Institut für Informatik
Lehrstuhl für Technische Informatik

Entwurf, Methoden und Werkzeuge für komplexe Bildverarbeitungssysteme auf Rekonfigurierbaren System-on-Chip-Architekturen

Dissertation
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

in der Wissenschaftsdisziplin
„Technische Informatik“

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von

Felix Mühlbauer

Potsdam, den 16.11.2011

Betreuer: Prof. Dr. rer. nat. Christophe Bobda

Dieses Werk ist unter einem Creative Commons Lizenzvertrag lizenziert:

Namensnennung - Keine kommerzielle Nutzung - Weitergabe unter gleichen Bedingungen 3.0 Deutschland

Um die Bedingungen der Lizenz einzusehen, folgen Sie bitte dem Hyperlink:
http://creativecommons.org/licenses/by-nc-sa/3.0/de/

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URL: http://opus.kobv.de/ubp/volltexte/2012/5992/
URN: urn:nbn:de:kobv:517-opus-59923
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-59923

Zusammenfassung

Bildverarbeitungsanwendungen stellen besondere Ansprüche an das ausführende Rechensystem. Einerseits ist eine hohe Rechenleistung erforderlich. Andererseits ist eine hohe Flexibilität von Vorteil, da die Entwicklung tendentiell ein experimenteller und interaktiver Prozess ist.

Für neue Anwendungen tendieren Entwickler dazu, eine Rechenarchitektur zu wählen, die sie gut kennen, anstatt eine Architektur einzusetzen, die am besten zur Anwendung passt. Bildverarbeitungsalgorithmen sind inhärent parallel, doch herkömmliche bildverarbeitende eingebettete Systeme basieren meist auf sequentiell arbeitenden Prozessoren. Im Gegensatz zu dieser *Unstimmigkeit* können hocheffiziente Systeme aus einer gezielten Synergie aus Software- und Hardwarekomponenten aufgebaut werden. Die Konstruktion solcher System ist jedoch komplex und viele Lösungen, wie z. B. grobgranulare Architekturen oder anwendungsspezifische Programmiersprachen, sind oft zu akademisch für einen Einsatz in der Wirtschaft.

Die vorliegende Arbeit soll ein Beitrag dazu leisten, die Komplexität von Hardware-Software-Systemen zu reduzieren und damit die Entwicklung hochperformanter on-Chip-Systeme im Bereich Bildverarbeitung zu vereinfachen und wirtschaftlicher zu machen. Dabei wurde Wert darauf gelegt, den Aufwand für Einarbeitung, Entwicklung als auch Erweiterungen gering zu halten. Es wurde ein Entwurfsfluss konzipiert und umgesetzt, welcher es dem Softwareentwickler ermöglicht, Berechnungen durch Hardwarekomponenten zu beschleunigen und das zu Grunde liegende eingebettete System komplett zu prototypisieren. Hierbei werden komplexe Bildverarbeitungsanwendungen betrachtet, welche ein Betriebssystem erfordern, wie z. B. verteilte Kamerasensornetzwerke. Die eingesetzte Software basiert auf Linux und der Bildverarbeitungsbibliothek OpenCV. Die Verteilung der Berechnungen auf Software- und Hardwarekomponenten und die daraus resultierende Ablaufplanung und Generierung der Rechenarchitektur erfolgt automatisch. Mittels einer auf der Antwortmengenprogrammierung basierten Entwurfsraumexploration ergeben sich Vorteile bei der Modellierung und Erweiterung. Die Systemsoftware wird mit OpenEmbedded/Bitbake synthetisiert und die erzeugten on-Chip-Architekturen auf FPGAs realisiert.

Abstract

Image processing applications have special requirements to the executing computational system. On the one hand a high computational power is necessary. On the other hand a high flexibility is an advantage because the development tends to be an experimental and interactive process.

For new applications the developer tend to choose a computational architecture which they know well instead of using that one which fits best to the application. Image processing algorithms are inherently parallel while common image processing systems are mostly based on sequentially operating processors. In contrast to this *mismatch*, highly efficient systems can be setup of a directed synergy of software and hardware components. However, the construction of such systems is complex and lots of solutions, like gross-grained architectures or application specific programming languages, are often too academic for the usage in commerce.

The present work should contribute to reduce the complexity of hardware-software-systems and thus increase the economy of and simplify the development of high-performance on-chip systems in the domain of image processing. In doing so, a value was set on keeping the effort low on making familiar to the topic, on development and also extensions. A design flow was developed and implemented which allows the software developer to accelerate calculations with hardware components and to prototype the whole embedded system. Here complex image processing systems, like distributed camera sensor networks, are examined which need an operating system. The used software is based upon Linux and the image processing library OpenCV.

The distribution of the calculations to software and hardware components and the resulting scheduling and generation of architectures is done automatically. The design space exploration is based on answer set programming which involves advantages for modelling in terms of simplicity and extensions. The software is synthesized with the help of OpenEmbedded/Bitbake and the generated on-chip architectures are implemented on FPGAs.

Inhaltsverzeichnis

Abbildungsverzeichnis vi

Tabellenverzeichnis vii

1 Einleitung 8

1.1 Problemstellung 10
1.2 These und wissenschaftliche Fragestellungen 11
1.3 Zielsetzung und Einschränkungen 13
1.4 Übersicht 14

2 Grundlagen und verwandte Arbeiten 15

2.1 Umsetzung von Bildverarbeitungssystemen 15
2.2 Formaler Entwurfsfluss für Hardware-Software-Systeme 20
2.3 Entwurf von On-Chip-Systemen 21
2.4 Verwandte Arbeiten 27
2.5 Diskussion 30

3 Entwurf von komplexen Bildverarbeitungssystemen 33

3.1 Eigenschaften von Bildverarbeitungsalgorithmen 35
3.2 Architektur für Bildverarbeitungssysteme 38
3.3 Entwurfsfluss 39

4 Entwurfsraumexploration 44

4.1 Problemstellung 44
4.2 Metainformationen 46
4.3 Modellierung in ASP 47
4.4 Experimentelle Ergebnisse 54

5 Systemsynthese 56

5.1 Softwaresynthese 56
5.2 Hardwaresynthese 66

6 Implementierung und Berechnungsplattformen **68**
6.1 Hardwarekomponenten 69
6.2 Beispielanwendung 76
6.3 Potsdam Intelligent Camera System (PICSY) 77
6.4 Plattform für Autonome Modellfahrzeuge (PAMo) 79

7 Zusammenfassung und Ausblick **84**
7.1 Zusammenfassung 84
7.2 Ausblick 86

Literaturverzeichnis **88**

Veröffentlichungen **94**

A ASP Modellierung **96**

B SDI-Kontroller **111**

Abbildungsverzeichnis

1 Gegenüberstellung der verschiedenen Prozessortechnologien. 15
2 FPGA Entwurfsfluss. Quelle: [Lü10] 19
3 Beispiel für einen Spezifikationsgraph 20
4 Grobe Übersicht über den Entwurfsraum von On-Chip-Mehrprozessorsystemen. Quelle: [Ish09] 22
5 Vom Problem zur Lösung mit Hilfe von Antwortmengenprogrammierung (ASP, answer set programming). Quelle: [AKLS05] 25
6 Vforce: Framework für die Integration von Spezialprozessoren, insbesondere FPGAs, im Bereich rekonfigurierbarer Supercomputer am Beispiel der Signalverarbeitungsbibliothek VSIPL++. Quelle: [LCC+06] 28
7 Entwurfsfluss für adaptive und rekonfigurierbare Mehrprozessorsysteme auf FPGAs (DFG-Projekt AMoC). Quelle: [BMAI10] 34
8 Modellierung des Datenflussverhaltens von Berechnungsfunktionen mit Hilfe eines synchronen Datenflussgraphen und Klassifizierung der Implementierung in verschiedene Typen T (fixed data rate/result, streaming/random access). 35
9 IPE Klasse: Datenstromorientierte Verarbeitung / Variabler Speicherzugriff 36
10 IPE Klasse: Konstante Datenrate / Ergebnisgröße 37
11 Beispielarchitektur: Software und Hardware IPEs, direkte (1) und indirekte (2) Speicherkommunikation, Koprozessorschnittstelle (3), Pipelining (4) . . 40
12 Vereinfachter Entwurfsfluss 41
13 Laufzeiten des ASP-Solvers in Abhängigkeit von der Anzahl der Knoten (rechts) in zufällig generierten Problemgraphen über 1000 Messungen. . . . 55
14 Der Zugriff einer Anwendung auf Hardwarekomponenten erfolgt über verschiedene Abstraktionsebenen. 61
15 Hardwarezugriff mit Hilfe der UIO (user IO) Erweiterung des Linux-Kernels. 64
16 Streaming Data Interface (SDI) 69
17 Interconnection Module für den PLB-Bus 73
18 Ergebnisarchitekturen für verschiedene Begrenzungen der Chipfläche. Die reine Softwarelösung benötigt 65 Zeiteinheiten. 77

19 SoCar2 (System-on-Chip Car) mit der Plattform für Autonome Modellfahrzeuge (PAMo) 80
20 SoCar2: Bildverarbeitung 81
21 SoCar2: Umweltrepräsentation und Wegeplanung 82
22 Entwurfsfluss 85

Tabellenverzeichnis

1 Hardwaremodule für die Verarbeitung von Datenströmen. 39
2 Statistische Werte zu den Laufzeiten des ASP-Solvers in Sekunden in Abhängigkeit von der Anzahl der Knoten in zufällig generierten Problemgraphen über 1000 Messungen. 55
3 Übersicht der Metainformationen 77
4 Softwareregister zur Steuerung des SDI-Kontrollers. 111
5 Statusregister des SDI-Kontrollers. 112
6 Kontrollregister des SDI-Kontrollers. 112

1 Einleitung

Die Zahl der technischen Geräte in unserem alltäglichen Umfeld wächst stetig an. Immer mehr Funktionen werden in immer kleineren Geräten untergebracht. Dafür kommen hauptsächlich digitale Systeme und Prozessorsysteme zum Einsatz und meistens eine Kombination aus Hardware- und Softwarekomponenten. Für die Entwicklung solcher Systeme spielen aus wirtschaftlicher Sicht zwei Faktoren eine wichtige Rolle, die sich maßgeblich auf die Kosten auswirken: a) Effizienz: Effiziente Systeme sind im Vergleich leistungsfähiger, im Sinne von z. B. Geschwindigkeit, Funktionsumfang oder Stromverbrauch, und b) Entwicklungszeit: Je eher ein Produkt auf den Markt kommt (time-to-market), d. h. je kürzer die Entwicklungszeit, desto mehr Einnahmen können während des Lebenszykluses des Produkts erwirtschaftet werden.

In der Vergangenheit wurde die Leistungssteigerung, insbesondere die Taktfrequenz, von Chips immer wieder durch physikalische Grenzen aufgehalten. Durch die Entwicklung neuer Herstellungsverfahren konnten immer kleinere Strukturen produziert und neue Materialkombinationen eingesetzt und damit Geschwindigkeit und Funktionsumfang der digitalen Chips weiter gesteigert werden. Auch im Bereich der Methoden zur Hardwareentwicklung ereigneten sich große Fortschritte. Während früher digitale Schaltungen aus einzelnen Gattern zusammengesetzt wurden, geschieht dies heute größtenteils mit Hilfe von Hardwarebeschreibungssprachen. Module werden als IPcores (intellectual property) bezeichnet und können, wie in einem Baukasten, zur Entwicklung ganzer Systeme herangezogen werden. Durch die Vereinheitlichung, welche die Benutzung von Hardwarebeschreibungssprachen mit sich führten, konnten sich Simulationswerkzeuge etablieren und sich zu leistungsstarken und wichtigen Helfern bei der weitgreifenden Überprüfung der Funktionalität von digitalen Schaltungen vor der Herstellung und auch später bei der Automatisierung von Produktionsschritten der Chips weiterentwickeln.

Auch die Softwareentwicklung stand nicht still. Mit der Leistungsfähigkeit von digitalen Schaltungen vergrößerten sich auch die Einsatzgebiete und damit die Anforderungen an Rechensysteme und die Softwareentwicklung begann an Bedeutung zu gewinnen. In ihren Anfängen war die Softwareentwicklung sehr aufwendig und fehleranfällig. Durch die Entwicklung neuer Entwurfsmethoden konnte die Entwicklungszeit von Programmen und Entwicklungswerkzeugen verkürzt und die Handhabbarkeit und Wiederverwendung von Programmcode erhöht werden. Hochsprachen vereinfachten die Programmierung, da nicht

mehr auf Maschinenebene beschrieben werden musste, sondern eine höhere Abstraktionsebene gewählt werden konnte. Dennoch wurde damals für hochperformante Programmteile noch lange Zeit auf die hardwarenahe Assemblerprogrammierung zurückgegriffen.

Wegen ihrer Flexibilität und Anpassbarkeit haben sich Mikrocontroller und Prozessoren in einer Unzahl von Anwendungsbereichen etabliert und sind heute allgegenwärtig, sei es in Telefon oder Waschmaschine oder im Fensterheber von modernen PKWs. Lange Zeit konnte die Geschwindigkeit von Prozessoren immer weiter gesteigert werden. Inzwischen sind jedoch die Möglichkeiten ausgereizt und der Trend geht hin zu Mehrkernprozessorsystemen. Die Schwierigkeit bei der Entwicklung von Mehrprozessorsystemen liegt in der Synchronisierung von Datenzugriffen und erstreckt sich sowohl auf Software- als auch Hardwarekomponenten, wie z. B. beim Cache-Kohärenz-Problem. Um sogenannte race conditions zu vermeiden, wurden Prozessoren um spezielle Befehle und Fähigkeiten erweitert, welche einen kontrollierten Zugriff auf Speicherzellen ermöglichen. Die Synchronisierung von Prozessen wird teilweise auch von Programmiersprachen unterstützt. Letztendlich bleibt es in der Hand des Programmierers die Synchronisierung von Datenzugriffen korrekt auszuführen. Die Fehlersuche ist komplex und nur mit Mehraufwand, verglichen mit linearen Programmen, zu bewerkstelligen. Für die parallele Berechnung von Algorithmen gibt es zwei Paradigmen, um Daten zwischen Prozessen auszutauschen: message passing interface (MPI) und shared memory (SMP). Ähnlich, wie beim Schritt von der prozeduralen Programmierung zur objektorientierten Programmierung, ist die Adaption einer neuen Denkweise erforderlich, welche in der Wirtschaft nur langsam Einzug erhält.

Die von-Neumann-Architektur ist allerdings nicht für alle Berechnungen von Algorithmen gleich gut geeignet. Allgemein ist die Herausforderung für die effiziente Berechnung eines Algorithmus, das Finden einer geeigneten Architektur. Im Hardware-Software-Co-Design werden Algorithmen analysiert und parallele und sequentielle bzw. datenstrom- und kontrolllastige Teile identifiziert. Daraus wird die Architektur des Rechensystems abgeleitet und die Aufgaben entsprechend der gewählten Aufteilung auf Hardwarebeschleuniger und Standardprozessoren verteilt. Die Synergie der beiden Welten ermöglicht niedrigere Taktraten, Energieeinsparung und hohe Verarbeitungsgeschwindigkeiten. Dabei spielt die Kommunikation zwischen den einzelnen Berechnungsmodulen eine wichtige Rolle und bestimmt maßgeblich die Effizienz des Gesamtsystems.

Mit der stetig steigenden Miniaturisierung und Rechenleistung von mobilen und eingebetteten Systemen nimmt ebenfalls deren Akzeptanz und langfristig deren Notwendigkeit

im alltäglichen Umfeld zu. Im Zusammenhang mit diesem Trend werden auch häufig *Pervasive Computing* und *Ubiquitous Computing* genannt. Die Miniaturisierung von elektronischen Bauteilen führte ebenfalls zu Veränderungen im Aufbau und der Montage von Platinen. Wurden am Anfang einzelne ICs durch Platinen gesteckt können heute vollständige Systeme in einem Chip untergebracht werden. Die Vorteile sind Kosten-, Strom- und Platzeinsparungen und eine höhere Fehlertoleranz.

Mit der zunehmenden Miniaturisierung von elektrischen Geräten wuchsen gleichzeitig auch die Anforderungen im Multimediabereich, insbesondere in der Bildverarbeitung. Beispiele dafür sind Bildtelefonie, Videoüberwachung, Gesichtserkennung in Digitalkameras zur automatischen Fokussierung bis hin zu Mobiltelefonen, welche ein Foto schießen, wenn ein Lächeln erkannt wurde. Bilddaten treten gewöhnlich in großen Mengen auf. Ein Video in mittlerer Qualität[1], hat bereits eine Datenrate von etwa 175 Megabit/s oder ein Foto einer einfachen 5 Megapixel Kamera entsprechend 40 Megabit. Zur Verarbeitung stehen dem gegenüber jedoch verhältnissmäßig ressourcenschwache eingebettete Systeme. Dieses Ungleichgewicht stellte den Entwurf und die Entwicklung bildverarbeitender eingebetteter Systeme vor ganz neue Herausforderungen.

1.1 Problemstellung

Den steigenden Ansprüche an eingebetteten Systemen kann durch Einsatz von Speziallösungen, wie Multimediaprozessoren, entsprochen werden. Dies erhöht jedoch die Anforderungen an Entwickler und Entwicklungswerkzeuge und verursacht höhere Einarbeitungszeiten und die Spezialisierung auf Nicht-Standards. Gleichzeitig ist beim Ausscheiden eines Mitarbeiters der Know-how-Verlust der Firma größer. Möglicherweise passt diese Speziallösung für die aktuelle, jedoch nicht für die nächste Anwendung. Für die Entwicklungsabteilung muss folglich ein Kompromiss gefunden werden.

Bildverarbeitungsanwendungen werden gewöhnlich in einem experimentellen und interaktiven Prozess entwickelt. Dies erfordert eine hohe Flexibilität an die zu Grunde liegende Entwicklungsplattform.

Außerdem tendieren Entwickler dazu, für neue Anwendungen eine Architektur zu wählen, die sie gut kennen, anstatt eine Architektur einzusetzen, die am besten zur Anwendung passt [Caa08]. Herkömmliche bildverarbeitende eingebettete Systeme basieren meist

[1] VGA-Auflösung, Farbtiefe 24 Bit und 25 Bildern pro Sekunde

auf sequentiell arbeitenden Prozessoren, doch Bildverarbeitungsalgorithmen sind inhärent parallel [BFRR01]. Im Gegensatz dazu können Systeme aus einer gezielten Synergie aus Software- und Hardwarekomponenten aufgebaut werden und somit hocheffizient arbeiten. Während für die Softwareentwicklung viele Werkzeuge und Experten zur Verfügung stehen, ist die Hardwareentwicklung fehleranfällig, schwierig zu debuggen, zeitaufwendig, und es gibt weniger Experten auf dem Markt. Ebenso erfordert die Integration von Hardware und Software spezielles Wissen aus beiden Welten.

Es existieren viele Methoden solche effizienten System zu konstruieren, wie z. B. grobgranulare Architekturen oder anwendungsspezifische Programmiersprachen, doch sind diese Ansätze oft zu akademisch für einen Einsatz in der Wirtschaft.

Während in den letzten Jahrzehnten die mögliche Anzahl von Transistoren auf einem Chip exponentiell anstieg, hinkt die Entwicklungsleistung einer Person, diese Transistoren auch zu nutzen, weit hinterher. Diese Lücke wird *productivity gap* genannt. Sie kann nicht durch die Vergrößerung des Entwicklungsteams geschlossen werden, denn mit steigender Anzahl an Personen, steigt auch der Kommunikationsaufwand und es ergibt sich eine maximale Produktivität bei einer Teamgröße von etwa 15 Personen (mythical man-month). Auf Grund des productivity gaps und des mythical man-month entsteht ein dringender Bedarf, durch geeignete Entwicklungswerkzeuge, die Produktivität bei der Entwicklung von eingebetteten Systemen wieder zu steigern. [VG02]

1.2 These und wissenschaftliche Fragestellungen

1.2.1 Die These und ihre Bestätigung

Die These der vorliegenden Arbeit lautet:

> *Mit Hilfe geeigneter Modelle und Methoden kann die Komplexität der Entwicklung von Bildverarbeitungssystemen auf rekonfigurierbaren SoC-Architekturen und die damit verbundene Hardware-Software-Partitionierung reduziert und dadurch wirtschaftlicher gestaltet werden.*

Der Nachweis der These erfolgt in folgenden Schritten:

Theoretische Analyse und Erstellung eines Entwurfsflusses. Durch die Definition einer flexiblen Systemarchitektur mit geeigneten Schnittstellen kann das notwendige Wissen das ein Entwickler für den Entwurf eines Hardware-Software-Systems benötigt reduziert werden. Diese Vereinfachung ermöglicht die Entwicklung komplexer Systeme mit geringem Einarbeitungsaufwand. Das betrifft sowohl die Anwendungsentwickler (Software) als auch die Entwickler von Hardwarebeschleunigern.

Automatisierung der Architekturgenerierung. Für eine effiziente Ausnutzung der Synergie aus Hardware- und Softwarekomponenten muss eine geeignete Partitionierung und Ablaufplanung gefunden werden. Die Parameter, welche Systemaufbau und -eigenschaften bestimmen, sind oft zu viele und die Zusammenhänge zu komplex, als dass ein Entwickler diese noch überblicken könnte. Deshalb ist eine automatisierte Entwurfsraumexploration oft sinnvoll.

Automatisierung der Systemsynthese. Auch die Synthese einer generierten Struktur zu einem lauffähigen System, was Hardware- und Softwarekomponenten einschließt, ist, insbesondere für die Cross-Kompilierung komplexer Software, wie z. B. ein Betriebssystem, nicht trivial. Durch die Automatisierung von Arbeitsschritten soll auch hier die Handhabung vereinfacht und mögliche Fehlerquellen vermieden werden.

Beispielimplementierung. An Hand von Beispielanwendungen soll die Umsetzbarkeit des vorgeschlagenen Entwurfsflusses gezeigt werden.

1.2.2 Wissenschaftliche Fragestellungen und technische Herausforderungen

Theoretische Analyse und Erstellung eines Entwurfsflusses. Für die Konzeption der oben genannten flexiblen Systemarchitektur und geeigneter Schnittstellen werden verwandte Arbeiten analysiert. Die Entwicklung eines Entwurfsfluss erfordert Spezialwissen in sehr vielen Bereichen und Abstraktionsebenen, von der Software-Anwendung bis hinunter zu den Gattern der Hardware.

Automatisierung der Architekturgenerierung. Bevor eine für die Anwendung passende Architektur gesucht werden kann, muss zuerst der Entwurfsraum definiert werden. Dazu sind weitere Informationen zu ermitteln, wie z. B. die Struktur der Anwendung oder die Eigenschaften der verschiedenen Implementierungen von Berechnungsfunktionen. Von den vielen Methoden den Entwurfsraum zu durchsuchen, muss eine für die Problemstellung in der vorliegenden Arbeit passende ausgewählt werden. Die Methode bzw. das Werkzeug soll leistungsstark sein und eine einfache und damit komfortabel erweiterbare Modellierung der Problemstellung ermöglichen.

Automatisierung der Systemsynthese. Nachdem die Struktur der Architektur und die Ablaufplanung generiert worden sind, erfolgt die Systemsynthese. Für die Cross-Kompilierung komplexer Software ist ein geeignetes Werkzeug auszuwählen und ggf. anzupassen, welches flexibel genug für generierte Architekturen ist. Dabei soll möglichst eine Gesamtlösung gefunden werden, welche alle benötigte Software verarbeiten kann. Eine weitere Herausforderung ist die Integration der Hardware- und Softwarekomponenten, insbesondere die Kommunikation zwischen Anwendung und Hardwarebeschleunigern in Gegenwart eines Betriebssystems.

Beispielimplementierung. Die Entwicklung von Hardware-Software-Systemen ist ein aufwendiger und vielschichtiger Prozess. Die Herausforderung liegt darin die Basis-Hardwarekomponenten zu implementieren und die eingesetzte Plattform für die Umsetzung der Beispielanwendungen in den Entwurfsfluss und die Werkzeuge zu integrieren oder anzupassen.

1.3 Zielsetzung und Einschränkungen

Die vorliegenden Arbeit soll einen Betrag dazu leisten, die Architektur von Rechensystemen dem Algorithmus anzunähern und nicht umgekehrt, wie es gewöhnlich passiert. Die dafür notwendigen generierten Architekturen können auf programmierbaren Logikbausteinen, wie FPGAs, realisiert werden. Insbesondere soll ein Beitrag geleistet werden, die Komplexität von Hardware-Software-Systemen zu reduzieren und damit die Entwicklung hochperformanter Systeme zu vereinfachen und wirtschaftlicher zu machen. Auf dem Weg

dorthin sollen neben den Gesichtspunkten Einarbeitungszeit, Wartung und Erweiterbarkeit auch Open-Source-Lösungen in Betracht gezogen werden, da häufig leistungsstarke Alternativen zu kostenintensiven, kommerziellen Lösungen existieren.

In der vorliegenden Arbeit werden komplexe Bildverarbeitungsanwendungen betrachtet, welche ein Betriebssystem erfordern, wie z. B. verteilte Kamerasensornetzwerke. Die Softwaresynthese und die Integration von Hardware ist hier komplizierter im Vergleich zu Stand-alone-Programmen. Der Fokus der vorliegenden Arbeit liegt weniger auf dem Finden der besten Architektur für eine Anwendung, sondern auf einer praktischen und dennoch effizienten Lösung.

1.4 Übersicht

Im Kapitel 2 wird kurz auf einige Grundlagen eingegangen und darauf folgend verwandte Arbeiten vorgestellt und diskutiert. Im Kapitel 3 *Entwurf von komplexen Bildverarbeitungssystemen* wird der in der vorliegenden Arbeit entwickelte Entwurfsfluss hergeleitet und vorgestellt. Danach folgen die Arbeitspakete 4 *Entwurfsraumexploration* und 5 *Systemsynthese*. Im Kapitel 6 *Implementierung und Berechnungsplattformen* werden praktische Ergebnisse präsentiert. Abschließend erfolgt die Zusammenfassung der Arbeit und ein Ausblick mit Ideen zur Fortführung der Arbeit.

Zur einfacheren Unterscheidung werden Literaturangaben in der Autor-Jahr-Notation zitiert, während eigene Veröffentlichungen mit arabischen Zahlen durchnummeriert sind.

Alle Marken und Warenzeichen sind Eigentum der jeweiligen Unternehmen.

2 Grundlagen und verwandte Arbeiten

Die Berechnung von Algorithmen enthält kontrollfluss- und datenflussrelevante Abschnitte. Für deren Realisierung mit Hilfe von Hardware- und Softwarekomponenten, stehen viele Möglichkeiten zur Auswahl. Im Folgenden wird auf den Entwurfsraum von bildverarbeitenden Systemen näher eingegangen.

2.1 Umsetzung von Bildverarbeitungssystemen

Für die Berechnung von Algorithmen steht ein breites Feld an Prozessortechnologien zu Verfügung. Diese lassen sich, wie in Abbildung 1 dargestellt, in reine Software- oder Hardwarelösung oder auch Mischformen der beiden Extreme einteilen. Auf der Softwareseite stehen GPPs (General Purpose Processor) mit einer hohen Flexibilität und Programmierbarkeit was die Umsetzung und auch Änderung einer Anwendung betrifft. Die Möglichkeit den gleichen Chip für verschiedene Anwendungsszenarien einsetzen zu können, macht sich durch hohe Stückzahlen entsprechend positiv im Preis bemerkbar. Alternativ kann für einen bestimmten Algorithmus auch ein spezieller Chip entwickelt werden (ASIC = Application-Specific Integrated Circuit). Hohe Performance und Energieeffizienz stehen einem hohen Entwicklungsaufwand und einer unveränderbaren Lösung gegenüber.

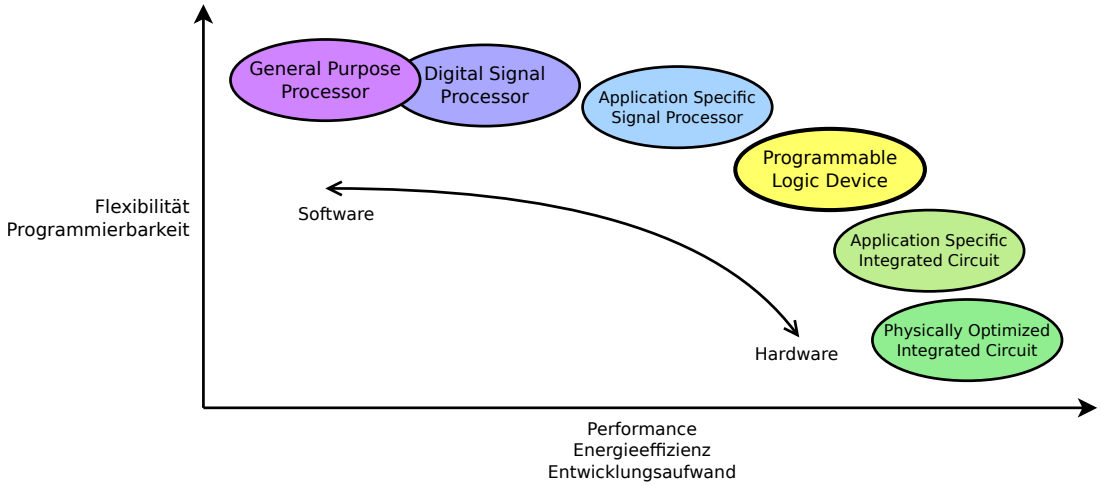


Abbildung 1: Gegenüberstellung der verschiedenen Prozessortechnologien.

2.1.1 Bildverarbeitungssysteme in Software

Neben kommerziellen Lösungen existiert *Open Source* Software, deren gesamter Quelltext unter einer freien[2] Lizenz, wie z. B. der GNU General Public License (GPL), steht. Für den Einsatz freier Software entstehen keine monetären Kosten. Außerdem ist sie in allen Teilen nachvollziehbar und individuell anpassbar, was für die Entwicklung von speziellen Systemen, wie in der vorliegenden Arbeit, von Vorteil sein kann.

Komplexe Bildverarbeitungsanwendungen werden, wie andere komplexe Software, nicht von Grund auf neu entwickelt, sondern basieren auf bereits existierender Software. Bei Softwaresystemen sind dies insbesondere Bibliotheken, die nützliche und wiederkehrende Funktionen für die Entwicklung von Anwendungen bereitstellen, und ebenso Betriebssysteme, für die Verwaltung von Systemressourcen und die Kommunikation zu anderen Systemen. Grob gesagt besteht die Systemsoftware aus Betriebssystem, Bibliotheken und Anwendungen.

Auf der Seite der Bildverarbeitungsbibliotheken existieren freie Lösungen. Zwei bekannte Vertreter werden im Folgenden kurz vorgestellt. Die *Vector, Signal and Image Processing Library (VSIPL)* nennt sich den offenen Industriestandard für Signalverarbeitung und wird von der High Performance Embedded Computing Software Initiative (HPEC-SI) als Open-Source-Projekt entwickelt. Neben der C-Implementierung existieren auch die C++-Implementierung *VSIPL++*, *Parallel VSIPL++* für Mehrprozessorsysteme und *GPU VSIPL* spezialisiert auf Graphikprozessoren der NVIDIA's CUDA Plattform. Es werden viele mathematische Grundoperationen und -funktionen abgedeckt. [VSIa]

Dagegen ist *OpenCV (Open Source Computer Vision)* eine in C++ implementierte Bibliothek mit Funktionen für Bildverarbeitung und darüber hinaus für Echtzeit Computer Vision. Sie hat Schnittstellen für C, C++, Python und Java und läuft auf Windows, Linux, Android und Mac. Die Bibliothek verfügt über mehr als 2500 optimierte Algorithmen und eine sehr große Community. Ihre Einsatzgebiete reichen von interaktiver Kunst über Mineninspektion bis hin zur Robotik. [OCV]

Auf Grund der meist eingeschränkten Systemressourcen benötigen eingebettete Systeme besondere Betriebssysteme. Es existieren viele Betriebssysteme, insbesondere solche mit sehr geringem Speicherverbrauch oder Echtzeitfähigkeit. Neben proprietären Lösungen, wie QNX oder Windows CE hat sich in den letzten zehn Jahren *Linux* immer weiter

[2] Frei im Sinne von Freiheit.

verbreitet. Das kommerzielle, auf Linux basierte Betriebssystem VxWorks hatte 2007 laut VDC Research den größten Marktanteil [Lina]. Daneben existieren auch noch viele andere Open-Source-Projekte, wie z. B. eCos oder TinyOS.

