Feld `alpha` gespeichert. Das Feld `bLeftSide` gibt an, ob sich der LAP und der Lenkwinkel an der linken Fahrbahnmarkierung orientieren oder nicht.

Die Struktur vom Typ `PipelineData` speichert hauptsächlich Informationen, die für alle Verarbeitungsstufen und Fahrspurbilder zu jedem Zeitpunkt allgemeingültig sind. Es wird nur eine Struktur diesen Typs definiert. Dabei werden Zugriffskonflikte vermieden,

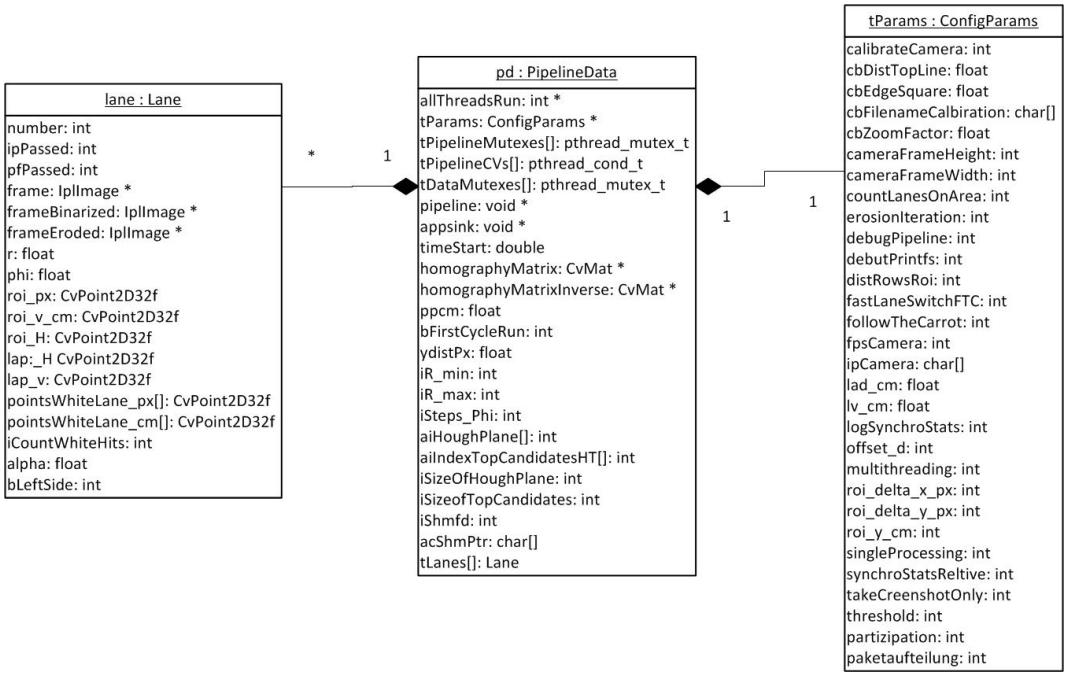

Abbildung 4.18.: Alle Informationen und Daten zur Pipeline werden in den drei Datenstrukturen gespeichert.

indem zu keinem Zeitpunkt von mehreren Funktionen zeitgleich auf dasselbe Feld dieser Struktur zugreifen:

- Mutexe und Condition Variablen zur Synchronisation der Zugriffe auf die `Lane` Elemente.
- Die während der Kalibrierung berechnete Homographie-Matrix sowie das Verhältnis zwischen Pixeln und cm sind werden in den Feldern `homographyMatrix` und `ppcm` abgelegt.
- Die Konstanten für die Einschränkung des Wertebereichs in der Hough Ebene sind in den Feldern `iR_min, iR_max` und `iSteps_Phi` gespeichert. Die Hough Ebene wird im eindimensionalen Array `aiHoughPlane` verwaltet.
- Der Deskriptor `iShmfd` erlaubt den Zugriff auf den Shared Memory Bereich für die Schnittstelle zur Hinderniserkennung.
- Das Feld `tLanes` ist der Pointer auf das erste Array-Element der Fahrspurbilder vom Typ `Lane`.

Der Programmfluss wird extern durch eine Menge von Konfigurationsparametern gesteuert. Diese werden bei Programmstart aus einer Datei eingelesen und in der dritten Datenstruktur vom Typ `ConfigParams` gespeichert. Die `PipelineData` Struktur integriert einen Pointer auf die `ConfigParams`. Eine Übersicht zu den Konfigurationsparametern ist in Kap. 4.6 aufgelistet.

Bei Programmstart werden alle Datenstrukturen dynamisch allokiert und initialisiert. Danach wird keine weitere speicherreservierende Transaktion ausgeführt. Bevor also die Threads starten und die Pipeline aufstellen, ist der Speicherbedarf deterministisch festgelegt, wodurch Fehlerfälle im Zusammenhang mit Überladung des Heap-Speicherbereiches ausgeschlossen werden können, und die Applikation somit an Robustheit gewinnt. Eine Übersicht zur Komposition der verschiedenen allokierten und initialisierten Datenstrukturen ist nochmals in Abbdildung 4.18 zusammengefasst.

4.4.2. Modulstruktur

Das zentrale Entwurfsprinzip der modularen Struktur ist die Trennung von Pipeline-Funktionen und Threads. Gemäß des jeweilig implementierten Entwurfsmusters für Multithreading führt ein Thread eine Reihe von Pipeline-Funktionen aus. Die Modulstruktur der Applikation ist in die drei Ebenen *Infrastruktur, Threads* und *Pipeline-Funktionen* strukturiert (vgl. Abb. 4.19). Alle Pipeline Funktionen sind im Modul `pipelineFunctions` definiert. Eine Ausnahme ist die Funktion `imaging()`, die zusammen mit allen kamerabildverarbeitenden Funktionen im Modul `cameraInterface` separiert ist.

Die Module der Ebene `Threads` enthalten die Thread-Funktionen, die in Kap. 4.5 erläutert werden. Die Pipeline-Funktionen sind in Kap. 4.3 beschrieben.

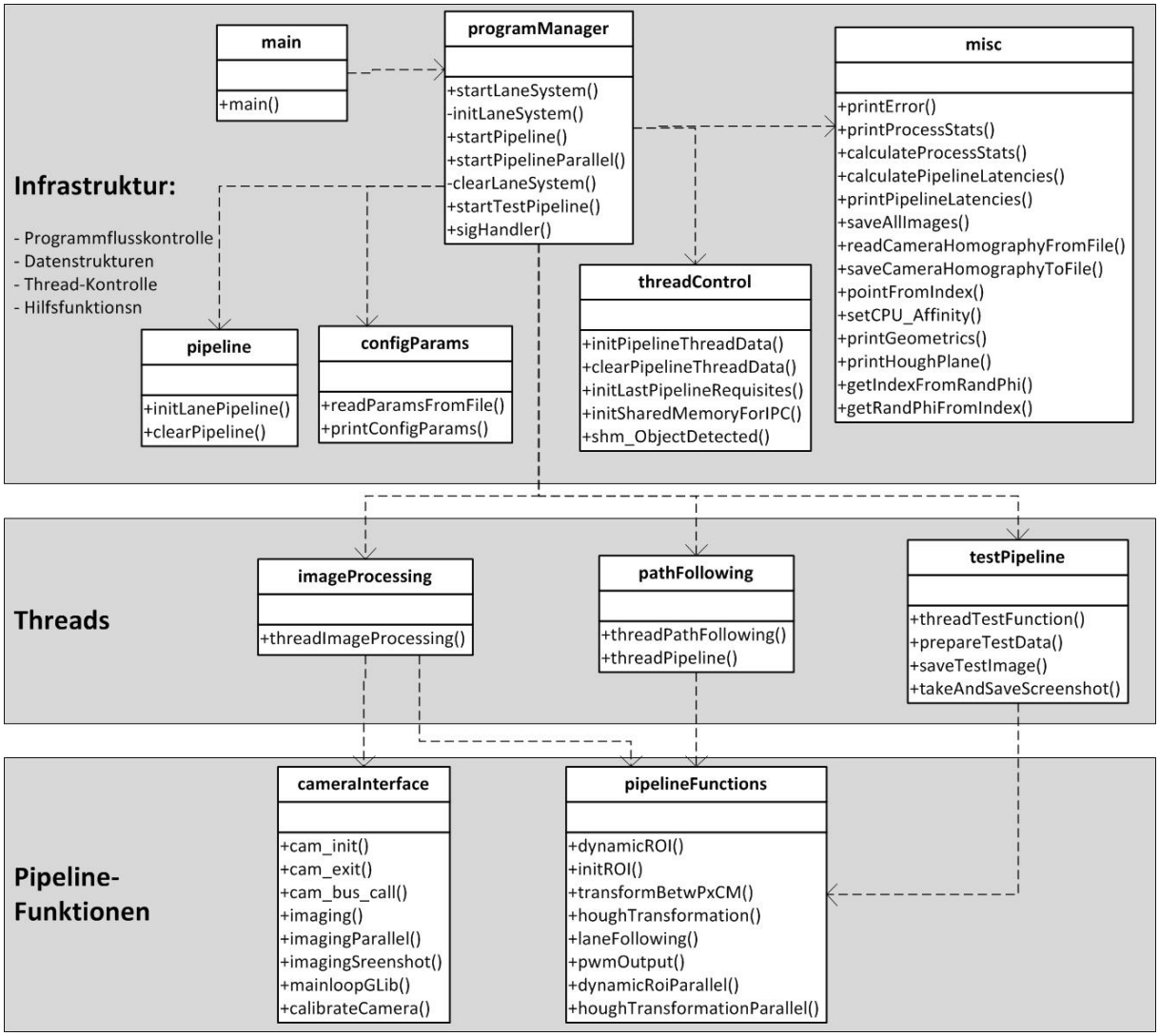

Abbildung 4.19.: Softwarearchitektur der Fahrspurführung auf drei logische Ebenen verteilt

Die Module der Ebene *Infrastruktur* beinhalten die Programmsteuerung wie z.B. das Starten und Beenden der Pipeline-Threads auf oberster Ebene oder Initialisierung der Datenstrukturen bei Programmstart:

- `programManager`: Steuerung der Initialisierung aller Datenstrukturen für die Pipeline sowie Starten und Beenden der Threads. Freigabe des allokierten Speichers aller Datenstrukturen nach Beendigung.
- `pipeline`: Allokierung und Initialisierung der `Lane`-Strukturen in einem Array. Aktuell entspricht die Anzahl der `Lane`-Elemente der Framerate der IP-Kamera

pro Sekunde, welche über den Konfigurationsparameter `fpsCamera` gesetzt wird. Aktuell übersendet die Kamera einen MPEG-4 Stream mit 30 Bildern pro Sekunde.

- `threadControl`: Initialisierung der Pipeline-Mutexe und Conditionvariablen sowie der Daten-Mutexe in Arrays. Die jeweils ersten Elemente aller Arrays werden in der Datenstruktur vom Typ `PipelineData` gespeichert.
- `configParams`: Einlesen der Konfigurationsparameter aus der Datei `params.config` und Speichern in der Datenstruktur vom Typ `ConfigParams`.
- `misc`: Definition von häufig benutzten Hilfsfunktionen, um somit Code-Sequenzen zu zentralisieren, da sie von mehreren Threads genutzt werden.

4.5. Programmablauf und Datenfluss in der Pipeline

Die Thread-Funktionen sind in den Modulen `imageProcessing`, `pathFollowing` und `testPipeline` definiert. Dabei entsteht die folgende Zuordnung zwischen einer Threadfunktion und einem in Kap. 4.1.2 vorgestellten Entwurfsmuster:

- Die Implementierung der Pipeline durch das Entwurfsmuster der funktionalen Parallelität durch zwei Threads mit expliziter CPU Zuweisung erfolgt durch gleichzeitiges Starten der Threads `threadImageProcessing()` und `threadPathFollowing()`.
- Die Ausführung der Pipeline nach dem Entwurfsmuster der datenabhängigen Parallelität innerhalb parallelisierbarer Pipeline-Funktionen erfolgt durch Starten des Threads `threadPipeline()`
- Das Testen, Debuggen und die Leistungsanalyse einzelner Pipelinefunktionen erfolgt im Thread `threadTestFunction()` im Modul `testPipeline`, vgl. Kap. 5.

Die explizite Zuweisung eines Threads an eine CPU ist vom Entwurfsmuster abhängig. Dem Linux-Scheduler wird damit die Information mitgeteilt, dass der betreffende Thread auf der gewählten CPU ausgeführt werden soll. Damit wird jedoch massiv in die Scheduling-Policy bei SMP Systemen eingegriffen, denn damit wird ein Load-Balancing zwischen mehreren CPU Kernen in SMP Systemen für den betreffenden Thread bewusst verhindert. Die explizite Zuweisung eines Threads an eine bestimmte CPU erfolgt aus dem Programm-Code heraus durch die Funktion `pthread_setaffinity_np()`, die als Interface der POSIX Threads API zur Kernelfunktion `sched_setaffinity()` fungiert.