Linux ist ebenfalls ein Open-Source-Projekt und steht unter der GPL. Linux nützt Hardware sehr effizient aus, ist daher sehr schnell im Vergleich zu anderen UNIX-ähnlichen Betriebssystemen und auch auf kostengünstigen oder ressourcenschwachen Systemen lauffähig. Die Entwickler arbeiten auf hohem Qualitätsniveau, was sich in sehr stabilen Systemen, geringen Fehlerraten und kurzen Wartungszyklen äußert. Darüber hinaus ist es wesentlich einfacher, Patches und Updates für Linux zu bekommen, als für andere proprietäre Betriebssysteme. [BC06]

Durch die große Community verfügt Linux über guten Support und viele Hilfestellungen im Internet. Wegen dieser Eigenschaften gewann Linux, neben dem Desktopbereich, auch im eingebetteten Bereich zunehmend an Bedeutung und ist dort inzwischen weit verbreitet [Linb]. Durch den modularen Aufbau ist Linux sehr gut an verschiedene Einsatzbereich anpassbar und auf Supercomputern bis hin zu Mikrocontrollern zu finden. Linux wurde für über 20 Architekturen portiert. Darin sind Mikrocontroller und Softcore-Prozessoren, wie z. B. OpenRISC, noch nicht eingerechnet. [ker]

2.1.2 Bildverarbeitungssysteme in Hardware

Die Entwicklung von komplexen digitalen Systemen lässt sich grob in die Abstraktionsebenen: Verhaltensebene, RTL-Ebene[3] und Gatterebene einteilen. Auf jeder Ebene erfolgt eine Beschreibung und Validierung durch Simulation des Systems, unterstützt durch Entwicklungswerkzeuge. Die Beschreibung kann mit Hilfe von sogenannten Hardwarebeschreibungssprachen, wie z. B. VHDL oder VERILOG, vorgenommen werden. Durch die vielen Abstraktionsebenen ist die Entwicklung eines Chips aufwendig und die damit verbundenen NRE-Kosten[4] sehr hoch.

Gegenüber Softwareimplementierungen mit GPPs oder DSPs und reinen Hardwarelösungen, wie ASICs, bieten programmierbare Logikbausteine einen Kompromiss zwischen

[3] RTL = Register Transfer Level
[4] NRE = Non-Recurring Engineering; Einmalige Kosten bei einer Serienproduktion

Programmierbarkeit und Geschwindigkeit (siehe Abbildung 1). Im Bereich der programmierbaren Logikbausteine sind insbesondere die FPGAs (Field-Programmable Gate Array) interessant. In ihnen wurden die Konzepte von PLDs[5] und MPGAs vereint. FPGAs bestehen aus vielen kleinen Logikzellen, sogenannten CLBs (Configurable Logic Block), die in einem Gitter angeordnet sind und über ein Netzwerk von Leitungen miteinander verbunden werden können. Dabei können die Funktion der einzelnen Zellen und die Verbindungen nach der Produktion des Chips frei konfiguriert werden. Die CLBs sind im Vergleich zu den Logikzellen von CPLDs wesentlich einfacher aufgebaut, weil viele Anwendungen tendenziell mehr Register und weniger Kombinatorik benötigen und so die Produktterme der PLDs nur geringfügig ausgenützt wurden. Diese *mehrstufige Logik* eignet sich gut für Datenpfadanwendungen. Im Gegensatz zum CPLD verläuft die Verdrahtung einer logischen Funktion durch mehrere Zellen hindurch. Die Signallaufzeiten sind dementsprechend auch unterschiedlich. Sie können erst nach der Aufteilung der logischen Funktion auf die einzelnen Zellen und deren Verdrahtung ermittelt werden (place & route). [Wan98]

FPGAs vereinen folgende Eigenschaften:

Hohe Flexibilität: Durch Dynamische Rekonfiguration können dem selben FPGA verschiedene Funktionalitäten einkonfiguriert und sogar während des Betriebs verändert werden. Die Flexibilität von FPGAs ist für die Entwicklung von Prototypen von Vorteil. Die Entwicklung der Hauptplatine einer Schaltung kann weitgehend unabhängig von der durch den FPGA zu realisierenden Logik abgeschlossen werden.

Hohe Leistungsfähigkeit: Alle Logikzellen arbeiten gleichzeitig. Somit können Aufgaben auch parallel ausgeführt und damit sehr hohe Geschwindigkeiten erreicht werden.

Geringe Kosten: FPGAs sind Standardbauteile und können in großen Stückzahlen hergestellt werden, da die Funktionalität erst später hinzugefügt wird. Dies ist insbesondere für Kleinserien interessant, für die die Produktion eines ASICs zu teuer wäre.

[5] (C)PLD = (Complex) Programmable Logic Device, MPGA = Mask Programmable Gate Array

Großes Sortiment: Die große Auswahl an FPGAs auf dem Markt ermöglicht eine Vielzahl von Einsatzgebieten. Die heutigen FPGAs sind je nach Modell mit Block-RAM, Multiplizierern, Digital Clock Managern und Hardcore-Prozessoren ausgestattet. Die Logikzellen werden in verschiedenen Techniken (LUT, Multiplexer) und Technologien (SRAM, EPROM, Antifuse) realisiert.

Unabhängigkeit: Die Entwicklung von Hardwarekomponenten kann durch Hardwarebeschreibungssprachen plattformunabhängig erfolgen. Reicht der eingesetzte FPGA im Laufe der Weiterentwicklung eines Produkts nicht mehr aus, kann mit wenig Änderungen am System auf einen größeren oder moderneren Chip gewechselt werden.

In Abbildung 2 ist der Lowlevel-Entwurfsfluss für FPGAs allgemein dargestellt. Die weit verbreitete Hardwarebeschreibungssprache VHDL erlaubt die Definition des Verhaltens eines Systems sowohl auf Gatter- und Registertransfer-, als auch auf Strukturebene. Das System wird auf die Struktur des eingesetzten Chips abgebildet und ein entsprechender *Bitstream* generiert, der die Konfiguration der Logikzellen und der Kommunikation enthält.

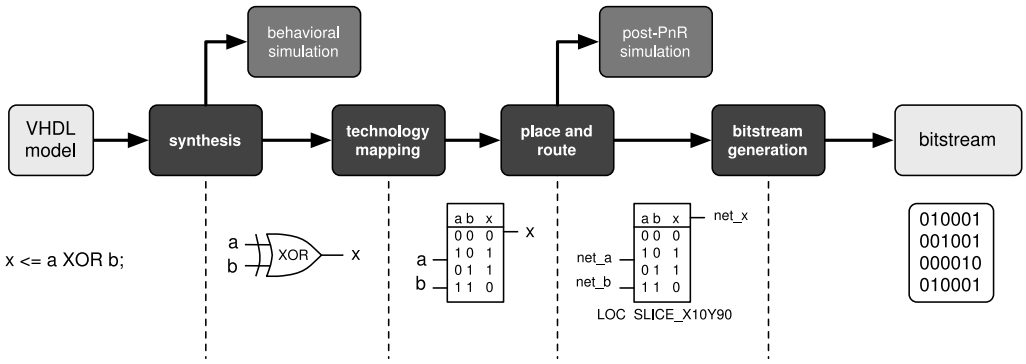
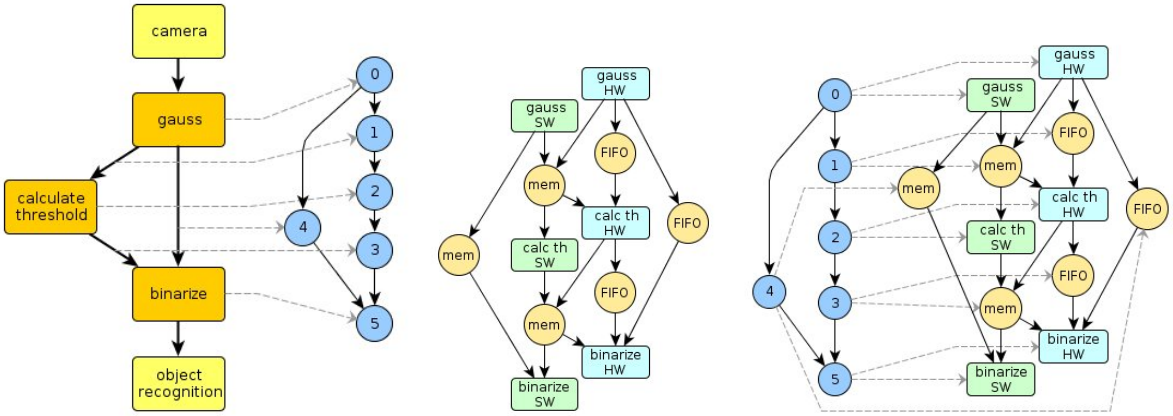


Abbildung 2: FPGA Entwurfsfluss: Die Hardwarebeschreibung wird zu einer Netzliste synthetisiert, welche anschließend auf die Logikzellen des Ziel-FPGAs abgebildet und dann geroutet und verdrahtet wird. Der FPGA wird mit dem resultierenden Bitstream konfiguriert. Eine Simulation ist in verschiedenen Zwischenstufen möglich.

Laut dem Marktführer Xilinx, sind FPGAs heutzutage so attraktiv, bezüglich Geschwindigkeit, Kostenstruktur und Energieeffizienz, dass sie mit ASICs konkurrieren können [Ele]. Die Eigenschaften von FPGAs ermöglichen außerdem Hardware-Software-Systeme auf einem Chip zu realisieren und damit die effizienzsteigernde Synergie der Extreme Software und Hardware auszunutzen.

2.2 Formaler Entwurfsfluss für Hardware-Software-Systeme

In [TH07] wird eine *formale* Spezifikation von Hardware-Software-Systemen und deren Entwurf vorgestellt. Im Folgenden wird deren Modell beispielhaft an Hand der Vorverarbeitungsstufe einer Objekterkennung beschrieben: Nach der Glättung des Bildes nach Gauß, zur Reduzierung von Artefakten, wird ein Schwellwert berechnet und damit anschließend das Grauwertbild der Kamera in ein Schwarzweißbild konvertiert (siehe Abbildung 3a).



(a) Datenflussgraph → Problemgraph (b) Architekturgraph (c) Spezifikationsgraph

Abbildung 3: Beispiel für einen Spezifikationsgraph (Ausschnitt): Die Einzelschritte der Binarisierung eines Bildes (orange) werden im Problemgraphen definiert und alle möglichen Architekturen bestehend aus Kommunikationskanälen, Software- und Hardwareimplementierungen im Architekturgraphen widergespiegelt. Beide Graphen werden im Spezifikationsgraph durch eine Abbildung verknüpft.

Die formale Spezifikation besteht aus einem Problemgraphen, einem Architekturgraphen, Abbildungsmöglichkeiten und Entwurfsbeschränkungen. Das Problem wird in einem Problemgraphen spezifiziert. Dabei stellen Knoten die Berechnungs- und Kommunikationsaufgaben und Kanten die Datenabhängigkeiten dar. Der Problemgraph kann z. B. aus einem Datenflussgraph abgeleitet werden, indem jede Kante aus dem Datenflussgraph durch einen zusätzlichen Knoten für die Kommunikation repräsentiert wird (siehe Abbildung 3a). Im Architekturgraph werden alle für die Realisierung des Systems zur Verfügung stehenden Komponenten als Knoten repräsentiert. Dies können z. B. Prozessoren und Hardwarebeschleuniger, als auch Kommunikationskanäle und Speicher sein (siehe Abbildung 3b). Der Spezifikationsgraph beinhaltet schließlich einen Problemgraph, einen Architekturgraph und eine Abbildung aller Knoten des Problemgraphen auf Knoten des Architekturgraphen (siehe Abbildung 3c). Diese Abbildung gibt an, welche Verarbeitungsschritte auf welchen Systemkomponenten abgewickelt werden sollen.

Der Spezifikationsgraph stellt den Entwurfsraum des Systems dar. Für den Entwurf eines digitalen Hardware-Software-Systems folgen anschließend die Arbeitsschritte Allokation, Bindung und Ablaufplanung. Bei der Allokation werden die für die Zielarchitektur zu verwendenden Komponenten im Architekturgraph ausgewählt. Die Bindung definiert darauf folgend, welche Berechnungen auf welchen Komponenten auszuführen sind. Dies erfolgt jeweils unter Berücksichtigung von Systemanforderungen bzw. -einschränkungen und Optimierungszielen. [TH07]

2.3 Entwurf von On-Chip-Systemen

Während des Entwurfs von On-Chip-Systemen müssen vielfältige Entscheidungen getroffen werden. Die zahlreichen Entscheidungen bestimmen die Eigenschaften des resultierenden Systems, z. B. in puncto Geschwindigkeit, Stromverbrauch, Chipgröße, etc. Entscheidungen aus verschiedenen Bereichen können sich beeinflussen und sogar Gegenspieler sein. Deshalb muss abgewogen werden, welches Ziel wichtiger ist. Abbildung 4 zeigt eine grobe Übersicht des Entwurfsraum von On-Chip-Mehrprozessorsystemen. Die einzelnen Bereiche sind dabei als Pfeile dargestellt. Ein konkretes System kann durch eine Kurve durch alle Kreissegmente repräsentiert werden, im Beispiel eine 3-Kern-Architektur.

Im Folgenden werden verschiedene Methoden vorgestellt, die Suche im Entwurfsraum zu automatisieren.

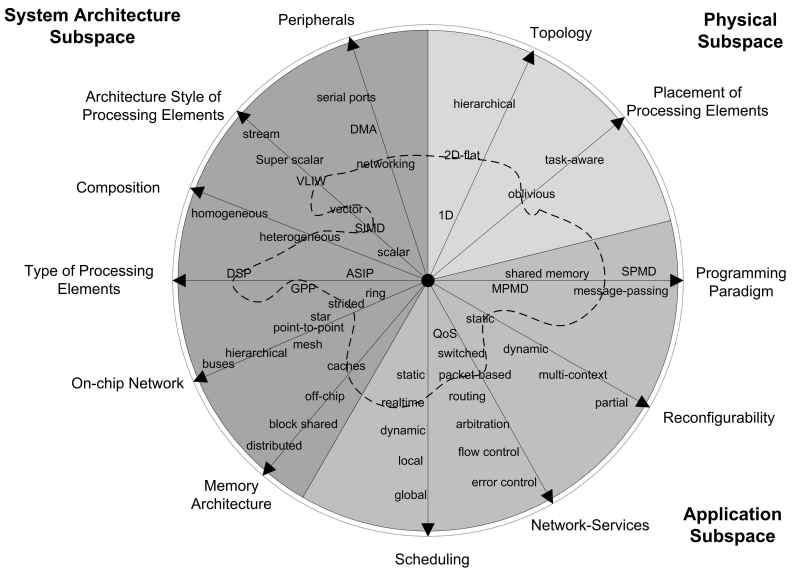


Abbildung 4: Grobe Übersicht über den Entwurfsraum von On-Chip-Mehrprozessorsystemen.

2.3.1 Suchalgorithmen

Suchalgorithmen, wie z. B. evolutionäre Algorithmen, sind in der Lage sehr komplexe Optimierungsprobleme zu lösen. Das Werkzeug PISA [pis] wurde hauptsächlich für die Mehrzielsuche konzipiert. Dabei ist das Optimierungsproblem durch eine Menge von konkurrierenden Zielen charakterisiert. PISA verfolgt einen generischen Ansatz und trennt problem- und lösungsspezifische Programmteile von Suchalgorithmen. So kann ein bestimmtes Problem mit mehreren Suchalgorithmen untersucht werden. Ein Suchalgorithmus wird hier als eine Methode definiert, welche durch folgende drei Schritte, versucht eine Lösung für ein vorgegebenes Problem zu finden:

- Die Bewertung der Lösungskandidaten,
- die Auswahl von viel versprechenden Kandidaten basierend auf der Bewertung und
- die Bildung einer neuen Generation von Kandidaten durch Variation.

In diese Kategorie von Suchalgorithmen fallen die meisten evolutionären Algorithmen und die simulierte Abkühlung. [BLTZ03]

Evolutionäre Algorithmen (EA) sind probabilistische Optimierungs- und Verbesserungsalgorithmen, die der Darwinschen Evolution nachempfunden sind. Ein Individuum entspricht einer Lösung und wird durch einen Zahlenvektor repräsentiert, ähnlich wie ein Lebewesen durch seine DNA. Durch Mutation, Rekombination und Selektion nähern sich die Individuen schrittweise einem Optimum an. Die Variationsoperatoren produzieren dabei genetische Diversität und die Selektion gibt der Evolution die Richtung. Dabei spielt die Bewertungsmethode, auch Fitnessfunktion genannt, eine grundlegende Rolle für die Qualität der Lösungen.

Eine weitere Methode ist die *simulierte Abkühlung* (engl.: simulated annealing). Sie ist ein heuristisches Optimierungsverfahren mit der Grundidee, den Abkühlungsprozesses eines glühenden Metalls nachzubilden. Beim Abkühlen werden stabile Kristalle gebildet und die Atome ordnen sich in einem energieärmeren Zustand nahe des Optimums an.

2.3.2 Constraint Satisfaction Problem

Für Probleme, deren Zielfunktion bekannt ist und denen eine gewisse Struktur innewohnt, gibt es andere Lösungswege. Insbesondere für kombinatorische Probleme ist das *Constraint Programming* und *Constraint Satisfaction Problem* (CSP) interessant.

Die Grundidee von *constraint satisfaction* ist die Spezifizierung eines Problems durch Fakten und Bedingungen (constraints). Dabei wird nicht beschrieben, *wie* das Problem zu lösen ist, sondern *was* das Problem ist (deklarative Problemlösung). Ziel ist es, für alle lösungsrelevanten Variablen einen Wert zu finden, ohne dass eine der Bedingungen verletzt wird. Constraint Programming ist ein mächtiges Paradigma, um kombinatorische Suchprobleme zu lösen und kann somit auf ein sehr weites Feld von Beweisen und Problemen, auch außerhalb der Informatik, wie z. B. das 8-Damen-Problem im Schach, Sudoku, oder Ablauf- und Routenplanung angewendet werden. [RvW06]

Eine klassische Definition des Constraint Satisfaction Problems (CSP) lautet[6]:
Ein CSP $\mathcal{P}$ ist definiert als ein Tripel $\mathcal{P} = \langle X, D, C \rangle$, mit dem n-Tupel $X = \langle x_1, x_2, \ldots, x_n \rangle$, dem n-Tupel der zugehörigen Wertebereiche $D = \langle D_1, D_2, \ldots, D_n \rangle$ mit $x_i \in D_i$ und dem t-Tupel der Bedingungen $C = \langle C_1, C_2, \ldots, C_t \rangle$. Eine Bedingung C_j ist ein Paar $\langle R_{S_j}, S_j \rangle$, wobei R_{S_j} eine Relation auf die Variablen in $S_i = scope(C_i)$ ist. In anderen Worten ist R_i eine Untermenge des kartesischen Produkts der Wertebereiche der Variablen in S_i.

[6] Übersetzt aus [RvW06]

Eine Lösung des CSPs $\mathcal{P}$ ist ein n-Tupel $A = \langle a_1, a_2, \ldots, a_n \rangle$, wobei $a_i \in D_i$ gilt und jede Bedingung C_j erfüllbar ist, d. h. dass R_{S_j} eine Projektion von A auf S_j ist. Mögliche Aufgabenstellungen können sein, alle Lösungen $sol(\mathcal{P})$, eine Lösung oder die Existenz einer Lösung zu finden. Ist die Menge der Lösungen leer, ist das CSP nicht erfüllbar.

Bei der Linearen Optimierung (auch Lineare Programmierung (LP)) wird versucht, lineare Zielfunktionen über einer Menge, die durch lineare Gleichungen und Ungleichungen eingeschränkt wird, zu optimieren. Die mathematische Formulierung eines LPs lautet: Gegeben sind die Matrix $A \in \mathbb{R}^{m,n}$ und die Vektoren $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$. Ziel ist es, einen Vektor $x \in \mathbb{R}^n$ zu finden, der die Ungleichungen

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

erfüllt und das Skalarprodukt $c^T x = c_1x_1 + \cdots + c_nx_n$ minimiert. [RvW06]

Eine Erweiterung ist die ganzzahlige lineare Optimierung (ILP, engl. integer linear programming) mit $x \in \mathbb{N}_0^n$, sowie die 0-1 ILP mit $x \in \{0, 1\}$, die bei einer Großzahl von Problemen Anwendung findet. Die Modellierung von Problemen mit ILP ist, u. a. durch die fehlende Komplementbildung, häufig jedoch groß und unübersichtlich. Außerdem beeinflusst die Qualität der Modellierung auch die Berechenbarkeit eines ILPs. [RvW06]

Eine Spezialisierung des CSP ist das *Erfüllbarkeitsproblem der Aussagenlogik* (SAT, engl. satisfiability). Hier sind die Wertebereiche C_i des CSPs auf $\{true, false\}$ beschränkt. Mit SAT konnten in der Vergangenheit besonders bei der automatischen Verifikation von Hard- und Software große Fortschritte erreicht werden. Insbesondere für Entscheidungsprobleme auf Bitvektoren, bezeichnet [BHvW09] effizientes SAT-Solving als die Schlüsseltechnologie des 21. Jahrhunderts. Durch *backtracking*-Algorithmen unterstützen SAT-Solver eine vollständige Suche, d. h. falls eine Lösung existiert, wird diese auch gefunden.

2.3.3 Antwortmengenprogrammierung (ASP)

Eine Weiterentwicklung der vorgestellten Konzepte ist die Antwortmengenprogrammierung (ASP, engl. answer set programming). Sie ist eine Kombination aus Wissensrepräsentation, deduktiven Datenbanken, nichtmonotonem Schließen und Logikprogrammierung mit Komplementbildung [Bar03, Gel08]. In Abbildung 5 ist der Weg vom Problem zur Lösung skizziert. Neben Fakten und Regeln werden auch Sprachkonstrukte verwendet, um ein Problem zu definieren. Die Vorteile von ASP liegen in seiner Einfachheit und der sich daraus ergebenden Transparenz, was sich positiv auf die Erstellung und Weiterentwicklung von Modellen auswirkt, und die Möglichkeit effektiv unvollständige Spezifikationen zu modellieren [Sch95, Jan06, LR06, AKLS05]. ASP wurde u. a. für die Erzeugung von Ablaufplänen verwendet [BLLN06].

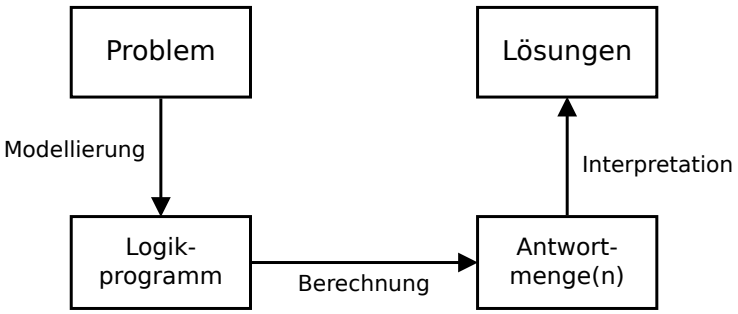


Abbildung 5: Vom Problem zur Lösung mit Hilfe von Antwortmengenprogrammierung (ASP, answer set programming).

Im Folgenden wird das Grundkonzept von ASP, wie in [AKLS05] ausführlich dargestellt, beschrieben. Eine Antwortmenge besteht aus einer Menge von Atomen. Die Definition von Regeln erfolgt, wie in normalen Logikprogrammen in der Form,

$$p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n \tag{1}$$

wobei $n \geq m \geq 0$ gilt und p_i, $i \in [0; n]$ die Atome darstellen. Für eine Regel r sind neben den Mengen $head(r) = p_0$ und $body(r) = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$ außerdem die Mengen $body^+(r) = \{p_1, \ldots, p_m\}$ und $body^-(r) = \{p_{m+1}, \ldots, p_n\}$ definiert.

Regel 1 ist so zu verstehen, dass $head(r)$ genau dann mit in die Antwortmenge aufgenommen wird, wenn alle Atome aus $body^+(r)$ bereits enthalten sind, jedoch kein Atom aus $body^-(r)$.

Ein Programm Π gilt als abgeschlossen unter einer Menge von Atomen X, wenn für alle $r \in \Pi$ und $body^+(r) \subseteq X$ auch $head(r) \in X$ gilt. Die kleinste solche Menge an Atomen wird als $Cn(\Pi)$ und als *Antwortmenge* von Π bezeichnet. Im Allgemeinen wird die Ableitung eines Programms Π von einer Menge an Atomen X benötigt:

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi,\ body^-(r) \cap X = \varnothing\} \tag{2}$$

Damit ist eine Menge von Atomen X die Antwortmenge eines Programms Π genau dann, wenn $Cn(\Pi^X) = X$ gilt. Eine weitere Besonderheit ist, dass durch die Formulierung $f \leftarrow not\ f$ Integritätsbedingungen definiert werden können. Somit kann die Integritätsbedingungen

$$\leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n \tag{3}$$

als Regel

$$f \leftarrow not\ f,\ p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n \tag{4}$$

geschrieben werden.

Der erste Schritt für die Berechnung einer Antwortmenge ist es, das Logikprogramm zu expandieren (grounding). Dabei werden alle Variablen eliminiert, indem jede Regel für jede passende Konstante dupliziert wird. Zum Beispiel wird das Logikprogramm

$$p(1). \quad p(2). \quad q(X) \leftarrow p(X). \tag{5}$$

zu

$$p(1). \quad p(2). \quad q(1) \leftarrow p(1). \quad q(2) \leftarrow p(2). \tag{6}$$

expandiert. Bei der Modellierung muss außerdem darauf geachtet werden, dass für jede Variable eine vollständige Domäne benötigt wird (im Beispiel X), welche ggf. explizit angegeben werden muss. Nach dem Grounden wird mit einem SAT-ähnlichen Solver die Antwortmenge berechnet.

Eine intuitive Vorgehensweise zur Modellierung von Logikprogrammen für ASP-Solver ist das *generate-and-test*-Paradigma. Durch bestimmte Regeln wird eine Menge an Fakten generiert, welche dann durch einschränkende Regeln auf die Lösung reduziert wird. Um

das zu ermöglichen verfügt ASP über spezielle Sprachkonstrukte, wie z. B. Integritätsbedingungen und *Aggregate* [GKK+08]:

$$l\ [\ v_0{=}a_0,\ v_1{=}a_1,\ \ldots,\ v_n{=}a_n\ ]\ u. \tag{7}$$

Die eckigen Klammern beschreiben eine gewichtete Summe der Atome $v_0, \ldots, v_n$ und den dazugehörigen Gewichten $a_0, \ldots, a_n$. Die Regel 7 definiert den Fakt, dass es eine Menge von wahren Atomen in $A \subseteq \{v_0, \ldots, v_n\}$ gibt, so dass deren gewichtete Summe im Intervall $[l; u]$ liegt. Eine Gewichtung mit 1 kann entfallen.

2.4 Verwandte Arbeiten

Im Folgenden werden Arbeiten aus einzelnen Bereichen vorgestellt. Die breite der Thematik spiegelt sich in der Vielfalt der dargestellten Projekte wieder.

Skelett-basierte Hardwarearchitekturen. Anstatt für eine vorgegebene Anwendung eine Hardwarearchitektur von Grund auf neu zu entwickeln, kann sie stattdessen aus einem vorgegebenen Grundgerüst (skeleton) aufgebaut werden. Die Basis ist ein Pool von parametrisierbaren Hardwaremodulen, welche eine Reihe von Grundfunktionen im Bereich der Bildverarbeitung implementieren, und eine Infrastruktur für die Kommunikation der einzelnen Module. Benkrid et al. reichern die Module um weitere Informationen an. Anschließend kann für eine, in der Struktur des Grundgerüsts beschriebenen, Anwendung automatisch eine effiziente Hardwarekonfiguration erzeugt werden. [BCSB01]

Darauf aufbauend, wurde in [BCBB02] eine Highlevel-Entwicklungsumgebung vorgestellt, welche die Logikprogrammiersprache Prolog mit einbezieht. Sie soll, gegenüber Hardwarebeschreibungssprachen wie VHDL, die Beschreibung von Strukturelementen vereinfachen und ermöglicht außerdem die Definition von herstellerspezifischen Details, wie z. B. über die Platzierung.

Solche Hardwarebaukästen sind auch z. B. bei der Firma Agility (ehemals: Celoxica), Entwickler der Highlevel-Hardwarebeschreibungssprache HANDELC, anzutreffen. Ihrer Entwicklungsumgebung bietet ein Framework und ein graphisches Werkzeug zur Verarbeitung von (Bild)Datenströmen mit dem Namen *PixelStreams* an. In der Software werden Verarbeitungsfunktionen als Rechtecke dargestellte. Diese können mit Linien verbunden

und damit die Datenabhängigkeiten definiert werden. Die Hardwareschnittstellen sind dokumentiert und so können auch eigene Hardwarebeschleuniger eingebunden werden. Neben der graphischen Programmierung kann diese auch in HANDELC erfolgen. [Cel]

Vforce. VSIPL++ ist eine in C++ implementierte Bibliothek für Vektor-, Signal- und Bildverarbeitung, basierend auf VSIPL. An der Northeastern University wurde ein Framework entwickelt, welches die Einbeziehung von Hardwarebeschleunigern in die Signalverarbeitung vereinfachen soll. FPGA-Platinenhersteller können damit ihre VSIPL-IPcores durch eine einfache und standardisierte Schnittstelle den Softwareentwicklern zur Verfügung stellen. Ziel des Projekts ist die Integration von SPPs (special purpose processor), insbesondere FPGAs, in VSIPL++ zu vereinfachen und einen Beitrag zur Unterstützung der Programmierung von rekonfigurierbaren Supercomputern zu leisten. [LCC+06, VSIb]

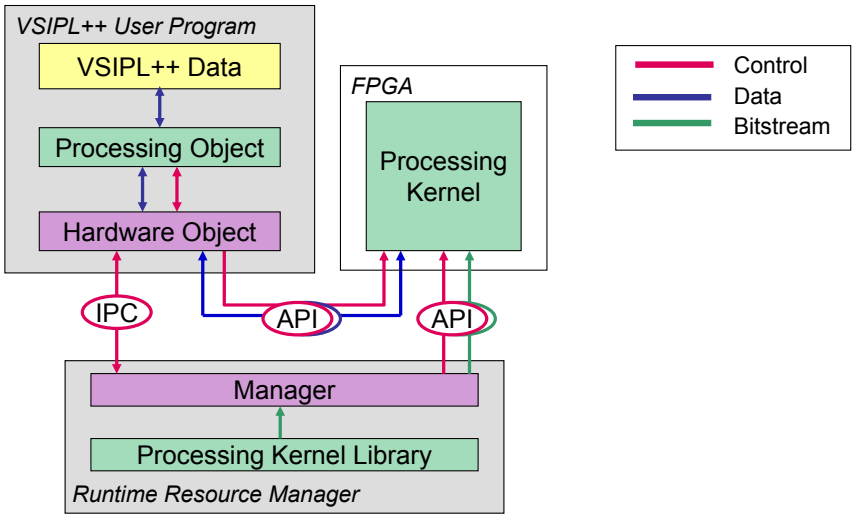


Abbildung 6: Vforce: Framework für die Integration von Spezialprozessoren, insbesondere FPGAs, im Bereich rekonfigurierbarer Supercomputer am Beispiel der Signalverarbeitungsbibliothek VSIPL++.

Hardwareplattformen werden in C++-Klassen gekapselt und dort die Konfiguration und Kommunikation der Hardware implementiert. Durch den Austausch der zugehörigen Header-Dateien im Hauptprogramm werden statt der ursprünglichen Softwareimplementierung entsprechende Hardwareressourcen oder Plattformen eingebunden. Später wurde das Framework in Vforce umbenannt, die Unterstützung auf GPUs erweitert und ein

Ressourcenmanager integriert, der einen FPGA zur Laufzeit entsprechend der Anwendung dynamisch umkonfigurieren kann [MCLK07]. Das Framework ist in Abbildung 6 dargestellt.

SystemCoDesigner (Haubelt et al.). Für die Exploration von eingebetteten Systemen wurde der SystemCoDesigner entwickelt [HFG+]. Dies ist ein Modul für das in Abschnitt 2.3.1 vorgestellte Werkzeug PISA. Als Eingabedaten für die Allokation und die Ablaufplanung dienen ein Problemgraph und ein Architekturgraph der Problemstellung, wie in Abschnitt 2.2 *Formaler Entwurfsfluss für Hardware-Software-Systeme* beschrieben. Eine grafische Oberfläche führt durch die vielen Parameter, welche für die Steuerung der evolutionären Algorithmen notwendig sind.

Ishebabi (Dissertation). In [Ish09, IB09] wurde der Entwurfsraum für Mehrprozessorsysteme analysiert. Die Problematik wurde sehr umfangreich dargestellt und mehrere Methoden zur Exploration des Entwurfsraums untersucht. Die Modellierung des Problems erfolgte in ILP. Diese skalierte jedoch nur schwach und die Anzahl der Aufgaben und Prozessoren der Problemstellung wirkte sich massiv auf die Laufzeit des Solvers aus, weshalb anschließend SAT-Solver-Techniken ausgenützt wurden. Dazu wurde der 0-1-ILP-fähige ASP-Solver `clasp` aus der Potassco-Suite [POT] verwendet. Abschließend wurden auch Greedy-ähnliche Heuristiken untersucht. Diese arbeiten im Allgemeinen schnell und flexibel, können jedoch keine optimalen Lösungen finden, was auch im Experiment bestätigt wurde.

Lübbers (Dissertation) / Hardware-Threads. Das Projekt *hthreads* beschäftigt sich mit der Integration von Hardware- und Software-Threads in einem Betriebssystem. Alle Threads werden auf die POSIX pthreads API abgebildet und die Synchronisierung und Ablaufplanung des Betriebssystems in Hardware ausgelagert. [Hth]

Lübbers verfolgt einen flexibleren Ansatz und stellt in [Lü10] die Ausführungsumgebung *ReconOS* vor, welche es ermöglicht, die Vorteile von Hardwarebeschleunigern und partiell rekonfigurierbaren Architekturen in Multithreaded-Betriebssysteme zu integrieren. Die Arbeit zeigt Synchronisations- und Kommunikationsmechanismen, um die Kooperation von in Software und in Hardware implementierten Threads zu ermöglichen. Das Konzept wird am Beispiel von eCos und Linux implementiert. Durch die partielle Rekonfiguration

ist es möglich, Hardwarekomponenten während der Laufzeit auszutauschen und somit die vorhandenen Ressourcen effizienter auszunutzen.

Caarls (Dissertation). Der steigende Bedarf an Anwendungen mit hohem Datendurchsatz, wie z. B. computergestützte Kameraüberwachung, erzeugt eine große semantische Diskrepanz zwischen universellen Programmiersprachen, wie C oder Java, und den zur Verfügung stehenden Architekturen von Rechensystemen. Gewöhnlich wird dieser Missstand durch die Einarbeitung in eine architekturspezifische Programmiersprache gelöst. Diese Vorgehensweise ist weder plattformunabhängig noch hilft sie bei der Auswahl einer passenden Architektur. Schlimmstenfalls stellt sich später die gewählte Architektur als unzureichend oder unpassend heraus. Statt die Programmiersprache der Architektur anzunähern, schlägt Caarls in seiner Dissertation [Caa08] vor, die Programmiersprache dem Algorithmus mit Hilfe von ASLs (algorithm-specific language) anzunähern, auch weil der Entwickler mit diesem besser vertraut ist.

Eine Anwendung besteht aus der Kombination von mehreren Algorithmen. Caarls empfiehlt für die Beschreibung jedes Algorithmus eine geeignete ASL zu benutzen. Diese plattformunabhängigen Implementierungen können nun mit seinem Werkzeug zusammengeführt und für bestimmte Zielarchitekturen kompiliert und simuliert werden. Die aus der Simulation gewonnenen Messwerte werden in einer Entwurfsraumexploration benutzt, um eine Pareto Front für die Zielarchitektur zu finden. Dazu wird der evolutionäre Algorithmus SPEA2 (Strength Pareto Evolutionary Algorithm 2) eingesetzt. Untersucht wurden verschiedene Prozessoren aus dem Bereichen: SIMD, MIMD, VLIW und GPU[7].

2.5 Diskussion

In [Ish09] wurde der Suchraum für Mehrprozessorsysteme untersucht und auf die Wichtigkeit eines automatisierten Syntheseprozesses hingewiesen. Die Komplexität des Problems sei selbst für erfahrene Entwickler schwer überschaubar und kann dazu führen, dass Details übersehen werden, die große Auswirkungen auf das Endergebnis haben.

Caarls Arbeit bestätigt, dass noch keine allgemeine und effiziente Lösung für die Ausführung von Berechnungen durch den Entwurf und die Programmierung von Rechensystemen

[7] SIMD = single instruction multiple data, MIMD = multiple instruction multiple data, VLIW = very long instruction word, GPU = graphics processing unit

gefunden wurde. Er beschäftigt sich mit der Abbildung von Algorithmen auf bestehenden Architekturen, während in der vorliegenden Arbeit die Architekturen selbst, auf einer feingranularen Ebene, generiert werden. Der Ansatz Hardwarebeschleuniger durch ASLs zu beschreiben, kann aufgegriffen werden.

Skelett-basierte Architekturen bieten durch die Modularisierung ähnliche Vorteile, wie Bibliotheken bei der Softwareentwicklung: Wiederverwendung von Code und wohldefinierte Schnittstellen, Kommunikation und Interaktion. Sie können als Basis für die Architekturgenerierung verwendet werden. Für die Entwicklung eines Systems muss die Partitionierung und die Integration von Hardware und Software manuell erfolgen.

Im Projekt Vforce wurde die Integration von Hardware und Software für den Software-Entwickler vereinfacht. Dies erfolgt durch den Austausch von internen, und somit für den Entwickler nicht bekannten, Softwareimplementierungen. Eine Entwurfsraumexploration findet nicht statt. Das Projekt zielt im Vergleich zur vorliegenden Arbeit auf Supercomputer und nicht auf SoCs ab, weshalb der Umfang des Frameworks wesentlich kleiner ausfällt.

Überscheidungen von Lübbers mit der vorliegenden Arbeit liegen in der Fragestellung vor, wie eine Anwendung (Software) mit Hardwarekomponenten kommunizieren kann und welche Anpassungen und Erweiterungen des Betriebssystems dafür notwendig sind. Lübbers verwendet teilweise sehr aufwendige Kommunikationsmodule, wie z. B. eine Hardware-MMU oder Datenzugriff über Gerätedateien, da sein Fokus auf der Abstraktion von Threads liegt. In der vorliegenden Arbeit steht dagegen die Verarbeitungsgeschwindigkeit im Vordergrund, weshalb direktere Kommunikationsmechanismen verwendet werden.

Für die Entwurfsraumexploration benutzt SystemCoDesigner evolutionäre Algorithmen. Ihre Vorteile sind die universelle Einsetzbarkeit bei akzeptabler Performance. Sie benötigen keine Angaben über die Zielfunktion oder fordern andere Voraussetzungen, wie Linearität, Stetigkeit oder Differenzierbarkeit. Deshalb sind sie für eine Vielzahl von Problemstellungen geeignet, einfach zu implementieren und zu parallelisieren. Durch Parallelität sind sie robust gegenüber ungültigen Lösungen. Das Ergebnis ist eine Gruppe von Lösungen, die außerdem für eine Pareto-Optimierung verwendbar sind. Dem gegenüber steht, dass die für die Berechnungen von EAs zu Grunde liegenden Heuristiken nur eine schwache theoretische Basis bilden. Oft sind viele Iterationen erforderlich, um gute Lösungen zu erzielen. Darüber hinaus sind die Eingangsparameter, welche die Konvergenz

maßgeblich beeinflussen, oft in großer Anzahl vorhanden und dementsprechend schwierig zu handhaben. Ohne Koevolution können Populationen leicht ihre Diversität verlieren und gegen (lokale) Maxima konvergieren. Letztendlich gibt es keine Garantie, dass eine optimale Lösung in endlicher Zeit gefunden wird. [Kel]

Die kombinatorische Natur des in der vorliegenden Arbeit bestehenden Optimierungsproblems spricht dagegen für die Verwendung der Constraints Programmierung und ist evolutionären Algorithmen oder anderen stochastischen Verfahren vorzuziehen. Dies bestätigen auch die Untersuchungen von Ishebabi. Die Komplexität seines Entwurfsraums ist dem der vorliegenden Arbeit sehr ähnlich. Bei seiner Modellierung war ein Problem der ganzzahlige Wertebereich der Gewichte. Sollen mehrere nicht ganzzahlige Gewichte in Relation gebracht werden, muss eine Normierung stattfinden, welche den gültigen Zahlenbereich der Gewichte überschreiten kann und somit möglicherweise zu nicht-optimalen Lösungen führt. Außerdem entstanden bei der Modellierung mit ILP teilweise sehr große Logikprogramme. Dem entsprechend war die Speicherauslastung sehr hoch und es konnten keine Lösungen in einem vorgegebenen Zeitfenster gefunden werden. Dies liegt vermutlich daran, dass die ILP-Formulierungen direkt in ASP übersetzt und somit die Vorteile von ASP nicht ausgereizt wurden. In der vorliegende Arbeit wird mit der Antwortmengenprogrammierung eine Methode gewählt, die gute Ergebnisse liefert und auch bei Erstellung und Weiterentwicklung der Modellierung Vorteile aufweisen kann. Darüber hinaus wird der datenstromorientierte Charakter der Architekturen besonders berücksichtigt.

Für die Umsetzung von komplexen Bildverarbeitungssystemen wurde für die vorliegende Arbeit der Betriebssystemkern *Linux* in Kombination mit der Bibliothek *OpenCV* für Echtzeit Computer Vision ausgewählt, weil beides mächtige Softwarelösungen sind, sie im Bereich eingebetteter Systeme bzw. Bildverarbeitung weit verbreitet sind, jeweils eine sehr große Community an Entwicklern und Firmen dahinter stehen und durch ihren Open-Source-Charakter frei verfügbar und individuell anpassbar sind [Linb, Lina]. Diese Basis stellt eine gewohnte und komfortable Umgebung für viele Softwareentwickler dar und ermöglicht darüber hinaus einen einfachen Einstieg in die Entwicklung auf eingebetteten Systemen. Außerdem ist es möglich, die Anwendung auf einem Computer zu entwickeln und zu testen, und nachfolgend mit wenig Aufwand auf ein eingebettetes System zu übertragen.

3 Entwurf von komplexen Bildverarbeitungssystemen

Im Rahmen des von der Deutschen Forschungsgemeinschaft (DFG) geförderten Projekts AMoC (Adaptive Multiprocessors-on-Chip) wurde ein Beitrag zum automatischen Entwurf von Mehrprozessorsystemen auf dynamisch rekonfigurierbaren Chips, insbesondere FPGAs, geleistet [DFGb, DFGa, PTI]. Neben der Untersuchung und Entwicklung von Modellen und Entwurfsmethoden wurde auch die Verwendung solcher Architekturen z. B. für zeitkritische und leistungsstarke Anwendung betrachtet. Die Systemarchitektur beeinflusst im Allgemeinen die Kommunikation maßgeblich. Deshalb wurde auch der Einsatz flexibler Kommunikationsnetzwerke und -protokolle untersucht, welche die Zustellung von Nachrichten auch nach der Umplatzierung eines Hardwaremoduls auf dem Chip gewährleisten können. Mit Hilfe eines neu entwickelten Werkzeugs kann eine Mehrprozessor-Systemarchitektur halbautomatisch aus einem C-Programm abgeleitet werden, um am Ende einen Bitstream zu synthetisieren. Der Entwurfsfluss ist in Abbildung 7 dargestellt.

Das Werkzeug arbeitet mit zwei Sichten auf das System: Der *abstrakten* und der *konkreten*. In der hardwareunabhängigen abstrakten Sicht werden die Komponenten eines Systems in folgende Klassen eingeteilt: Prozessoren, Speicher, Kommunikationsmedien, Peripheriebausteine und Hardwarebeschleuniger. Zur Erstellung eines neuen Systems werden Komponenten instantiiert und die Verbindungen untereinander spezifiziert. Diese abstrakte Systembeschreibung wird für die funktionale Simulation und die Validierung herangezogen. In einem zweiten Schritt wird anschließend die abstrakte Sicht konkretisiert. Dazu wird jeder Komponente eine bestimmte Implementierung zugeordnet, z. B. CPU0 → PowerPC, Mem0 → DDR-RAM, Mem1 → Block-RAM, Periph0 → RS232. Diese konkrete Beschreibung kann in eine herstellerspezifische Version übergeführt und anschließend synthetisiert werden.

Dieser Teil des Entwurfsflusses wurde in [2] veröffentlicht. Weitere Arbeiten des Projekts, wie die Analyse von C- und MPI-Programmen und die Entwurfsraumexploration mit einem ILP-Modell, wurden in [BMAI10] veröffentlicht.

Bildverarbeitende eingebettete Systeme operieren meist im Verbund mit anderen Systemen, was ihre Komplexität erhöht. Aus diesem Grund ist der Einsatz eines Betriebssystems zweckmäßig, da es grundlegende Funktionen, wie z. B. den Zugriff auf Speicher- und Kommunikationsmedien oder eine Prozess- und Speicherverwaltung bereitstellen kann. In

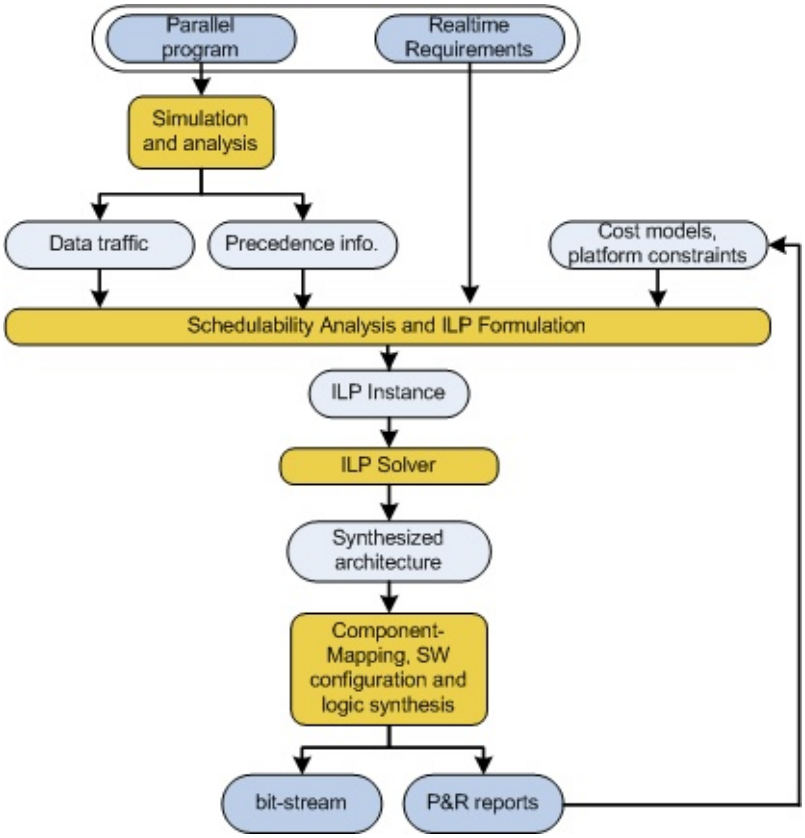


Abbildung 7: Entwurfsfluss für adaptive und rekonfigurierbare Mehrprozessorsysteme auf FPGAs (DFG-Projekt AMoC).

der vorliegenden Arbeit liegt der Fokus auf komplexen Bildverarbeitungssystemen, welche ein Betriebssystem erfordern.

Die, u. a. von Felix Mühlbauer, im Rahmen des AMoC-Projekts geleisteten Vorarbeiten und Ergebnisse wurden in der vorliegenden Arbeit zu einem durchgehenden Entwurfsfluss für komplexe Bildverarbeitungssysteme weiterentwickelt. Durchgehend bedeutet, dass alle Arbeitsschritte von der Anwendung bis hin zum lauffähigen System betrachtet werden. Soweit möglich und sinnvoll werden Vorgänge automatisiert, um die Entwicklung zu vereinfachen und Fehlerquellen zu reduzieren. Nach einer Betrachtung der Besonderheiten von bildverarbeitenden Systemen, wird eine passende Struktur der Systemarchitektur und der Entwurfsfluss vorgestellt.

3.1 Eigenschaften von Bildverarbeitungsalgorithmen

Bilddaten lassen sich sehr gut in Datenströmen organisieren und verarbeiten [BFRR01, Pel]. In der vorliegenden Arbeit wurde die Klassifizierung von Systemkomponenten aus dem AMoC-Projekt weiterentwickelt und an datenstromorientierte Systeme angepasst.

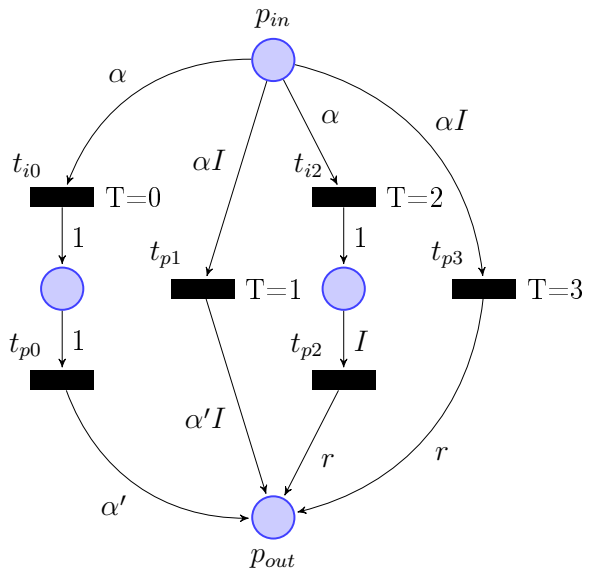
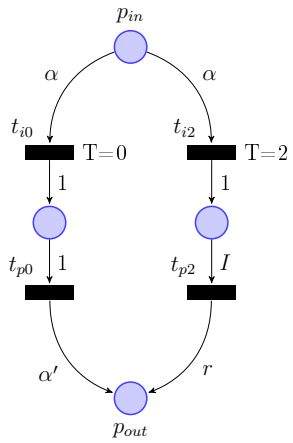


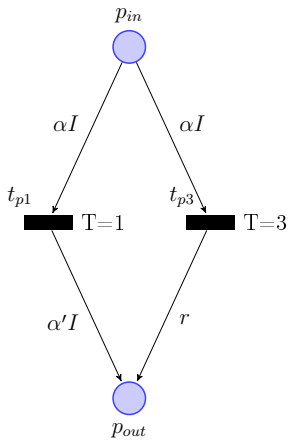
Abbildung 8: Modellierung des Datenflussverhaltens von Berechnungsfunktionen mit Hilfe eines synchronen Datenflussgraphen und Klassifizierung der Implementierung in verschiedene Typen T (fixed data rate/result, streaming/random access).

Dazu werden im Folgenden Systemkomponenten, welche Bildverarbeitungsfunktionen implementieren, *IPE (image processing element)* genannt. Auf Grund ihres Verhaltens können IPEs in vier Klassen eingeteilt werden. Das Verhalten eines IPEs wurde als synchroner Datenflussgraph modelliert und ist in Abbildung 8 dargestellt. Dabei wird ein Datenbit durch eine Marke repräsentiert. Die Marken bewegen sich entsprechend dem Typ der Implementierung T durch einen der vier Pfade von p_{in} nach p_{out}. Dabei gibt α die Anzahl der Bits pro Bildpunkt und I die Anzahl der Bildpunkte in einem Bild an. Die vier Typen lassen sich wiederum in verschiedenen Klassen zusammenfassen, welche im Folgenden einzeln dargestellt werden. In Abbildung 8 sind alle Typen und Klassen zusammengefasst.

Typ 0 und 2 (siehe Abbildung 9a) sind datenstromorientierte Implementierungen (streaming), welche Bildpunkte einzeln einlesen und verarbeiten. Sie sind meist gut für eine



(a) Datenstromorientierte Verarbeitung



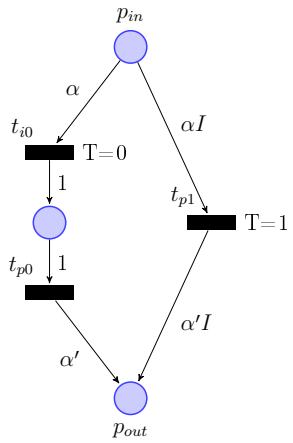
(b) Verarbeitung mit nicht linearem Speicherzugriff

Abbildung 9

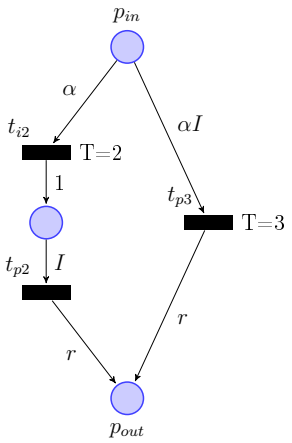
Realisierung in Hardware geeignet. IPEs vom Typ 0 produzieren darüber hinaus auch wieder einen Datenstrom und können deshalb wie in einer Pipeline hintereinander geschaltet parallel arbeiten. Nach einer gewissen Initialverzögerung t_i werden Ergebnisse mit einer Rate von $\frac{1}{t_p}$ produziert. Beispielsweise benötigt ein Faltungsfilter mit einer 3×3 Matrix, zuerst ein Quadrat von 9 Punkten aus der Bildecke des ersten Bildpunkts (meist oben links), um das erste Ergebnis berechnen zu können. Dies ist der Fall, nachdem das IPE aus dem eingehenden Datenstrom zwei Zeilen und weitere drei Bildpunkte eingelesen hat.

Typ 1 und 3 (siehe Abbildung 9b) sind Implementierungen, welche nicht linear auf das Bild zugreifen (random access). Dieser Sachverhalt wurde dadurch modelliert, dass die Transitionen, t_{p1} bzw. t_{p3}, Marken entsprechend eines ganzen Bildes auf einmal konsumieren und auch das Ergebnis in einem Schritt produzieren. Alternativ kann die Modellierung als eine Sperre auf den Speicherbereich der Bilddaten interpretiert werden. Damit ist neben der aus der Modellierung direkt ableitbaren Synchronisierung zwischen IPEs keine weitere notwendig. Diese Erkenntnis kann in die Codegenerierung während der Systemsynthese mit einfließen. Die Zeit, welche die IPE für die Verarbeitung benötigt, wird ebenfalls durch die Transition t_{p1} bzw. t_{p3} modelliert. Zu dieser Klasse gehören z. B. kontrolllastige Algorithmen, welche sich deshalb gut in Software realisieren lassen.

Typ 0 und 1 (siehe Abbildung 10a) beschreiben filterähnliche Operationen, welche ein Bild mit α Bits pro Bildpunkt konsumieren und ein Bild, ggf. mit einer anderen Datenrate



(a) (Durchschnittlich) Konstante Datenrate



(b) Konstante Ergebnisgröße

Abbildung 10

α', wieder ausgeben (fixed data rate). Beispiele sind Glättungsfilter, Gamma- oder Kontrastanpassung oder einfache Kantenfilter. Eine Veränderung der Datenmenge ($\frac{\alpha}{\alpha'} \neq 1$) tritt beispielsweise bei der Konvertierung eines Graustufenbildes in ein Schwarzweißbild oder bei Kompressionsalgorithmen auf.

Typ 2 und 3 (siehe Abbildung 10b) beschreiben Operationen, welche aus einem Bild ein Ergebnis der Größe r erzeugen (fixed result), wie z. B. die durchschnittliche Helligkeit eines Bildes oder ein Histogramm. Dabei gilt $r \ll I$, da ansonsten das IPE eher der vorhergehenden Klasse zugeordnet werden kann.

Je nach Implementierungstyp (streaming / random access) werden die Daten von IPE zu IPE weitergeleitet oder müssen andernfalls in einem Speicher gepuffert werden. Folglich müssen beim Architekturentwurf Prozessoren, PUs und Speicher miteinander und untereinander verbunden werden können. Da Bilddaten in großen Mengen auftreten, sind sie meist in einem externen Speicher untergebracht. Diese Eigenschaften müssen für die Strukturierung einer Architektur für bildverarbeitende Systeme berücksichtigt werden.

3.2 Architektur für Bildverarbeitungssysteme

Auf der Basis der bisher beschriebenen Eigenschaften von Bildverarbeitungsanwendungen wird in diesem Kapitel eine konkrete Struktur für Systemarchitekturen dieser Anwendungen vorgestellt. Ein Minimalsystem hat mindestens einen Hauptprozessor, auf dem das Betriebssystem ausgeführt wird, einen Systembus und einen Speicher in dem die Software und die Bild- bzw. Ergebnisdaten untergebracht werden können. Durch die Erweiterung des Systems um Hardwarebeschleuniger können Berechnungen in Hardware ausgelagert werden.

Wie im Abschnitt 3.1 definiert werden in der vorliegenden Arbeit Systemkomponenten, welche Bildverarbeitungsfunktionen implementieren, *IPE (image processing element)* genannt. Dabei soll zwischen Softwareimplementierungen und Hardwareimplementierungen unterschieden werden. Insbesondere werden im Folgenden Software-ausführende IPEs als *Prozessoren* und datenstromorientierte, in Logik implementierte IPEs als *PU (processing unit)* bezeichnet.

Neben der Implementierung von IPEs spielt auch die Kommunikation zwischen den IPEs eine wichtige Rolle für die Effizienz eines Systems. In datenstromorientierten Architekturen treten zwei Arten von Parallelität auf:

Multicore: Mehrere IPEs arbeiten unabhängig voneinander parallel und

Pipelining: Mehrere IPEs sind in einer Kette hintereinander geschaltet und die Daten werden verarbeitet, während sie die Kette durchfließen. Dafür sind insbesondere datenstromorientierte PUs geeignet.

Um die Wiederverwendbarkeit von PUs für verschiedene Anwendungen bzw. Systemarchitekturen zu gewährleisten, müssen PUs eine einheitliche Schnittstelle für die Kommunikation aufweisen. Dafür wurde in der vorliegenden Arbeit das Streaming Data Interface (SDI) entworfen (siehe auch Abschnitt 6.1.1). Die Schnittstelle wurde bewusst einfach gehalten, benutzt eine parallele Datenübertragung und weist eine Flusskontrolle zur Handhabung unterschiedlicher Verarbeitungsgeschwindigkeiten und zur Vermeidung von Verlusten bei Datenstaus auf.

Um die Kommunikation zwischen Hardwaremodulen, sowie zwischen Hardware und Software zu ermöglichen, sind bestimmte Verbindungselemente notwendig. Diese Komponenten können in zwei Kategorien eingeteilt werden:

Verarbeitung	Beispiele
PUs (processing unit), SDI ⇆ SDI	
Berechnung	Faltung, Histogramm, Farbraumkonvertierung, ...
Datenflusskontrolle	Teilen/Zusammenführen, (De)Serialisieren, Puffer
IMs (interconnection module), SDI ⇆ □	
Prozessor	FSL (MicroBlaze), APU (PowerPC), ...
Speicher indirekt	DDR-RAM via Systembus, z. B. PLB, AMBA, ...
Speicher direkt	BlockRAM, SRAM, ...

Tabelle 1: Hardwaremodule für die Verarbeitung von Datenströmen.

- Module, die den Datenstrom manipulieren (PUs) und
- Module zur Ankopplung eines Datenstroms an andere Systemkomponenten (IMs).

In Abbildung 11 ist eine Beispielarchitektur mit verschiedenen Schlüsselkomponenten dargestellt.

Neben der Änderung der Daten in einem Strom, z. B. durch Berechnungen, kann der Datenstrom auch z. B. verzweigt, synchronisiert oder in seiner Breite verändert werden. Dagegen werden zur Ankopplung eines Datenstroms an z. B. Speicher oder Prozessoren Adaptermodule benötigt, welche hier *Interconnection Moduls (IM)* genannt werden. Der Transfer eines Datenstroms in einen Speicher kann direkt, z. B. im Falle eines lokalen BlockRAMs, oder indirekt, z. B. über den Systembus auf den Hauptspeicher, erfolgen. Viele eingebettete Prozessoren sind mit einer Koprozessorschnittstelle ausgestattet, welche ebenfalls zur Anbindung von SDI-Kanälen benutzt werden kann. In Tabelle 1 sind die verschiedenen Komponenten mit Beispielen zusammengefasst.

3.3 Entwurfsfluss

Für den in der vorliegenden Arbeit entwickelten Entwurfsfluss wurde folgende Betrachtungsweise und Vorgehensweise verfolgt: Zu Beginn steht eine OpenCV-Anwendung (Programm), für die ein effizientes Rechensystem gefunden und anschließend das Gesamtsystem, d. h. Software- und Hardwarekomponenten, synthetisiert werden soll. Für die Realisierung derart vielfältiger Architekturen werden FPGAs eingesetzt. Als Ausgangspunkt wird eine reine Softwarelösung angenommen, d. h. alle Berechnungen laufen auf dem

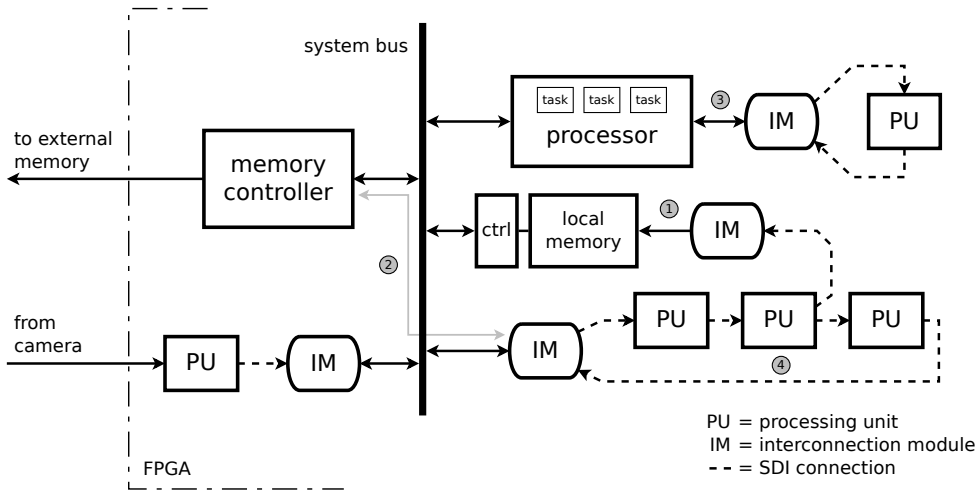


Abbildung 11: Beispielarchitektur: Software und Hardware IPEs, direkte (1) und indirekte (2) Speicherkommunikation, Koprozessorschnittstelle (3), Pipelining (4)

Hauptprozessor ab. Als Softwarefundament dient ein *Linux*-basiertes Betriebssystem in Kombination mit der Bibliothek *OpenCV* für Echtzeit Computer Vision. Anschließend werden ausgewählte Programmteile der Bildverarbeitung durch die Erweiterung der Basisarchitektur um dedizierte Hardwarebausteine beschleunigt. Die Integration der Hardware erfolgt durch Modifikation der Implementierung der OpenCV-Bibliothek und bleibt damit vor dem Anwendungsprogrammierer verborgen. Mit Hilfe einer Entwurfsraumexploration wird ermittelt, welche Berechnungen in Software und welche in Hardware erfolgen sollen, um vorgegebenen Optimierungszielen zu genügen. Der Entwurfsfluss ist in Abbildung 12 vereinfacht dargestellt und wurde in [14] veröffentlicht. Am Ende der Arbeit ist eine ausführliche Version zu finden (siehe Abbildung 22*).

Aufbau des Hardwarepools. Für den Aufbau eines Pools von Hardwarekomponenten müssen die für eine Realisierung in Hardware geeigneten Berechnungen in der OpenCV-Bibliothek identifiziert und implementiert werden. Die Bibliothek gliedert sich in mehrere Module, welche teilweise aufeinander aufbauen. Grundfunktionen werden von Funktionen auf höheren Abstraktionsebenen aufgerufen. Werden Grundfunktionen beschleunigt, wirkt sich dies folglich auch positiv auf höhere Funktionen aus. Die Auswahl geeigneter Funktionen erfolgt per Intuition oder mit Hilfe geeigneter Profiling-Werkzeuge. Mittels freier

*auf Seite 85

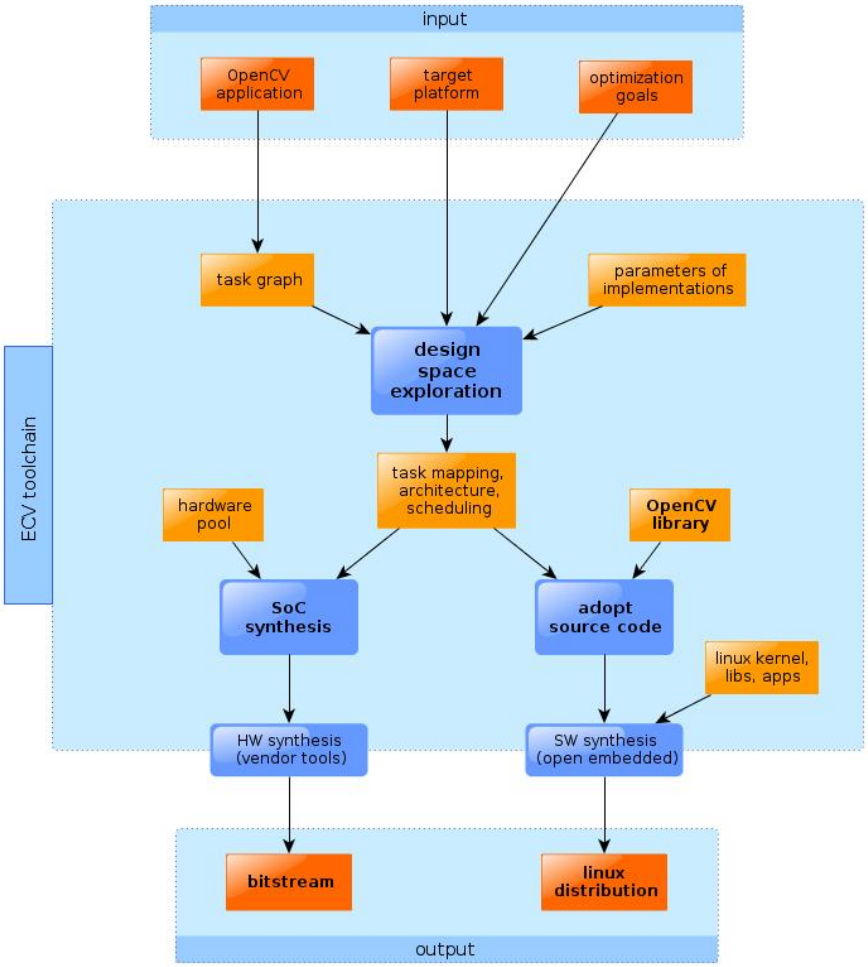


Abbildung 12: Vereinfachter Entwurfsfluss

Software, wie z. B. `gprof` oder VALGRIND, können Abhängigkeiten und Ausführungszeiten, während der Laufzeit der Anwendung, einfach bestimmt und visualisiert werden. Neben bestimmten Bildverarbeitungsfunktionen, die komplett für eine Realisierung in Logik geeignet sind, ist an anderer Stelle nur eine teilweise Implementierung einer Funktion in Hardware sinnvoll. Gegebenenfalls sind auch mehrere Implementierungen der selben Funktion mit unterschiedlichen Eigenschaften, wie z. B. Datendurchsatz und Ressourcenverbrauch, zweckmäßig.