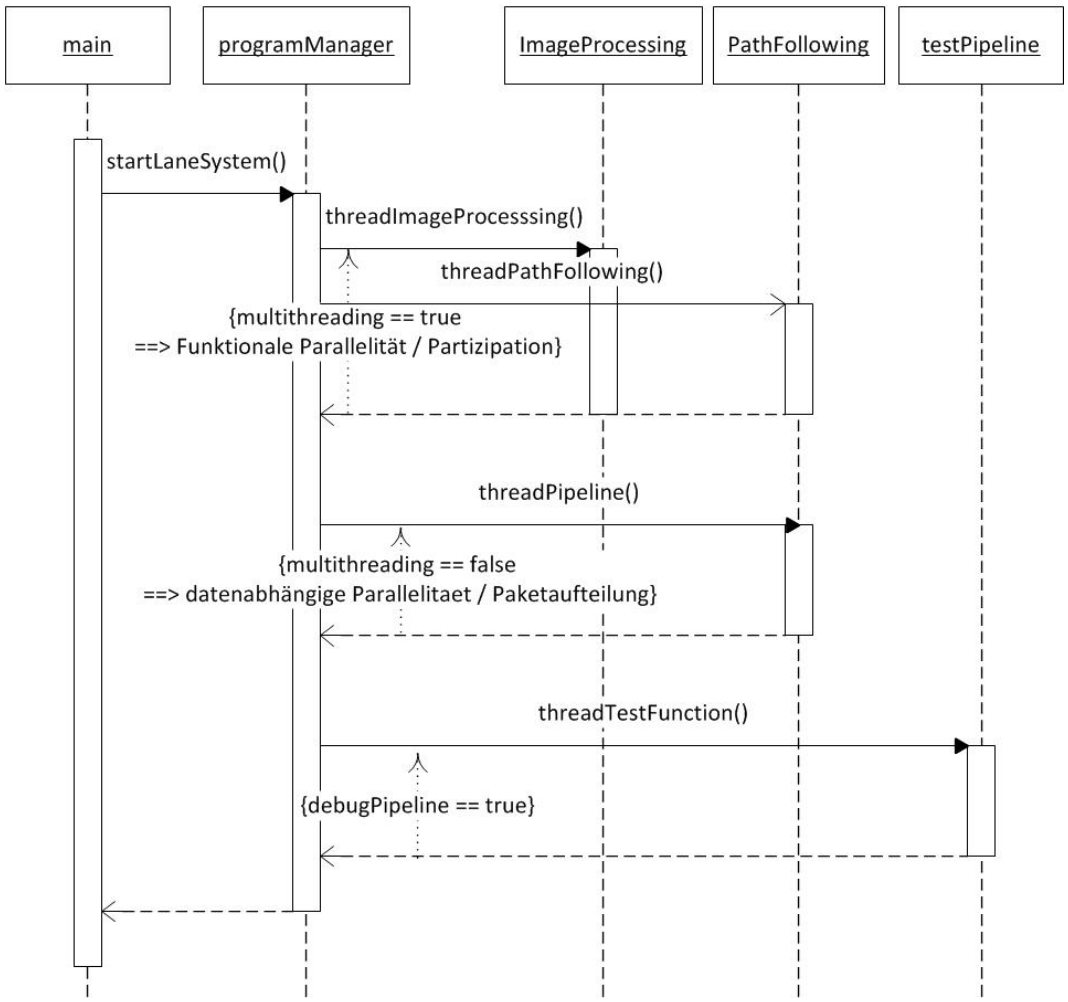

Abbildung 4.20.: Starten der Threadfunktionen gemäß des durch Konfigurationsparameter festgelegten Entwurfsmusters

Im folgenden Abschnitt aus dem Quellcode des Threads `threadImageProcessing` wird der Thread dem zweiten CPU-Kern (CPU1) zugewiesen:

```
1  int s;
2  cpu_set_t cpumask;
3  CPU_ZERO(&cpumask);
4  CPU_SET(1, &cpumask);
5
6  s = pthread_setaffinity_np(
7    pthread_self(),
8    sizeof(cpu_set_t),
9    &cpumask);
10
```

```
11  if(s != 0)
12  {
13    perror(NULL);
14    /* ERROR HANDLING */
15  }
```

Listing 4.8: Explizite Zuweisung eines Threads an eine CPU

In Zeile 2-3 wird eine Bitmaske angelegt, welche die zur Verfügung stehenden CPUs repräsentiert und durchweg mit 0 initialisiert. In Zeile 4 wird die Bitmaske mit dem Wert 1 gefüllt, welche für die zweite CPU (CPU 1) steht. Ein Füllen der Bitmaske mit der ersten CPU (CPU 0) würde das Übergeben der Zahl 0 bedeuten. Somit legt die gefüllte Bitmaske fest, auf welcher CPU der Thread ausgeführt werden soll. Die Bitmaske wird der POSIX-Funktion `pthread_setaffinity_np()` (Zeile 6-9) übergeben. Dabei liefert die POSIX-Funktion `pthread_self()` die ID des aufrufenden Threads.

Die Synchronisation der Threads zur Realisierung der Pipeline anhand des Konzepts der funktionalen Parallelisierung durch die zwei Threads `threadImagePipelining` und `threadPathFollowing` ist in Abbildung 4.21 dargestellt. In diesem Konzept gibt es also zwei Threadfunktionen, wobei jeder Threadfunktion ihre Pipeline-ID `id` zugeordnet wird. Diese Zuordnung erfolgt gemäß der Rangfolge innerhalb der Pipeline. Hier hat `imageProcessing` die `id = 0` und `pathFollowing` die `id = 1`. Für die Synchronisation wird ein Pipeline-Mutex und eine Pipeline-Conditionvariable benutzt. Die Anzahl der Array-Elemente in der `PipelineData`-Struktur für die Pipeline-Synchronisation entspricht also der Anzahl Threads - 1. Die Anzahl Threads ist im Makro `NOPIPLINEPROCESSES` in der globalen Headerdatei `globals.h` festgelegt. Der erste Thread der Pipeline (in diesem Fall `imageProcessing`) synchronisiert sich mit seinem Nachfolge-Thread (hier `pathFollowing`) über den Mutex und die Condition-Variable jeweils an der 0. Stelle der Arrays. Der zweite Thread indiziert dieselben Elemente durch seine `id - 1`.

Zum Zeitpunkt T1 wollen beide Threads denselben Pipeline-Mutex sperren. Damit die Pipeline aber in der richtigen Reihenfolge aufgestellt wird, muss dafür gesorgt werden, dass der zweite Thread (`pathFollowing`) den Mutex zuerst sperrt. Dies wird dadurch erreicht, indem der zweite Thread in der Funktion `startLaneSystem` im Modul `programManager` vor dem ersten Thread gestartet wird. Nach dem erfolgreichen Lock des Pipeline-Mutexes wartet der zweite Thread zum Zeitpunkt T2 auf das Signal über die Conditionvariable, die sich an der gleichen Position im Array der Pipeline-

Conditionvariablen befindet, wie der Pipeline-Mutex. Über die gleiche ID existiert somit

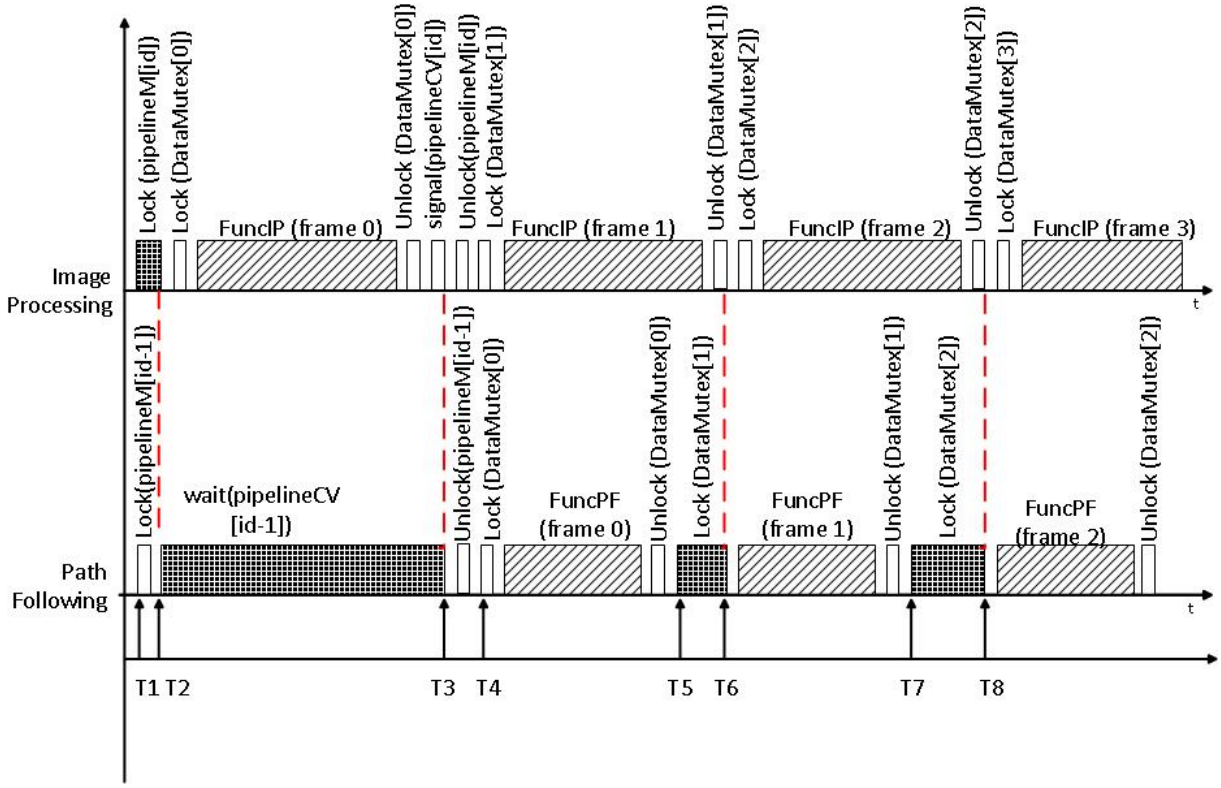

Abbildung 4.21.: Realisierung der Pipeline durch Synchronisierung der Threads über Mutexe und Conditionvariablen

eine 1:1 Zuordnung zwischen einer Pipeline-Conditionvariablen und einem Pipeline-Mutex. Die `wait`-Funktion ist blockierend und bewirkt zudem ein Entsperren des Pipeline-Mutexes. Somit kommt die blockierende Lock-Funktion im ersten Thread nahezu zeitlich zum Zeitpunkt T2 zurück. Der zweite Thread wartet nun auf das Signal über die Conditionvariable zum Zeitpunkt T3. Aus der Sicht des ersten Threads gibt es zu diesem Zeitpunkt keine konkurrierenden Threads um dieselbe Ressource. Er kann somit den Daten-Mutex für das 0. Frame sperren. Bis zum Zeitpunkt T3 führt der erste Thread die erste Pipeline-Stufe der Fahrspurführung auf das 0. Frame-Element im `Lane`-Array aus. `FuncIP()` steht hierbei für das Ausführen einer oder mehrerer Pipeline-Funktionen.

Nach Ausführung der ersten Pipeline-Stufe und Entsperren des 0. Daten-Mutex schickt der erste Thread das Signal über die Conditionvariable an den zweiten Thread. Dieser sperrt ebenfalls den 0. Daten-Mutex während der erste Thread den 1. Daten-Mutex sperrt. Somit ist die Pipeline aufgebaut. Die weitere Synchronisation erfolgt nur noch über die

Daten-Mutexe. Um sicherzustellen, dass zu jedem Zeitpunkt der richtige Thread den richtigen Daten-Mutex sperren kann, soll die Latenzzeit des jeweiligen Nachfolge-Threads für die Bearbeitung eines Frames kürzer sein, als die Latenzzeit des Vorgänger-Threads. Dadurch verlängert sich die Zeit des Sperrens auf einen Daten-Mutex, weil dieser noch vom Vorgänger-Thread gesperrt ist, wie in den Zeitspannen zwischen T5 und T6, sowie zwischen T7 und T8 ersichtlich ist.

4.6. Funktionssteuerung über Konfigurationsparameter

Die Datei `params.config` beinhaltet eine Liste von Konfigurationsparametern zur Steuerung und Konfiguration des Daten- und Programmflusses in der Fahrspur-Pipeline. Die Parameter werden bei Programmstart eingelesen und an die Threads übergeben. Eine Übersicht zu den Konfigurationsparametern ist in der folgenden Tabelle aufgeführt.

Name	Beschreibung
`calibrateCamera`	Legt fest, ob bei Programmstart eine Kamerakalibrierung durchgeführt werden soll
`cbDistTopLine`	Abstand der Querlinie der obersten inneren Ecken auf dem Schachbrett zum Fahrzeugursprung
`cbEdgeSquare`	Kantenlänge eines Quatrats auf dem Schachbrett
`cbFilenameCalibr`	Name der Datei, in welche die Kalibrierungsparameter gespeichert werden sollen.
`cbZoomFactor`	Zoom-Faktor, welcher mit der Homographie-Matrix multipliziert wird
`cameraFrameHeight`	Höhe eines Kamerabildes
`cameraFrameWidth`	Breite eines Kamerabildes
`countLanesOnArea`	Anzahl weißer Fahrbahnmarkierungen auf dem Fahrbahnuntergrund
`erosioniteration`	Anzahl der Durchführungen des Erosionsfilters
`debugPipeline`	Legt fest, ob die Test-Pipeline zu Debug-Zwecken gestartet werden soll
`debugPrintfs`	Legt fest, ob Debug-Ausgaben geschrieben werden sollen

Name	Beschreibung
`distRowsRoi`	Zeilenabstand in Pixeln für die Iteration innerhalb `dynamicROI()`
`fastLaneSwitchFTC`	Legt fest, ob bei einem Fahrspurwechsel der Algorithmus *Follow the Carrot* verwendet werden soll
`followTheCarrot`	Legt fest, ob der Algorithmus *Follow the Carrot* im Normalbetrieb verwendet werden soll
`fpsCamera`	Bildrate der Kamera
`ipCamera`	IP Adresse der Kamera
`lad_cm`	*Look Ahead Distance* in cm
`lv_cm`	Abstand der Fahrzeugachsen
`logSynchrostats`	Legt fest, ob Zeitmessungen der Threadsynchronisation vorgenommen werden sollen
`offset_d`	Abstand des LAP im Hilfkoordinatensystem zur Fahrspur
`multithreading`	Legt fest, nach welchem Entwurfsmuster die Threads gestartet werden sollen
`roi_delta_x_px`	Länge der ROI-Kante vom Ursprung in x-Richtung
`roi_delta_y_px`	Länge der ROI-Kante vom Ursprung in y-Richtung
`roi_y_cm`	Initialer Abstand des ROI in Fahrzeugkoordinaten vom Fahrzeug in y-Richtung
`singleProcessing`	Legt fest, ob der Thread `threadPipeline` die parallelisierbaren Pipelinefunktionen ausführen soll
`synchroStatsRelative`	Legt fest, ob die Zeitmessungen der Threadsynchronisation relativ zu einem initialen Startzeitpunkt gemessen werden
`takeSreenshotOnly`	Gibt an, ob statt dem Start der Pipeline ein Sreenshot durch die Kamera gespeichert werden soll
`threshold`	Schwellwert für den Kantenübergang der Binarisierung
`partizipation`	Gibt an, ob die Threads nach dem Entwurfsmuster der Partizipation ausgeführt werden sollen.
`paketaufteilung`	Gibt an, ob die Threads nach dem Entwurfsmuster der Paketaufteilung ausgeführt werden sollen.