Die Entwicklung von Hardwarekomponenten (PUs) kann mit Hilfe verschiedener Methoden und Sprachen erfolgen, wie z. B. VHDL, VERILOG, HANDELC oder Generatorwerkzeugen für endliche Automaten, MATLAB, SYSTEMC, C und andere, welche eine

Hardwarebeschreibung oder Netzlisten erzeugen können. Dies erlaubt eine einfache und vielfältige Möglichkeit, ein Sortiment an Hardwaremodulen aufzubauen, zu erweitern und auch für andere Anwendungen wiederzuverwenden.

Neben PUs für Berechnungen sind auch PUs zur Kontrolle des Datenstroms zu implementieren. Durch die gemeinsame Schnittstelle (SDI) sind PUs plattformunabhängig. Weiter müssen IMs passend zu den Kommunikationsmöglichkeiten in der Zielarchitektur und entsprechende Betriebssystem-Treiber implementiert werden.

Analyse der Anwendung. Voraussetzung für die Entwurfsraumexploration ist der Problemgraph der OpenCV-Anwendung (siehe Abbildung 3a), welcher die Datenabhängigkeiten der Funktionsaufrufe repräsentiert. Die Ermittlung der Datenabhängigkeiten kann manuell erfolgen, da der Anwendungsentwickler seinen Algorithmus kennen sollte, oder auch (halb)automatisch mit Hilfe entsprechender Werkzeuge, wie z. B. dem SUIF (Stanford University Intermediate Format) Compiler [SUI].

Entwurfsraumexploration. Für jede Bildverarbeitungsfunktion existiert mindestens eine Implementierung in Software (OpenCV-Bibliothek) und ggf. weitere in Hardware. Die Eigenschaften der Implementierungen, wie z. B. Ausführungszeit, Datendurchsatz oder Ressourcenverbrauch (siehe 4.2 *Metainformationen*) und der Problemgraph der OpenCV-Anwendung fließen zusammen mit Angaben über Zielplattform und Optimierungsziele in die Suche nach effizienten Lösungen für die Systemarchitektur ein. Optimierungsziele sind z. B. die zur Verfügung stehende Chipfläche und die Suche nach der schnellsten Architektur.

In Systemen mit mehreren parallel- und zusammenarbeitenden Einheiten spielt die Kommunikation zwischen den einzelnen Einheiten eine wichtige Rolle und hat maßgeblichen Einfluss auf die Effizienz des Gesamtsystems. Die Kommunikation muss folglich, neben der Auswahl von Hardwaremodulen, ebenfalls in den Entwurf der Systemarchitektur einfließen. Aus diesem Grund sind Allokation, Bindung und Ablaufplanung stark miteinander verbunden und müssen gleichzeitig gelöst werden, um zu einer optimalen Lösung zu gelangen.

Die Parameter, welche Systemaufbau und -eigenschaften bestimmen, sind oft von zu großer Anzahl und die Zusammenhänge zu komplex, als dass ein Entwickler diese noch überblicken könnte. Deshalb ist eine automatisierte Entwurfsraumexploration oft sinnvoll

[Ish09]. In der vorliegenden Arbeit wird mit Hilfe von ASP (answer set programming) ein neuer Ansatz verfolgt, den Entwurfsraum zu durchsuchen. Die Ergebnisse werden in der Systemsynthese weiterverarbeitet.

Highlevel-Synthese. Die Systemsynthese betrifft sowohl Hardware als auch Software. Auf der Hardwareseite wird die SoC-Architektur generiert. Der Aufbau der Architektur geht aus den Ergebnissen der Entwurfsraumexploration hervor und beinhaltet, die zu verwendenden Hardwarekomponenten und wie diese miteinander verbunden werden sollen. Aus einem Minimalsystem und dem Hardwarepool werden anschließend die konkreten Implementierungen zusammengeführt.

Auf der Softwareseite wird der Quelltext der OpenCV-Bibliothek modifiziert. Programmteile, in welchen die Berechnung gemäß den Ergebnissen der Entwurfsraumexploration in Hardware ausgelagert werden soll, werden angepasst. Die ursprüngliche Implementierung wird durch eine andere, welche die Kommunikation mit der entsprechenden Hardwarekomponente durchführt, ersetzt. Ebenfalls erfolgt die damit verbundene Konfiguration der Treiber.

Lowlevel-Synthese. Aus der in der Highlevel-Synthese vorbereiteten SoC-Architektur kann nun durch die Synthesewerkzeuge des FPGA-Herstellers ein *Bitstream* erzeugt werden.

Ebenfalls muss eine Linux-Distribution erzeugt werden. Diese beinhaltet neben dem Linux-Kernel hauptsächlich auch Anwendungen und Bibliotheken. Die gesamte Software muss für das Zielsystem kompiliert werden, was einen komplexen Vorgang darstellt. In der vorliegenden Arbeit wurde das Werkzeug *OpenEmbedded* an die verwendeten FPGA-Architekturen angepasst. Dadurch konnte der gesamte Softwaresyntheseprozess automatisiert werden.

4 Entwurfsraumexploration

Dieses Kapitel behandelt das Suchen nach der besten Architektur für eine vorgegebene Bildverarbeitungsberechnung, gemessen an bestimmten Optimierungszielen. Letztendlich geht es darum, für jede Funktion der Berechnung eine geeignete Implementierung (Software oder Hardware) auszuwählen und die Kommunikation im System festzulegen. Jede Entscheidung kann weitreichenden Konsequenzen auf die Eigenschaften des Gesamtsystems, wie z. B. Platzbedarf oder Geschwindigkeit haben. Ausgehend von der Vielzahl an Rekombinationsmöglichkeiten der Hardwarekomponenten wird die Suche durch die Anforderungen an das System eingeschränkt. Trotzdem wächst der Suchraum mit der Anzahl der Hardwarekomponenten und der Verbindungsmöglichkeiten sehr stark an und es ist schwierig, alle möglichen Lösungen in einem angemessenen Zeitraum zu prüfen.

Für die Entwurfsraumexploration gibt es verschiedene Methoden, Lösungen zu finden. Wie in Abschnitt 2.4 gegenübergestellt, decken die Methoden den Entwurfsraum unterschiedlich ab und die Lösungen unterscheiden sich in ihrer Qualität verglichen mit dem Optimum. In der vorliegenden Arbeit wird mit ASP (answer set programming) ein neuer Ansatz verfolgt.

4.1 Problemstellung

In der vorliegenden Arbeit wird die Methode von [TH07] aufgegriffen, welche den Entwurf von digitalen Hardware-Software-Systemen allgemein in die Arbeitsschritte Allokation, Bindung und Ablaufplanung einteilt. Bei der Allokation werden die für die Zielarchitektur zu verwendenden Hardwarekomponenten identifiziert und festgelegt. Die Bindung definiert, welche Teile der Berechnung auf welchen Systemkomponenten auszuführen sind, und die Ablaufplanung wann die Berechnungen stattfinden sollen. Dies muss jeweils unter Berücksichtigung von Systemanforderungen bzw. -einschränkungen und Optimierungszielen erfolgen. In den meisten Anwendungsfällen sind das die begrenzte FPGA-Chipfläche und eine hohe Verarbeitungsgeschwindigkeit.

Bei der Suche nach einer geeigneten Architektur sind einige Zusammenhänge zu berücksichtigen. Für die Bindung stehen für jede Funktion im Problemgraphen verschiedene Implementierungen in Software oder Hardware zur Verfügung, gegebenenfalls auch mehrere mit unterschiedlichem Ressourcenverbräuchen. Gewöhnlich ist die Implementierung

einer Funktion in Hardware seinem Softwarependant in puncto Verarbeitungsgeschwindigkeit deutlich überlegen. Jedoch ist es nicht vorteilhaft oder möglich, alle Berechnungen in Hardware auszulagern. Einerseits wird die Allokation der Hardwarekomponenten durch die vorhandenen FPGA-Ressourcen (Logikzellen, DSP-Einheiten, Speicher, ...) beschränkt. Andererseits können nicht alle Hardwarekomponenten gleichzeitig auf den Speicher zugreifen, da dann die Kommunikation zum Engpass wird. Darüber hinaus kann möglicherweise bei gleichem Ressourceneinsatz durch zwei kleine Module insgesamt eine höhere Geschwindigkeit erreicht werden, als durch den Einsatz eines großen Moduls für eine andere Funktion. Es muss folglich eine Lösung gefunden werden, welche die vorhanden Ressourcen möglichst effizient nutzt.

Wie bereits angesprochen, spielt die Kommunikation in Hardware-Software-Systemen für die Performance eine essentielle Rolle. Durch die Zeitverzögerung beim Datentransfer zwischen Modulen (IPEs) kann es schneller sein, die Daten lokal zu verarbeiten, als sie zu einem benachbarten Spezialmodul zu transferieren, dort vergleichsweise schneller zu verarbeiten und anschließend zurück zu transferieren. Gegenüber Problemen mit einer vorgegebenen Systemarchitektur ist die Komplexität der Ablaufplanung für die in der vorliegenden Arbeit entworfenen Systeme durch die variabel definierbare Architektur folglich um ein vielfaches höher.

Nicht nur die Architektur ist nicht statisch, sondern auch die Verarbeitungsgeschwindigkeit der Module. Durch die Bindung einer Funktion an eine Hardwarekomponente ist lediglich die maximale Verarbeitungsgeschwindigkeit definiert, jedoch nicht die Tatsächliche. Der Datendurchsatz eines Hardwaremoduls wird maßgeblich vom Datendurchsatz der direkt angeschlossenen Komponenten beeinflusst. Einerseits spielen hier die Enden eines SDI-Pfads eine Rolle. Die direkte Kommunikation, z. B. zu einem internen Speicher oder über die Coprozessorschnittstelle, ist meist schneller als indirekt über den Systembus, z. B. zum externen Speicher. Da die Bilddaten auf Grund der großen Datenmengen im externen Speicher untergebracht sind, wird beim simultanen Zugriff mehrerer Komponenten der Speicherkontroller zu einem Engpass und die Datenverarbeitung der Module wird gedrosselt. Andererseits wird der Datenfluss durch Warten und Staus beeinträchtigt. In einer Kette von PUs kann ein Modul nur so schnell arbeiten, wie die Daten vom Vorgängermodul geliefert werden. Sind nachfolgende PUs langsamer, können folglich die Module davor nicht ihre maximal mögliche Geschwindigkeit ausreizen. Das langsamste Glied in der Kette bestimmt die Geschwindigkeit.

Für die Erstellung der Ablaufplanung ist daher die Planung der gesamten Kommunikation im System erforderlich. Die dafür notwendigen Parameter, wie z. B. Datendurchsatz und maximale Verarbeitungsgeschwindigkeit, sind aus den Metadaten der Module (siehe Kapitel 4.2) ersichtlich.

Wie in Kapitel 3.2 *Architektur für Bildverarbeitungssysteme* dargestellt, können PUs, je nach Implementierung, auch in einer Verarbeitungskette (pipelining) betrieben werden. Dabei greifen Start- und Endzeitpunkte von, von einander abhängigen, Berechnungen ineinander, was bei der Ablaufplanung gesondert berücksichtigt werden muss und sich damit von der Ablaufplanung von Mehrprozessorsystemen, wie z. B. die Verteilung eines MPI-Programms auf mehrere Prozessoren, grundlegend unterscheidet.

Hardwaremodule können nicht beliebig miteinander verbunden werden, wenn sie sich in ihrem Zugriffstyp (streaming/random access) unterscheiden. Mit Hilfe eines ausreichend großen Zwischenspeicher kann der Zugriff auf die Daten zwischen den beiden Klassen konvertiert werden, da er beide Zugriffsarten unterstützt. Somit ergeben sich noch weitere Möglichkeiten, wie sich eine Berechnung in der Architektur widerspiegeln kann.

4.2 Metainformationen

Für die Exploration des Entwurfsraums sind Metainformationen über die einzelnen Implementierungen von Bildverarbeitungsfunktionen notwendig. Es bietet sich an, diese in einer Datenbank zu speichern. Im Folgenden werden die Eigenschaften von Implementierungen zusammengefasst, welche teilweise bereits im Datenflussmodell in Abbildung 8* beschrieben wurden:

Name: Jede Implementierung hat einen eindeutigen *Namen*.

Funktionsname: Jede Implementierung ist einer bestimmten Berechnungsfunktion mit einem eindeutigen *Funktionsnamen* zugeordnet. Diese Funktionsnamen werden auch im Problemgraph verwendet. Da für jede Berechnung eine Softwareimplementierung existiert, bietet es sich an, diesen Funktionsnamen zu verwenden. Es können mehrere Implementierungen für eine Funktion existieren, z. B. Hardwaremodule mit verschiedenen Verarbeitungsgeschwindigkeiten und Ressourcenverbräuchen.

*auf Seite 35

Implementierungstyp: Jede Implementierung kann entsprechend der Klassifizierung im Datenflussmodell einem Typ T zugeordnet werden.

Verarbeitungsgeschwindigkeit: Die Verarbeitungsgeschwindigkeit wird durch die Verarbeitungsverzögerung t_p und ggf. einer Initialverzögerung t_i bestimmt. Bei Logikimplementierungen ist darüber hinaus die maximale Taktgeschwindigkeit von Bedeutung.

Datendurchsatz: Die Menge an Daten, welche eingelesen bzw. ausgegeben wird, spiegelt sich in den Parametern α und α' bzw. r wieder.

Ressourcenverbrauch: Implementierungen in Logik lassen sich im Wesentlichen durch den Verbrauch von Logikzellen, internem Speicher (BlockRAM) und DSP-Blöcken charakterisieren, während bei Softwareimplementierungen hauptsächlich der Speicherbedarf wichtig ist.

Nicht alle Eigenschaften sind offensichtlich und müssen ggf. erst ermittelt werden. Bestimmte Eigenschaften können von der Bildgröße abhängig sein, wie z. B. die Verarbeitungsgeschwindigkeit bei einem Histogramm oder die Initialverzögerung eines Faltungsfilters. Die Erfassung der Zeitparameter erfolgt durch Simulation, Messungen oder Schätzung. Bei Logikimplementierungen können die Parameter sehr genau durch zyklenakurate Simulation gewonnen werden. Für die Softwareimplementierungen hingegen können in einem laufenden System mit Hilfe von geeigneten Profilingwerkzeugen, wie `gprof` oder VALGRIND, Zeitmessungen durchgeführt werden.

4.3 Modellierung in ASP

Die Antwortmengenprogrammierung bedient sich Fakten, Regeln und anderen Sprachkonstrukten, um ein Problem zu spezifizieren. Basierend auf dieser formalen Beschreibung kann ein ASP-Solver die Menge von Fakten bestimmen, welche zu dem gegebenen Modell passen. Es bedarf einer gewissen Erfahrung bei der Modellierung, um bei der Suche eine Balance zwischen Ausweiten und Einschränken des Suchraums zu finden, so dass der ASP-Solver nicht die vorhandenen Rechenressourcen überstrapaziert.

Die Modellierung in der vorliegenden Arbeit ist in drei Teile gekapselt und auch im Anhang zu finden. Durch die Aufteilung ist die Modellierung einfach für verschiedene Anwendungsfälle wiederverwendbar:

Problembeschreibung: Problemgraph und Rahmenbedingungen für die gesuchte Architektur, wie z. B. Chipfläche (siehe Listing 1*)

Metainformationen: Parameter der Implementierungen und der Kommunikationsmodule (siehe Listing 2)

Antwortmengenprogramm: Regeln zur Lösung des Problems (siehe Listing 3).

Die Grundidee für die Modellierung ist, für jeden Berechnungsschritt aus dem Problemgraph eine Implementierung auszuwählen, die zugehörigen Komponenten miteinander zu verbinden und schließlich die Abhängigkeiten und die Ablaufplanung zu prüfen. Als Optimierungsziel wurde die Laufzeit gewählt. Die ASP-Formulierungen und Laufzeitverbesserungen werden im Folgenden beschrieben. Zur besseren Lesbarkeit sind die dargestellten Formeln teilweise vereinfacht. Die Entwicklungen wurde in [13] veröffentlicht.

4.3.1 Allokation von Komponenten

Zuerst wird jeder Verarbeitungsschritt aus dem Problemgraphen genau einer IPE zugeordnet. Die IPEs können Software- oder Hardwareimplementierungen sein. Diese Zuordnung wird in ASP durch ein Atom repräsentiert. Sei N die Anzahl der Komponenten, die für eine Architektur benötigt werden, so gibt es $N!$ Permutationen für diese Atome. Um diese Symmetrien einzuschränken und damit den Suchraum zu vereinfachen, wird eine Komponente durch ein Atom c_{ij} definiert. Die beiden Indizes sind der Implementierungstyp i und ein Instanzzähler $j \in [1; J_i]$, wobei J_i die maximale Anzahl an Instanzen der Komponenten vom Typ i ist. Somit reduziert sich die Anzahl der Permutationen auf $J_1! \cdot \ldots \cdot J_n!$. Die Werte von J_i können aus dem Problemgraph abgeleitet werden. In den meisten Fällen ist es nicht notwendig, eine Komponente mehrfach zu instantiieren, und damit ist $J_i = 1$.

Für jede instantiierte Komponente c_{ij} und jeden Berechnungsschritt t_n (=Knoten im Datenflussgraph) wird ein Atom $M_{t_n c_{ij}}$ definiert. Folglich muss für jeden Berechnungsschritt t_i die Summe der zugeordneten Komponenten genau eins sein:

$$1 \; [\; M_{t_i c_{11}}, \ldots, M_{t_i c_{kl}} \;] \; 1. \tag{8}$$

* auf Seite 96

4.3.2 Datenfluss

Nachdem alle Berechnungskomponenten instantiiert wurden, müssen sie miteinander verbunden werden. Verbindungen werden aus den Kanten des Problemgraphs abgeleitet und können einfache Punkt-zu-Punkt-Verbindungen sein, oder andererseits auch noch weitere Komponenten erfordern. Sollen zum Beispiel Daten von einer PU zu einem Speicher transferiert werden, muss dazwischen ein IM geschaltet werden, um die verschiedenen Schnittstellen zu verbinden.

Für jeden Transfer n wird ein Atom $C_{nc_{ij}c_{kl}}$ definiert, welches anzeigt, dass Data von der Komponente c_{ij} zur Komponente c_{kl} geschickt werden. Dieses Atom existiert nur, wenn der Transfer auch tatsächlich stattfindet:

$$0\ [\ C_{nc_{ij}c_{kl}}\ ]\ 1. \tag{9}$$

Nachdem diese Atome generiert wurden, müssen sie auf sinnvolle eingeschränkt werden. Die Modellierung dieser Einschränkungen ist denen von Wegfindungsalgorithmen ähnlich. Angenommen der Transfer n beschreibt die Abhängigkeiten zwischen den beiden Berechnungsschritten t_x (Quelle) und t_y (Senke), dann müssen folgende Integritätsbedingungen gelten:

$$\leftarrow M_{t_x c_{ij}},\ [\ C_{nc_{ij}c_{kl}}\ ]\ 0. \tag{10}$$

$$\leftarrow M_{t_y c_{ij}},\ [\ C_{nc_{kl}c_{ij}}\ ]\ 0. \tag{11}$$

Das heißt, wenn ein datenproduzierender Berechnungsschritt t_x der Komponente c_{ij} zugeordnet wurde, dann muss eine Komponenten c_{kl} existieren, zu der im Transfer n von c_{ij} Daten gesendet werden: $\sum C_{nc_{kl}c_{ij}} > 0$ (siehe Regel 10). Ebenso, wenn ein datenkonsumierender Berechnungsschritt t_y der Komponente c_{ij} zugeordnet wurde, dann muss eine Komponenten c_{kl} existieren, die im Transfer n zu c_{ij} Daten sendet (siehe Regel 11).

Darüber hinaus stellt das Modell sicher, dass für jeden Transfer ein Pfad zwischen den Quell- und Zielkomponenten existiert und außerdem unsinnige Verbindungen ausgeschlossen werden, wie z. B. im Falle von inkompatiblen Schnittstellen zwischen zwei Komponenten.

4.3.3 Zeit

Um die Geschwindigkeit einer Architektur zu evaluieren, ist es notwendig, alle Berechnungsschritte in eine Abfolge zu bringen, die in einer minimalen Laufzeit des Algorithmus resultiert. Die Modellierung des Zeitverhaltens wird mit Hilfe einer Zeitdomäne realisiert, welche eine diskrete und endliche Menge an Zeitfenstern definiert. Jeder Berechnungsschritt t_i wird genau einem Zeitfenster k zugeordnet, um dessen Startzeitpunkt festzulegen. Dies wird durch das Atom $T_{t_i k}$ ausgedrückt, wobei die Konstante m die Gesamtzahl der zur Verfügung stehenden Zeitfenster ist:

$$1\,[\,T_{t_i 1},\ \ldots,\ T_{t_i m}\,]\,1. \tag{12}$$

Um die Abhängigkeiten im Problemgraph zu berücksichtigen, darf kein Berechnungsschritt vor seinem Vorgänger starten:

$$\leftarrow T_{t_x k_x},\ T_{t_y k_y},\ k_y < k_x. \tag{13}$$

Der Problemgraph wird außerdem um zwei Knoten erweitert, um den Start und das Ende der gesamten Berechnung zu markieren. Während der Startknoten dem Zeitfenster 0 zugeordnet wird, zeigt das Zeitfenster des Endknotens die Gesamtlaufzeit der Berechnung an und wird als Optimierungsziel für den ASP-Solver verwendet.

Einen praktischen Wert für das Zeitintervall eines Zeitfensters festzulegen ist schwierig. Eine kürzere Zeitspanne resultiert in einer sehr präzisen Ablaufplanung, jedoch explodiert dabei die Anzahl der Möglichkeiten, die der Solver verarbeiten muss. Als Kompromisslösung wird ein normalisierter Wert gewählt, bezogen auf die schnellste Komponente: Ein Zeitfenster ist die Zeitspanne, die die schnellste Komponente benötigt, um eine bestimmte Menge an Daten zu verarbeiten. Belegt eine Komponente zwei Zeitfenster, dann arbeitet sie nur halb so schnell, verglichen mit der Schnellsten.

4.3.4 Synchronisation

Wie in Abschnitt 4.1 dargestellt, ist die tatsächliche Geschwindigkeit einer Berechnungskomponente von ihren umgebenden Komponenten abhängig und nicht immer die maximal mögliche Geschwindigkeit. Das Atom $S_{t_i k}$ definiert den Geschwindigkeitsfaktor k für den

Berechnungsschritt t_i, wobei aus gleichen Gründen, wie bei der Modellierung der Zeit ein relativer Wert verwendet wurde:

$$1\,[\,S_{t_i1},\ \ldots,\ S_{t_ip}\,]\,1. \tag{14}$$

Dabei bedeutet der Wert $k = 1$, dass die schnellste Komponente genau ein Zeitfenster benötigt, um eine bestimmte Menge an Daten zu verarbeiten. Die Konstante p ist der höchste Wert und demnach der Geschwindigkeitsfaktor der langsamsten Komponente.

Der Geschwindigkeitsfaktor einer Komponente ist von deren Vorgänger abhängig. Sind die Berechnungsschritte t_x und t_y zwei aufeinander folgenden Komponenten zugeordnet, so müssen die Geschwindigkeitsfaktoren gleich sein:

$$\leftarrow S_{t_xk_x},\ S_{t_yk_y},\ k_x \neq k_y. \tag{15}$$

Für die Ablaufplanung sind weitere Hilfsatome notwendig. Mit Hilfe des Startzeitpunkts und der Geschwindigkeit eines Berechnungsschritts kann dessen Endzeitpunkt E ermittelt werden. Der Berechnungsschritt t_i startet im Zeitfenster k_s und benötigt d Zeitfenster für die Berechnung:

$$E_{t_ik_e} \leftarrow S_{t_id},\ T_{t_ik_s},\ k_e = k_s + d. \tag{16}$$

Für die lokale Ablaufplanung auf den einzelnen Komponenten ist es wichtig, dass sich keine zwei Berechnungen überlappen. Sind zwei Berechnungsschritte t_x und t_y der selben Komponente zugeordnet und beginnt t_y nach t_x, so muss t_x vor dem Start von t_y beendet worden sein:

$$\leftarrow T_{t_xk_s},\ E_{t_xk_e},\ T_{t_yk_y},\ k_s \leq k_y,\ k_y < k_e. \tag{17}$$

4.3.5 Busauslastung

Mit den bisher vorgestellten Regeln können bereits gültige Architekturen erzeugt werden. Im Folgenden werden weitere Regeln eingeführt, die die Qualität der Lösung beeinflussen und außerdem die allgemeinen Systemanforderungen berücksichtigen. Diese Faktoren betreffen insbesondere Speicher- bzw. Busauslastung, Chipressourcen und Gesamtlaufzeit.

Ein wichtiger Punkt ist die Auslastung des Systembusses bzw. Bus des Hauptspeichers, da im Hauptspeicher ein Großteil der Daten gespeichert ist und so der Bus leicht zu

einem Flaschenhals werden kann. Für jeden Zeitpunkt bzw. jedes Zeitfenster muss gewährleistet werden, dass der Bus nicht überlastet und gegebenenfalls die entsprechenden Komponenten gedrosselt werden.

Die Geschwindigkeit des Systembusses wird durch die Konstante s_b festgelegt. Für jedes Zeitfenster wird die aktuelle Belastung durch alle aktiven Transfers aufsummiert und mit der Kapazität des Systembusses verglichen. Die Busbelastung einer Komponente ist umgekehrt proportional zu ihrem Geschwindigkeitsfaktor. Arbeitet eine Komponente z. B. vier mal langsamer als der Bus, so ist die Busauslastung ein Viertel. Dies wird durch folgende Ungleichung ausgedrückt:

$$\frac{1}{s_{tc_1}} + \ldots + \frac{1}{s_{tc_n}} \geq \frac{1}{s_b} \tag{18}$$

In jedem Zeitfenster t belasten die Komponenten $c_1, \ldots, c_n$ den Bus gemäß ihrem individuellen Geschwindigkeitsfaktor s_{tc_k}.

Um gebrochene Zahlen im Antwortmengenprogramm zu vermeiden, wird die Buskapazität diskret als eine Menge von Belegungseinheiten modelliert. Die Konstante p (siehe auch Gleichung 14) wird aus der Geschwindigkeit der langsamsten Komponente abgeleitet und damit ist die kleinste Busauslastung $1/p$, für $s_b = 1$. Weiter ergibt sich damit die Anzahl der benötigen Belegungseinheiten gleich p bzw. p/s_b für $s_b \neq 1$.

Dies ergibt sich aus folgendem Sachverhalt: Es ist möglich, alle Geschwindigkeitsfaktoren, inklusive dem Maximalwert p, durch s_b zu normalisieren, um die neuen Werte $p' = p/s_b$ und $s'_b = 1$ zu erhalten. Für eine Komponente c_k mit dem Geschwindigkeitsfaktor s_{tc_k} ist der normalisierte Koeffizient s_{tc_k}/s_b. Folglich benötigt sie s_b/s_{tc_k} der Buskapazität und damit

$$\frac{s_b}{s_{tc_k}} \cdot p' = \frac{s_b}{s_{tc_k}} \cdot \frac{p}{s_b} = \frac{p}{s_{tc_k}} \tag{19}$$

Belegungseinheiten. Durch die Normalisierung wird die Ungleichung 18 zu einer Integritätsbedingung mit ganzen Zahlen:

$$\leftarrow \lceil \frac{p}{s_{tc_1}} \rceil + \ldots + \lceil \frac{p}{s_{tc_n}} \rceil \geq \lfloor \frac{p}{s_b} \rfloor. \tag{20}$$

4.3.6 Ressourcenverbrauch

Ein weiterer Faktor bezüglich der Rahmenbedingungen ist die vorhandene Chipfläche. Wie oben beschrieben, ist der Ressourcenverbrauch r jeder Komponente in den Metainformationen enthalten. Für jede instantiierte Komponente c_k wird der Wert r_k durch das Atom $R_{c_k r_k}$ repräsentiert. Mit der Gesamtchipfläche R_u lehnt die Integritätsbedingung

$$\leftarrow R_{c_1 r_1}, \ldots, R_{c_n r_n}, R_u, r = \sum_n r_n, u \leq r. \tag{21}$$

Architekturen ab, die zu viele Ressourcen verbrauchen. Entsprechend gibt es im Modell Regeln für andere Ressourcen wie z. B. BRAM.

4.3.7 Laufzeitminimierung

Um die Suche für den Solver zu vereinfachen, erfolgt die Belegung der Zeitfenster T zuerst in einer geringeren zeitlichen Auflösung (z. B. Faktor 10). Ist für das grobe Zeitraster keine Lösung zu finden, muss die feine Auflösung nicht mehr durchsucht werden. Die entsprechenden Regeln werden mit Hilfe der *div* und *mod* Operatoren formuliert.

Um schließlich die optimale Architektur zu erhalten, muss die Gesamtlaufzeit minimiert werden. Als Indikator für die Laufzeit wurde in Regel 16 der Zeitpunkt des Endknotens eingeführt. Im ASP-Modell wird ein Aggregat benutzt, um diesen Zeitpunkt t_e zu finden:

$$minimize\ [\ T_{t_e 1} = 1, \ldots, T_{t_e m} = m\ ]. \tag{22}$$

Jedes Atom $T_{t_e k}$ wird entsprechend seiner Zeitfensternummer k gewichtet. Weil nur ein Atom wahr sein kann, ergibt sich aus der Summe (eckige Klammern) die Fensternummer des Endzeitpunkts. Multipliziert mit dem Zeitintervall eines Fensters lässt sich die Gesamtlaufzeit für die Berechnung des Problemgraphs herleiten.

4.3.8 Einschränkungen

Wie oben beschrieben, ist die Zeit diskret modelliert. Das Zeitverhalten der implementierten Architektur kann deshalb von dem des ASP-Solvers abweichen. Insbesondere treten in der Praxis sehr kleine Verzögerungen auf, welche im Modell nicht berücksichtigt werden,

weil sie klein gegenüber einem Zeitfensterintervall sind. Diese sind z. B. die Zeiten zum Füllen oder Leeren von FIFOs in PU-Ketten oder IMs von wenigen Taktzyklen.

Komponenten vom Typ *fixed result* (siehe Kapitel 3.1 *Eigenschaften von Bildverarbeitungsalgorithmen*), wie z. B. ein Histogramm, haben meist nur eine geringe Menge an Ergebnisdaten, verglichen mit einem ganzen Bild. Deshalb fließt in der Modellierung diese Menge und ebenso die Verzögerungszeiten nicht in die Berechnung des Datenflusses mit ein. In dem Logikprogramm werden entsprechende Komponenten (bzw. Kanten im Problemgraph mit dem Schlüsselwort wait_for) markiert, was für den ASP-Solver heißt, auf diese Komponente zu warten, bevor der nachfolgende Berechnungsschritt starten kann.

Die Einschränkung der Ressourcen bzw. die Berechnung des Ressourcenverbrauchs ist nicht 100%ig genau. Der tatsächliche Verbrauch einer Komponente ist z. B. abhängig davon, welche Ausdehnung eine Komponente auf dem Chip hat, in welcher Nähe bestimmte Ressourcen, wie z. B. Speicher oder DSP-Einheiten verfügbar sind und wie gut bei der Platzierung und Verdrahtung (place & route) optimiert werden kann. Dies liegt jedoch im Ermessen der verwendeten Synthesewerkzeuge der jeweiligen FPGA-Hersteller.

4.4 Experimentelle Ergebnisse

Als ASP-Solver wird die Implementierung `gringo` aus der Potassco-Suite verwendet, welche bereits mehrere Preise gewonnen hat [GKK+11, POT].

Für die Auswertung der Lösung wurden in der Modellierung bestimmte Atome und Hilfsatome markiert, welche der ASP-Solver am Ende der Suche ausgibt (siehe Ende von Listing 3). Aus diesen Daten sind die Architektur und die Ablaufplanung direkt ableitbar, welche dann für die Hardware- und Softwaresynthese weiterverarbeitet werden, wie im Kapitel 5 *Systemsynthese* beschrieben.

Um die Skalierbarkeit der Modellierung zu untersuchen, wurden zufällig Problemgraphen erzeugt und die Laufzeit des ASP-Solvers auf einem Desktop mit einem Intel Core2 Duo Prozessor 3,16 GHz und 3,2 GB RAM gemessen. Für Graphen mit einer Größe von 5 bis 10 Knoten, wurden jeweils 1000 Probleme betrachtet und dazu deren Berechnungsfunktionen und Verbindungen, sowie die Metainformationen der Implementierungen zufällig generiert. Abbildung 13 und Tabelle 2 zeigen einen exponentiellen Anstieg der Zeiten des Solvers bezüglich der Anzahl der Knoten im Problemgraph. In der Tabelle sind außerdem die Standardabweichung und die Extrema gelistet, die mit steigende Knotenzahl ebenfalls

Laufzeit \ Knoten	5	6	7	8	9	10
μ	3,279	8,264	20,52	56,1	166	540
σ	0,688	2,274	5,50	16,6	52	232
min	1,88	3,14	5,82	10,9	18	104
max	5,99	19,79	42,82	111,3	429	1541

Tabelle 2: Statistische Werte zu den Laufzeiten des ASP-Solvers in Sekunden in Abhängigkeit von der Anzahl der Knoten in zufällig generierten Problemgraphen über 1000 Messungen.

steigen. Dies ist auf die unterschiedliche Komplexität der entstehenden Systeme zurückzuführen. Die durchschnittliche Laufzeit μ bei einer größeren Anwendung mit 10 Knoten ist mit 9 Minuten noch relativ gering gegenüber der Laufzeit der Synthesesoftware ISE mit ca. 45 Minuten.

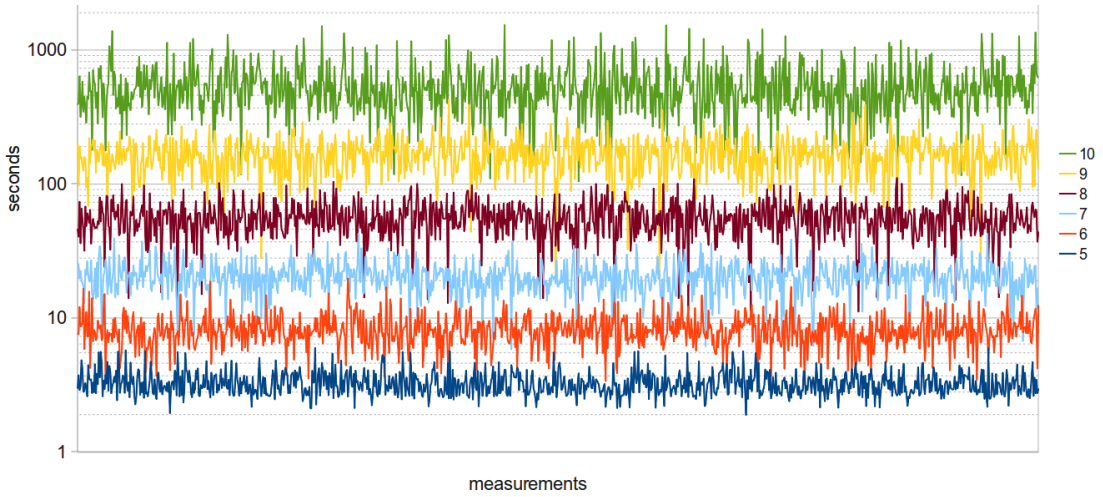


Abbildung 13: Laufzeiten des ASP-Solvers in Abhängigkeit von der Anzahl der Knoten (rechts) in zufällig generierten Problemgraphen über 1000 Messungen.

5 Systemsynthese

Nachdem die Arbeitsschritte aus dem Problemgraphen konkreten Implementierungen zugeordnet wurden und die Ablaufplanung feststeht, kann das System synthetisiert werden. Ein System besteht aus Hardware- und Softwareteilen. Deren Synthese und die notwendigen Konzepte und Abläufe werden in den folgenden Unterkapiteln beschrieben.

5.1 Softwaresynthese

Die Softwaresynthese untergliedert sich in den Betriebssystem relevanten Teil und die Interaktion zwischen Software und Hardware.

5.1.1 Betriebssystem

Sehr einfache Anwendungen können direkt auf einem eingebetteten System gestartet werden. Die nötige Entwicklungsumgebung wird meist vom Hersteller mitgeliefert. Mit steigender Komplexität der Anwendung wird der Einsatz eines Betriebssystems sinnvoll, da es bestimmte Grundfunktionen, wie z. B. zeitlich gesteuerte Abläufe, Hardwareabstraktion, Zugriff auf Speicher- und Kommunikationsmedien oder eine Prozess- und Speicherverwaltung, bereitstellt, und so die Entwicklung von Anwendungen mit höheren Anforderungen vereinfacht und beschleunigt.

Für die Lauffähigkeit eines Systems mit Betriebssystem sind folgende Komponenten notwendig: Boot-Loader, Kernel, Bootkonfiguration, Systemkonfiguration, Systemdienste, Bibliotheken und schließlich die Anwendungen. All diese Komponenten werden in sogenannten Distributionen zusammengefasst. Schließlich wird die gewünschte Auswahl an Komponenten in einem Dateisystem gespeichert und darauf folgend in Form eines *Image*s z. B. in einem nichtflüchtigen Speicher des eingebetteten Systems untergebracht.

Die Generierung eines Images ist sehr komplex. Software für eingebettete Systeme wird für gewöhnlich, z. B. aus Mangel an Ressourcen, nicht auf dem Zielsystem erzeugt, sondern auf einem Entwicklungssystem mit Hilfe eines *Cross-Kompilers*. Das bedeutet, der Compiler läuft auf einem (Host)Prozessor und kompiliert für einen anderen (Target)Prozessor. Nachdem die Abhängigkeiten zwischen den Softwarepaketen aufgelöst und eine Kompilierreihenfolge gefunden wurde, können die einzelnen Softwarepakete kompiliert werden. Dazu wird eine *Toolchain* benötigt. Sie beinhaltet, am Beispiel der GNU Compiler Collection

(gcc): (Cross-)Compiler, binutils (Linker, Assembler, ...) und die C-Standardbibliothek `glibc`. Alternativen zur Standardbibliothek sind die, speziell für eingebettete Systeme entwickelten, Bibliothek `uClibc` und `eglibc`. Der Aufwand, eine Toolchain zu kompilieren, ist oft hoch und bei Cross-Toolchains noch höher [Keg]. In manchen Fällen kann auf eine fertige Toolchain im Internet zurückgegriffen werden. Ansonsten, oder wenn spezielle Patches benötigt werden, muss die Toolchain selbst kompiliert werden. Teilweise ist dafür unterstützende Software verfügbar (z. B. crosstools [Keg]).

Eine Software besteht gewöhnlich aus mehreren Quelltextdateien, die kompiliert und dann zu der eigentlichen Anwendung zusammengeführt (gelinkt) werden. Diese Arbeitsschritte werden praktischerweise automatisch durchgeführt, z. B. mit Hilfe von `make`. Für die plattformunabhängige Kompilierung (z. B. Linux, UNIX, ...) können die Entwickler von Softwarepaketen auf diverse Werkzeuge, wie das GNU Build System (Autotools), CMake, jam, Apache Ant, SCons, u. a., zurückgreifen. Viele Software ist nicht für die Cross-Kompilierung vorbereitet und muss angepasst werden. Problematisch ist insbesondere, dass häufig kleine Hilfsprogramme benutzt werden, welche bei der Cross-Kompilierung nicht lauffähig sind, weil sie nicht für das Hostsystem kompiliert. Des Weiteren muss die Verwendung einer bestimmten Toolchain für die Zielarchitektur oder auch die Konfiguration korrekter Suchpfade zu Bibliotheken gewährleistet werden.

Ebenfalls der Linux-Kernel kann konfiguriert werden. Er ist modular aufgebaut und kann an die Bedürfnisse des Zielsystems angepasst werden. So kann die Unterstützung von bestimmter Hardware oder anderen Funktionen, wie z. B. Bluetooth, Sound oder IPv6, in den Kernel integriert, als Modul kompiliert oder ganz ausgeschaltet werden. Somit kann der Kernel auf das notwendigste Maß verkleinert werden. Eingebettete Systeme sind meist statisch aufgebaut und dementsprechend kann ein Kernel ohne Modulunterstützung eingesetzt werden, was seine Größe weiter reduziert. Andererseits kann durch das dynamische Laden und Entladen von Modulen, je nach Bedarf, Speicher gespart werden, was ebenfalls bei eingebetteten System von Bedeutung ist.

Sind alle Softwarepakete kompiliert, müssen diese zusammen mit dem Kernel und der Konfiguration des Betriebssystems in einem Dateisystem (*root filesystem*) installiert werden. Auch hierbei muss sichergestellt werden, dass die richtigen Verzeichnispfade verwendet werden. Zusammenfassend lässt sich feststellen, dass die Erzeugung einer Linux-Distribution sehr komplex ist und möglichst automatisch durchgeführt werden sollte.

Für die Generierung von Linux-Distributionen bzw. Images für eingebettete Systeme gibt es Softwarelösungen, wie z. B. ELDK (Embedded Linux Development Kit), buildroot, crosstools-NG, PTXdist oder Bitbake. In der vorliegenden Arbeit wurde zunächst PTXdist eingesetzt. Die Konfiguration der Distribution und die Einbindung neuer Pakete ist komfortabel. Die Toolchain ist jedoch, wie bei den anderen genannten Werkzeugen, nicht integriert. Deshalb wurde zu OpenEmbedded gewechselt.

OpenEmbedded ist eine umfangreiche Sammlung von Metadaten, um die Kompilierung der Software und die Generierung von Distributionen zu steuern. Mit Hilfe des Werkzeugs *Bitbake* können, auf der Basis des Quelltextes der Software, einzelne Pakete (wie bei den großen Linux-Distributionen) bis hin zu fertigen Images vollautomatisch erzeugt werden. Bei der Entwicklung von OpenEmbedded standen Flexibilität und Funktionsumfang im Vordergrund. Es vereint im Vergleich zu den oben genannten Lösungen folgende Eigenschaften [OEM]:

- Unterstützung der Cross-Kompilierung und Einbindung von Patches
- Unterstützung von verschiedenen Prozessoren, Architekturen, Plattformen und Betriebssystemen
- Erzeugung von Toolchains, alternativ Einbindung von externen Toolchains
- Erzeugung von Paketen in verschiedenen Formaten (tar, rpm, deb, ipk, opk, ...) für einzelne Software
- Erzeugung von Images aus Paketen (unter Berücksichtigung von Paketabhängigkeiten) in verschiedenen Formaten und Dateisystemen
- Erzeugung von bootbaren Images in verschiedenen Formaten (jffs2, ext2.gz, squashfs, ...)
- Metadaten sind einfach erstellbar, erweiterbar, vererbbar, wiederverwendbar und hochgradig konfigurierbar
- Mit Metadaten werden einzelne Software, Zielplattformen, Gruppierungen oder auch ganze Distributionen konfiguriert
- Durchführung von Plausibilitätstests
- Unabhängig: Alle benötigten Werkzeuge werden automatisch heruntergeladen und kompiliert

- Reproduzierbarkeit von Images, Paketen, ...(besonders im produktiven Einsatz wichtig)
- Schnelle Integration von (kleinen) Änderungen
- Momentan mehr als 1000 Softwarepakete mitgeliefert

Wurden alle Einstellungen vorgenommen, erfolgt die Generierung eines Images vollautomatisch. Ein typischer Ablauf von bitbake ist: Quelltext herunterladen (fetch), Entpacken und Patches einspielen (patch), Kompilierung vorbereiten (configure), Software kompilieren (compile), Softwarepaket erstellen (build), Software installieren (install) und Image erstellen (build-image). Dieser Prozess erfolgt sowohl für Software, die Bitbake für seine eigene Funktionalität benötigt (native packages), als auch für Software für das Zielsystem (target packages).

Wie aufgezeigt wurde, ist die manuelle Generierung eines Images ein langwieriges und fehleranfälliges Unterfangen. Mit Hilfe von OpenEmbedded können sehr viele Aufgaben automatisiert werden und die Entwickler können sich auf die Konfiguration der Distribution konzentrieren. Die Einarbeitungszeit für die Erstellung eigener Anpassungen und Erweiterungen mit OpenEmbedded ist im Vergleich zu anderen Lösungen höher, zahlt sich jedoch durch die sehr viele höhere Flexibilität und Mächtigkeit wieder aus. Insbesondere der Einsatz im Bereich Programmierbarer Logik mit Softcore-Prozessoren und variablen Systemarchitekturen stellt hohe Anforderungen an ein build-System, dem sich OpenEmbedded jedoch stellt.

5.1.2 Zugriff auf Hardwarekomponenten

Damit eine Bildverarbeitungsanwendungen sich Hardwarebeschleuniger zu Nutze machen kann, muss eine Zugriffsmöglichkeit geschaffen werden. Da der Einsatz von Hardware vor dem Softwareentwickler der Anwendung versteckt werden soll, wird die Bildverarbeitungsbibliothek um diese Fähigkeit erweitert und der Quelltext entsprechen modifiziert. Der Zugriff auf Hardwarekomponenten ist jedoch von der Anwendung bzw. Bibliothek aus nicht direkt möglich. Im Folgenden werden die Hintergründe erklärt (mehr Details sind in [BC06, Ven08] nachzulesen) und im Anschluss daran wird eine effiziente Lösung vorgestellt.

Die wichtigsten Hauptaufgaben eines Betriebssystems sind:

- Die Verwaltung und Bereitstellung von Interaktionsmöglichkeiten zu den Betriebsmitteln und
- die Bereitstellung einer Ausführungsumgebung für Anwendungen.

In den momentan verbreiteten Betriebssystemen, wie UNIX-Varianten, Linux, MacOS oder MS Windows, werden die Betriebsmittel verborgen und abstrahiert. Dies vereinfacht die Entwicklung von Anwendungen und erhöht die Sicherheit und Portierbarkeit. Anwendungen können nicht direkt auf Hardwarekomponenten zugreifen, sondern müssen ihre Anfragen über das Betriebssystem bzw. den Betriebssystemkern (kernel) abwickeln. Um diese Vorgehensweise zu unterstützen, verfügen moderne Prozessoren über spezielle Modi, welche z. B. unterschiedliche Speicherzugriffsberechtigungen haben. Der Linux-Kernel kommt mit zwei Modi aus, einem Privilegierten für Betriebssystemoperationen und einem Nicht-privilegierten für Anwendungen, welche dementsprechend als *kernel mode* bzw. *user mode* bezeichnet werden.

Der Linux-Kernel interagiert mit I/O-Geräten per Gerätetreiber (device driver). Gerätetreiber sind als Kernelmodule implementiert und laufen ebenfalls im kernel mode. Sie abstrahieren den Zugriff auf Geräte, wie z. B. Festplatte, Tastatur oder USB-Schnittstelle. Der Kernel spricht die Gerätetreiber über eine einheitliche (Programmier)Schnittstelle an, was u. a. die Entwicklung und Wartung von Treibern und treibernahen Kernelteilen vereinfacht. Geräte werden der Anwendung als Dateien in einem *virtuellen Dateisystem* (`/dev`) zur Verfügung gestellt, welche über die Funktionen `open`, `close`, `read`, `write` und `ioctl` angesprochen werden können.

Linux unterstützt das Konzept des *virtuellen Speichers*. Dem Kernel und jeder Anwendungen steht ein eigener (und geschützter) virtueller Adressraum zur Verfügung, der durch die MMU (memory management unit) des Prozessors auf den physikalischen Speicher abgebildet wird. Der physikalische Speicher ist in Speicherseiten (bei Linux mit der typischen Größe von 4 kB oder 8 kB) organisiert und der Kernel gewährleistet, dass sich die momentan benötigten Speicherseiten auch tatsächlich im physikalischen Speicher befinden. Dazu können andere Speicherseiten verdrängt und ausgelagert werden. Infolgedessen wird der Speicher fragmentiert. Der Adressraum von im user mode laufender Software wird als *user space* und der des Kernels als *kernel space* bezeichnet.

Der Zugriff einer user-mode-Anwendung auf Hardwarekomponenten erfolgt durch mehrere Abstraktionsebenen (siehe Abbildung 14). An zentraler Stelle steht der Systemaufruf

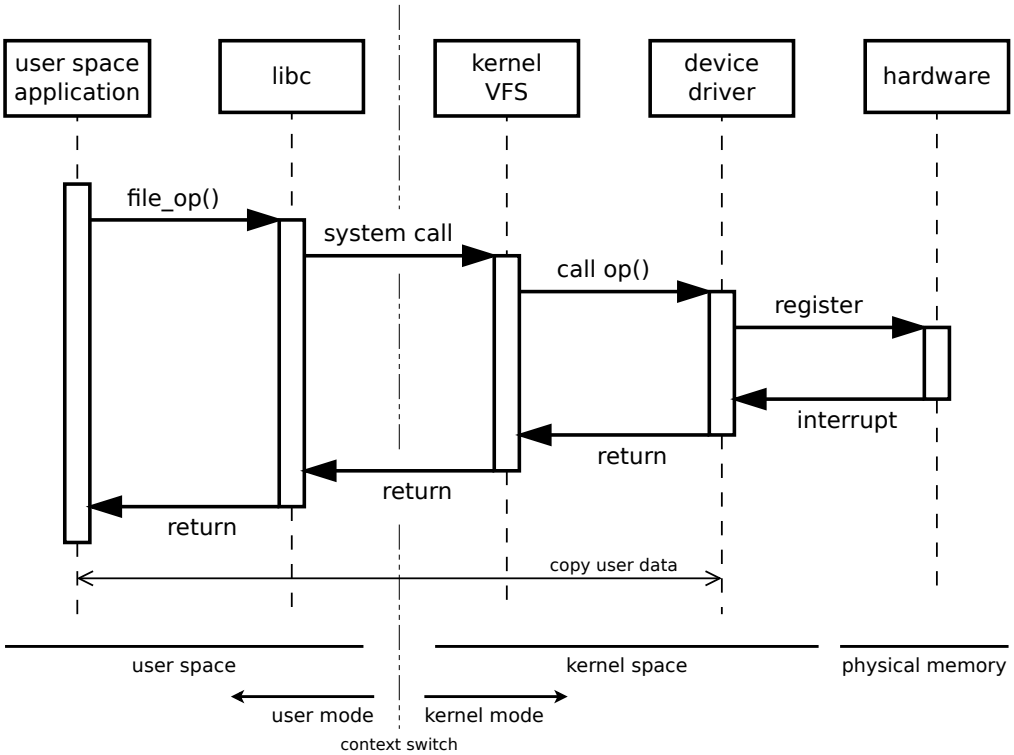


Abbildung 14: Der Zugriff einer Anwendung auf Hardwarekomponenten erfolgt über verschiedene Abstraktionsebenen.

(system call), welcher mit einem Wechsel vom user mode in den kernel mode (context switch) verbunden ist und von der Standard-C-Bibliothek initiiert wird. In der VFS-Schicht (virtual filesystem) wird anschließend der zugehörige Gerätetreiber bestimmt und die entsprechende Funktion aufgerufen. Zur weiteren Verarbeitung der Anfrag müssen die Parameter der Anwendung vom user space in den kernel space kopiert werden. Schließlich wird vom Gerätetreiber die Hardwareanfrage durchgeführt, z. B. durch Kommunikation über Hardware-Register. Meist stehen die Ergebnisdaten nicht sofort zur Verfügung und der aktuelle Prozess wird schlafen gelegt, bis die Hardware einen Interrupt liefert. Die Ergebnisdaten werden anschließend in den user space zurück kopiert und der Aufruf beendet.

Auch die Verteilung der Daten im Speicher hat Einfluss auf die Verarbeitungsgeschwindigkeit. Nach der Entwicklung der ersten Prozessoren erhielten diese bald Unterstützung

von sogenannten DMA-Kontrollern (direct memory access). Diese können Daten unabhängig vom Hauptprozessor von und zu I/O-Bereichen und Speichern transferieren und werden im PC z. B. für den Datenaustausch zwischen Hauptspeicher und Netzwerk- oder Soundkarten eingesetzt. So wird der Hauptprozessor entlastet. Ursprünglich wurden DMA-Transfers durch Quell- und Zieladresse und der Anzahl von zu übertragenden Bytes festgelegt. Spätere DMA-Kontroller mit Scatter/Gather-Modus sind in der Lage, mehrere Datenblöcke zu verarbeiten, welche nicht fortlaufend im Speicher liegen. Dazu muss im Speicher eine verkettete Liste von Deskriptoren angelegt werden, welche Informationen über die einzelnen Datenblöcke enthalten. Der DMA-Kontroller liest die Liste sukzessiv ein und führt die entsprechenden Blocktransfers durch. Die höhere Flexibilität wird durch einen komplexeren Kontroller und zusätzlichen Speicherbedarf für die Deskriptoren erkauft.

Für große Datenmengen, wie im vorliegenden Fall von Bild- oder Videodaten, wird die Anzahl der Listenelemente = $\frac{\text{Bilddatenmenge}}{\text{Speicherseitengröße}}$ entsprechende groß und die damit verbundenen zusätzlichen Lesezugriffe des DMA-Kontrollers ebenso. Darüber hinaus ist der zur Verfügung stehende Speicher im kernel mode sehr eingeschränkt.

Der beschriebene gewöhnliche Ablauf für die Datenverarbeitung durch Hardwaremodule hat einige Nachteile. Die Probleme sind im Folgenden noch einmal zusammengestellt:

- Ein Systemaufruf bringt das Durchlaufen mehrerer Abstraktionsebenen des Betriebssystems (inklusive content switch) und das Umkopieren von Daten zwischen user space und kernel space mit sich, was sich entsprechend negativ auf den Datendurchsatz auswirkt.
- Daten im user space liegen zwar logisch fortlaufend, physikalisch jedoch fragmentiert vor. Deshalb müsste ein bildverarbeitendes Hardwaremodul (bzw. IM) Kenntnis über die Verteilung der Speicherseiten erhalten und außerdem sich bei jedem Zugriff mit dem Betriebssystem synchronisieren.
- Die Kernel-Entwickler sind stets offen für neue Konzepte und Methoden. In der Vergangenheit kam es deshalb öfter vor, dass sich die Kernel-APIs verändert haben. Selbstentwickelte Gerätetreiber müssen bei einer Aktualisierung des Kernels entsprechend angepasst werden.

- Beim gleichzeitigem Zugriff mehrerer Einheiten, wie z. B. von Hauptprozessor und Hardwarekomponenten, auf den selben Speicherbereich kann das Cache-Kohärenz-Problem auftreten.
- Alle erweiterten Funktionen eines Gerätetreibers müssen durch einen Aufruf von `ioctl()` realisiert werden.
- Die Funktion `ioctl()` hat nur einen Parameter, weshalb alle zu übergebenden Variablen in diesen einen gekapselt werden müssen.

In der vorliegenden Arbeit wurde eine alternative Vorgehensweise für die Organisation der Datenverarbeitung durch Hardwarekomponenten aus einer user-space-Anwendung heraus, mit dem Fokus auf Datendurchsatz, entwickelt.

Um das Problem der Fragmentierung und der Koordinierung mit dem Betriebssystem zu vermeiden, wurde der physikalische Speicher in zwei Bereiche eingeteilt: Betriebssystemspeicher und Bildspeicher. Dazu wird dem Kernel beim booten eine geringere Hauptspeichergröße vorgegeben als tatsächlich physikalischer Speicher vorhanden ist. Als Konsequenz ergibt sich ein Speicherbereich im oberen Adressraum, der nicht von Linux verwaltet und benutzt wird. Dieser Speicher steht nun explizit für die Bildverarbeitung frei zur Verfügung. Folglich können Bilddaten fortlaufend und als ein Block gespeichert werden, was sich positiv auf die Komplexität der Bus-IMs und die Verarbeitungsgeschwindigkeit auswirkt.

Es ist möglich, einen physikalischen Speicherbereich in den user space einzublenden und damit den Bildspeicher einer Anwendung direkt zur Verfügung zu stellen. Dazu muss für eine physikalische Adresse zuerst eine virtuelle Adresse im kernel space erzeugt werden, indem die MMU mit Hilfe entsprechender Kernelfunktionen konfiguriert wird. Anschließend kann über den Kernelbefehl `mmap()` die virtuelle Adresse in den user space eingeblendet werden. Auf die gleiche Art und Weise kann eine Anwendung direkten Zugriff auf die Register von Hardwarekomponenten erhalten, da die Register mittels *memory mapped IO* in den normalen Adressraum eingeblendet werden. Die Anwendung hat nun zusätzlich die Aufgabe, den Bildspeicher zu verwalten. Auf der anderen Seite können durch die volle Kontrolle und Flexibilität auch einfach *double buffering* oder andere nützliche Methoden realisiert werden.

Als letztes Interaktionselement zwischen Anwendungen und Hardwarekomponenten ist der Rückkanal zu betrachten. Während die Anwendung nun direkt auf skalare Ergebniswerte und auf Ergebnisbilder im Bildspeicher zugreifen kann, sind Interrupts nicht vom user space aus erreichbar. Dies kann jedoch über eine entsprechende Gerätetreiberfunktion erfolgen, welche auf einen bestimmten Interrupt wartet und anschließend zurückkehrt. Um das Cache-Kohärenz-Problem zu vermeiden, muss, bevor der Verarbeitungsprozess der Hardwaremodule gestartet wird, der Schreib-Cache des Prozessors zurückgeschrieben und nach Beenden der Lese-Cache invalidiert werden.

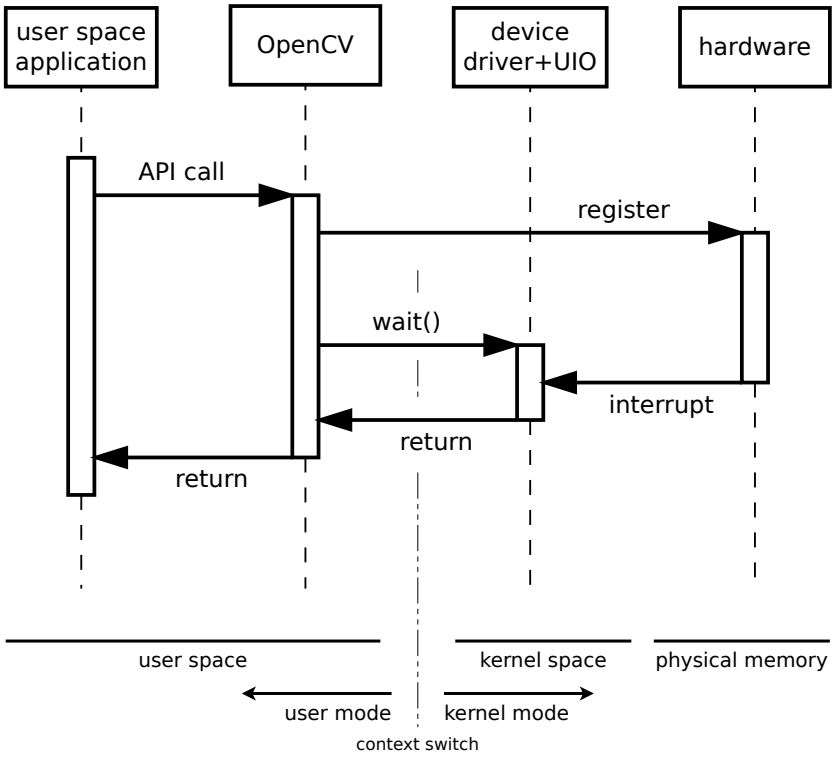


Abbildung 15: Hardwarezugriff mit Hilfe der UIO (user IO) Erweiterung des Linux-Kernels.

Die beschriebene Vorgehensweise bezüglich Speicherzugriff und Interruptbehandlung wurde im UIO-Framework (user space IO) implementiert und erhielt Einzug in den Vanilla Kernel seit der Version 2.6.23[8]. Der Zugriff auf Hardwarekomponenten kann somit komplett in einer Bibliothek, hier in die Bildverarbeitungsbibliothek OpenCV integriert, und einem minimalen Gerätetreiber gekapselt werden (siehe auch Abbildung 15). Die Funktionen der API können frei typisiert werden und es müssen nicht alle Aufrufe durch

[8]Veröffentlichung: 9. Oktober 2007

die `ioctl()`-Funktion gezwängt werden. Dies erhöht die Programmierbarkeit und reduziert gleichzeitig die Fehleranfälligkeit. Außer für die Interruptbehandlung ist kein *content switch* notwendig. Der minimale Gerätetreiber kann aus einem Beispiel des UIO-Frameworks abgeleitet werden und übernimmt die Verwaltung der Speicherbereiche und Interrupts, wie oben beschrieben. Alle anderen Funktionalitäten finden in der Bibliothek Platz. Als Teil des Kernels wird der Programmcode des UIO-Frameworks auch vom Kernel-Entwicklerteam gewartet und Änderungen an Kernel-Interna wirken sich nicht auf die Bibliothek oder den Gerätetreiber aus. Die Wartung des Gerätetreibers wird dadurch vereinfacht.

SDI-Busmodule sollten eine einfache Bereichsprüfung implementieren, ob die angegebenen Adressen tatsächlich im Bildspeicher liegen. Gegebenenfalls erhält die Anwendung Rückmeldung über die Zugriffsverletzung über ein entsprechendes Flag im Statusregister. Der Speicherbereich des Betriebssystems wird dadurch bei fehlerhafter Programmierung geschützt.

5.1.3 Anpassung der Bildverarbeitungsbibliothek

Nachdem in der Entwurfsraumexploration für bestimmte Funktionen eine Hardwareimplementierung ausgewählt wurde, muss die Bildverarbeitungsbibliothek entsprechend modifiziert werden. Dazu wird die ursprüngliche Implementierung ersetzt und stattdessen der oben beschriebene Treiber verwendet, um die betroffenen IMs zu konfigurieren. Der Funktionsaufruf kehrt mit dem Interrupt des IMs zurück. Erfolgt die Verarbeitung in einer Kette von mehr als zwei PUs, kann die Softwareimplementierung der mittleren Funktionen leer bleiben. Die erste Funktion startet die Hardware und die letzte wartet auf den Interrupt.

5.2 Hardwaresynthese

Aus der Entwurfsraumexploration geht neben softwarerelevanten Parametern insbesondere die Systemarchitektur hervor (siehe Entwurfsfluss in Abbildung 12*). Für die Umsetzung der generierten Systemarchitektur werden FPGAs eingesetzt. Die Architekturbeschreibung muss abhängig vom eingesetzten Synthesewerkzeug in ein passendes Eingabeformat übergeführt werden. Das Ergebnis der Synthese ist ein sogenannter *Bitstream*, mit dem die Zielplattform konfiguriert wird.

Die Beschreibung der Systemarchitektur besteht:

- Aus der Menge der eingesetzten Hardwaremodule: Prozessoren, PUs, IMs, Kommunikationskanäle, Speicher und Peripheriebausteine,
- den Verbindungen zwischen diesen Modulen und
- der Parametrisierung der Module.

Für die Synthese des Bitstreams sind außerdem noch plattformabhängige Informationen notwendig, wie z. B. die Zuordnung der Anschlüsse des toplevel-Designs zu den Chipanschlüssen oder Timingparameter. Teilweise können diese Einstellungen nur von den System-on-Chip-Werkzeugen der FPGA-Hersteller bestimmt werden. Darüber hinaus werden von den Herstellerwerkzeugen auch automatische systemspezifische Anpassungen durchgeführt, wie z. B. das Einfügen von Signalpuffern zwischen Modulen. In den Forschungsarbeiten des AMoC-Projekts wurde dies ebenfalls untersucht. [2]

Die Werkzeuge, welche für den in der vorliegenden Arbeit konzipierten Entwurfsfluss implementiert wurden, sollen die Realisierung von Systemen möglichst umfassend durch automatische Abläufe unterstützen. Um die Projektdateien für die SoC-Werkzeuge vollständig zu generieren, müssten jedoch, wie eben beschrieben, Programmteile nachimplementiert werden, und das für jede Version neu. Dies wäre mit erheblichem Reengineeringaufwand verbunden und ist nicht sinnvoll. Statt dessen wird hier ein alternativer Weg gewählt und ein Grundgerüst als Basissystem verwendet. Zum Beispiel bestehend aus Hauptprozessor, Systembus, Speicherkontroller (für externen Speicher) und serieller Schnittstelle, enthält ein Basissystem alle Komponenten, die für alle möglichen (oder eine Untermenge davon) Systemarchitekturen gleich sind, inklusive der externen Anschlüsse.