Tabelle 4.2.: Übersicht zu den Konfigurationsparametern für die Steuerung des Daten- und Programmflusses der Pipeline

5. Ergebnisanalyse

Durch die Leistungsanalyse soll die Anzahl der Pipeline-Funktionen ermittelt werden, die auf einem Kern des Cortex A9 DualCore Prozessors zwischen zwei aufeinanderfolgenden Kamerabildern sequentiell ausgeführt werden können. Diese Erkenntnis setzt die Grundlage für die Durchführbarkeit der in Kap. 4.1.2 vorgestellten Entwurfsmuster. Hierzu werden für jede Pipeline-Funktion, bzw. für eine Sequenz von Pipeline-Funktionen die beiden Kennzahlen *Latenzzeit* und *CPU-Auslastung* gemessen. Dieses Kapitel beschreibt zum einen die allgemeine Vorgehensweise, wie diese beiden Kennzahlen ermittelt werden, sowie die Ergebnisse der Leistungsanalyse der einzelnen Pipelinefunktionen und der Pipelinezyklen nach den entsprechenden Entwurfsmustern.

5.1. Vorgehensweise zur Messung von Latenzzeit und CPU Auslastung

Die *Latenzzeit* ist die Zeitdifferenz zwischen zwei Zeitstempeln, die im Quellcode innerhalb eines Threads vor und nach der Ausführung einer Pipeline-Funktion, bzw. einer Sequenz von mehreren Pipeline-Funktionen gesetzt wurden. Der folgende Abschnitt aus dem Quellcode des Threads `threadImageProcessing()` zeigt das Setzen der Zeitstempel um die Pipeline-Funktionen *Imaging* und *Binarize*. Letztere wird von der OpenCV-Funktion `cvThreshold()` implementiert.

```
1 clock_gettime(CLOCK_REALTIME, &timeStartStep);
2
3 imaging(appsink, &(runner->frame) , thresholdCV);
4
5 cvThreshold(
6   laneRunner->frame,
7   laneRunner->frame, thresholdCV, 255, CV_THRESH_BINARY);
```

```
8
9 clock_gettime(CLOCK_REALTIME, &timeStopStep);
```

Listing 5.1: Zeitstempel um Pipeline-Funktion

Die Felder `timeStartStep` und `timeStopStep` sind vom Typ `struct timesec`, deklariert in `sys/time.h`. Das Setzen der Zeitstempel wird durch die Funktion `clock_gettime()` ausgeführt. Die Berechnung der Latenzzeit wird von der Hilfsfunktion `calculatePipelineLatencies()` im Modul `misc` durchgeführt. Durch dieses Verfahren lässt sich die Latenzzeit jeder Pipeline-Funktion messen.

Die erste Pipeline-Funktion der Fahrspurverarbeitungskette - *Imaging* - bildet bei der Messung ihrer Latenzzeit einen Sonderfall. Das Lesen der Bilddaten aus dem Videopuffer der GStreamer-Pipeline ist solange blockierend, bis der Puffer durch GStreamer mit den nächsten Bilddaten gefüllt wird. Die IP-Kamera liefert einen MPGEG-4 Stream mit der maximalen Auflösung von 30 fps (Frames per Second). Aus dieser Bildratenfrequenz von 30 Hz ergibt sich somit eine Verarbeitungszeit von 33 ms pro Bild. Um einen Kern des ARM Cortex-A9 Prozessors bestmöglich auszulasten, wird diese Verarbeitungszeit mit der größtmöglichen Sequenz von Pipeline-Funktionen ausgekleidet, welche hintereinander einen zusammenhängenden Teil der Verarbeitungskette pro Bild darstellen. Wird die erste Pipeline-Funktion *Imaging* nun nach einer geringeren Verarbeitungszeit als 33 ms iterativ aufgerufen, erhöht sich deren Latenzzeit aufgrund des blockierenden Lesens aus dem Video-Puffer. Um die kleinstmögliche Latenzzeit messen zu können, die gleichzeitig der Latenzzeit unter Vollauslastung des Prozessorkerns entspricht, muss die Verarbeitungszeit bis zur nächsten Zeitintervallgrenze von 33 ms beispielsweise mit einer `sleep`-Funktion überbrückt werden. Die kleinstmögliche Latenzzeit der Pipeline-Funktion *Imaging* wurde ermittelt, indem die Überbrückungszeit solange erhöht wurde, bis die gemessene Latenzzeit gegen einen unteren Grenzwert konvergiert.

Die zweite Kennzahl *CPU-Auslastung* wird mit dem ARM STREAMLINE Analyzer ermittelt, einer Komponente innerhalb ARM DS-5. STREAMLINE kommuniziert während der sog. *Capture*-Phase mit dem Kernelmodul *gator* [33] innerhalb des Linux Betriebssystems auf der Zielplattform. Gator misst auf der Plattform eine Reihe von CPU-Statistiken pro Millisekunde aus und übermittelt diese dem STREAMLINE Analyzer, der sie visualisiert ausgibt, darunter auch die CPU-Auslastung in Prozent für jeden der beiden Prozessorkerne. Die CPU-Auslastung pro Millisekunde ist der Anteil in Prozent, in dem die CPU

eine Sequenz von Instruktionen ausführt, die nicht zum Leerlaufprozess *idle* in Linux gehören. Beträgt die CPU-Auslastung für eine gemessene Millisekunde 0%, bedeutet dies somit, dass auf der CPU innerhalb dieser Millisekunde ausschließlich der Leerlaufprozess *idle* ausgeführt wird. Beträgt umgekehrt die Latenzzeit eines Pipeline-Prozesses 3 ms, so soll die CPU-Auslastung für drei direkt aufeinanderfolgenden Millisekunden 100% betragen. Die CPU-Auslastung ist somit eine ergänzende Darstellung, bzw. Verifikation der Latenzzeit. Um eine gemessene Latenzzeit mithilfe der CPU-Auslastung verifizieren zu können, muss sichergestellt werden, dass 100% CPU-Auslastung ausschließlich von der Pipeline-Funktion verursacht wurde, für welche die Latenzzeit gemessen wurde (s.u).

Im Folgenden wird nun eine allgemeine Vorgehensweise zur Messung einer Latenzzeit einer bestimmten Pipeline-Funktion und ihrer Verifikation durch die CPU-Auslastung beschrieben:

1. Setzen der Zeitstempel um die zu messende Pipeline-Funktion im Quellcode innerhalb des Threads `threadPipeline` (vgl. Listing 5.1)
2. Starten des Threads `threadPipeline` und somit zyklische Verarbeitung durch die implementierten Pipeline-Funktionen, iteriert über das `Lane`-Array. Aktivierung der Ausgabe der gemessenen Latenzzeit (Konfigurationsparameter `debugPrintfs = 1` setzen)
3. Alle vorausgehenden Pipeline-Funktionen vor der zu messenden Funktion in der Verarbeitungskette in der Funktion `prepareTestData()` im Modul `testPipeline` aufrufen und die zu testende Pipeline-Funktion aus dem Thread `threadTestFunction()` im gleichen Modul aufrufen.
4. Die im 1. Schritt gemessene Latenzzeit von der gesamten Verarbeitungszeit 33 ms abziehen und die Differenz an die `usleep()` Funktion in Mikrosekunden übergeben, welche im Anschluss an die zu testende Pipeline-Funktion aufgerufen wird. Dies entspricht der Überbrückungszeit.
5. Start des STREAMLINE Analyzers und Ausführung der Applikation im Debug-Modus (Konfigurationsparameter `debugPipeline = 1` setzen)

Das Modul `testPipeline` stellt ein Testwerkzeug innerhalb der Applikation dar. Zuerst wird jedes Element des `Lane`-Arrays in der Funktion `prepareTestData()` mit testweisen Bilddaten aus dem Filesystem gefüllt, ohne die GStreamer-Pipeline aktivieren zu müssen, und anschließend durch alle vorausgehende Pipeline-Funktionen vor der zu messenden

Funktion verarbeitet. Somit stehen der zu testenden Pipeline-Funktion im Thread valide Testdaten zur Verfügung.

5.2. Übersicht zum Ressourcenverbrauch der Pipelinefunktionen

Die Ergebnisse der Leistungsanalyse der Pipeline-Funktionen sind in der folgenden Tabelle 5.1 zusammengefasst. Alle Funktionen wurden auf der CPU 1 ausgeführt. Daher bezieht sich die CPU-Auslastung in Prozent ausschließlich auf die CPU 1. Für diese Analyse wurde zunächst keine Optimierungsoption (vgl. Kap. 5.4) berücksichtigt.

Pipeline-Funktion	Latenzzeit in ms	CPU-Auslastung in %
`imaging()`	11	15.5
`cvThreshold()`	1.8	5.8
`cvErode()`	11.7	37.68
`dynamicROI()`	1.1	2.42
`houghTransformation()`	4.2	17.05
`laneFollowing()`	0.1	2
`pwmOutput()`	0.1	2
Summe:	30	82,45
Verbleibend:	3	17.55

Tabelle 5.1.: Leistungsanalyse der Pipeline Funktionen ohne Optimierungen

Somit kann ein Fahrspurbild durch die Pipeline nach dieser Implementierung unter Linux auf dem ARM Cortex A9 Prozessor mit einer Taktung von 1 GHz innerhalb der durch die Kamerabildfrequenz vorgegebenen 33 ms auf nur einer CPU sequentiell verarbeitet werden. Zugleich besteht keine Notwendigkeit, die Pipeline nach den Entwurfsmustern *Partizipation* und *Paketaufteilung* auszuführen. Die Leistungsanalyse der Pipeline mit Anwendung aller in Kap. 5.4 erläuterten Optimierungsoptionen ist in folgender Tabelle 5.2 aufgelistet, wobei die wichtigste Optimierung im Entfernen des Erosionsfilters aus der Pipeline sowie dessen Kompensierung durch `dynamicROI()` liegt, vgl. Kap. 5.4.

Im Vergleich zur OMAP Plattform wird der ARM Cortex A9 Prozessor auf der Zync Plattform mit 600 MHz getaktet, was für die Pipeline einen Anstieg der Latenzzeit um

Pipeline-Funktion	Latenzzeit in ms	CPU-Auslastung in %
`imaging()`	11	15.5
`cvThreshold()`	1.8	5.8
`dynamicROI()`	0.1	2
`houghTransformation()`	1,2	2.45
`laneFollowing()`	0.1	2
`pwmOutput()`	0.1	2
Summe:	14.3	29.75
Verbleibend:	18.7	70.25

Tabelle 5.2.: Leistungsanalyse der Pipeline Funktionen mit Optimierungen

60 % zur Folge hat. Beispielsweise ergibt sich für die optimierte Pipeline statt 14.3 ms eine Latenzzeit von 22,88 ms. Bei einer Kamerabildfrequenz ab 50 Bildern pro Sekunde und der daraus abgeleiteten Berechnungszeit von 20 ms pro Bild kann die Pipeline nicht mehr auf einer CPU sequentiell ausgeführt werden. Hierfür ergeben sich für die Pipeline folgende Optionen:

- Multithreading Variante der Pipeline mit funktionaler Parallelität durch zwei Threads
- Ressourcenteilung mit der Hinderniserkennung nach den Entwurfsmustern *Paketaufteilung* und *Partizipation*
- HW/SW Co-Design durch Ausführen von kamerabildverarbeitenden Funktionsblöcken [35] auf dem FPGA der Zynq Plattform

Die durch den ARM STREAMLINE Analyzer erstellte grafische Darstellung der CPU Auslaustung durch die sequentiell ausgeführte Pipeline ohne Multithreading auf der CPU 1 ist in der folgenden Abbildung 5.1 zu sehen. Der oberste Bildabschnitt enthält eine Zeitachse, die auf eine Schrittlänge von 100 ms skaliert ist. Unterhalb der oberen Zeitachse befinden sich in der Kategorie *CPU Aktivity* zwei horizontal verlaufende Diagrammflächen mit schwarzem Hintergrund, wobei die Auslastung der CPU 0 in der oberen und die Auslastung der CPU 1 in der unteren Diagrammfläche in Form von grünen Balkenausschlägen pro Zeitschritt von 100 ms dargestellt werden. Die Auslastung der CPU 0 zwischen 1,8 und 2,4 Sekunden wurden von der Initialisierungsphase verursacht. Zwischen 2,5 und 3 Sekunden wird keine CPU Auslastung verursacht. In dieser Zeitspanne wird auf die Fertigstellung der Initialisierung der Co-Prozessoren, die

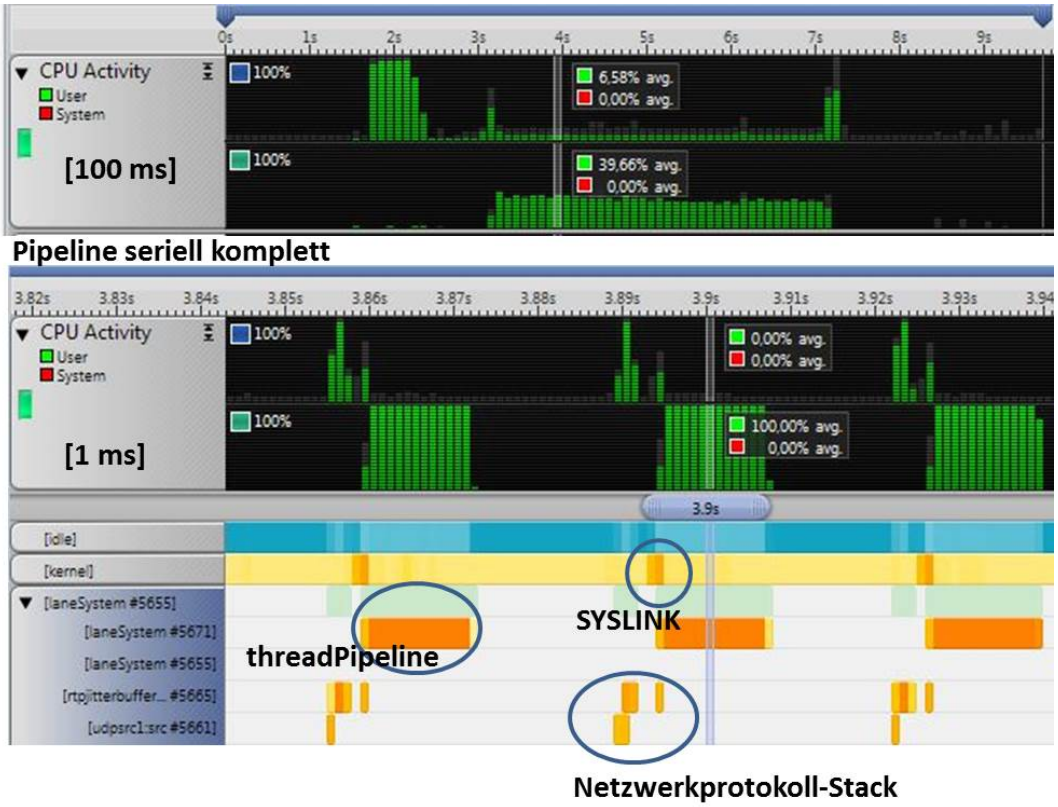

Abbildung 5.1.: CPU Auslastung der sequentiellen Pipeline im Thread `threadPipeline`

an der GStreamer-Pipeline für die Kamerabildverarbeitung beteiligt sind, gewartet. Nach 3 Sekunden startet die sequentielle Pipeline auf der CPU1 mit einer Verarbeitungszeit von 4 Sekunden und einer Auslastung der CPU 1 von 39,66 %. In diesem Durchgang wurde die Pipeline ohne den Erosionsfilter und ohne die restlichen Optimierungsoptionen gestartet. Eine Durchführung mit den restlichen Optimierungsoptionen entspricht einer CPU Auslastung von 29,75 %, wie in Tabelle 5.2 aufgelistet. Der mittlere Diagrammbereich für beide CPUs entsteht aus dem oberen Diagrammbereich durch Verändern der zeitlichen Skalierung auf die minimale Schrittlänge von 1 ms. Der auf der CPU0 gemessene Ausschlag zwischen 3,889 und 3,891 Sekunden wird durch den Empfang der MPEG-4 Pakete eines Kamerabildes in den RTP-Puffer auf einem UDP Socket verursacht. Die Lücke zwischen 3,892 und 3,894 Sekunden entspricht dem Warten auf das Dekodieren des Bilddatenpuffers durch die Co-Prozessoren. Stellvertretend für alle Pipeline-Funktionen wird die grafische Darstellung der CPU Auslastung für die Pipeline-Funktionen `imaging()`, `cvThreshold()` und `houghTransformation()` in Abbildung 5.2 zusammengefasst. Im obersten Diagramm der CPU Auslastung durch `imaging()`

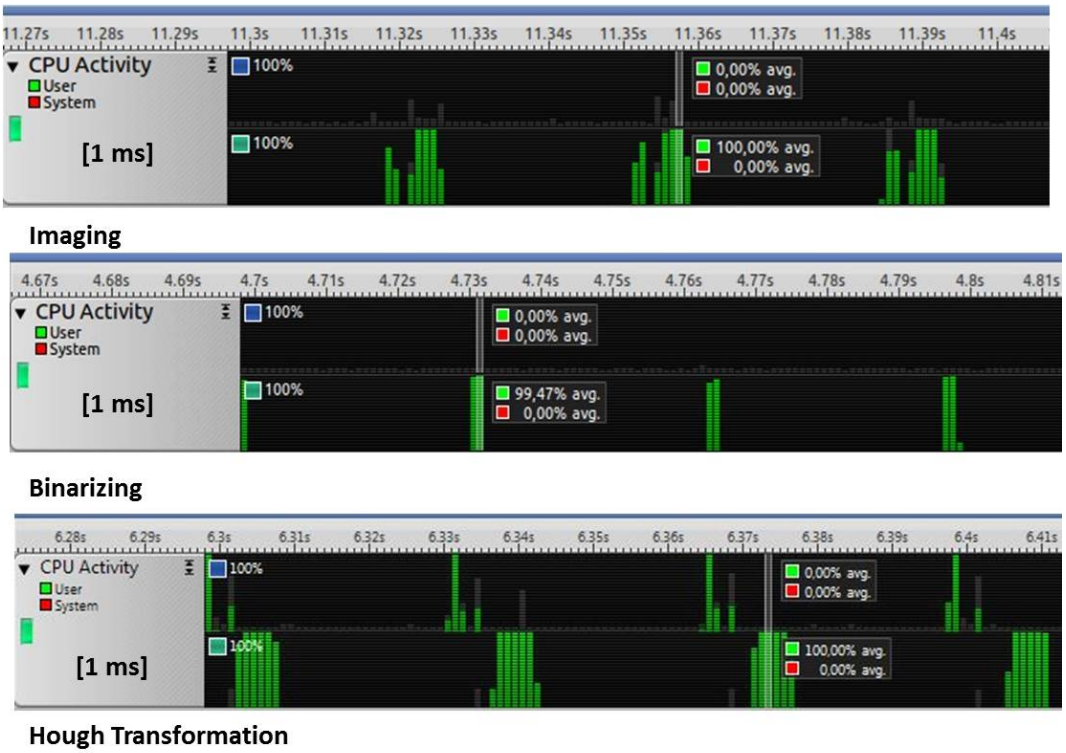

Abbildung 5.2.: CPU Auslastung dreier Pipelinefunktionen pro 1 ms

pro 1 ms sind drei zusammenhängende Auslastungsbereiche zu erkennen, die mit jedem Aufruf von `imaging()` verursacht werden. Die ersten beiden Ausschläge werden durch Netzwerkprotokollstack über RTP und UDP verursacht. Die Lücke danach entspricht der Verarbeitung durch Syslink auf den Co-Prozessoren. Die vier folgenden Ausschläge werden durch das Kopieren der Bilddaten vom dekodieren GStreamer-Puffer in den OpenCV Buffer innerhalb der Funktion `imaging()` verursacht. Im mittleren Diagramm verursacht der Aufruf der Funktion `cvThreshold()` für zwei Millisekunden jeweils einen Ausschlag von knapp 100 % pro Millisekunde. Damit wird die gemessene Latenzzeit von 1,8 ms verifiziert. Der Aufruf der Funktion `houghTransformation()` verursacht im unteren Diagramm einen Ausschlag von 100 % über 5 ms, was der Latenzzeit ohne Optimierung von 4,2 ms nahekommt. Im Gegensatz zum Analysedurchlauf für die Binarisierung ist bei der Hough Transformation die GStreamer Pipeline noch nicht beendet worden, was durch die Ausschläge im unteren Diagramm der CPU 0 erkennbar wird.

5.3. Vergleich der parallelen Entwurfsmuster

Nach dem Entwurfsmuster der funktionalen Parallelität durch zwei Threads wird der Ressourcenverbrauch auf zwei CPUs verteilt, wie in der folgenden Abb. 5.3 zu sehen ist.

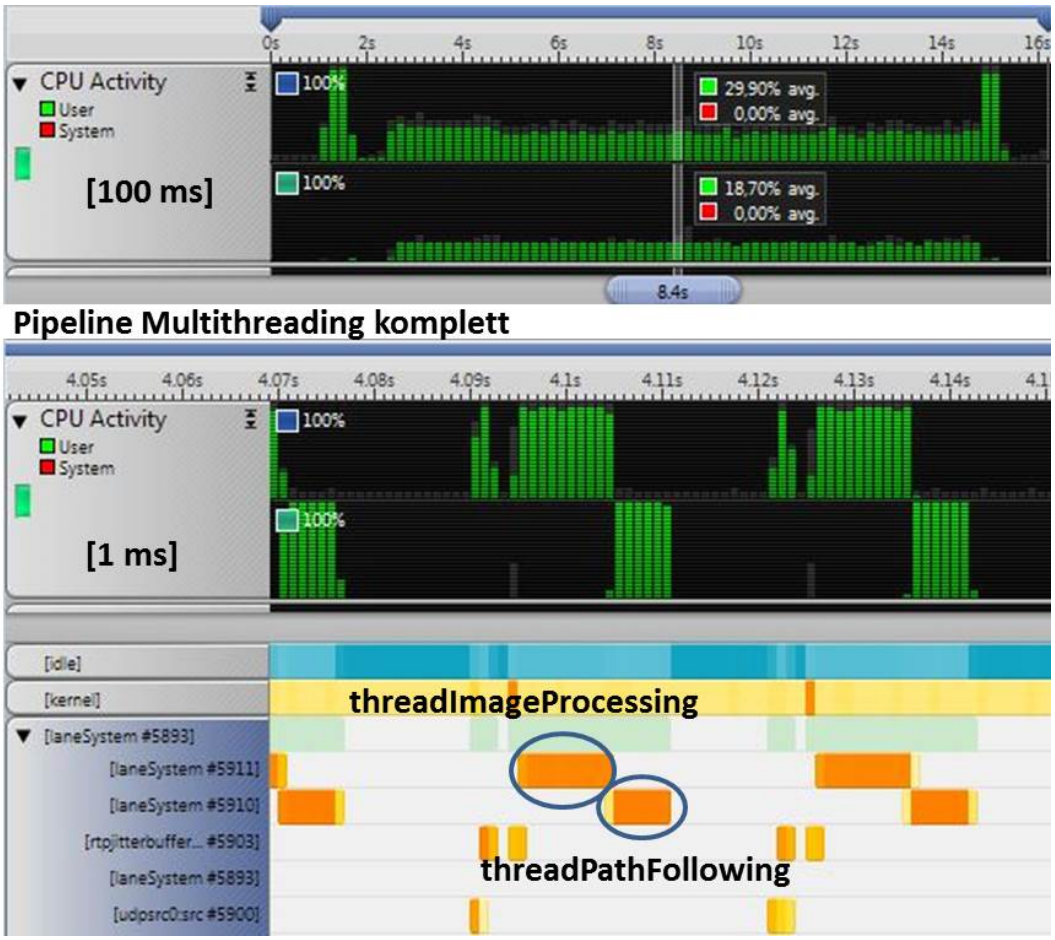

Abbildung 5.3.: CPU Auslastung beider CPUs bei parallelem Ausführen der Threads `threadImageProcessing` und `threadPathFollowing`

Die Auslastung der CPU 0 von 29,9 % pro 100 ms wird durch den Netzwerkprotokollstack, Syslink und den Thread `threadImageProcessing()` verursacht, der die Pipeline-Funktionen `imaging()`, `cvThreshold()` und `dynamicROI()` ohne Erosionsfilter und ohne restliche Optimierungen ausführt. Die Latenzzeit wird dabei auf 15.5 ms pro Durchgang reduziert. Im Thread `threadPathFollowing()` auf der CPU 1 werden die Pipelinefunktionen `houghTransformation()`, `laneFollowing()` und `pwmOutput()` ohne Optimierungen mit einer Latenzzeit von 6 ms pro Durchgang und einer Auslastung der

CPU 1 von 18,7 % ausgeführt. Ebenfalls erkennbar ist die Ablösung der Bearbeitung eines Fahrspurbildes durch beide Threads sowie deren Synchronisation über den zugehörigen Daten-Mutex. Beispielsweise wird der Daten-Mutex bei 4,105 Sekunden vom Thread `threadImageProcessing()` entsperrt und vom Thread `pathFollowing()` gesperrt. In der Zeitspanne, in der auf der CPU 0 keine Auslastung gemessen wird, z.B. zwischen 4,106 und 4,122 Sekunden, wartet die Funktion `imaging()` blockierend auf das nächste Kamerabild.