*auf Seite 41

Das Basissystem ist außerdem für eine reine Softwarelösung (ohne Hardwarebeschleuniger) einsetzbar und Anwendungssoftware kann damit getestet werden.

Die vom SoC-Werkzeug erzeugten Projektdateien eines Basissystems können nun modifiziert und um die Komponenten aus der Entwurfsraumexploration ergänzt werden. Somit sind alle für die Synthese notwendigen Daten gegeben, um den Bitstream zu erzeugen.

6 Implementierung und Berechnungsplattformen

In diesem Kapitel wird auf verschiedene Implementierungsdetails eingegangen. Neben der Umsetzung von IMs und PUs wird die Benutzung der ASP-Werkzeuge und die ASP-Modellierung an Hand eines Beispiels veranschaulicht. Anschließend folgen neue Berechnungsplattformen, auf denen der vorgestellte Systemsyntheseprozess durchgeführt wurde.

Die PowerPC-Architektur wurde 1991 von IBM, Motorola und Apple spezifiziert (heute: Freescale) und ist sowohl im High Performance Computing (HPC) als auch im Bereich Eingebetteter Systeme weit verbreitet. Für die Implementierung des in der vorliegenden Arbeit entwickelten Systems standen verschiedene Entwicklungsboards mit Xilinx FPGAs zur Verfügung. Da der Linux-Kernel für die PowerPC-Architektur portiert wurde [PPC] und einige Virtex FPGAs über einen eingebetteten PowerPC-Hardcore-Prozessor verfügen, wurde der PowerPC 405 als leistungsstarker Hauptprozessor für die in Abschnitt 3.2 *Architektur für Bildverarbeitungssysteme* in Abbildung 11* gezeigte Beispielarchitektur ausgewählt. Insbesondere wurden folgende Plattformen verwendet:

XUP Virtex-II Pro Development System:

- Xilinx Virtex-II (XC2VP30: 13969 slices, 2448 Kb Block-RAM, 136 Multiplier, 2 PowerPC 405)
- 512 MB DDR-RAM
- Erweiterungsboard zur Digitalisierung von analogen Videosignalen

PICSY (Potsdam Intelligent Camera System):

- Xilinx Virtex-4 (XC4VFX20: 8544 slices, 1224 Kb Block-RAM, 32 XtremeDSP, 1 PowerPC 405)
- 128 MB DDR-RAM
- Digitaler Bildsensor: 3MP, Farbe

PAMO (Plattform für Autonome Modellfahrzeuge):

- Xilinx Virtex-4 (XC4VFX12: 5472 slices, 648 Kb Block-RAM, 32 XtremeDSP, 1 PowerPC 405)
- 32 MB DDR-RAM
- Digitaler Bildsensor: 3MP, Farbe

*auf Seite 40

6.1 Hardwarekomponenten

Im Folgenden wird auf die Implementierung des Streaming Data Interfaces (SDI) und wichtiger PUs (processing unit) und IMs (interconnection module) eingegangen.

6.1.1 Streaming Data Interface (SDI)

Das Streaming Data Interface (SDI) ist die gemeinsame Schnittstelle für die im Kapitel 3.2 *Architektur für Bildverarbeitungssysteme* beschriebenen Hardwarekomponenten, welche für Berechnungen als auch für die Kommunikation zuständig sind. Eine einheitliche Schnittstelle erlaubt die Rekombination und Wiederverwendung der Hardwarekomponenten für verschiedene Anwendungsfälle. In Analogie zu den UNIX-Shell Tools und Pipelines, können so auch sehr kleine spezifische Berechnungen in einem Modul implementiert und anschließend durch die Verkettung der Module komplexere Berechnungen einfach aufgebaut werden.

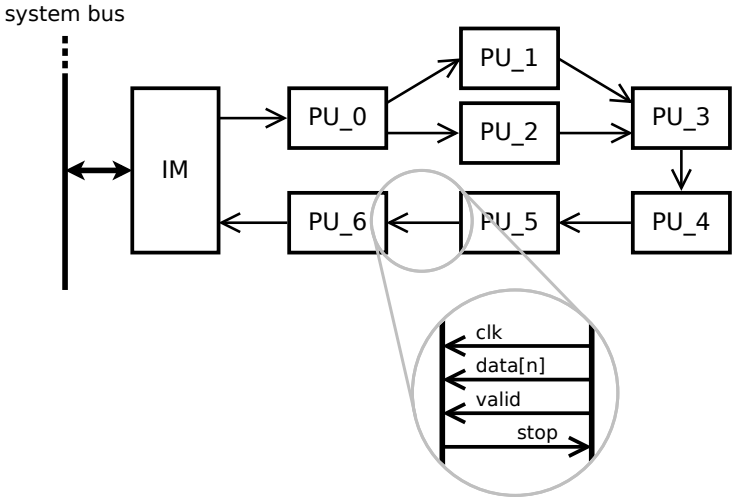


Abbildung 16: Streaming Data Interface (SDI): Gemeinsame Schnittstelle zwischen verschiedenen Modulen zur Berechnung (1–2, 4–6), Datenflusskontrolle (0, 3) und Systemanbindung (IM).

Die Definition des SDI orientiert sich an bereits bekannten Lösungen, wie z. B. die HUNT Library [HUN] oder PixelStreams [Cel]. Zur Vereinfachung werden jedoch so wenig Signale wie nötig verwendet und auf Indikatorsignale z. B. für Zeilen- oder Fensterbegrenzungen verzichtet. Module, welche diese zusätzlichen Informationen benötigen, müssen

diese selbst erzeugen. Abbildung 16 zeigt ein Beispiel für eine Kette von PUs mit verschiedenen Auswirkungen auf den Datenstrom. Über ein Bus-IM erfolgt die Anbindung an den Hauptspeicher bzw. Systembus. Alle Module haben eine SDI-Schnittstelle. Die Übertragung des Datenstroms erfolgt parallel und synchron, wobei die Busbreite pro Verbindung individuell festgelegt werden kann. Das Signal `clk` gibt den Takt für die Daten auf dem n-breiten Bus `data[n]` an und `valid` die Gültigkeit des gerade anliegenden Datenworts.

PUs arbeiten möglicher Weise mit unterschiedlichen Geschwindigkeiten oder haben keine konstante Datenrate beim Lesen oder auch Schreiben. Um den Datenfluss zu kontrollieren, wird das asynchrone Signal `stop` verwendet, welches sich entgegen des Datenstroms ausbreitet. Kann ein Modul gerade keine Daten aufnehmen, aktiviert es `stop` und hält damit das Vorgängermodul an, welches das Signal, falls notwendig, unmittelbar weiterleitet. Damit werden Datenverluste vermieden. Ist die PU-Kette zu lang, so dass das Stoppsignal nicht innerhalb eines Takts bis zum ersten Modul einer Staukette propagiert wird, kann dies im Design durch einen Zwischenpuffer für ein Datenwort ausgeglichen werden.

Die Einfachheit der Schnittstelle erleichtert ebenfalls die Einarbeitung und somit die Implementierung neuer SDI-Module.

Eine Erweiterung der vorliegenden Arbeit stellt die dynamische Änderung der Architektur unter Verwendung der partiellen Rekonfiguration dar (siehe [3, 9] und 7.2 *Ausblick*). Die für die Kommunikation zwischen variablen und festen Bereichen des FPGAs notwendigen sogenannten Busmacros[9] sind mengenmäßig begrenzt und nur an bestimmten Koordinaten im Chip verfügbar. Für den Fall, dass die Busmacros nicht ausreichen, kann die Breite des SDI-Datenstroms für den Übergang einfach reduziert werden. Dies erfolgt mit Hilfe von (De)Serializer-Modulen.

6.1.2 Processing Units (PU)

Im Rahmen von verschiedenen Lehrveranstaltungen und Abschlussarbeiten wurden diverse Funktionen in Hardware implementiert, welche in Bildverarbeitungsalgorithmen häufig benutzt werden. In den Projekten [3, 12, 8, 7, 10, 11, 4, 5, Rec07, And10] wurden weitere Funktionen implementiert. Interessante Funktionen werden im Folgenden näher erläutert.

[9] bei Xilinx

Farbraumkonvertierung: Es gibt viele Modelle um Farbräume mathematisch darzustellen. Wichtige im Bereich der Bildverarbeitung sind u. a. RGB (red, green, blue), YCrCb (Helligkeit und Chrominanz), Graustufen und schwarz/weiß (binär). Je nachdem in welchem Format die Bilddaten von der Kamera geliefert werden und wie die Daten von den Bildverarbeitungsalgorithmen weiterverarbeitet werden, ist die Konvertierung in einen anderen Farbraum notwendig.

generische Faltung: Viele Filter in der Bildverarbeitung basieren auf der Faltung. Mit Hilfe des *sliding window* Verfahrens kann die Faltung sehr effizient in Hardware implementiert werden. Dazu wird bei einer 3×3 Matrix der eingehende Bilddatenstrom durch zwei Bildzeilenpuffer geleitet. Nach einer Initialverzögerung zur Füllung der Puffer kann pro Takt ein Ergebniswert berechnet werden. An den jeweiligen Enden der Puffer können jetzt die 9 Werte für die Matrixoperation der Faltung abgegriffen und weiterverarbeitet werden. Auf diese Weise sind sowohl lineare als auch nicht lineare Operationen möglich, wie z. B. Glättung (Gauß, Mittelwert), Kantendetektion (Gradient, Laplace, Sobel, Kompass) oder Farbraumkonvertierungen, sowie beliebige Matrixoperationen, als auch Rangordnungsoperationen (Erosion, Dilatation, Median). Entsprechend können Filter für größere Matrizen entworfen werden.

Binarisierung: Für die Konvertierung eines Graustufenbilds in ein Schwarzweißbild ist ein Schwellwert zu bestimmen, mit dem der Grauwert jedes Bildpunkts verglichen wird. Dies kann einmal für das ganze Bild erfolgen oder z. B. bei großen Helligkeitsunterschieden auch pro Bildpunkt oder Bildbereich. Das statistische Verfahren von Otsu [Ots79] optimiert die Streuung zwischen den Klassen (hier schwarz und weiß) mit Hilfe der Varianz. Dazu ist ein Histogramm notwendig.

Histogramm: Mit einem kleinen Speicher für 256 Werte kann sehr einfach das Histogramm für ein 8 Bit Graustufenbild in Hardware implementiert werden. Pro Bildpunkt wird der aktuelle Wert in einer Speicherzelle inkrementiert, wobei der Grauwert des Bildpunkts die Zellenadresse angibt.

Segmentierung: Die Segmentierung nach Horprasert et al. [HHD99] erkennt bewegte Objekte und deren Schatten vor einem statischen Hintergrund in Farbbildern und ist sehr robust. Um Vordergrund und Hintergrund in einem Video zu trennen, wird das aktuelle Bild mit einem gelernten Hintergrund verglichen. Die Entscheidung,

welcher Klasse ein Bildpunkt zugeordnet wird, basiert auf Helligkeits- und Farbzahländerungen, ist gut für eine datenstromorientierte Verarbeitung geeignet und wurde in [3] implementiert.

Gesichtserkennung: In [Rec07] wurde eine Gesichtserkennung implementiert. Dabei werden die Augen auf Grund der stärker wechselnden Grauwerte auf einer Linie erkannt und der Mund auf der Basis von bestimmten Chrominanzverhältnissen im roten und blauen Farbbereich kombiniert mit einer Kantendetektion gefunden. Entwickelt wurden neben den eben genannten Detektoren für Augen und Mund auch ein Modul für die Bereichsauswahl (Aus dem Gesamtbild wird ein bestimmter Bereich ausgeschnitten und nur dieser weitergeleitet).

Kontrolle des Datenstroms: In [Rec07] u. a. wurden mehrere PUs zur Datenflusskontrolle, wie z. B. Teilen/Zusammenführen und Zwischenpuffer implementiert.

Integralbild, Bildhelligkeit: In [And10] wurde ein Fahrerassistenzsystem entwickelt, welches beim Wechsel der Fahrspur andere Fahrzeuge detektiert und ggf. den Fahrer warnt. Dazu wurden sowohl Filter für die minimale und die durchschnittliche Helligkeit implementiert, als auch die Berechnung des Integralbilds (Summe der Grauwerte aller Bildpunkte in einem rechteckigen Bereich). Interalbilder können auch zur effizienten Berechnung von Haar-Filtern eingesetzt werden (Bereich: Merkmals- und Objekterkennung).

Hilfsmodule: Weitere Hilfsmodule für Quellen und Senken eines SDI-Datenstroms wurden implementiert. Dazu gehören ein VGA-Ausgabemodul für verschiedene Auflösungen und Farbmodi, als auch Kameramodule für digitale Bildsensoren (siehe auch Kapitel 6.3 *PICSY* und 6.4 *PAMo*) und für Decoderchips, die analoge Videosignale digitalisieren.

6.1.3 Interconnection Module (IM) für den PLB-Bus

Für die Implementierung des ersten IMs standen für die Xilinx FPGAs verschiedene Busse zur Auswahl. Der Processor Local Bus (PLB) ist Teil der CoreConnect Spezifikation und der Standardbus für PowerPC-Architekturen. Das System-on-Chip Entwicklungswerkzeug EDK von Xilinx setzt standardmäßig den Multi-Port Memory Controller (MPMC) als

Speicherkontroller für externen DDR-RAM ein. Der MPMC unterstützt folgende Schnittstellen: PLB, Native Port Interface (NPI), Soft Direct Memory Access (SDMA) und Video Frame Buffer Controller (VFBC).

Die Schnittstellen NPI und VFBC erreichen hohe Geschwindigkeiten und haben ein einfaches Protokoll. Allerdings sind sie wenig verbreitet, so dass es dafür kaum Hardwaremodule gibt. SDMA unterstützt einen Scatter/Gather-Modus und kann dadurch mehrere im Speicher verteilte Blöcke in einer Transaktion verarbeiten. Das Modul ist dadurch sehr viel komplexer bezüglich des Protokolls und auch der benötigten Chipfläche. Weiter erfordert dieser Modus auch auf Softwareseite (Betriebssystem) durch die Vorbereitung der Speicherdescriptoren einen höheren Aufwand.

Für die Implementierung des IMs wurde der PLB gegenüber den anderen Schnittstellen vorgezogen, weil er hohe Geschwindigkeiten erreicht und ein Standardbus ist, was auch eine gewisse Sicherheit für zukünftige Entwicklungen mit sich bringt. In der Spezifikation findet sich außerdem eine ressourcensparende Variante für die Punkt-zu-Punkt-Kommunikation. Die Implementierung des PLB-basierten IMs wird im Folgenden SDI-Kontroller genannt.

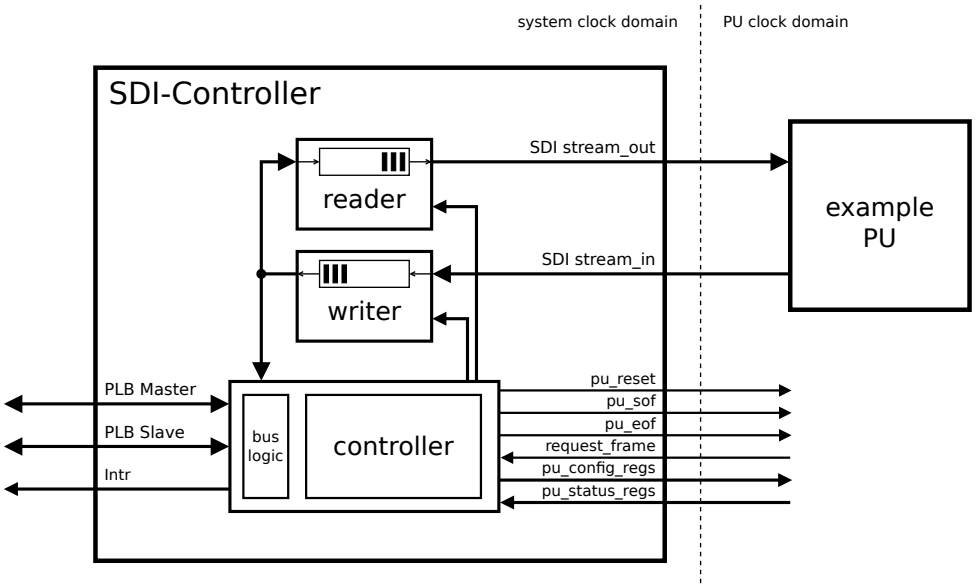


Abbildung 17: Interconnection Module für den PLB-Bus

Der SDI-Kontroller besteht aus den drei Komponenten: *Reader*, *Writer* und *Controller*, wie in Abbildung 17 dargestellt. Er wurde so konzipiert, dass er Datenströme vom Hauptspeicher zu einem SDI-Modul und umgekehrt verarbeiten kann.

Der *Controller* koordiniert alle Lese- und Schreibzugriffe auf den PLB und verarbeitet Steuerbefehle von der Software. *Reader* und *Writer* arbeiten unabhängig von einander, haben jeweils einen Zwischenpuffer und stellen eine SDI-Schnittstelle bereit. Die Zwischenpuffer (FIFO) sind asynchron und folglich können SDI-Kontroller und angeschlossene PUs mit unterschiedlichen Taktfrequenzen arbeiten. Der Datentransfer erfolgt über die PLB-Master-Schnittstelle und ermöglicht so eine Direktverbindung zum Speicherkontroller. Zur effizienten Nutzung des Systembusses erfolgen die Datentransfers im Burst-Modus, d. h. in einer Transaktion werden viele Datenworte übertragen und dadurch der Zeitverlust durch die Busarbitrierung insgesamt reduziert. In [Rec07] wurde der Datendurchsatz bezüglich der Burst-Länge und der Anzahl der aktiven Master an einem Bus untersucht. Um Ressourcen zu sparen, können die beiden Datenkanäle (*Reader* und *Writer*) durch generics in VHDL unabhängig von einander aktiviert werden.

Der Zugriff auf die Softwareregister erfolgt über die PLB-Slave-Schnittstelle. PLB Master und Slave können optional am selben PLB hängen. Alle Register und Flags sind im Anhang in Tabelle 4*, 5 und 6 zusammengefasst. Durch Status- und Kontrollregister kann der Software-Treiber den SDI-Kontroller überwachen und steuern. Um z. B. ein Bild durch eine Kette von PUs zu leiten, werden Start- und Endadresse von jeweils Quell- und Zieladresse im Bildspeicher übergeben und der SDI-Kontroller über das Kontrollregister gestartet. Über folgende Signale können zusätzliche Funktionen des SDI-Kontrollers genutzt werden. Nach wichtigen Ereignissen, wie z. B. der vollständigen Übertragung eines Datenblocks, kann der Treiber durch die Auslösung eines Interrupts benachrichtigt werden. Die Interrupts sind maskierbar. Das Signal `pu_reset` kann vom Software-Treiber ausgelöst werden und dazu verwendet werden, die PUs in einen definierten Zustand zurückzusetzen. Die asynchronen Ausgänge `pu_sof` und `pu_eof` (start/end of frame) markieren den Anfang und das Ende eines Datenblocks z. B. für PUs, welche sich für jeden Block initialisieren müssen. Für die individuelle Parametrisierung und Rückmeldung von PUs per/zur Software gibt es die frei benutzbaren PU Status- und Kontrollregister `pu_status_regs[n]` bzw. `pu_control_regs[m]`, deren Anzahl durch generics in VHDL festgelegt werden kann.

* auf Seite 111

Somit können z. B. Parameter einer generischen Faltung übergeben oder die Anzahl der verarbeiteten Bildpunkte abgefragt werden.

Der SDI-Kontroller unterstützt zwei Betriebsarten: *single mode* und *continuous mode*. Der single mode arbeitet wie oben beschrieben. Im continuous mode wird der Datentransfer nach Abschluss automatisch neu gestartet. Der Neubeginn kann sofort oder alternativ durch ein externes Signal, den Eingang `request_frame`, angestoßen werden. Dieser Modus ist z. B. für Videodatenströme von einem Bildsensoren oder zu einem VGA-Ausgang sinnvoll.

Neben der Start- und Endadresse eines Datenblocks kann optional noch eine weitere Adresse (`trigger_address`) gesetzt werden. Wird diese Speicheradresse vom *Controller* während der Verarbeitung eines Datenblocks erreicht, wird ein Interrupt ausgelöst. Dieser Mechanismus kann z. B. für *double buffering* eingesetzt werden.

Der *Controller* verfügt des weiteren über einen Taktzähler, welcher über oben genannte Ereignisse gestartet bzw. gestoppt werden kann. Damit sind z. B. Performancemessungen möglich.

Die Implementierung des SDI-Kontrollers erfolgte ursprünglich für den PLB der Version 3.4, der später von Xilinx nicht weiter unterstützt wurde. Für den Nachfolger PLB 4.6 wurde der SDI-Kontroller komplett neu implementiert und um einige Funktionen erweitert. Der *Controller* unterstützt einen Systemtakt von 100 MHz. In der vollen Konfiguration (mit *Reader* und *Writer*) werden ca. 1200 slices und 2 BRAMs benötigt[10]. Der SDI-Kontroller wurde von Felix Mühlbauer konzipiert und zusammen mit Ali A. Zarezadeh implementiert. Mehr Implementierungsdetails sind in [Zar11] beschrieben.

6.1.4 Interconnection Module (IM) für die Coprozessorschnittstelle des PowerPCs

Der PowerPC 405 verfügt über eine Coprozessorschnittstelle, die über den APU Kontroller (Auxiliary Processor Unit) an den Prozessorkern angekoppelt ist, und an die ein sogenanntes Fabric Coprocessor Module (FCM) angeschlossen werden kann. In einem FCM können verschiedene vordefinierte als auch benutzerdefinierte APU-Prozessorbefehle verarbeitet und somit der Befehlssatz des Prozessor erweitert werden. Ein IM kann

[10] Gemessen auf einem Virtex-4 FPGA von Xilinx

als FCM implementiert werden und damit den PowerPC-Prozessor direkt an einen SDI-Kanal ankoppeln.

Die APU des auf dem XUP Entwicklungsboard eingebauten Virtex-II FPGAs wurde nicht mit den Logikzellen verbunden und kann nicht benutzt werden. Beim Virtex-4 kann die APU verwendet werden. Xilinx stellt dafür Patches für den GCC-Compiler und Wrapperfunktionen bereit. In diesen Funktionen werden die selben Assemblerbefehle aufgerufen, wie für den FSL des MicroBlaze-Prozessors. Der modifizierte Compiler übersetzt diese dann in entsprechende APU-Befehle. Für dieses Experiment mussten die Patches von Xilinx in die OpenEmbedded Toolchain eingebaut werden. Warum der Umweg über die MicroBlaze-Implementierung gegangen und nicht direkt APU-Befehle verwendet wurden ist unklar.

Während der Entwicklung stellte sich jedoch heraus, dass die APU-Befehle nur im Privilegierten Modus des PowerPCs ausgeführt werden können. Das entspricht bei Linux dem kernel mode und hat zur Folge, dass das FCM nicht direkt aus der Anwendung (user mode) angesprochen werden kann und stellt die Zweckmäßigkeit der Entwicklung eines solchen IMs in Frage. Der notwendige Betriebssystemaufruf (system call) muss für jedes 32-Bit Datenwort einzeln erfolgen und verursacht damit, wie bereits in Kapitel 5.1.2 *Zugriff auf Hardwarekomponenten* beschrieben, einen erheblichen Aufwand, der die Bandbreite, Daten über das APU zu transferieren, zu sehr reduziert.

6.2 Beispielanwendung

Zur Veranschaulichung des Entwurfsraums wurde als Beispielanwendung ein Canny-Kantendetektor mit folgenden Verarbeitungsschritten gewählt: Kamera → Gauß-Filter (Entrauschen) → Sobel (Kantendetektion) → Gradient der Kanten berechnen → Nachbarpixel vergleichen (Spur). Während die ersten Schritte sehr schnell in Hardware implementiert werden können, enthält der letzte Schritt keine linearen Speicherzugriffe, weshalb nur eine Softwareimplementierung angenommen wird. In Tabelle 3 sind der (reziproke) Datendurchsatz und die Ressourcenverbräuche der verschiedene Komponenten zusammengefasst.

Abbildung 18 zeigt zwei Interessante der Ergebnisarchitekturen für unterschiedliche Begrenzungen der Chipfläche (siehe Beschriftung rechts unten). Es ergibt sich eine Gesamtlaufzeit von 21 Zeiteinheiten für die schnellste Architektur, bzw. 37 Zeiteinheiten für eine

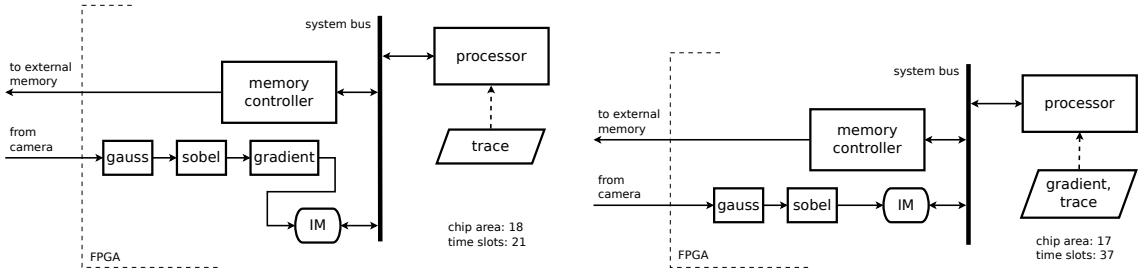


Abbildung 18: Ergebnisarchitekturen für verschiedene Begrenzungen der Chipfläche. Die reine Softwarelösung benötigt 65 Zeiteinheiten.

	1/Datendurchsatz			
	Software	Hardware	HW-Ressourcen	
Funktion			slices	BRAM
Gauß	16	1	2	2
Sobel	16	1	2	2
Gradient	8	2	1	0
Spur	16	-	-	-
Systembus			10	
IM (Bus)			3	2

Tabelle 3: Übersicht der Metainformationen

Architektur mit weniger Chipfläche, während die reine Softwarelösung 65 Zeiteinheiten benötigt.

6.3 Potsdam Intelligent Camera System (PICSY)

Gegenüber den herkömmlichen zentralisierten Mehrkamerasystemen, verfügen *Intelligente Kameras* über genug Rechenleistung, um Bilddaten lokal vorzuverarbeiten und gewünschte Informationen zu extrahieren. Da diese Datenmengen normalerweise sehr gering gegenüber den Bilddaten ausfallen, wird die erforderliche Bandbreite für die Kommunikation zu einer Zentrale oder auch zwischen Kamerasystemen erheblich reduziert. Anwendungen sind in vielen Bereichen wie Automotive, Sicherheit, Navigation und Assisted Living anzutreffen, wie z. B. die optische Qualitätsprüfung in Produktionsanlagen.

In der Arbeitsgruppe von Professor Bobda wurde das intelligente Kamerasystem PICSY entwickelt. Die Plattform wurde in Hinblick auf Adaptivität, Flexibilität, Programmierbarkeit, Größe, Stromverbrauch und Skalierbarkeit entworfen. Das System basiert deshalb auf FPGAs, insbesondere wegen deren Rechenleistung und Flexibilität, und ist für high-performance Anwendungen ausgelegt. Weiter können mit diesem System Kamerasensornetzwerke aufgebaut werden. PICSY wurde von Felix Mühlbauer zusammen mit Marco Hirsch konzipiert und mit der Hilfe von Ali A. Zarezadeh realisiert. Es wurden mehrere Systeme hergestellt.

Das PICSY ist modular und stapelbar aufgebaut und besteht aus einer Schnittstellenplatine, einer Hauptplatine und optional aus weiteren Berechnungsplatinen.

Schnittstellenplatine. Auf der Schnittstellenplatine sind die Stromversorgung, mit den unterschiedlichen Spannungen für FPGAs und andere Chips, eine 100 MBit Ethernetschnittstelle und ein high-speed USB Hostadapter mit mehreren Ports untergebracht.

Hauptplatine. Das Herzstück der Hauptplatine ist ein Virtex-4 FX12 FPGA mit einem eingebetteten PowerPC 405, 100 MHz Systemtakt und 128 MB DDR-RAM und einem direkt angeschlossenem 3 Megapixel Farbbildsensor. Die Programmierung kann über verschiedene Wege erfolgen. Neben dem klassischen Plattform Flash PROM und der JTAG-Schnittstelle kann der FPGA per Microkontroller über den auf einer SD-Karte gespeicherten Bitstreams konfiguriert werden. Ebenso kann ein aus dem PROM geladenes Basissystem weitere Bitstream von der SD-Karte holen und per partieller Rekonfiguration laden. Über LVDS, ein Hochgeschwindigkeitsbus, wird eine sehr schnelle Kommunikation mit der Schnittstellenplatine und ggf. weiteren Berechungsplatinen ermöglicht. Es ist vorgesehen, dass auf dem PowerPC ein Betriebssystem läuft, welches die komplexe Kommunikation mit der Außenwelt, z. B. über IP oder USB, ermöglicht. Zur Beschleunigung von Bildverarbeitungsanwendungen können die FPGA-Ressourcen verwendet werden. Der Hauptprozessor dient als Koordinator und verteilt Aufgaben auf die anderen Berechnungsplatinen, welche dann kein Betriebssystem benötigen.

Berechnungsplatine. Der FPGA auf der Berechnungsplatinen stellt eine Art Coprozessor für den Hauptprozessor dar und ist von mehreren Speichern umgeben. Damit können Speicherengpässe umgangen werden, die häufig bei der Bildverarbeitung entstehen, weil

die verschiedenen IPcores jeweils auf einen eigenen Speicher zugreifen können. Der FPGA kann über das lokale Flash PROM oder über JTAG von der Hauptplatine aus konfiguriert werden. Berechnungsplatinen sind stapelbar und die Kommunikation erfolgt über einen Punkt-zu-Punkt-Bus von Platine zu Platine. Somit können Daten durchgeschleift, oder Aufgaben an obere Prozessoren deligiert werden.

Die Werkzeuge und Implementierung aus der vorliegenden Arbeit wurden für PICSY portiert und weitere Anwendungen im Bereich (verteilte) Kamerasysteme, insbesondere Personenverfolgung, Fahrerassistenzsysteme, hardwareunterstützte Middleware und rekonfigurierbare Architekturen entwickelt [3, 6, 15] und [And10, ZB10a, ZB10b, HMMB10, ZB11a, ZB11b]. Eine detaillierte Beschreibung und weitere Anwendungen von PICSY wurden in [Zar11] veröffentlicht.

6.4 Plattform für Autonome Modellfahrzeuge (PAMo)

Das Forschungsgebiet Autonomes Fahren hat in den letzten Jahren stark an Bedeutung zugenommen. Nach der *DARPA Grand Challenge (Defence Advanced Research Projects Agency)* durch die Wüste im Jahr 2004, folgte 2007 die *Urban Challenge*, bei der autonome Fahrzeuge durch eine Stadt mit Gebäuden und Verkehr navigieren mussten. Das Team der Universität Braunschweig nahm als bestes europäisches Team teil. 2008 startete die Universität Braunschweig den *Carolo-Cup*, ein Wettbewerb für alle deutschen Studenten, dessen Ziel es ist, ein Modellfahrzeug (Maßstab 1:10) zu entwickeln, welches in der Lage ist, einer modellierten Straße zu folgen, Hindernissen auszuweichen und eine Parklücke zu finden und einzuparken. [CC]

Währenddessen wurde von Philipp Mahr und Felix Mühlbauer an der Universität Potsdam die Lehrveranstaltung System-on-Chip Car (SoCar) aufgebaut. Studenten der Technischen Informatik sollten insbesondere im Bereich Eingebettete Systeme die Entwicklung von Software- und Hardwarekomponenten nähergebracht werden. Dazu wurde ein Spartan3E-Entwicklungsboard von Xilinx auf einem Modellfahrzeug montiert. Über ein selbstentwickeltes Erweiterungsboard wurde eine Schnittstelle zu Ultraschallsensoren, Drehzahlmesser, Fernbedienungsempfänger, Motor und Lenkservo hergestellt. Im Rahmen des Projekts sollten die Studenten eine Steuerung für das Modellfahrzeug realisieren. Das Fahrzeug sollte seine Umgebung mit Hilfe der Nahbereichssensoren erfassen und einen vorgegebenen Parkur ohne Kollision durchfahren können. Die Implementierung der

Steuerung erfolgte komplett im FPGA, der zwischen Fernbedienungsempfänger und Motor/Lenkservo geschaltet war. So konnte im Fehlerfall auf die manuelle Steuerung per Fernbedienung umgeschaltet werden. Beginnend mit einem PWM-Pulsgenerator für die Servos bis hin zur Steuerungssoftware auf einem MicroBlaze-Prozessor konnten die Studenten in der Praxis einen sehr guten Einblick in die Entwicklung im Hardware-Software-Co-Design bekommen.