Die datenabhängige Parallelisierung der Pipeline-Funktionen `dynamicROI()` und `houghTransformation()` durch OpenMP führt nicht zum erwünschten Speedup der Latenzzeit. Wird die Pipeline nach diesem Entwurfsmuster ausgeführt, ist eine Latenzzeit für die Verarbeitung eines Bildes von ca. 39 ms messbar und somit sogar höher als die Latenzzeit bei einer sequentiellen Pipeline ohne Parallelisierung durch OpenMP. Die Ursache für den Anstieg liegt im erhöhten Synchronisations- und Verwaltungsaufwand seitens des Kernels. Die Ausführung einer Pipeline-Funktion mit einem durch die `#pragma` Direktive parallelisierbaren Codeabschnitt bewirkt ein Starten und Stoppen eines weiteren Threads, sowie das Kopieren der Datenstrukturen in dessen Stack und die Synchronisation der gemeinsam genutzten Daten über den L2 Cache. Mit insgesamt zwei parallelisierbaren Funktionen verdoppelt sich dieser Zusatzaufwand pro Bild und sorgt für den Anstieg der Bearbeitungszeit. Aus diesem Grund ist die zyklische Parallelisierung einzelner Pipeline-Funktionen durch OpenMP für diese Applikation nicht geeignet. Stattdessen wird die funktionale Parallelität gegebenenfalls in Verbindung mit dem Muster der Partizipation durch explizite Zuweisung eines Threads an eine CPU bevorzugt, da der Ressourcenverbrauch auf beide CPUs verteilt wird und die Verteilung der Pipeline-Funktionen flexibel gestaltet wird.

5.4. Parametervergleich zur Optimierung der Zykluszeit

Die Entfernung des Erosionsfilters aus der Pipeline stellt die größte Ressourcenersparnis innerhalb der Optimierungsoptionen dar. Den Ausschlag für das Entfernen und Kompensieren gab die gemessene Latenzzeit des Erosionsfilters durch die OpenCV Funktion `cvErode()` von 11,7 ms, die in keinem Verhältnis zu den Zeiten der anderen Pipeline-Funktionen steht, vgl. Tab. 5.1. Hinzu kommt, dass der gewünschte Effekt des

Erosionsfilters, nämlich die Kantenschärfung durch die Reduzierung weißer Vordergrundpixel, mit einer Iteration nicht erfüllt werden kann, vgl. Abb. 4.12. Mit Erhöhung des Iterationsparameters steigt ebenfalls die gemessene Latenzzeit:

Iterationen	Latenzzeit in ms
1	11,2
2	11,7
3	12,9
4	14,2

Tabelle 5.3.: Ansteig der Latenzeit mit der Anzahl der Iteration des Erosionsfilters

Die Kantenschärfung durch den Erosionsfilter sollte ursprünglich der Hough Transformation zuarbeiten, damit weniger weiße Pixel in die Hough-Ebene übertragen werden müssen. Dies wird innerhalb der Funktion `dynamicROI()` kompensiert, indem pro definierte Pixelzeile nur ein weißes Vordergrundpixel der Hough Transformation übergeben wird. Hiermit wird die Kante maximal geschärft, indem sie die Breite eines Pixels erhält. Zusätzlich wird für die Funktion `dynamicROI()` die Möglichkeit geschaffen, zeilenweise über die Pixelmenge nicht Pixel-weise (Byte-weise) zu iterieren sondern über einen Block von vier benachbarten Bytes, die im Datentyp `uint32_t` zusammengefasst werden können. Dadurch reduziert sich die Anzahl der Schleifendurchgänge um das vierfache. Die Iteration über einen Pixelblock setzt voraus, dass die weiße Fahrspur im Bild innerhalb des ROI in x-Richtung eine Breite von mindestens 8 weißen Pixeln besitzt, damit der Vierer-Block in jedem Byte-Offset hineingelegt werden kann. Da die Funktion `dynamicROI()` auf dem perspektivisch verzerrten Schrägbild der Kamera operiert und nur einzelne Punkte perspektivisch transformiert, reduziert sich die Pixelbreite der Fahrspur mit zunehmendem Abstand. Insofern ist eine Reduktion der weißen Pixel durch den Erosionsfilter allein aus diesem Grund ungünstig für diese gewählte Art der Kantenidentifikation. Eine 2 cm breite Fahrbahnmarkierung im Abstand von 90 cm zum Fahrzeug entspricht bei dieser Kameraauflösung einer Pixelbreite von ca. 10 Pixeln, was die Iteration über einen Vierer-Block noch nicht beeinträchtigt.

Eine weitere Optimierungsoption für die Iteration in y-Richtung über die Pixelzeilen innerhalb der Funktion `dynamicROI()` besteht in der Konfiguration des Zeilenabstandes durch den Konfigurationsparameter `distRowsRoi`. Damit wird die Anzahl der Zeilen innerhalb des ROI festgelegt über die iteriert werden soll. Dies beeinflusst wiederum

die Anzahl der gefundenen weißen Pixel und somit die Anzahl der Punkte, die durch die Hough Transformation in die Hough Ebene übertragen werden. Somit beeinflusst der Konfigurationsparameter `distRowsRoi` die Anzahl der Schleifendurchgänge und die Latenzzeit beider Pipeline-Funktionen, wie in der folgenden Tabelle aufgelistet ist:

`distRowsRoi`	Latenzzeit dyn.ROI in ms	gefundene weiße Pixel	Latenzzeit HT in ms	Top-Kandidaten	Maximum HT
1	1,1	95	4,2	1	34
2	0,54	30	3	1	20
4	0,3	16	2	[1..2]	10
6	0,24	10	1,6	[1..4]	6
8	0,18	8	1,2	[1..4]	6
10	0,1	7	1,2	[1..6]	4

Tabelle 5.4.: Einfluss des Zeilenabstandes auf die Berechnungszeiten der Pipeline-Funktionen `dynamicROI()` und `houghTransformation()`

Je höher der in `distRowsRoi` festgelegte Abstand der Pixelzeilen ist, desto geringer fällt die Anzahl der Schleifendurchgänge in beiden Pipeline-Funktionen aus, da die Zahl der gefundenen weißen Pixel abnimmt. Die Latenzzeit für `dynamicROI()` verringert sich um 1 ms, die Latenzzeit der Hough Transformation um 4 ms, jeweils zwischen einem Zeilenabstand von 1 bis 10 Zeilen. Jedoch wird das Ergebnis der Hough Transformation mit Zunahme des Zeilenabstandes unschärfer, wie die letzten beiden Spalten zeigen. Die Top-Kandidaten sind die Punkte der Hough Ebene, welche mit dem gemeinsamen Schnittpunkt-Maximum der Hough Transformation belegt sind, was der Anzahl an Punkten entspricht, durch die die Gerade des Hough-Punktes mit den Koordinaten r und ϕ gelegt werden kann. Bei einem Zeilenabstand von 10 beispielsweise variiert die Zahl der Top-Kandidaten zwischen 1 und 6 mit einem Schnittpunkt-Maximum von 4. Das bedeutet, dass bis zu 6 Geraden durch jeweils 4 Punkte von insgesamt 7 gefundenen Punkten gelegt werden können. Mit Abnahme des Zeilenabstandes erhöht sich die Genauigkeit des Ergebnisses, indem die Anzahl der Top-Kandiaten abnimmt und das Schnittpunkt-Maxium zunimmt. Bei einem Zeilenabstand von einer Zeile kann eine Gerade durch 34 Punkte von insgesmat 95 gefundenen Punkten gelegt werden. Mit einem Aabstand von zwei Zeilen liegen 20 von insgesamt 30 Punkten auf einer Geraden.

Für die Abwägung des Verhältnisses zwischen optimierter Latenzzeit und Ergebnis-

schärfe spielt die insgesamt zur Verfügung stehende Zykluszeit pro Bildaufnahme die wichtigste Rolle. Steht genug Zeit zur Verfügung, so wird mit dem Zeilenabstand von einer Pixelzeile das präziseste Ergebnis erzielt. Verringert sich die Berechnungszeit etwa durch eine höhere Bildrate, so kann die Latenzzeit zweier Pipeline-Funktionen durch nur einen Konfigurationsparameter optimiert werden, solange das Ergebnis der Hough Transformation (vgl. Tab. 5.4), welches die Berechnungsgrundlage für den Lenkwinkel darstellt, den Toleranzbereich nicht übersteigt.

6. Zusammenfassung

Diese Arbeit dokumentiert den Entwurf und die Implementierung einer Fahrspurführungsapplikation in C unter Linux auf einem ARM Cortex A9 DualCore Prozessor. Sie dient zur Vorbereitung der Portierung einer Systemarchitektur des SAV Projekts für verschiedene Fahrassistenzsysteme auf die Zynq 7000 Plattform, die den ARM Prozessor mit einem FPGA-Baustein integriert. In dieser Arbeit wurde der ARM Prozessor innerhalb der OMAP 4430 Plattform von Texas Instruments als Vorbereitung für die Zynq Plattform eingesetzt, bis diese verfügbar ist. Das verwendete Betriebssystem Linaro ist eine für ARM spezialisierte Ubuntu Distribution mit Hardware Unterstützung für die OMAP Plattform durch die Auslagerung und Verwaltung von Multimediaverarbeitung auf Co-Prozessoren der OMAP Plattform durch das SYSLINK Framework im Linaro Kernel. Hauptbestandteil der Entwicklungsumgebung ist das ARM Development Studio (DS) 5, mit der die Applikation für die Zielplattform entwickelt wurde, und welche sowohl die OMAP als auch die Zynq Plattform untertützt. Die Entwicklung der Softwarelösung liefert für das HW/SW Co-Design auf der Zynq Plattform die Entscheidungsgrundlage für die Zuweisung von Funktionen auf den FPGA oder auf den ARM Prozessor und ermöglicht eine Aussage über die Leistungsfähigkeit des ARM Cortex A9 DualCore Prozessors bezüglich der Applikationsanforderungen der Fahrspurführung.

Die Applikation ist nach dem Entwurfsmuster einer fließbandartigen Verarbeitung eines Datenelements durch mehrere Pipeline-Funktionen gemäß der Verarbeitungskette der Fahrspurführung strukturiert. Durch die Verfügbarkeit von zwei CPUs wird die Pipeline in verschiedenen Multithreading Varianten jeweils nach einem Entwurfsmuster parallelisiert. Nach dem Entwurfsmuster der funktionalen Parallelität durch einen Thread pro CPU wird ein Datenelement innerhalb des Threads auf der CPU 1 durch die sequentielle Ausführung der zweiten Hälfte der Pipeline-Funktionen verarbeitet, während zeitgleich das nachfolgende Datenelement auf der CPU 0 durch die erste Hälfte der Pipeline-Funktionen bearbeitet wird. Beide Threads synchronisieren den gmeinsamen Zugriff auf dieselbe

Datenstruktur durch Mutexe und bauen die Verarbeitungsreihenfolge gemäß der Pipeline durch den Multithreading Mechanismus der Conditions-Variablen auf. Eine IP Kamera sendet über das RTP Protokoll einen MPEG-4 Stream mit einer Rate von 30 Bildern pro Sekunde, der durch das SYSLINK Framework auf dem IVA HD Beschleuniger der OMAP Plattform dekodiert wird und auf dem ARM Cortex A9 durch das GStreamer Multimediaverarbeitungs-Framework verwaltet wird. Der dekodierte Videostream wird in einzelne Bilder aufgeteilt, die jeweils in einer eigenen Datenstruktur gespeichert werden und jede Funktion der Pipeline sequentiell durchlaufen. Die Implementierung der Pipeline-Funtionen wird entweder durch das OpenCV Framework realisiert oder im Rahmen dieser Arbeit implementiert. Nachdem ein einzelnes Kamerabild aus dem GStreamer Puffer kopiert wird, wird die Pixelmenge von Graustufenwerten in Schwarz/Weiß Werte binarisiert, sodass im Bild ein eindeutiger Kantenübergang zwischen der weißen Fahrspurmarkierung und dem schwarzen Fahrbahnuntergrund hervorgehoben wird. Innerhalb des Bildes wird eine *Reqion of Interest* (ROI) mit konfigurierbarem Ursprung in Form eines Rechtecks mit Kanten in x-Richtung und y-Richtung des Bildkoordinatensystems um die Fahrspur gelegt und pro Kamerabild an deren Lagenänderung angepasst. Innerhalb des ROI wird pro Pixelzeile ein Bereich von mehreren benachbarten weißen Pixlen gesucht und daraus ein Punkt extrahiert. Anschließen wird nach dem Verfahren der Hough Transformation die Fahrspur-Gerade ermittelt, die durch die meisten gefundenen weißen Punkte verläuft. Die ermittelte Gerade ist die Grundlage für die Berechnung des Lenkwinkels durch zwei verschiedene Fahrspurführungsalgorithmen.

Für die Realisierung nach dem Multithreading Entwurfsmuster der datenabhängigken Parallelisierung innerhalb von Pipeline-Funktionen wurden die zwei schleifenintensivsten Pipeline-Funktionen mit vielen Iterationsschritten durch das Shared Memory Modell OpenMP parallelisiert. Jedoch ergab die Leistungsanalyse der parallelisierten Funktionen nicht den gewünschten Speedup, sodass dieses Entwurfsmuster und die Parallelisierung durch OpenMP für diese Anwendung nicht geeignet sind, da der Synchronisationsaufwand für die Verwaltung der parallelen Threads zu groß wird. Durch die Bildrate der Kamera ergibt sich pro Bild eine Verarbeitungszeit durch die gesamte Pipeline von 33 ms. Die gemessene Berechnungszeit der sequentiell ausgeführten Pipeline auf nur einer CPU ohne Multithreading bleibt mit 30 ms unterhalb der Obergrenze. Durch eine Reihe von Optimierungen innerhalb der Pipeline wird diese Zeit zusätzlich unterboten.

A. Installation der Entwicklungsumgebung

A.1. Installation von ARM DS-5

A.2. Instalation unter Windows

1. ARM DS-5 auf der ARM Products Seite unter `http://www.arm.com/products/tools/software-tools/ds-5/ds-5-downloads.php` downloaden.
2. Archiv entpacken und die Datei `setup.exe` ausführen.
3. Beim Start von `Eclipse for DS-5` den Workplace wählen
4. In Eclipse zur Workbench wechseln
5. Unter *Help - ARM License Manager* den Button *Add License Server* klicken.
6. Im nächsten Fenster die Checkbox *License Server* auswählen und Rechnername oder IP Adresse sowie Port des Lizenz-Servers eingeben und mit *Add* bestätigen. Im Zweifelsfall die Angaben aus der `SERVER` Zeile in `license.dat` verwenden.
7. Das grüne Häkchen bestätigt die erfolgreiche Lizenzsierung und Kommunikation mit dem Lizenz-Server.

A.3. Installation und Linux

1. Linux Version von ARM DS-5 auf der ARM Products Seite unter `http://www.arm.com/products/tools/software-tools/ds-5/ds-5-downloads.php` downloaden.

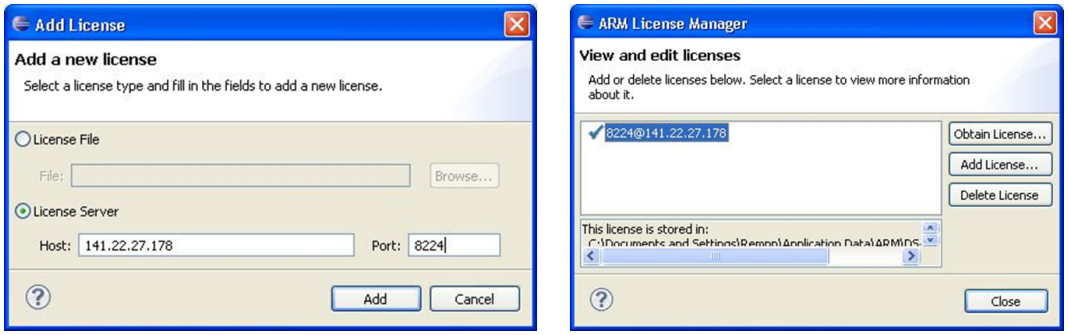

Abbildung A.1.: Lizenzierung von ARM DS-5

2. Archiv entpacken und ins entpackte Verzeichnis wechseln
3. Mit Admin-Rechten das Shellskript `install.sh` ausführen und Anweisungen folgen
4. ARM DS-5 wird standardmäßig in */usr/local/DS-5/* installiert.
5. Nach Installation die Linux-Umgebungsvariable `$PATH` um das Installationsverzeichnis von ARM DS-5 ergänzen, z.B. in `/home/<user>/.bashrc`:
   `$PATH=$PATH:/usr/local/DS-5/bin`
6. Nach Programmstart von *Eclipse for DS-5* wie unter Windows die Lizenzierungsschritte ausführen.
7. Die von ARM DS-5 verwendete GNU/ARM Toolchain befindet sich im Verzeichnis */usr/local/DS-5/bin/*.

A.4. Installation eines ARM Lizenz-Servers

Um das ARM Development Studio (ARM DS-5) nutzen zu können, muss wie bei allen ARM Tools zuvor eine valide Lizenzdatei installiert werden. Für das Lizenz Management benutzt ARM die Flexera Software FLEX*net*. ARM unterscheidet zwischen den folgenden beiden Licensing Schemata:

- Node-locked (*NL*) (Nur Windows)
- Floting-License (*FL*) (Windows und Linux)

Eine Node-locked Lizenz riegelt die Tools auf einem bestimmten, dedizierten Computer, welcher durch seine *Host ID* identifiziert wird. Somit eignet sich dieses Lizenzierungsschema am besten für eine Installation, die nur von einem Anwender auf einem Rechner benutzt wird. Die Host ID kann nicht geändert werden, ohne dass die zugehörige Lizenzdatei erneut von ARM angefordert und ersetzt wird. Das Floating-License Schema ist flexibler und wird typischerweise bei mehr Benutzern als erworbenen Lizenzen eingesetzt. Eine Floating-License Intallation besteht aus einem Lizenz-Server und mehreren Client-Rechnern, auf denen die ARM Entwicklungswerkzeuge installiert sind und ausgeführt werden. Die Lizenz-Server Software kontrolliert die Anzahl der Kopien der Lizenzdatei, die von von den Clients angefordert werden, um die auf den Clients installierten ARM Tools auszuführen.

A.5. Generierung der Lizenzdatei

In diesem Projekt wurden im November 2011 für den Einsatz von ARM DS-5 eine Floating-License und eine Node-locked Lizenz erworben, jeweils mit einer Gültigkeitsdauer von einem Jahr. Die beiden Lizenz-Zertifikate, in Form PDF Dokumenten enthalten jeweils eine eigene Seriennummer, mit welcher die Lizenzdatei auf der Licensing Seite von ARM unter https://silver.arm.com/licensing/generate.tm generiert werden muss. Eine komplette Anleitung zur Generierung der Lizenzdatei ist unter http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0065d/I475327.html ersichtlich. Das Verwenden der IP Adresse statt des Rechnernamens für die Host-ID ist nicht ratsam, wenn die beteiligten Rechner ihre IP Adresse per DHCP beziehen.

Der Lizenz Server wurde zunächst auf einem Labor-Notebook unter Windows installiert. Wegen Hardwareschaden und eingeschränkter Verfügbarkeit wurde der Lizenz Server auf eine Server Maschine der Hochschule unter Linux umgezogen. Deshalb wird im folgenden die Installation des Lizenz Servers sowohl in Windows als auch in Linux beschrieben.

A.6. Installation unter Windows

Die Dateien der FLEX*net* Server Software befinden sich im Installationsverzeichnis von ARM DS-5 (im Folgenden <*installDir_armDS5*> genannt), nachdem es installiert wurde.

1. Ein selbst zu wählendes Installationsverzeichnis für die FLEX*net* Server Software erstellen. (Im folgenden <*installDir_flexnet*> genannt)
2. Die FLEX*net* Utils von <*installDir_armDS5*>*sw**FLEXnet_11.9.1.1**windows-32bit* nach <*installDir_flexnet*> kopieren.
3. Lizenzdate `license.dat` ebenfalls nach <*installDir_flexnet*> kopieren und überprüfen, ob die `SERVER` Zeile entsprechend richtig editiert wurde (vgl. Abschnitt A.5).
4. Das Verzeichnis <*installDir_flexnet*> der Windows-Umgebungsvariablen `PATH` hinzufügen.
5. Die ausführbare Datei `lmtools.exe` starten. Sie ist die Administrations- und Konfigurationsoberfläche für den Lizenz Server.
6. Im Tab *Service/License File* die Option *Configuration using Services* auswählen.
7. Im Tab *Config Services* die Pfade zu den jeweiligen Dateien eintragen und mit *Save Service* speichern.
8. Im Tab *Start/Stop/Reread* wird der Server gestartet bzw. beendet.

A.7. Installation unter Linux

Zu Beginn des SoSe 2012 wurde der Linux-Server *tiserver-l.cpt.haw-hamburg.de* für dieses Projekt und die Installation des Lizenz Servers zur Verfügung gestellt. Der Zugriff auf den Linux-Server erfolgt über SSH und erfordert einen gültigen User-Account. Da der Lizenz Server zuvor mit derselben Seriennummer für die FL Lizenz installiert wurde, ist bei einer erneuten Installation auf einem anderen Rechner ein Re-Hosting der FL Lizenz notwendig (vgl. Abschnitt A.8).
Unter Linux sind folgende Installationsschritte notwendig:

1. FLEX*net* Utils im Verzeichnis *<installDir_armDS5>\sw\FLEXnet_11.9.1.1\redhat-enterprise-32bit* zusammen mit der neuen Lizenzdatei `license.dat` z.B. via SCP in ein Verzeichnis auf dem Linux-Server kopieren.
2. In diesem Fall wurden die Utils in das Verzeichnis */ti/usr/Rempp/bin* kopiert, da es schon in der Linux-Umgebungsvariable $PATH enthalten ist, was notwendig ist, und der User *Rempp* nur innerhalb seines *Home* Verzeichnisses Installationsrechte besitzt.
3. in dieses Verzeichnis wechseln und das Shellskript `makelinks.sh` ausführen.
4. Server wird gestartet mit: `nohup lmgrd -c license.dat -l logfile`
5. Server wird gestoppt mit: `lmutil lmdown -q -c license.dat`

A.8. Re-Hosting und Troubleshooting

Damit ein Umzug der Server Software vom Labor-Notebook zum Linux-Server erfolgreich durchgeführt werden kann, muss zuvor eine neue Lizenz-Datei von ARM mit der Host ID des neuen Rechners angefordert werden. Dies erfolgt über die Funktion *Re-Host* auf der ARM Licensing Seite unter https://silver.arm.com/licensing/rehost.tm. Bei eventuellen Problemen bei der Lizenzverwaltung auf Seiten von Server oder Client werden Lösungen zu einigen Problemfällen unter http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0065d/I475327.html angeboten. Eine komplette Dokumentation zum ARM Lizenzierungsverfahren ist unter http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0065d/I475327.html ersichtlich.

B. Inbetriebnahme und Bootvorgang

B.1. Installation des Linaro Betriebssystems auf der SD Karte

Im Rahmen dieser Arbeit wurde ein vorkompiliertes Linaro Image auf den Host-PC heruntergeladen und zusammen mit einem Hardware Support Package auf die SD Karte installiert, von der aus die Zielplattform bootet. Im Folgenden werden die einzelnen Installationsschritte hierzu einzeln aufgeführt:

- Download der Linaro Buildwerkzeuge

```
$ sudo apt-repositroy ppa:linaro-maintainers/tools
sudo apt-get update
sudo apt-get install linaro-image-tools
```

- Download des neuesten vorkompilierten Linaro Images für die OMAP Plattform auf dem Pandaboard vom Linaro Repository: http://snapshots.linaro.org/ubuntu/pre-built/panda/latest.
- Download des neuesten Hardware Support Packages (`HWPACK`), welches das SYS-LINK Framework und den Gator Treiber enthält: http://snapshots.linaro.org/ubuntu/hwpacks/panda/latest
- Einstecken der SD Karte in einen SD Kartenleser oder Adapter auf dem Host-PC und Identifikation des richtigen Gerätenamens, z.B. `/dev/sdb`, nicht der Partitionsname `/dev/sdb1`.

- Im gleichen Verzeichnis in das das gezippte Linaro Image und das HWPACK heruntergeladen wurden, folgenden Befehl zur Installation des Betriebssystems auf der SD Karte ausführen:

```
$ sudo linaro-media-create -rootfs ext3 -mmc /dev/sdb -binary
    <linaro-image>.tar.gz -hwpack <hwpack>.tar.gz -dev PANDA
```

B.2. Boot-Vorgang

Die Entwicklungs- und Installationsschritte im vorherigen Kapitel liefern eine bootfähige SD Karte mit einem Kernel Image und der Ubuntu Umgebung innerhalb des Filesystems. In diesem Kapitel wird nun der Bootvorgang von der SD Karte beschrieben. Erläutert werden zunächst die Bootloader für das 1.st Level und 2.nd Level Booting, sowie deren Funktionen innerhalb des Bootvorgangs. Daraufhin folgt die Beschreibung der Schnittstelle zur Steuerung des Bootens aus der Entwicklungsumgebung auf dem Host PC heraus.

Das Booten des Linux Kernels von der SD Karte hat den Vorteil, dass die OMAP Plattform für den Bootvorgang nicht an ein Netzwerk und Netzwerkdienste wie einen GDB-Server für das Laden des Kernels gebunden ist. Somit kann die Mobilität des Fahrspurführungssystems erhalten bleiben. Nachdem Linux auf die SD Karte gebracht wurde existieren auf der ersten FAT32-Partition der SD Karte folgende Dateien:

- `MLO` (X-Loader von Texas Instruments[40])
- `u-boot.bin` (U-Boot Boadloader [27])
- `uImage` (Linux Kernel Image)
- `boot.scr` (Boot Skript)

Beim Systemstart (Power-Up) bootet die Plattform nach Ausführung des Reset Vektors vom internen 48KB Boot ROM des ARM Cortex-A9 Prozessors (vgl. Abbildung 3.1) auf der CPU 0. Dieser Boot Code im ROM wurde während der Herstellung des Prozessors fest angebracht und kann nicht verändert werden. Der Boot ROM sucht nach dem ersten externen Bootloader. Die Bootreihenfolge beinhaltet NAND-Flash, UART (Serielle Schnittstelle) und zuletzt SD/MMC Card. Beim Booten von der SD Karte, wird der X-Loader (`MLO`) auf der ersten FAT32-Partition gefunden und in das On-Chip SRAM des OMAP Prozessorsystems geladen und ausgeführt (vgl Abbildung B.1). Der

X-Loader ist klein genug, dass er in das 56KB On-Chip SRAM passt, ohne dass externe Speicherinitialisierung notwendig ist. Er initialisiert den Off-Chip SDRAM, um somit genügend Speicherplatz für die Ausführung des zweiten Bootloaders U-Boot zu schaffen. Der X-Loader lädt nun U-Boot ebenfalls von der ersten Partition in den initialisierten