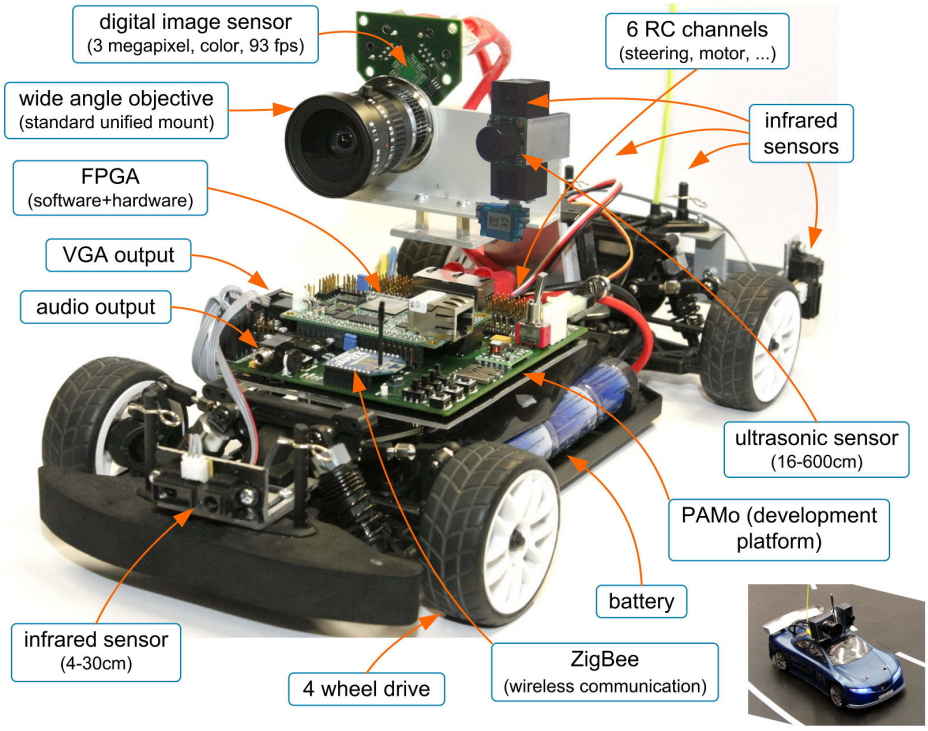
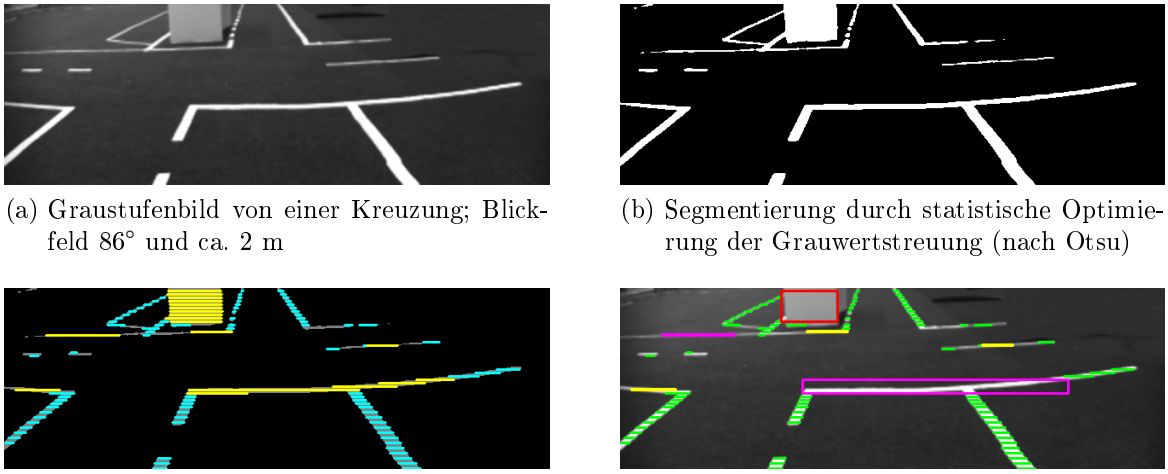


Abbildung 19: SoCar2 (System-on-Chip Car) mit der Plattform für Autonome Modellfahrzeuge (PAMo)

Beim Nachfolger SoCar2 steht eine hochauflösende Kamera für die Erfassung der Umgebung im Mittelpunkt. Das Herzstück der Steuerung ist die ebenfalls FPGA-basierte und von Max Frohberg und Felix Mühlbauer entwickelte *Plattform für Autonome Modellfahrzeuge (PAMo)*. In Abbildung 19 sind der Aufbau und die verschiedenen Schnittstellen und Sensoren dargestellt. Der 3 Megapixel Farbbildsensor arbeitet mit einer Bayer-Matrix und erreicht eine maximale Arbeitsfrequenz von 48 MHz, was bei einer Auflösung von 640×480



(a) Graustufenbild von einer Kreuzung; Blickfeld 86° und ca. 2 m

(b) Segmentierung durch statistische Optimierung der Grauwertstreuung (nach Otsu)

(c) Zusammenfassung von Liniensegmenten

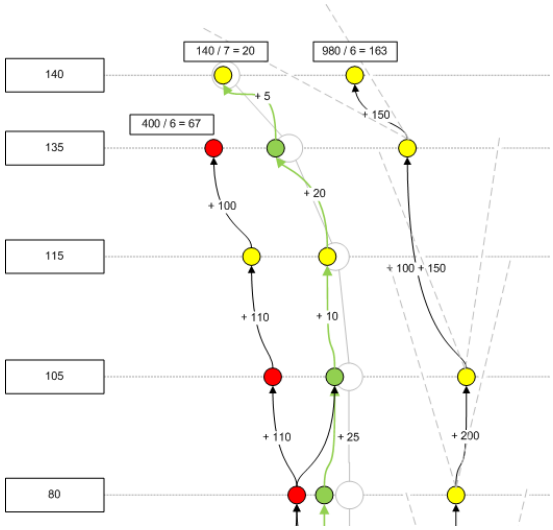
(d) Klassifizierung (Straßenmarkierung, Hindernis, Stopplinie) und Merkmalsextraktion

Abbildung 20: SoCar2: Bildverarbeitung

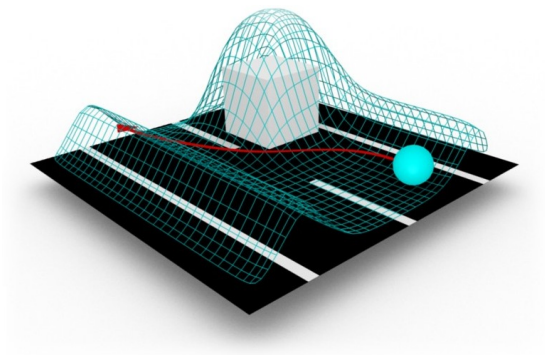
Bildpunkten 93 Bilder pro Sekunde bedeutet[11]. Die Auflösung und viele weitere Parameter, z. B. für Farbkorrektur, werden über I^2C programmiert. Durch das Weitwinkelobjektiv kann auch vorausschauend in Kurven *gesehen* werden. Der Bildsensor ist über LVDS, ein Hochgeschwindigkeitsbus, an den FPGA angeschlossen. LVDS arbeitet mit differenziellen Signalen und ist daher auch über größere Strecken sehr störungsunempfindlich. Auf PAMo sitzt das Entwicklungsboard Suzaku SZ410 mit einem Virtex-4 (XC4FX12) FPGA und 32 MB DDR-RAM. Der Virtex-4 hat einen eingebetteten PowerPC 405 Hardcore-Prozessor, welcher mit bis zu 450 MHz getaktet werden kann, die Logikzellen mit bis zu 500 MHz.

PAMo bietet vielfältige Debug-Schnittstellen an: Neben JTAG gibt es zwei serielle Schnittstellen (RS232), drahtgebunden und per Funk, mit jeweils 115 kbs und einen 100 MBit Ethernetanschluss. Ein VGA-Ausgang mit 24 Bit Farbe ist bei der Verarbeitung der Kamerabilder sehr hilfreich. Über einen Audio-Ausgang, inkl. Lautsprecher, und LEDs stehen weitere Rückmeldemöglichkeiten zur Verfügung.

[11] Der gleiche Bildsensor wurde auf PICSY eingesetzt.



(a) Rekonstruktion der Straße (grau) und Klassifizierung in linke, rechte und Mittelmarkierung (Farbe).



(b) Hindernisse werden als Potentialfelder und das Fahrzeug als Kugel modelliert.

Abbildung 21: SoCar2: Umweltrepräsentation und Wegeplanung

Für die Entwicklung der Steuerung wurde ein Simulator aufgebaut. Nach der Untersuchung verschiedener Lösungen wurde die Open Source 3D-Grafik-Software *Blender* eingesetzt [Ble]. Mit ihr können dreidimensionale Welten modelliert und gerendert werden. Die integrierte *Blender Game Engine Python API* ermöglicht eine interaktive Kamerafahrt durch eine modellierte Welt. Darüber hinaus kann das gerenderte Kamerabild über die API ausgelesen und durch eigene Software weiterverarbeitet werden. Die entwickelte Steuerungssoftware konnte somit am PC getestet und eins zu eins auf den eingebetteten PowerPC übertragen werden. Für die Bildverarbeitung wurde die Open Source Bibliothek *OpenCV* verwendet [OCV].

Anschließend wurde die Software analysiert und rechenintensive Teile identifiziert. Daraufhin konnte ein Großteil der Bildverarbeitung in Hardware ausgelagert und die Performance erheblich gesteigert werden. In Abbildung 20 sind die verschiedenen Arbeitsschritte dargestellt. Für die Differenzierung zwischen den Straßenmarkierungen und dem Hintergrund wurde das statistische Verfahren von Otsu verwendet. Mit Hilfe der Varianz wird die Streuung zwischen den Graustufenklassen optimiert und somit ein Schwellwert für die nachfolgende Binarisierung bestimmt. Im resultierenden Schwarzweißbild erfolgt zeilenweise eine Zusammenfassung von untereinander liegenden horizontalen Liniensegmenten.

Die Daten werden in mehreren Puffern der Software zu Verfügung gestellt. Die neuen Objekte werden anschließend in Straßenmarkierungen, Hindernisse und Stopplinien klassifiziert. Die Koordinaten werden perspektivisch entzerrt und in eine Karte eingetragen. Die Rekonstruktion der Straße erfolgt mit einem probabilistischen Modell, welches die aus dem Bild erkannten Straßenmarkierungen der linken, rechten oder mittleren Linie zuordnet und ggf. fehlende Linien ergänzt (siehe Abbildung 21a).

Daten der Distanzsensoren werden ebenfalls in die Karte eintragen. Die Wegeplanung des Fahrzeugs basiert auf der Potentialfeldmethode. Abbildung 21b zeigt die Modellierung von Hindernissen und das Fahrzeug als Kugel, die durch ein Tal rollt.

Das Fahrzeug war auf regionalen Veranstaltungen, wie der Tag der offenen Tür der Universität Potsdam oder der Langen Nacht der Wissenschaften Berlin/Brandenburg auch mehrfach auf dem Carolo-Cup [CC] und auf der CeBit und der internationalen DATE Konferenz [12] in Aktion zu sehen.

7 Zusammenfassung und Ausblick

Bei der Entwicklung neuer Anwendungen wird oft auf die bereits bekannten Architekturen zurückgegriffen, anstatt eine Architektur zu verwenden, welche am besten für die Anwendung geeignet ist. Auf der anderen Seite ist es schwierig, in der Vielzahl von Architekturen die Richtige zu finden. Ähnlich verhält es sich mit universellen Programmiersprachen, wie C oder Java, welche, insbesondere für Anwendungen mit hohem Datendurchsatz oder hoher Parallelität, nicht die effizienteste Wahl sind. Der Nutzung von effizienten, an der Anwendung orientierten, Architekturen stehen häufig wirtschaftliche Interessen, wie die Einarbeitungszeit der Entwickler und Kosten für Werkzeuge entgegen.

7.1 Zusammenfassung

In der vorliegende Arbeit wurden Werkzeuge, Methoden und der Entwurf für komplexe Bildverarbeitungssysteme auf rekonfigurierbaren System-on-Chip-Architekturen vorgestellt. Der Entwurfsfluss ist abschließend detailliert in Abbildung 22 dargestellt.

Komplexe Bildverarbeitungssysteme sind z. B. verteilte Kamerasensornetzwerke, welche neben der Bildverarbeitung auch höhere Anforderungen an die Kommunikation haben und deshalb ein entsprechendes Betriebssystem voraussetzten. Ausgehend von einer reinen Softwarelösung, basierend auf Linux und OpenCV, wird die Bildverarbeitungsbibliothek durch Hardwarebeschleuniger unterstützt und die entsprechenden Softwareteile durch die Kommunikation mit Hardwaretreibern ersetzt. Bildverarbeitung ist inhärent parallel und lässt sich gut in Datenströmen organisieren. Eine dafür passende Struktur und Schnittstellen für die Generierung entsprechender Architekturen wurde vorgestellt. Die Hardware-Software-Partitionierung und die Ablaufplanung wird mit Hilfe von ASP (answer set programming) automatisch gelöst. Die Cross-Kompilierung der gesamten Software erfolgt durch geeignete Werkzeuge ebenfalls automatisiert. Das resultierende SoC wird auf einem FPGA realisiert.

Neben funktionalen wurden auch wirtschaftliche Aspekte, wie Einarbeitungsaufwand, Wartung und Erweiterbarkeit berücksichtigt. Durch die Definition geeigneter Hardware- und Softwareschnittstellen und die größtenteils automatisierten Abläufe kann die Einarbeitungszeit in den vorgestellten Entwurfsfluss gering gehalten und für neue Anwendungen

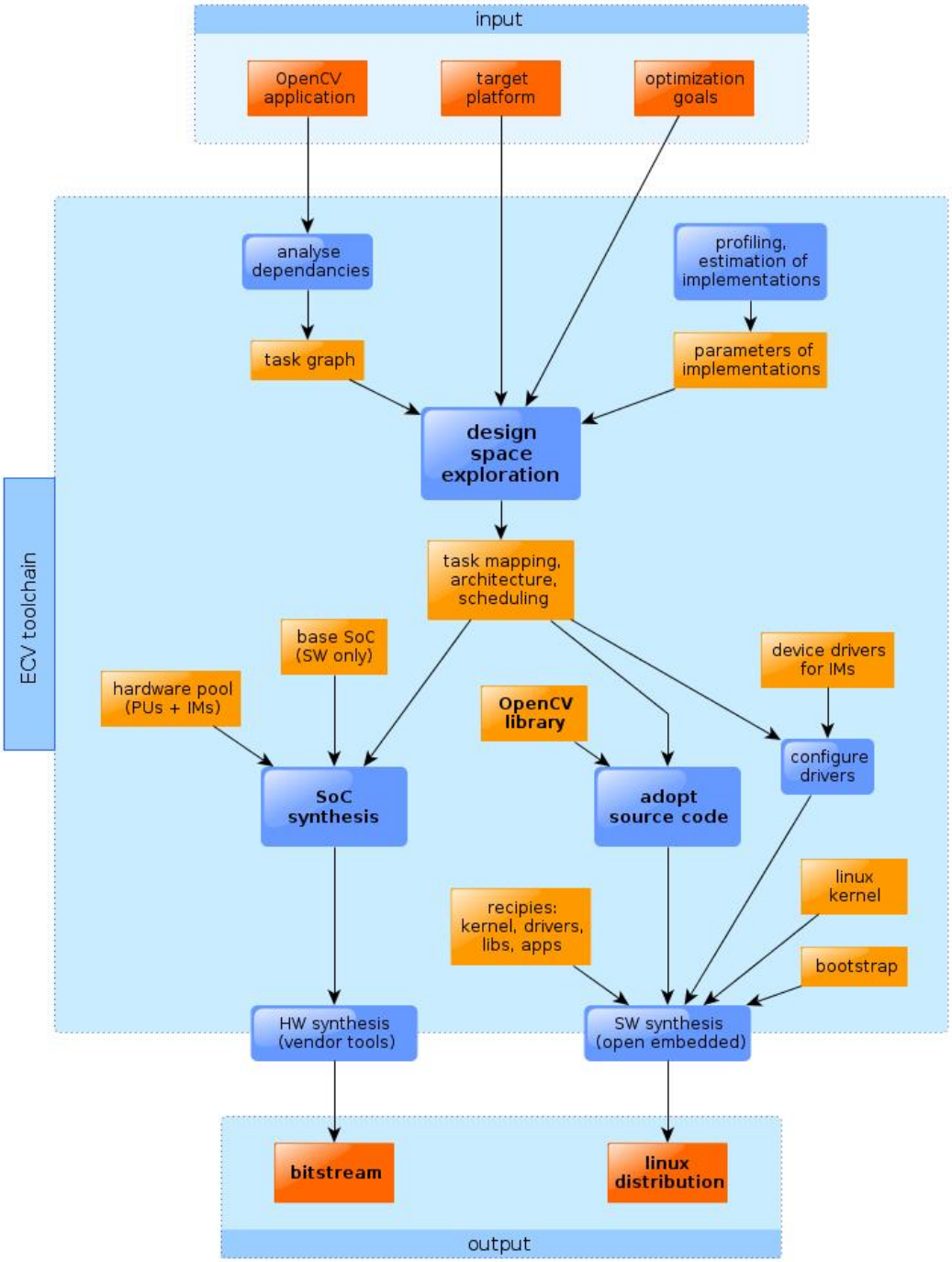


Abbildung 22: Entwurfsfluss

einfach erweitert werden. Der Einsatz von Open-Source-Software war durchgängig möglich.

7.2 Ausblick

Im Folgenden werden einige Ideen aus verschiedenen Bereichen für die Fortführung der vorliegenden Arbeit vorgestellt.

7.2.1 Systemarchitektur

Der SDI-Kontroller hat momentan jeweils einen Lese- bzw. Schreibkanal. Um Logikzellen zu sparen, könnte anstatt mehrere SDI-Kontroller in einem System zu verwenden, der Kontroller für mehrere Kanäle erweitert werden. Die Buslogik und der Steuerungsteil werden damit nicht unnötig vervielfacht.

Neben den herkömmlichen Beschreibungssprachen könnten auch ASLs (algorithm-specific language) eingesetzt werden, um Hardwaremodule zu implementieren.

Der Marktführer Xilinx wechselt in der neuen Generation von FPGAs für die eingebetteten Prozessoren von PowerPC zu ARM. Der passende Systembus AIX basiert auf dem bekannten AMBA. Dieser unterstützt auch Punkt-zu-Punkt- und Streaming-Modi und könnte SDI ersetzen.

Der ARM Cortex-A9 Prozessor in den PSoCs von Altera verfügt über einen sogenannten Accelerator Coherence Port. Dieser ermöglicht den Zugriff von Hardwarebeschleunigern auf den Hauptspeicher ohne das Cache-Kohärenz-Problem zu verursachen. [Goo]

Auch im Bereich der Hardware gibt es Open-Source-Projekte. Bei den Softcore-Prozessoren ist neben OpenSPARC und Leon, insbesondere der OpenRISC ein interessanter Kandidat. Wishbone ist der zugehörige Systembus und unterstützt ebenfalls Punkt-zu-Punkt-Kommunikation. Außerdem erhielten seit der Version 3.1 die Architekturen ARM und OpenRISC Einzug in den offiziellen Linux-Kernel. Damit könnte eine komplette Open-Source-Lösung erstellt werden.

Neben dem Bereich Bildverarbeitung kann der vorgestellte Entwurfsfluss ebenso für andere Anwendungen mit einer datenstromorientierten Verarbeitung benutzt werden.

7.2.2 ASP Modellierung

Ein Problem bei der Modellierung realer Abläufe ist die Beschränkung auf den ganzzahlige Wertebereich der Gewichte. Sollen mehrere nicht-ganzzahlige Gewichte in Relation gebracht werden, muss eine Normierung stattfinden, welche den gültigen Zahlenbereich der Gewichte überschreiten kann und somit möglicherweise zu nicht-optimalen Lösungen führt. Die Entwickler der Potassco Suite arbeiten derzeit an der Integration von gebrochenen Zahlen. Unter Ausnutzung dieser Erweiterung könnte das vorliegende ASP-Modell weiterentwickelt und insbesondere die zeitliche Auflösung verbessert werden.

7.2.3 Partielle Rekonfiguration

Die Partielle Rekonfiguration ermöglicht die Wiederverwendung von FPGA-Ressourcen während der Laufzeit. Dazu wird der FPGA in einen statischen und einen dynamischen Bereich eingeteilt. Die Kommunikation dazwischen erfolgt über sogenannte Busmacros[12], welche gegenüber einer Rekonfiguration persistent sind. Für den Fall, dass die Busmacros nicht ausreichen, kann die Breite des SDI-Datenstroms für den Übergang einfach, mit Hilfe von (De)Serializer-Modulen, reduziert werden. Die Technologie ermöglicht den Austausch von PUs während das System läuft und somit den Einsatz von mehr Hardwarebeschleunigern. Dies kann in der Ablaufplanung mit berücksichtigt werden und ebenso die Verzögerungszeit, welche für die Rekonfiguration benötigt wird. Die notwendigen Anpassungen des ASP-Modells ist geringfügig. In [9, 1, 6, 3] wurde dafür bereits ein Grundstein gelegt.

[12] bei Xilinx

Literaturverzeichnis

[AKLS05] Anger, Christian ; Konczak, Kathrin ; Linke, Thomas ; Schaub, Torsten: A Glimpse of Answer Set Programming. In: *Künstliche Intelligenz* (2005), Nr. 1/05, S. 12–17

[And10] Anders, Jakob: *Advanced Driver Assistance Systems using Multiple Cameras*, University of Potsdam, Diplomarbeit, 2010

[Bar03] Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003

[BC06] Bovet, Daniel P. ; Cesati, Marco: *Understanding the Linux Kernel.* 3rd ed. O'Reilly Media, 2006

[BCBB02] Benkrid, K. ; Crookes, D. ; Benkrid, A. ; Belkacemi, S.: A Prolog-Based Hardware Development Environment. In: Glesner, Manfred (Hrsg.) ; Zipf, Peter (Hrsg.) ; Renovell, Michel (Hrsg.): *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream* Bd. 2438. Springer Berlin / Heidelberg, 2002. – ISBN 978–3–540–44108–3, S. 3–42

[BCSB01] Benkrid, K. ; Crookes, D. ; Smith, J. ; Benkrid, A.: High Level Programming for FPGA Based Image and Video Processing Using Hardware Skeletons. In: *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM).* Los Alamitos, CA, USA : IEEE Computer Society, 2001. – ISBN 0–7695–2667–5, S. 219–226

[BFRR01] Bräunl, Thomas ; Feyrer, Stefan ; Rapf, Wolfgang ; Reinhardt, Michael: *Parallel Image Processing.* Springer-Verlag, 2001

[BHvW09] Biere, A. (Hrsg.) ; Heule, M. (Hrsg.) ; van Maaren, H. (Hrsg.) ; Walsh, T. (Hrsg.): *Frontiers in Artificial Intelligence and Applications.* Bd. 185: *Handbook of Satisfiability.* IOS Press, 2009

[Ble] *Blender.* http://www.blender.org, Abruf: 10.02.2011

[BLLN06] Baptiste, Philippe ; Laborie, Philippe ; Le Pape, Claude ; Nuijten, Wim: Constraint-Based Scheduling and Planning. In: [RvW06], Kapitel 22, S. 759–797

[BLTZ03] Bleuler, Stefan ; Laumanns, Marco ; Thiele, Lothar ; Zitzler, Eckart: PISA – A Platform and Programming Language Independent Interface for Search Algorithms. In: Fonseca, Carlos M. (Hrsg.) ; Fleming, Peter J. (Hrsg.) ; Zitzler, Eckart (Hrsg.) ; Deb, Kalyanmoy (Hrsg.) ; Thiele, Lothar (Hrsg.): *Evolutionary Multi-Criterion Optimization (EMO 2003)*. Berlin : Springer, 2003 (Lecture Notes in Computer Science), S. 494 – 508

[BMAI10] Bobda, Christophe ; Mahr, Philipp ; Andres, Benjamin ; Ishebabi, Harold: Application-driven architecture synthesis of on-chip Multiprocessor systems. In: [SM10], S. 591–598

[Caa08] Caarls, Wouter: *Automated Design of Application-Specific Smart Camera Architectures*, Technische Universität, Delft, Diss., 2008

[CC] *Carolo-Cup - Studenten entwickeln autonome Modellfahrzeuge*. http://www.carolo-cup.de, Abruf: 10.02.2011

[Cel] Celoxica: *PixelStreams*. http://www.celoxica.com/products/pxs, Abruf: 10.02.2011

[DFGa] DFG: *Adaptive Multiprocessor on-Chip (AMoC) Project*. http://gepris.dfg.de/gepris/OCTOPUS/;jsessionid=1?module=gepris&task=showDetail&context=projekt&id=34855752, Abruf: 11.09.2011

[DFGb] DFG: *Adaptive Multiprocessor on-Chip (AMoC) Project - Schwerpunktprogramm, Teilprojekt zu SPP 1148: Rekonfigurierbare Rechensysteme*. http://gepris.dfg.de/gepris/OCTOPUS/;jsessionid=1?module=gepris&task=showDetail&context=projekt&id=113212753, Abruf: 10.09.2011

[DVB+09] Denecker, M. ; Vennekens, J. ; Bond, S. ; Gebser, M. ; Truszczyński, M.: The Second Answer Set Programming Competition. In: [ELS09], S. 637–654

[Ele] ElectronicsWeekly.com: *FPGAs are ready to take on ASICs, says Xilinx CTO*. http://www.electronicsweekly.com/Articles/26/10/2011/52133/fpgas-are-ready-to-take-on-asics-says-xilinx-cto.htm, Abruf: 26.10.2011

[ELS09] Erdem, E. (Hrsg.) ; Lin, F. (Hrsg.) ; Schaub, T. (Hrsg.): *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. Bd. *5753*. Springer-Verlag, 2009 (Lecture Notes in Artificial Intelligence)

[Gel08] GELFOND, M.: Answer Sets. In: [LvP08], Kapitel 7, S. 285–316

[GKK+08] GEBSER, Martin ; KAMINSKI, Roland ; KAUFMANN, Benjamin ; OSTROWSKI, Max ; THIELE, Sven ; SCHAUB, Torsten: *A User's Guide to gringo, clasp, clingo, and iclingo*, November 2008

[GKK+11] GEBSER, M. ; KAMINSKI, R. ; KAUFMANN, B. ; OSTROWSKI, M. ; SCHAUB, T. ; SCHNEIDER, M.: Potassco: The Potsdam Answer Set Solving Collection. In: *AI Communication* 24 (2011), Nr. 2, S. 105–124

[Goo] GOODACRE, John: *The Effect and Technique of System Coherence in ARM Multicore Technology.* http://www.mpsoc-forum.org/2008/slides/8-6%20Goodacre.pdf, Abruf: 26.10.2011

[HFG+] HAUBELT, Christian ; FALK, Joachim ; GLASS, Michael ; LUKASIEWYCZ, Martin ; STREUBÜHR, Martin ; TEICH, Jürgen: *SystemCoDesigner.* http://www.tik.ee.ethz.ch/pisa/variators/scd/documentation.html, Abruf: 10.02.2011

[HHD99] HORPRASERT, T ; HARWOOD, D ; DAVIS, L S.: A statistical approach for real-time robust background subtraction and shadow detection. In: *IEEE ICCV* Bd. 99, 1999, S. 1–19

[HMMB10] HARTMANN, Robert ; MACHOT, Fadi A. ; MAHR, Philipp ; BOBDA, Christophe: Camera-based system for tracking and position estimation of humans. In: *DASIP*, 2010, S. 62–67

[Hth] *HybridThreads (hthreads).* http://hthreads.csce.uark.edu/wiki/About_Hthreads, Abruf: 17.02.2011

[HUN] *HUNT ENGINEERING Image Processing VHDL modules – Hunt Library.* http://www.hunteng.co.uk/products/ip/imaging_ip.htm, Abruf: 10.02.2011

[IB09] ISHEBABI, Harold ; BOBDA, Christophe: Automated architecture synthesis for parallel programs on FPGA multiprocessor systems. In: *Microprocess. Microsyst.* 33 (2009), Nr. 1, S. 63–71. – ISSN 0141–9331

[Ish09] ISHEBABI, Harold: *Architecture Synthesis for Adaptive Multiprocessor Systems on Chip*, University of Potsdam, Diss., 2009

[Jan06] JANHUNEN, T.: Some (in)translatability results for normal logic programs and propositional theories. In: *Journal of Applied Non-Classical Logics* 16 (2006), Nr. 1–2, S. 35–86

[Keg] KEGEL, Dan: *Building and Testing gcc/glibc cross toolchains.* http://kegel.com/crosstool, Abruf: 17.02.2011

[Kel] KELLER, David: *Stochastische Verfahren und Evolutionäre Algorithmen – Parametrisierungkonzepte für Evolutionäre Algorithmen.* https://www1.ethz.ch/structures/education/master/intro/compulsory/optimization/FS2011PDFs/ea_2011_V11.pdf, Abruf: 06.06.2011

[ker] *The Linux Kernel Archives.* http://www.kernel.org, Abruf: 17.02.2011

[LCC+06] LEESER, Miriam ; CONTI, Al ; CORDES, Ben ; MOORE, Nicholas ; KING, Laurie S.: VSIPL++ Support for Programming Reconfigurable Supercomputers. In: *Reconfigurable Systems Summer Institute (RSSI 2006) Conference.* Urbana, IL, 7 2006

[Lina] LINUXDEVICES.COM: *Linux still top embedded OS.* http://www.linuxfordevices.com/c/a/News/Linux-still-top-embedded-OS/, Abruf: 10.02.2011

[Linb] LINUXDEVICES.COM: *Snapshot of the embedded Linux market April 2007.* http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Snapshot-of-the-embedded-Linux-market-April-2007, Abruf: 10.02.2011

[LR06] LIFSCHITZ, V. ; RAZBOROV, A.: Why Are There So Many Loop Formulas? In: *ACM Transactions on Computational Logic* 7 (2006), Nr. 2, S. 261–268

[LvP08] LIFSCHITZ, V. (Hrsg.) ; VAN HERMELEN, F. (Hrsg.) ; PORTER, B. (Hrsg.): *Handbook of Knowledge Representation.* Elsevier, 2008

[Lü10] LÜBBERS, Enno: *Multithreaded Programming and Execution Models for Reconfigurable Hardware*, University of Paderborn, Diss., 2010

[MCLK07] MOORE, Nicholas ; CONTI, Albert ; LEESER, Miriam ; KING, Laurie S.: Vforce: An Extensible Framework for Reconfigurable Supercomputing. In: *Computer* 40 (2007), March, S. 39–49

[OCV] *Open Source Computer Vision Library.* http://www.intel.com/research/mrl/research/opencv/, Abruf: 10.02.2011

[OEM] *OpenEmbedded user manual.* http://docs.openembedded.org/usermanual/usermanual.html, Abruf: 10.02.2011

[Ots79] OTSU, Nobuyuki: A Threshold Selection Method from Gray-Level Histograms. In: *IEEE Transactions on Systems, Man and Cybernetics* Bd. 9, 1979, S. 62–66

[Pel] PELLERIN, David: *Streams-Based Programming accelerates FPGA-Based Embedded Applications.* http://www.xilinx.com/publications/magazines/emb_04/xc_pdf/p29-31_4emb-pellerin.pdf, Abruf: 10.02.2011

[pis] *PISA - A Platform and Programming Language Independent Interface for Search Algorithms.* http://www.tik.ee.ethz.ch/pisa, Abruf: 10.02.2011

[POT] *Potassco - Tools for Answer Set Programming.* http://potassco.sourceforge.net, Abruf: 10.02.2011

[PPC] *PenguinePPC: the home of the linux/ppc port.* http://penguinppc.org/, Abruf: 10.02.2011

[PTI] *Adaptive Multiprocessor on-Chip (AMoC) Project.* http://www.cs.uni-potsdam.de/techinf/projects/amoc/index.html, Abruf: 17.02.2011. – Universität Potsdam, Technische Informatik

[RC01] RUBINI, Alessandro ; CORBET, Jonathan: *Linux Device Drivers.* 2nd Ed. O'Reilly Associates, 2001

[Rec07] RECH, Lucio Otavio M.: *A System on Chip for Image Processing using OpenCV*, Universidade Federal do Rio Grande do Sul und Technische Universität Kaiserslautern, Diplomarbeit, 2007. – Betreuung: Prof. Dr. Flavio Rech Wagner, Prof. Dr. Christophe Bobda, Felix Mühlbauer

[RvW06] ROSSI, F. (Hrsg.) ; VAN BEEK, P. (Hrsg.) ; WALSH, T. (Hrsg.): *Handbook of Constraint Programming.* Elsevier, 2006

[Sch95] SCHLIPF, J.: The expressive powers of the logic programming semantics. In: *Journal of Computer and System Sciences* 51 (1995), S. 64–86

[SM10] SMARI, Waleed W. (Hrsg.) ; MCINTIRE, John P. (Hrsg.): *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010, June 28 - July 2, 2010, Caen, France.* IEEE, 2010