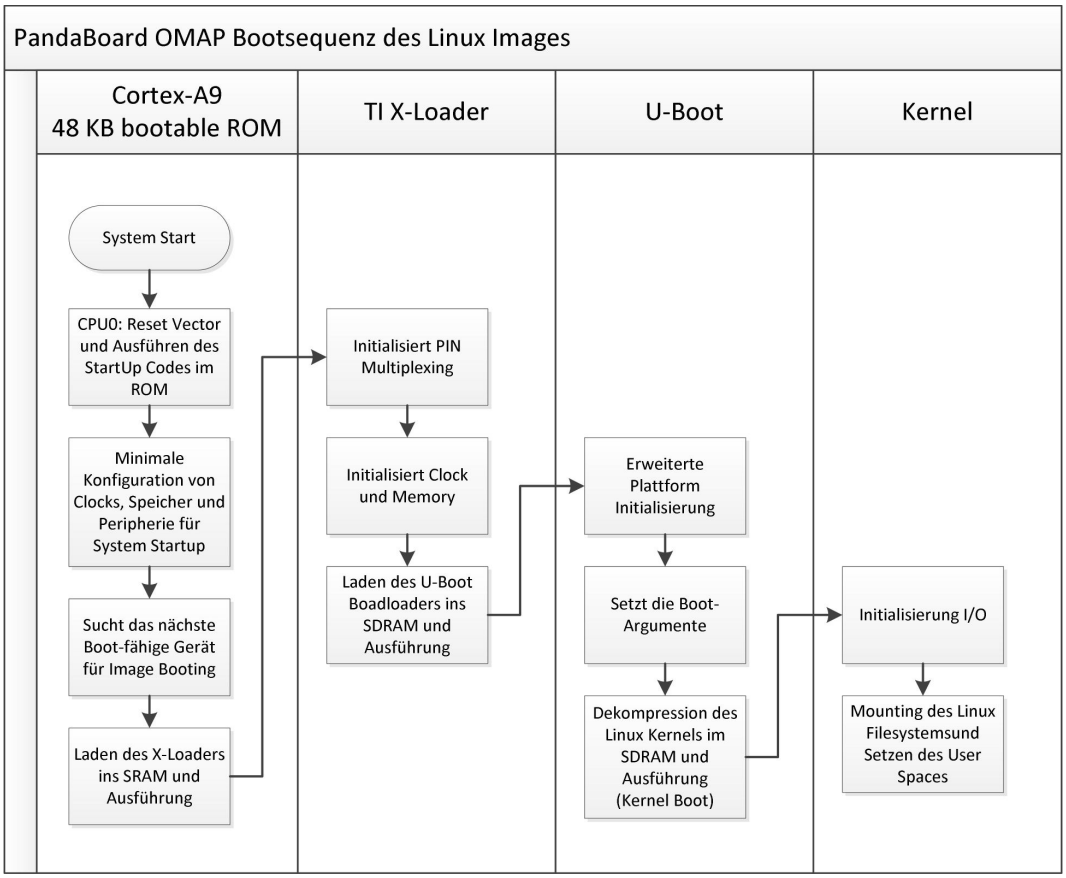

Abbildung B.1.: OMAP 4430 Bootsequenz von der SD Karte

Hauptspeicher und lässt ihn ausführen. Die Hauptaufgabe von U-Boot besteht darin, den Linux Kernel zu booten und ihn mit Informationen aus dem Linux Filesystem auszustatten. U-Boot kann so konfiguriert werden, den Linux Kernel von verschiedenen Speichermedien zu laden, entweder von NAND-Flash, SD/MMC Card, über eine serielle Schnittstelle (UART3) oder über Ethernet via TFTP. Hier wird der Linux Kernel ebenfalls aus der ersten Partition der SD-Karte geholt wird. U-Boot startet nun den gefundenen Kernel (`uImage`) ebenfalls im SDRAM. Der Kernel mountet nun das Linux Filesystem, dessen Pfad er von U-Boot per Bootargumente übergeben bekommen hat. U-Boot kann ein Root Filesystem spezifizieren, dass in NAND-Flash, SRAM (ramdisk), SD/MMC Card (2. ext3-Partition) oder über IP (NFS) lokalisiert ist.

Der Bootvorgang des vorkompilierten Ubuntu-Desktop Images erfolgt automatisch sobald das PandaBoard über sein Netzteil mit Strom versorgt wird. Zuvor werden ein Bildschirm über HDMI, Maus und Tastatur über USB, sowie die SD Karte in den Kartenleser auf dem Pandaboard eingesteckt (vgl. Abbildung B.2). Bei erstmaligem Booten werden einige abschließende Installations- und Konfigurationsschritte von Ubuntu vorgenommen, wie z.B. Anlegen des Benutzer-Accounts.

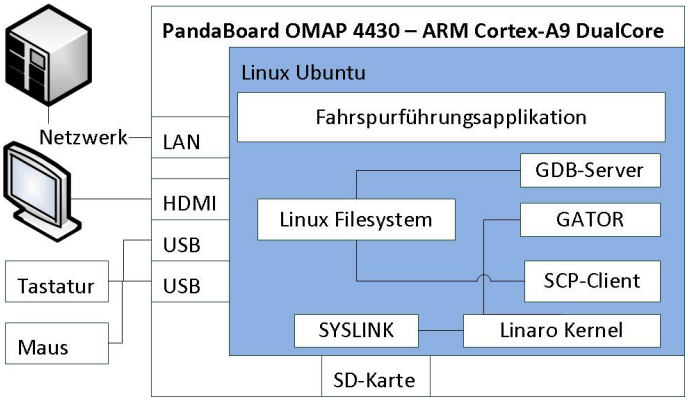

Abbildung B.2.: Linaro Image auf der OMAP Plattform mit I/O Peripherie

Der automatische Bootvorgang kann über die serielle Schnittstelle mitverfolgt, bzw. unterbrochen werden, da U-Boot dem Benutzter einige Sekunden Zeit lässt, den automatischen Bootvorgang zu unterbrechen, bevor dieser fortgesetzt wird. Durch Unterbrechung und manuelle Fortsetzung können z.B. weitere Boot-Argumente an U-Boot übergeben werden. Ansonsten benutzt U-Boot den Boot-Befehl inklusive Boot-Argumente aus dem Bootskript `boot.scr`. Die Überwachung des Bootvorgangs vom Host-PC aus erfolgt in

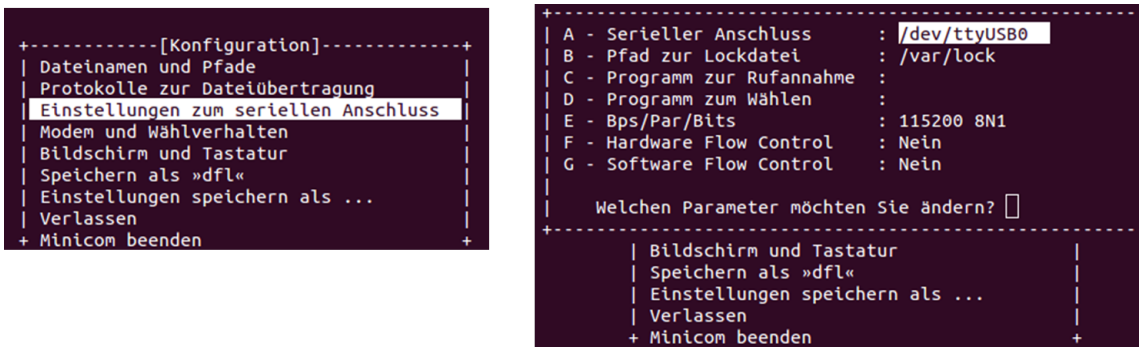

Abbildung B.3.: Konfiguration der Terminal-Sitzung in Minicom

Linux über das Terminal `minicom`. Im Folgenden werden die einzelnen Schritte zum Aufbau dieser Schnittstelle beschrieben:

1. USB-2-Serial Kabel zwischen Entwicklungs-PC und OMAP Plattform verbinden.
2. Auf dem Entwicklungs-PC unter Linux das Programm `minicom` starten:
   ```
   sudo minicom -s
   ```
3. Im angezeigten Konfigurations-Menü mit den Pfeiltasten zu *Einstellungen zum seriellen Anschluss* wechseln und mit *Enter* bestätigen.
4. Im nächsten Menü überprüfen, ob das richtige Gerät für den USB-2-Serial Anschluss gewählt wurden, in diesem Fall `/dev/ttyUSB0`. Bestätigen mit Enter.
5. Zurück im alten Menü zu *Verlassen* wechseln.
6. Nachdem das Terminal-Modem initialisiert wurde, das Pandaboard mit dem Strom-Netzteil verbinden.

Nachdem der Kernel gebootet wurde, kann der Login des Linux Benutzers auf der Zielpalttform via SSH erfolgen.

B.3. Netzwerkverbindung auf der Zielplattform

Die Kommunikation zwischen dem DS-5 Debugger über den *Remote System Explorer* erfordert eine Netzwerkanbindung in Linux auf der Zielplattform. Als einfachste Netzwerkverbindung hat sich während dieser Projektarbeit DHCP über das HAW Netzwerk herausgestellt, da es auf dem Host-PC keine zusätzlichen Netzwerkkonfigurationen erfordert. Bei Einsatz von Linux mit grafischer Oberfläche kann das Netzwerk-Interface auf dem Ubuntu Desktop mit DHCP konfiguriert und neugestartet werden.

Wird Linaro oder eine *Headless* Kernel Version eingesetzt, erfolgt die Konfiguration des Netzwerk-Interfaces bei erstmaligem Booten von Linux auf der OMAP Plattform über die serielle SSH-Schnittstelle manuell:

1. Config-Datei des Netzwerk-Interfaces (`eth0`) mit einem Editor (`vi`) öffnen:
   ```
   sudo vi /etc/network/interfaces
   ```

2. Die Zeile beginnend mit `auto` und die Konfigurationszeile für `eth0` folgendermaßen editieren: `auto eth0` (eventuell `auto lo` durch `auto eth0` ersetzen)
   ```
   iface eth0 inet dhcp
   ```
3. Datei speichern und Netzwerk-Interface neustarten mit:
   ```
   sudo /etc/init.d/networking restart
   ```

Das Netzwerk-Kabel kann auch schon beim Booten eingesteckt werden, es beeinflusst die Boot-Reihenfolge nicht. Jedoch dauert der Bootvorgang des Linux Kernels länger, weil das Netzwerk-Interface mit DHCP konfiguriert wird.

```
1  Texas Instruments X-Loader 1.5.0 (Apr 11 2011 - 09:48:22)
2  Reading boot sector
3  Loading u-boot.bin from mmc
4
5  U-Boot 2011.03 (Apr 20 2011 - 07:37:43)
6  CPU  : OMAP4430
7  Board: OMAP4 Panda
8  I2C:   ready
9  DRAM:  1 GiB
10 MMC:   OMAP SD/MMC: 0
11
12 In:    serial
13 Out:   serial
14 Err:   serial
15 reading boot.scr
16 350 bytes read
17 Running bootscript from mmc0 ...
18 ## Executing script at 82000000
19
20 reading uImage
21 4004992 bytes read
22
23 reading uInitrd
24 8302246 bytes read
25 ## Booting kernel from Legacy Image at 80000000 ...
26    Image Name:   Linux-3.1.5+
27    Image Type:   ARM Linux Kernel Image (uncompressed)
28    Data Size:    4004928 Bytes = 3.8 MiB
29    Load Address: 80008000
30    Entry Point:  80008000
31    Verifying Checksum ... OK
32 ## Loading init Ramdisk from Legacy Image at 81600000 ...
33    Image Name:   Ubuntu Initrd
34    Image Type:   ARM Linux RAMDisk Image (uncompressed)
35    Data Size:    8302182 Bytes = 7.9 MiB
36    Load Address: 00000000
37    Entry Point:  00000000
38    Verifying Checksum ... OK
39    Loading Kernel Image ... OK
40
41 Starting kernel ...
42 Uncompressing Linux... done, booting the kernel.
43 Ubuntu 11.04 PandaboardUbuntu ttyO2
44 PandaboardUbuntu login:
```

Abbildung B.4.: Ausgabe der Bootloader über die serielle Schnittstelle

C. Projektstruktur, Erstellung und Debugging

Die hierarchische Ordnerstruktur (vgl. Abb. C.1) gliedert sich in die Verzeichnisse `bin`, `include` und `src`. Im Verzeichnis `bin` befindet sich die Datei `params.config` mit den Konfigurationsparametern sowie die kompilerte ausführbare Datei `laneSystem` der Fahrspurführungsapplikation. Die zwei anderen Verzeichnisse enthalten die Quellcode-Dateien, die in Headerdateien und Source-Dateien getrennt sind. Innerhalb beider Quellcode-Verzeichnisse werden die Dateien zusätzlich in die Verzeichnisse `util` und `processes` unterteilt. Die Dateien im Verzeichnis `processes` befinden sich in der Softwarearchitektur auf den Ebenen *Threads* und *Pipeline Funktionen*. Die Dateien in den `util` Verzeichnissen gehören zur Ebene *Infrastruktur*.

Im `Makefile` sind alle Regeln zur Kompilierung der Applikation zusammengefasst. Übersetzt wird das Projekt durch den Befehl `make` im obersten Projektverzeichnis. Jedes Modul wird zunächst zu einer Object-Datei (.o) compiliert. Anschließend werden alle Object-Dateien zu einer ausführbaren Datei kompiliert, die nach erfolgreichem Erstellen in das `bin` Verzeichnis verschoben wird. Die Makefile Variable `$CC` gibt den Compiler an, mit dem kompiliert werden soll. Alle zusätzlichen Compilerflags werden in der Variable `$CFLAGS` zusammengefasst, alle verwendeten Bibliotheken in der Variablen `$LDFLAGS`.

In der Debug Konfiguration für das Zielsystem wird unter RSE Connection entweder die IP Adresse der Zielplattform, oder die gespeicherte Verbindung zur Zielplattform über den Remote Systems Explorer angegeben. Als Debugging Plattform wird der GDB-Server ausgewählt. Die Debug Operation besteht somit aus dem Starten des GDB-Servers und Debuggen der auf der Zielplattform existierenden Fahrspurführungsapplikation.

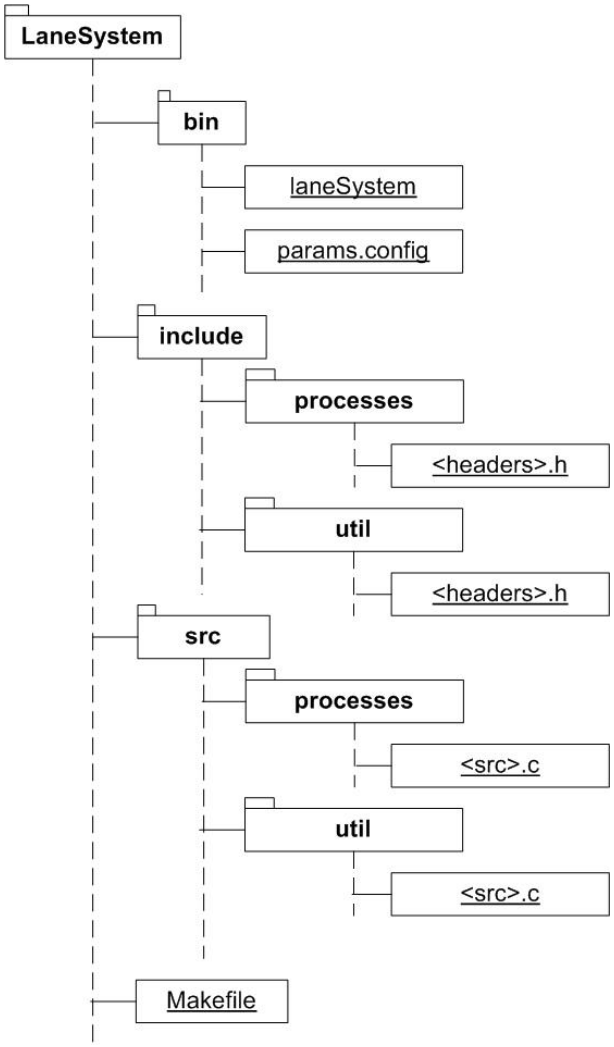

Abbildung C.1.: Odnerstruktur der Fahrspurführungsapplikation

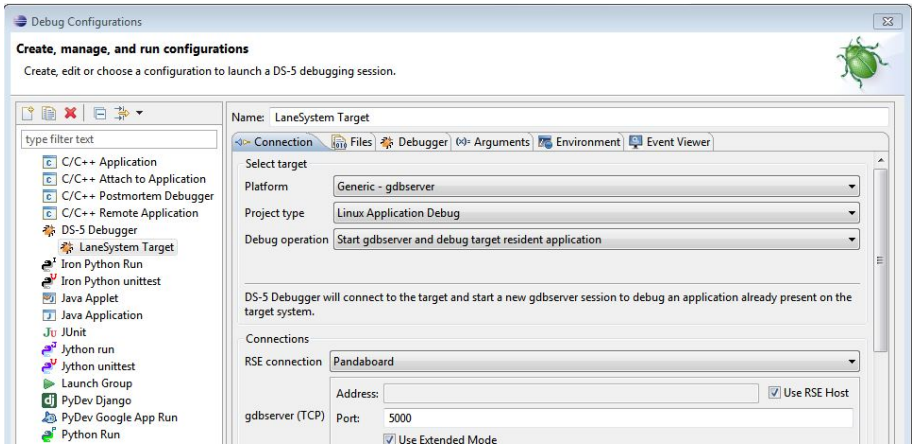

Abbildung C.2.: Debug Konfiguration, Verbindung und Operation

Innerhalb der Debug Konfiguration wird der Pfad zur Applikation im Dateisystem auf der Zielplattform angegeben. Diese wird nach jedem Erstellen vor einer Debug-Session per SCP auf die Zielplattform kopiert. Die Debug Symbole werden von der erstellen Datei auf dem Host-PC geladen.

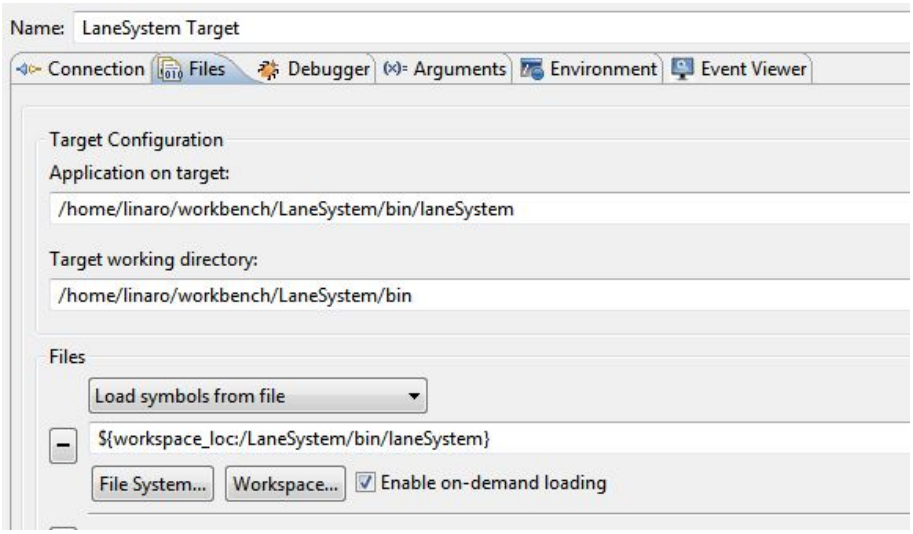

Abbildung C.3.: Debug Konfiguration, Dateien und Debug Symbole

Literaturverzeichnis

[1] ALTERA: *Alteras Cyclone FPGA*. 2012. – URL http://www.altera.com/devices/fpga/cyclone/cyc-index.jsp

[2] Andresen, Erik: *Dekodierung von MPEG4-Videos mit Hardware-Beschleunigern unter Verwendung des AMP Frameworks RPMsg und Multimedia-Framework GStreamer*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelor Thesis, 2012

[3] ARM, Inc.: *ARM Application Binary Interface*. 2012. – URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036b/IHI0036B_bsabi.pdf

[4] ARM, Inc.: *ARM Compiler Toolchain*. 2012. – URL http://www.arm.com/products/tools/software-tools/rvds/updates-patches/arm-compiler-41-patch.php

[5] ARM, Inc.: *ARM Development Studio 5 (DS-5)*. 2012. – URL http://www.arm.com/products/tools/software-tools/ds-5/index.php

[6] ARM, Inc.: *ARM DS-5 Debugger*. 2012. – URL http://www.arm.com/products/tools/software-tools/ds-5/debugger/index.php

[7] ARM, Inc.: *ARM DSTREAM Debug and Trace Unit*. 2012. – URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0481f/CHDCJEFH.html

[8] ARM, Inc.: *Setting up the GATOR Deamon*. 2012. – URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0482b/BABEJAAI.html

[9] Barr, Michael: *Embedded Systems Glossary*. 2012. – URL http://www.netrino.com/Embedded-Systems/Glossary/index.php

[10] Bovet, D.: *Understanding the Linux Kernel*. 3. Edition. O'Reilly, 2006

[11] Bovet, Daniel ; Cesati, Marco: *Understanding the Linux Kernel*. 3. Edition. O'Reilly, 2006

[12] Foundation, Open M.: *The Open MP API specification for parallel programming*. Website. 2012. – URL http://openmp.org/wp/. – Zugriffsdatum: 20.08.2012

[13] freedesktop: *GStreamer open source multimedia framework*. 2012. – URL http://gstreamer.freedesktop.org/

[14] Gary, Bradski ; Adrina, Kaehler: *Learning OpenCV*. 1. Edition. O'Reilly, 2008

[15] Gleim, Urs ; Schüle, Tobias: *Multicore-Software*. d.punkt Verlag, 2012

[16] Intel ; Garage, Willow: *OpenCV open source Computer Vision*. 2012. – URL http://gstreamer.freedesktop.org/

[17] Jalier, C. ; Lattard, D.: *Heterogeneous vs Homogeneous MPSoC Approaches for a Mobile LTE Modem*. 2010. – URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5457213

[18] Jerraya, A. ; Wolf, M.: *Multiprocessor Systems-on-Chip*. Morgan Kaufmann Publishers, 2005

[19] Jestel, Andre: *Software-Partitionierung einer Laserscanner-basierten Objekterkennung auf einer MPSoC-Plattform mit Android*, Hochschule für Angewandte Wissenschaften Hamburg, Masterthesis, 2013

[20] Khatib, I. ; Poletti, F.: *A multiprocessor system-on-chip for real-time biomedical monitoring and analysis: architectural design space exploration*. 2006. – URL http://dl.acm.org/citation.cfm?id=1146909.1146947&coll=DL&dl=ACM

[21] Kirschke, Marko: *FPGA-basierte MPSoC-Plattform zur Integration eines Antikollisionssystems in die Fahrspurführung eines autonomen Fahrzeugs*, Hochschule für Angewandte Wissenschaften Hamburg, Masterarbeit, 2012

[22] Kopetz, H.: *Real-Time Systems, Design Principles for Distributed Embedded Applications*. 2. Edition. Springer, 2011

[23] Linaro, Project: *Linaro Kernel for ARM Linux*. 2012. – URL http://www.linaro.org/

[24] Märtin, C: *Rechnerarchitekturen - CPUs, Systeme, Software-Schnittstellen*. Fachbuchverlag Leipzig, 2001

[25] Mentor, Graphics: *GNU ARM Toolchain Mentor Graphics*. 2012. – URL http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/

[26] Nischwitz, Alfret ; Fischer, Max ; Haberaecker, Peter ; Socher, Gudrun: *Computergrafik und Bildverarbeitung: Band 1*. 3. Auflage. Vieweg + Teubner, 2011

[27] OMAPPedia, Project: *Bootloader Project*. 2012. – URL http://omappedia.org/wiki/Bootloader_Project

[28] OMAPPedia, Project: *OMAP Ducati Subsystem*. 2012. – URL http://omappedia.org/wiki/Ducati_For_Dummies

[29] OMAPPedia, Project: *OMAP pre-comiled Ubuntu Image*. 2012. – URL http://www.ti.com/tool/dspbios

[30] Pandaboard: *Intruducing BandaBoard based on OMAP 4430 mobile processor*. – URL http://pandaboard.org/

[31] Popovici, K. ; Rousseau, F. ; Jerraya, A. ; Wolf, M.: *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. Springer, 2010

[32] Rauber, Thomas ; Rünger, Gudula: *Parallele Programmierung*. 2. Edition. Springer, 2007

[33] Rempp, Steffen: *Inbetriebnahme einer ARM-basierten MPSoC-Plattform für Bildverarbeitungsanwendungen*, Hochschule für Angewandte Wissenschaften Hamburg, Projektbericht, 2012

[34] Riemenschneider, F.: *(Fast) eineiige Zwillinge - Alteras Antwort auf Zynq*. elektroniknet. 2012. – URL http://www.elektroniknet.de/bauelemente/technik-know-how/halbleiter/article/88868/0/Fast_eineiige_Zwillinge_-_Alteras_Antwort_auf_Zynq/

[35] Salathe, Jasper: *Integration einer CMOS-Kamera mit parallelem Interface in ein System-on-Chip basiertes Spurführungssystem*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelor Thesis, 2012

[36] Schneider, Christian: *Ein System-on-Chip-basiertes Fahrspurführugnssystem*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelor Thesis, 2011

[37] Shabbir, A. ; Stuijk, S. ; Kumar, A. ; Corporall, H. ; Mesman, B.: *An MPSoC design approach for multiple use-cases of throughput constrainted applications*. 2011. – URL http://dl.acm.org/citation.cfm?id=2016604.2016628&coll=DL&dl=ACM&CFID=97826098&CFTOKEN=46856378

[38] Stallings, W.: *Computer Organization and Architecture - Designing for Performance*. 8. Edition. Pearson, 2010

[39] Syslink, Project: *Syslink Project for OMAP*. 2012. – URL http://omappedia.org/wiki/Syslink_Project

[40] TI, Inc.: *Boot Sequence of TI processors*. 2012. – URL http://processors.wiki.ti.com/index.php/Boot_Sequence

[41] TI, Inc.: *DSP/BIOS Real-Time Operating System (RTOS)*. 2012. – URL http://www.ti.com/tool/dspbios

[42] TI, Inc.: *OMAP 4430 Technical Reference Manual*. 2012. – URL http://info.arteris.com/blog/bid/64282/TI-OMAP4430-and-OMAP4460-TRM-Technical-Reference-Manual-Update. – Page 325

[43] TI, Inc.: *Open Multimedia Application Platform 4430*. 2012. – URL http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12843&contentId=53243

[44] Wolf, Juergen: *Linux-UNIX-Programmierung*. 2. Auflage. Gallileo Computing, 2006

[45] XILINX: *XILINX Zynq 7000 Extensible Processing Platform*. 2012. – URL http://www.xilinx.com/publications/prod_mktg/zynq7000/Product-Brief.pdf

[46] Yaghmour, K.: *Building Embedded Linux Systems*. O'Reilly, 2003

[47] Yaghmour, Kaled: *Building Embedded Linux Systems*. O'Reilly, 2003

[48] Yiu, Joseph ; Frame, Andrew: *32-Bit Microcontroller Code Size Analysis*. 2013. – URL http://www.arm.com/files/pdf/ARM_Microcontroller_Code_Size_(full).pdf

[49] Youssef, M. ; Yoo, S. ; Jerraya, A. ; Sasongko, A. ; Paviot, Y: *Debugging HW/SW Interface for MPSoC: Video Encode System Design Case Study*. DAC 2004, San Diego, 2004

[50] Yue, H. ; Qian, D.: *Design of an Architecture for Multiprocessor System-on-Chip (MPSoC)*. 2006. – URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1707566

[51] ZVEI: *Nationale Roadmap Embedded Systems*. 2010. – URL http://www.bitkom.org/files/documents/NRMES_2009_einseitig.pdf

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 17. Mai 2013 Steffen Rempp