[SUI] *SUIF (Stanford University Intermediate Format) Compiler.* http://suif.stanford.edu/suif, Abruf: 10.02.2011

[TH07] TEICH, Jürgen ; HAUBELT, Christian: *Digitale Hardware/Software-Systeme: Synthese und Optimierung.* Springer, Berlin, 2007

[Ven08] VENKATESWARAN, Sreekrishnan: *Essential Linux Device Drivers (Prentice Hall Open Source Software Development).* 1. Prentice Hall International, 2008

[VG02] VAHID, Frank ; GIVARGIS, T.: *Embedded system design: a unified hardware/-software introduction.* Wiley, 2002. – ISBN 9780471386780

[VSIa] *Vector Signal Image Processing Library.* http://www.vsipl.org, Abruf: 01.10.2011

[VSIb] *VSIPL++ Interface For FPGAs.* http://www.ece.neu.edu/groups/rcl/projects/vsipl/vsipl.html, Abruf: 17.02.2011

[Wan98] WANNEMACHER, Markus: *Das FPGA-Kochbuch.* Bonn : International Thomson Publishing GmbH, 1998

[Zar11] ZAREZADEH, Ali A.: *Enabling Communication Infrastructure on Embedded Distributed Smart Cameras*, University of Potsdam, Diss., 2011

[ZB10a] ZAREZADEH, Ali A. ; BOBDA, Christophe: Hardware ORB Middleware for Distributed Smart Camera Systems. In: *ERSA*, 2010, S. 104–116

[ZB10b] ZAREZADEH, Ali A. ; BOBDA, Christophe: Performance analysis of Hardware/Software Middleware in Network of Smart Camera Systems. In: *ReConfig10: International Conference on ReConFigurable Computing and FPGAs*, 2010

[ZB11a] ZAREZADEH, Ali A. ; BOBDA, Christophe: Enabling Communication Infrastructure and Protocol on Embedded Distributed Smart Cameras. In: *ICDSC 2011 - Fifth ACM/IEEE International Conference on Distributed Smart Cameras.* Ghent, Belgium : Institute of Electrical and Electronics Engineers, 2011

[ZB11b] ZAREZADEH, Ali A. ; BOBDA, Christophe: Probabilistic Framework for Person Tracking on Embedded Distributed Smart Cameras. In: *ICDSC 2011 - Fifth ACM/IEEE International Conference on Distributed Smart Cameras.* Ghent, Belgium : Institute of Electrical and Electronics Engineers, 2011

Veröffentlichungen

[1] Bobda, Christophe ; Cheng, Kevin ; Mühlbauer, Felix ; Drechsler, Klaus ; Schulte, Jan ; Murr, Dominik ; Tanougast, Camel: Enabling Self-Organization in Embedded Systems with Reconfigurable Hardware. In: *International Journal of Reconfigurable Computing* 2009 (2009), S. 9. – DOI 10.1155/2009/161458

[2] Bobda, Christophe ; Haller, Thomas ; Jung, Simon ; Rech, Dennis ; Mühlbauer, Felix: Design of Adaptive Multiprocessor on Chip Systems. In: *SBCCI conference*, 2007

[3] Bobda, Christophe ; Zarezadeh, Ali A. ; Mühlbauer, Felix ; Hartmann, Robert ; Cheng, Kevin: Reconfigurable Architecture for Distributed Smart Cameras. In: *ERSA - Engineering of Reconfigurable Systems and Algorithms*. Las Vegas, Nevada, USA, July 2010

[4] Chati, Harding D. ; Mühlbauer, Felix ; Braun, Tim ; Bobda, Christophe ; Berns, Karsten: Hardware/Software co-design of a key point detector on FPGA. In: *FCCM*, 2007

[5] Chati, Harding D. ; Mühlbauer, Felix ; Braun, Tim ; Bobda, Christophe ; Berns, Karsten: SoPC architecture for a Key Point Detector. In: *FPL*, 2007

[6] Cheng, Kevin ; Zarezadeh, Ali A. ; Mühlbauer, Felix ; Tanougast, Camel ; Bobda, Christophe: Auto-Reconfiguration on Self-Organized Intelligent Platform. In: *NASA/ESA AHS'2010 Conference*. Anaheim, California, June 2010

[7] Middendorf, Lars ; Mühlbauer, Felix ; Umlauf, Georg ; Bobda, Christophe: Embedded Vertex Shader in FPGA. In: *Embedded System Design: Topics, Techniques and Trends, IFIP International Federation for Information Processing* Bd. 231, Springer Boston, 2007, S. 155–164

[8] Middendorf, Lars ; Mühlbauer, Felix ; Umlauf, Georg ; Bobda, Christophe: SoPC for 3D Point Rendering. In: *DASIP*, 2007

[9] Murr, Dominik ; Mühlbauer, Felix ; Dressler, Falko ; Bobda, Christophe: Utilizing Reconfigurable Hardware to optimize Workflows in Networked Nodes. In: *Embedded System Design: Topics, Techniques and Trends, IFIP International Federation for Information Processing* Bd. 231, Springer Boston, 2007, S. 373–386

[10] Mühlbauer, Felix ; Bobda, Christophe: Design and Implementation of an Object Tracker on a Reconfigurable System on Chip. In: *17th IEEE International Workshop On Rapid System Prototyping (RSP'06)*. Chania, Crete, 2006

[11] Mühlbauer, Felix ; Bobda, Christophe: A Dynamic Reconfigurable Hardware/-Software Architecture for Object Tracking in Video Streams. In: *EURASIP Journal on Embedded Systems* 2006 (2006), S. 8. – DOI 10.1155/ES/2006/82564

[12] Mühlbauer, Felix ; Frohberg, Max ; Bobda, Christophe: SoCar: System-on-a-Chip Car. In: *University Booth of Design, Automation & Test in Europe 2009 (DATE'09)*. Nice, France, April 2009

[13] Mühlbauer, Felix ; Grosshans, Michael ; Bobda, Christophe: Rapid Prototyping of OpenCV Image Processing Applications using ASP. In: *22th IEEE International Workshop On Rapid System Prototyping (RSP'11)*. Karlsruhe, Germany, 2011

[14] Mühlbauer, Felix ; Rech, Lucio O. M. ; Bobda, Christophe: Hardware Accelerated OpenCV on Systems on Chip. In: *Reconfigurable Communication-centric Systems-on-Chip Workshop 2008*. Barcelona, Spain, July 2008

[15] Zarezadeh, Ali A. ; Mühlbauer, Felix ; Cheng, Kevin ; Bobda, Christophe: Potsdam Intelligent Camera System (PICSY) – FPGA-based High Performance Smart Camera. In: *Workshop on Smart Cameras and Real-Time Image Processing* Humboldt-Universität Berlin, 2009

A ASP Modellierung

Die Modellierung ist in drei Teile gekapselt. Durch die Aufteilung ist die Modellierung einfach für verschiedene Anwendungsfälle wiederverwendbar:

Problembeschreibung: Problemgraph und Rahmenbedingungen für die gesuchte Architektur, wie z. B. Chipfläche (siehe Listing 1)

Metainformationen: Parameter der Implementierungen und der Kommunikationsmodule (siehe Listing 2)

Antwortmengenprogramm: Regeln zur Lösung des Problems (siehe Listing 3).

Listing 1: ASP: Beispielproblem `problem.lp`

```
%%% taskgraph

% tasks
node(0,start).
node(1,togray).
node(2,otsu).
node(3,scan).
node(4,end).
% communication: from, to, amount of data in/out
edge(6,0,1,1).
edge(7,1,2,1).
edge(8,1,3,1).
edge(10,2,4,0).
edge(12,3,4,0).
wait_for(3,2). % task 3 has to wait for result of task 2

%%% ressource constraints
place(slices,40).
place(hc_proc,1).
place(bram,8).

%%% guessed size of domains
dom(ppc,1..1).
dom(pu_otsu,1..1).
```

```
dom(pu_scan,1..1).
dom(pu_togray,1..1).
dom(plb_bus,1..1).
dom(ddr,1..1).
dom(sdi_c,1..3).
```

Listing 2: ASP: Komponenten und Metainformationen `components.lp`

```
%%% components
component(ppc).
component(pu_a).
component(pu_b).
component(pu_c).
component(pu_d).
component(pu_e).
component(pu_otsu).
component(pu_scan).
component(pu_togray).
component(pu_gauss).
component(pu_sobel).
component(pu_gradient).
communicate(sdi_c).
communicate(plb_bus).
communicate(ddr).

%%% bus components
group(ppc,bus_con).
group(ddr,bus_con).
group(sdi_c,bus_con).
group(plb_bus, bus).

%%% processing units
group(pu_a,sdi).
group(pu_b,sdi).
group(pu_c,sdi).
group(pu_d,sdi).
```

```
group(pu_e,sdi).
group(pu_otsu,sdi).
group(pu_scan,sdi).
group(pu_togray,sdi).
group(pu_gauss,sdi).
group(pu_sobel,sdi).
group(pu_gradient,sdi).

%%% component implements task + throughput
implement(ppc,a,16).
implement(ppc,b,8).
implement(ppc,c,8).
implement(ppc,start,0).
implement(ppc,end,0).
implement(pu_a,a,2).
implement(pu_b,b,2).
implement(pu_c,c,2).
implement(pu_d,d,2).
implement(pu_e,e,2).
implement(ppc,otsu,16).
implement(ppc,scan,4).
implement(ppc,togray,16).
implement(pu_otsu,otsu,1).
implement(pu_scan,scan,1).
implement(pu_togray,togray,1).
implement(pu_gauss,gauss, 1).
implement(ppc,gauss, 16).
implement(pu_sobel,sobel, 1).
implement(ppc,sobel, 16).
implement(pu_gradient,gradient, 2).
implement(ppc,gradient, 8).
implement(ppc,hysteresis, 16).

%%% components which change the data rate of the stream:
% data_rate(name,m,n) => d_out = d_in * n / m
```

```
data_rate(a,1,2).
data_rate(b,1,1).
data_rate(c,1,0).
data_rate(d,1,1).
data_rate(e,1,0).
data_rate(otsu,1,0).
data_rate(scan,1,0).
data_rate(togray,1,1).
data_rate(sobel,1,2).
data_rate(gradient,2,1).
data_rate(hysteresis,1,1).
data_rate(gauss,1,1).

% connections between groups of components
connect(X,Y) :- group(X,sdi), group(Y,sdi).
connect(sdi_c, X) :- group(X,sdi).
connect(X, sdi_c) :- group(X,sdi).
connect(X,plb_bus) :- group(X, bus_con).
connect(plb_bus,X) :- group(X, bus_con).

% throughput
tp(plb_bus, 1).

%%% resource usage

% resources
resource(bram).
resource(hc_proc).
resource(slices).

uses(ppc,hc_proc,1).
uses(pu_a,slices,3).
uses(pu_b,slices,7).
uses(pu_c,slices,2).
uses(pu_d,slices,3).
```

```
uses(pu_e,slices,3).
uses(pu_otsu,slices,10).
uses(pu_otsu,bram,1).
uses(pu_scan,slices,1).
uses(pu_togray,slices,1).
uses(pu_togray,bram,2).
uses(pu_gauss,slices,2).
uses(pu_gauss,bram,2).
uses(pu_sobel,slices,2).
uses(pu_sobel,bram,2).
uses(pu_gradient,slices,1).
uses(sdi_c, slices, 3).
use_dom(C) :- uses(X,Y,C).

% if no explicit usage is give then 0 is assumed
size(Y,X,C):- uses(Y,X,C).
size(Y,X,0):- resource(X), component(Y),    [uses(Y,X,C) : use_dom(C ↩
    ) : C > 0] 0.
size(Y,X,0):- resource(X), communicate(Y), [uses(Y,X,C) : use_dom(C ↩
    ) : C > 0] 0.

% resource usage of certain components
use(plb,1,map,slices,10).

% port count constraints
wired(C1,K1,C2,K2) :- send(N,C1,K1,C2,K2).
wired(C1,K1,C2,K2) :- load(N,C1,K1,C2,K2).

% port count not equal to one
non_one_port(pu_d,1,2).
non_one_port(pu_e,2,0).
non_one_port(pu_c,1,0).
non_one_port(pu_otsu,1,0).
non_one_port(pu_scan,1,0).
```

```
ports(X,1,1) :- component(X), [non_one_port(X,T1,T2)]0.
ports(X,I,O) :- non_one_port(X,I,O).

% SDI controller
:- group(C,sdi), dom(C,K), N+1 [ wired(C,K,C1,K1) : dom(C1,K1) ], ↩
   ports(C,T,N), 1 [map(X,C,K)].
:- group(C,sdi), dom(C,K), [ wired(C,K,C1,K1) : dom(C1,K1) ] N-1, ↩
   ports(C,T,N), 1 [map(X,C,K)].
:- group(C,sdi), dom(C,K), N+1 [ wired(C1,K1,C,K) : dom(C1,K1) ], ↩
   ports(C,N,T), 1 [map(X,C,K)].
:- group(C,sdi), dom(C,K), [ wired(C1,K1,C,K) : dom(C1,K1) ] N-1, ↩
   ports(C,N,T), 1 [map(X,C,K)].

:- dom(sdi_c,K), 2 [wired(sdi_c,K,C1,K1):dom(C1,K1):C1!=plb_bus].
:- dom(sdi_c,K), 2 [wired(C1,K1,sdi_c,K):dom(C1,K1):C1!=plb_bus].
```

Listing 3: ASP: Solver-Logik `solver.lp`

```
#const time = 100.
#const tp = 64.
#const split = 8.

%%% time & throughput
time(0..time).
tp_dom(1).
tp_dom(Y) :- tp_dom(X), Y = 2*X, Y <= tp.
max_dom(tp).
mod_tp(split).

%%% choose desing parameters
1 [map(N,C,K) : implement(C,O,TP) : dom(C,K)] 1 :- node(N,O).
1 [map(N,T) : time(T)] 1 :- node(N,O).

1 [send_at(N,T) : time(T)] 1 :- edge(N,I,O,D), 1 [ send(N,CI,KI,CO, ↩
   KO) ].
```

```
1 [load_at(N,T) : time(T)] 1 :- edge(N,I,O,D), load(N,plb_bus,Kb, ↩
    sdi_c,Kc).

0 [send(N,C1,K1,C2,K2)] 1 :- dom(C1,K1), dom(C2,K2), connect(C1,C2) ↩
    , edge(N,I,O,Dummy), Dummy != 0.
0 [load(N,C1,K1,C2,K2)] 1 :- dom(C1,K1), dom(C2,K2), connect(C1,C2) ↩
    , edge(N,I,O,Dummy), Dummy != 0.

% task on processor needs memory
1 [ ext_mem(N,K) : dom(ddr,K)] 1 :- node(N,O), map(N,ppc,Kp).

% search data flow parameters for PUs
1 [ work_tp(N,Tp) : tp_dom(Tp)] 1:- node(N,O), O != start, O != end ↩
    .
1 [ send_tp(N,Tp) : tp_dom(Tp)] 1:- edge(N,I,O,D), D != 0, send(N, ↩
    sdi_c,Kc,plb_bus,Kb).
1 [ load_tp(N,Tp) : tp_dom(Tp)] 1:- edge(N,I,O,D), D != 0, load(N, ↩
    plb_bus,Kb,sdi_c,Kc).

% connect components
con(N,C1,K1,C2,K2) :- send(N,C1,K1,C2,K2).
con(N,C1,K1,C2,K2) :- load(N,C1,K1,C2,K2).
con(N,C1,K1,C3,K3) :- send(N,C1,K1,C2,K2), con(N,C2,K2,C3,K3).
con(N,C1,K1,C3,K3) :- load(N,C1,K1,C2,K2), con(N,C2,K2,C3,K3).

sdi_con(C1,K1,C2,K2) :- load(N,C1,K1,C2,K2), C1 = sdi_c.
sdi_con(C1,K1,C3,K3) :- send(N,C2,K2,C3,K3), C2 != plb_bus, sdi_con ↩
    (C1,K1,C2,K2).

% use same SDI controller for a PU chain
:- dom(sdi_c,K), 1 [sdi_con(sdi_c,K,sdi_c,K1) : dom(sdi_c,K1): K1 ↩
    != K], not sdi_con(sdi_c,K,sdi_c,K).

% helper elements
map(N,C,K,T) :- map(N,C,K), map(N,T).
```

```
real_tp(N,C,K,T) :- map(N,C,K), work_tp(N,T).
bus_tp(N,C1,K1,plb_bus,K2,T) :- send_tp(N,T), send(N,C1,K1,plb_bus, ↩
    K2).
bus_tp(N,C1,K1,plb_bus,K2,T) :- load_tp(N,T), load(N,C1,K1,plb_bus, ↩
    K2).
transfer(N,C,K,C1,K1,T) :- send(N,C,K,C1,K1), send_at(N,T).
transfer(N,C,K,C1,K1,T) :- load(N,C,K,C1,K1), load_at(N,T).

temp_store(E) :- edge(E,I,O,D), 1 [load(E,Cb,Kb,Cd,Kd)], 1[send(E, ↩
    Cb,Kb,Cd,Kd)].

use(C,K,map,R,U)    :- size(C,R,U), dom(C,K), 1 [map(N,C,K) : node(N ↩
    ,Op)].
use(sdi_c,K,sdi,R,U) :- size(sdi_c,R,U), dom(sdi_c,K), 1 [send(N, ↩
    sdi_c,K,C1,K1) : dom(C1,K1)].
use(sdi_c,K,sdi,R,U) :- size(sdi_c,R,U), dom(sdi_c,K), 1 [load(N, ↩
    sdi_c,K,C1,K1) : dom(C1,K1)].

utilization(R,U) :- resource(R), U = [ use(C,K,T,R,U1) = U1].

%%% force connections
% ppc <> ppc: RAMs connected?
:- edge(N,I,O,Dummy), ext_mem(I,M1), ext_mem(O,M2), M1 != M2, not ↩
    con(N,ddr,M1,ddr,M2), Dummy != 0.
% ppc > other
:- edge(N,I,O,Dummy), ext_mem(I,M), C1 != ppc, map(O,C1,K1), not ↩
    con(N,ddr,M,C1,K1), Dummy != 0.
% other > ppc
:- edge(N,I,O,Dummy), ext_mem(O,M), C1 != ppc, map(I,C1,K1), not ↩
    con(N,C1,K1,ddr,M), Dummy != 0.
% other > other: real connection
:- edge(N,I,O,Dummy), dom(C2,K2), dom(C1,K1), C1 != ppc, C2 != ppc, ↩
     map(I,C1,K1), map(O,C2,K2), not con(N,C1,K1,C2,K2), Dummy != 0.

%%% permit sensless connections
```

```
% are only allowed in certain cases
:- edge(N,I,O,Dummy), dom(C1,K1), 1 [ con(N,C2,K2,C1,K1) ], [ con(N ↩
    ,C1,K1,C2,K2) ] 0, not map(O,C1,K1), C1 != ddr.
:- edge(N,I,O,Dummy), dom(C1,K1), 1 [ con(N,C2,K2,C1,K1) ], [ con(N ↩
    ,C1,K1,C2,K2) ] 0, not ext_mem(O,K1) , C1 = ddr.

:- edge(N,I,O,Dummy), dom(C2,K2), 1 [ con(N,C2,K2,C1,K1) ], [ con(N ↩
    ,C1,K1,C2,K2) ] 0, not map(I,C2,K2), C2 != ddr.
:- edge(N,I,O,Dummy), dom(C2,K2), 1 [ con(N,C2,K2,C1,K1) ], [ con(N ↩
    ,C1,K1,C2,K2) ] 0, not ext_mem(I,K2) , C2 = ddr.

% to or from PPC
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(ddr,M), ext_mem(O ↩
    ,M), con(N,C1,K1,C2,K2), not con(N,C2,K2,ddr,M), C2 != ddr.
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(ddr,M), ext_mem(O ↩
    ,M), con(N,C1,K1,C2,K2), not con(N,C2,K2,ddr,M), K2 != M.
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(ddr,M), ext_mem(I ↩
    ,M), con(N,C2,K2,C1,K1), not con(N,ddr,M,C2,K2), C2 != ddr.
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(ddr,M), ext_mem(I ↩
    ,M), con(N,C2,K2,C1,K1), not con(N,ddr,M,C2,K2), K2 != M.

% to or from SDI
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(C,K), map(O,C,K, ↩
    T), con(N,C1,K1,C2,K2), not con(N,C2,K2,C,K), C2 != C, C != ppc.
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(C,K), map(O,C,K, ↩
    T), con(N,C1,K1,C2,K2), not con(N,C2,K2,C,K), K2 != K, C != ppc.
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(C,K), map(I,C,K, ↩
    T), con(N,C2,K2,C1,K1), not con(N,C,K,C2,K2), C2 != C, C != ppc.
:- edge(N,I,O,Dummy), dom(C1,K1), dom(C2,K2), dom(C,K), map(I,C,K, ↩
    T), con(N,C2,K2,C1,K1), not con(N,C,K,C2,K2), K2 != K, C != ppc.

%%% send and receive
% use port only once
:- edge(N,I,O,Dummy), send(N,C1,K1,C2,K2), load(N,C1,K1,C2,K2).
```

```
% no simultanious transfer between two components
:- dom(C1,K1), dom(C2,K2), send(N,C1,K1,C2,K2), send(N,C2,K2,C1,K1) ↩
    , edge(N,I,O,D).
:- dom(C1,K1), dom(C2,K2), load(N,C1,K1,C2,K2), load(N,C2,K2,C1,K1) ↩
    , edge(N,I,O,D).

% components
:- edge(N,I,O,Dummy), dom(C,K), component(C), send(N,C,K,C1,K1), ↩
    not map(I,C,K).
:- edge(N,I,O,Dummy), dom(C,K), component(C), load(N,C,K,C1,K1), ↩
    not map(I,C,K).
:- edge(N,I,O,Dummy), dom(C,K), component(C), send(N,C1,K1,C,K), ↩
    not map(O,C,K).
:- edge(N,I,O,Dummy), dom(C,K), component(C), load(N,C1,K1,C,K), ↩
    not map(O,C,K).

% PPC
:- 1 [ send(N,C,K,ppc,Kp) ].
:- 1 [ load(N,C,K,ppc,Kp) ].

% RAM
:- 1 [ send(N,ddr,K,plb_bus,Kb) ].
:- 1 [ load(N,plb_bus,Kb,ddr,K) ].

% communication without RAM
:- dom(C,K), communicate(C), C != ddr, 1 [ send(N,C1,K1,C,K) ], [ ↩
    send(N,C,K,C2,K2) ] 0, edge(N,I,O,D).
:- dom(C,K), communicate(C), C != ddr, 1 [ load(N,C1,K1,C,K) ], [ ↩
    load(N,C,K,C2,K2) ] 0, edge(N,I,O,D).

% SDI controller
:- 1 [ send(N,plb_bus,Kb,sdi_c,K) ].
:- 1 [ load(N,sdi_c,K,plb_bus,Kb) ].

:- 1 [ load(N,C,Kb,sdi_c,K) ], group(C,sdi), component(C).
```

```
:- 1 [ send(N,sdi_c,K,C,Kb) ], group(C,sdi), component(C).

:- send(N,sdi_c,Kc,plb_bus,Kb), [ send(N,C,K,sdi_c,Kc) ] 0.
:- load(N,sdi_c,Kc,C,K), [ load(N,plb_bus,Kb,sdi_c,Kc) ] 0.

% SDI, only one direction
:- edge(N,I,O,D), map(O,C,K), 2 [ send(N,C1,K1,C,K) ].
:- edge(N,I,O,D), map(O,C,K), 2 [ load(N,C1,K1,C,K) ].
:- edge(N,I,O,D), map(O,C,K), 1 [ send(N,C1,K1,C,K) ], 1 [ load(N, ↩
   C1,K1,C,K) ].

% loads only while accessing RAM
:- load(N,C1,K1,C2,K2), [load(N, ddr, Kd, plb_bus, Kb)] 0.

%%% delete invalid throughput
% selected throughput is not better than given one
:- node(N,Op), dom(C,K), map(N,C,K), 1 [ work_tp(N,Rtp) : tp_dom( ↩
   Rtp) : Rtp < Tp], implement(C,Op,Tp).

% SDI -> SDI
:- edge(N,I,O,Dummy), group(CI,sdi), group(CO,sdi), map(I,CI,KI), ↩
   map(O,CO,KO), send(N,CI,KI,CO,KO), work_tp(I,T1), work_tp(O,T2), ↩
    node(I,Op), data_rate(Op, Din, Dout), Cin = T1 * Din, Cout = T2 ↩
    * Dout, Cin != Cout.

% SDI -> bus
:- edge(N,I,O,Dummy), dom(CI,KI), group(CI,sdi), dom(sdi_c,KO), map ↩
   (I,CI,KI), send(N,CI,KI,sdi_c,KO), work_tp(I,T1), send_tp(N,T2), ↩
    node(I,Op), data_rate(Op, Din, Dout), Cin = T1 * Din, Cout = T2 ↩
    * Dout, Cin != Cout, Dout != 0.

% bus -> SDI
:- edge(E,I,O,Data), work_tp(O,TpO), load_tp(E,Tp), TpO != Tp.

%%% transfer timing
```

```
% task dependancies
dep(X,Y) :- node(X,O1), node(Y,O2), edge(K,X,Y,Dummy).
dep(X,Z) :- node(X,O1), node(Y,O2), node(Z,O3), edge(K,X,Y,Dummy), ↩
    dep(Y,Z).

% task execution order
:- map(X,TX), dep(X,Y), 1 [map(Y,TY) : TY < TX].

% set start task to 0
:- node(X,start), 1 [ map(X,TX) : TX > 0 ].

% balance transfer (in - out)
:- map(X,CX,KX,TX), CX != ppc, edge(N,X,Y,DN), 1 [send_at(N,TN) : ↩
    time(TN): TN != TX].
:- map(X,CX,KX,TX), CX != ppc, load(N,ddr,Kd,plb_bus,Kb), edge(N,Y, ↩
    X,DN), 1 [load_at(N,TN) : time(TN): TN != TX].
:- map(X,CX,KX,TX), CX != ppc, [ load(N,ddr,Kd,plb_bus,Kb) ] 0, ↩
    edge(N,Y,X,DN), 1 [send_at(N,TN) : time(TN): TN != TX].
:- map(X,ppc,KX,TX), edge(N,X,Y,DN), CX = ppc, load(N,ddr,K,pl_bus, ↩
    KY) , 1 [ load_at(N,TN) : time(TN): TN < TX], ext_mem(X,K).
:- map(X,ppc,KX,TX), edge(N,Y,X,DN), CX = ppc, send(N,plb_bus,KY, ↩
    ddr,K), 1 [ send_at(N,TN) : time(TN): TN > TX], ext_mem(X,K).

%%% local scheduling
task_at_time(N,T) :- map(N,Time), work_tp(N,Tp), edge(E,I,N,Data), ↩
    time(T), T >= Time, T < (Time + (Tp * Data)).
task_at_time(N,0) :- node(N,start).

:- dom(C,K), task_at_time(N,T), map(N,C,K), 1 [map(N2,C,K,T) : node ↩
    (N2,Op) : N2 != N].

% end time must be later
:- edge(E,I,O,D), time(T), task_at_time(I,T), not task_at_time(O,T) ↩
    , 1 [task_at_time(O,T1) : T1 < T].
```

```
:- edge(E,I,O,D), map(O,ppc,K), task_at_time(I,T), task_at_time(O,T ↩
    ).

%%% bus sharing
% incomming traffic: bus_tp * data rate + local load
% model speed up: group <split> slots together to reduce search ↩
    space

work_div(local,N,K,T,W1):- task_at_time(N,T), map(N,ppc,K), work_tp ↩
    (N,Tp), W= tp / Tp, W1 = W / split.
work_mod(local,N,K,T,W1):- task_at_time(N,T), map(N,ppc,K), work_tp ↩
    (N,Tp), W= tp / Tp, W1 = W #mod split.
work_div(write,sdi,E,T,W1):- edge(E,I,O,Data), task_at_time(I,T), ↩
    tp_dom(Tp), send_tp(E,Tp), W= tp / Tp, W1 = W / split.
work_mod(write,sdi,E,T,W1):- edge(E,I,O,Data), task_at_time(I,T), ↩
    tp_dom(Tp), send_tp(E,Tp), W= tp / Tp, W1 = W #mod split.
work_div(load,ddr,E,T,W1):- edge(E,I,O,Data), task_at_time(O,T), ↩
    tp_dom(Tp), load_tp(E,Tp), W= tp / Tp, W1 = W / split.
work_mod(load,ddr,E,T,W1):- edge(E,I,O,Data), task_at_time(O,T), ↩
    tp_dom(Tp), load_tp(E,Tp), W= tp / Tp, W1 = W #mod split.

% check load of certain time slot
div_tp_dom(0..X) :- max_dom(X1), mod_tp(X2), X = X1 / X2.
komp_tp_dom(0..W) :- tp_dom(W).
work(div,T,W) :- time(T), div_tp_dom(W), W= [work_div(X,N,K,T,W1) ↩
    = W1].
work(mod,T,W) :- time(T), komp_tp_dom(W2), W2 = [work_mod(X,N,K,T, ↩
    W1) = W1], W = W2 #mod split.
work(car,T,W) :- time(T), komp_tp_dom(W2), W2 = [work_mod(X,N,K,T, ↩
    W1) = W1], W = W2 / split.

work_kom(T,W) :- time(T), div_tp_dom(W), W= [work(X,T,W1) = W1 : X ↩
    != mod].

:- time(T), [work_kom(T,W)] 0.
```

```
:- time(T), [work(mod,T,W)] 0.
:- time(T), [work(div,T,W)] 0.
:- time(T), [work(car,T,W)] 0.

:- tp(plb_bus,TP), time(T), work_kom(T,W), W * TP * split > tp.
:- tp(plb_bus,TP), time(T), work_kom(T,W), W * TP * split == tp, 1 ↩
   [work(mod,T,W1) : W1 != 0].

%%% wait for components
:- wait_for(X,Y), task_at_time(X,T), task_at_time(Y,T).
:- wait_for(X,Y), map(X,T), 1 [map(Y,TY) : TY > T].

% falls I auf PC, dann O erst danach!
:- edge(E,I,O,D), map(I,ppc,K), task_at_time(I,T), task_at_time(O,T ↩
   ).

%%% area constraints
:- 1 [utilization(R,U1) : U1 > U2], place(R,U2).

%%% minimize and output variables
:- task_at_time(X,T), node(N,end), 1 [map(N,TY) : TY <= T].

#minimize[map(N,T) = T : node(N,end)].

#hide.
#show map/4.
#show transfer/6.
#show real_tp/4.
#show bus_tp/6.
#show ext_mem/2.
#show node/2.
#show edge/4.
#show max_dom/1.
#show transfer_work/3.
#show local_work/3.
```

```
#show use/5.
#show utilization/2.
#show wait_for/2.
#show group/2.
#show port/5.
```

B SDI-Kontroller

Einige Details zu den Software-Registern des SDI-Kontrollers.

Index	Register	Zugriff	Funktion
0	hardware register	R	Versionsnummer, Magic Word (0xABCD)
1	status register	R	Statusflags
	control register	W	Kontrollregister
2	reader start address	R/W	Speicheradressen für den Reader
3	reader end address	R/W	
4	reader trigger address	R/W	optional zur Generierung von Interrupts
5	writer start address	R/W	Speicheradressen für den Writer
6	writer end address	R/W	
7	writer trigger address	R/W	optional zur Generierung von Interrupts
8	tick counter	R	Steuerbarer Taktzähler für Messungen
9	PU status register 0	R	Benutzerdefiniertes Statusregister
	PU control register 0	W	Benutzerdefiniertes Steuerregister
10	PU status register 1	R	⋮
	PU control register 1	W	
	...		

Tabelle 4: Softwareregister zur Steuerung des SDI-Kontrollers.

Bit	Funktion
0	Interrupt: Transfer abgeschlossen (Reader)
1	Reader ist aktiv
2	Interrupt: Transfer abgeschlossen (Writer)
3	Writer ist aktiv
4	Interrupt: FIFO ist leer (Reader)
5	Interrupt: Trigger-Adresse erreicht (Reader)
6	Interrupt: Trigger-Adresse erreicht (Writer)
24	PLB Fehler
25	PLB Timeout
26	Startadresse ungültig (Reader)
27	Endadresse ungültig (Reader)
28	Startadresse ungültig (Writer)
29	Endadresse ungültig (Writer)

Tabelle 5: Statusregister des SDI-Kontrollers.

Bit	Funktion
0	Reader starten
1	Writer starten
2	FIFO leeren (Reader)
3	FIFO leeren (Writer)
4	Signal `pu_reset` auslösen
5	Taktzähler starten
6	Taktzähler zurücksetzen
7	Taktzähler stoppen
8	Taktzähler stoppen wenn Reader fertig
9	Taktzähler stoppen wenn Writer fertig
10	continous mode aktivieren (Reader)
11	continous mode aktivieren (Writer)

Tabelle 6: Kontrollregister des SDI-Kontrollers.