

Draft

**Automatisierter Entwurf von
Varianten eines
HW/SW-Systems**

FB 35 AB-1/95

Draft

Automatisierter Entwurf von Varianten eines HW/SW-Systems

*W. Hardt, R. Merker, M. Kortke, D. Finckel,
C. Trautwein, J. Wedeck, W. Rosenstiel*

*Fallstudie zum Entwurf eines HW/SW-Systems unter
Ausnutzung von Alternativen zur Parametrisierung*

Technische Universität Dresden

Institut für Grundlagen der Elektrotechnik und Elektronik
Mommsenstraße 17
01062 Dresden

Dipl.-Ing. habil. Renate Merker

Telefon: 0351/463-3108

E-Mail: merker@e-technik.tu-dresden.de

Mathias

Telefon: 0351/463-3108

E-Mail: kortke@e-technik.tu-dresden.de

Universität Gesamthochschule Paderborn

Fachbereich 17 (Informatik)

Fachbereich Informatik

Warburger Strasse 100

33098 Paderborn

Dipl.-Inform. Wolfram Hardt

Telefon: 05251 60 3348

Fax: 05251 60 3519

E-Mail: hardt@uni-paderborn.de

Universität Tübingen

Wilhelm-Schickard-Institut

Fakultät für Informatik

Arbeitsbereich Technische Informatik Sand 13

72076 Tübingen

Prof. Dr. Wolfgang Rosenstiel

Telefon: 07071/29-5482

Fax: 07071/610399

E-Mail: rosenstiel@peanuts.informatik.uni-tuebingen.de

Dipl.-Inform. Christoph Trautwein

Telefon: 0721/9654-414

FAX: 0721/9654-409

E-Mail: trautw@fzi.de

Dipl.-Inform. Jörg Wedeck

Telefon: 07071/29-4015

Fax: 07071/610399

E-Mail: jw@peanuts.informatik.uni-tuebingen.de

Draft

Inhaltsverzeichnis

I. Einleitung

1.1 Überblick11
1.2 Entwurfsvarianten und Entwurfsabläufe11

II. Beispiel - Anwendung: Fuzzy Inferenz Prozeß

III. Partitionierung

3.1 Partitionierung für Parallelrechner19
3.1.1 Partitionierung für einen Workstation Cluster19
3.1.2 Partitionierungskriterium20
3.1.3 Partitionierungsmethode22
3.1.4 Partitionierung des Beispiels26
3.2 Partitionierung für Prozessorfelder31
3.2.1 Partitionierungskriterien31
3.3 HW/SW -Partitionierung für erweiterte Standard-Architekturen32
3.3.1 Partitionierungsmethode32
3.3.2 Partitionierung des Beispiels34
3.4 Schlußfolgerung37

IV. Synthese

4.1 Synthese als sequentielle Hardware41
4.2 Synthese als paralleles VLSI-Strukturfeld42
4.3 Schlußfolgerung52

V. Abbildung auf FPGAs

5.1 Zielarchitektur55
5.2 Entwurfsablauf55
5.2.1 Entwurfsablauf der Hardware56
5.2.2 Entwurfsablauf der Software59
5.3 Implementierung59
5.3.1 Implementierung der Hardware59
5.3.2 Implementierung der Software60
5.3.3 Inklusiver Nutzung60
5.4 Beurteilung der Ergebnisse60
5.4.1 Der Übersetzungsvorgang60
5.4.2 Die Implementierung61

VI. Anhang

6.1 Das C-Programm „fuzzy.c“65

Draft

Abbildungsverzeichnis

Abbildung 1-1: Zielarchitekturen	12
Abbildung 1-2: Entwurfsabläufe	13
Abbildung 2-1: N-Regeln Fuzzy-System	15
Abbildung 2-2: Ablaufplan des C - Programms für den Fuzzy-Inferenzprozeß	17
Abbildung 3-1: Überblick über den Parallelisierer	23
Abbildung 3-2: Beispiel Goedel-Implikation	24
Abbildung 3-3: Dreiadreßcodefragment	24
Abbildung 3-4: Algorithmus für den Aufbau der Eingangs- und Ausgangsmengen von Variablen	25
Abbildung 3-5: Beispiel für generierten Single-Assignment Code	26
Abbildung 3-6: Abhängigkeitsgraph der einzelnen Basisblöcke, resultierend aus der Schleifenexpansion	27
Abbildung 3-7: Abhängigkeitsgraph der einzelnen Blöcke der FOR-Partition von goe	27
Abbildung 3-8: Modifizierter Datenflußgraph	28
Abbildung 3-9: Hauptprogramm der Fuzzy-Inferenz Applikation	29
Abbildung 3-10: Workstation-Cluster für Fuzzy-Inferenz Applikation	30
Abbildung 3-11: Spezifikations-Analyse	33
Abbildung 3-12: Modulgraph der Fuzzy-Inferenz Applikation	35
Abbildung 3-13: Modulgraph der Goedel-Inferenz Partition	36
Abbildung 3-14: Anordnung der Partitionierungen gemäß der verwendeten Datenstruktur	38
Abbildung 3-15: Anordnung der Partitionierungen gemäß der intendierten Zielarchitekturen	38
Abbildung 4-1: Die C-Partition „goedel-inference“	45
Abbildung 4-2: Die Gödel-Inferenz in der Sprache „DAGS“	46
Abbildung 4-3: System von Rekurrenzgleichungen	47
Abbildung 4-4: Gemeinsamer Indexbereich	47
Abbildung 4-5: Reduzierter Abhängigkeitsgraph	48
Abbildung 4-6: Vereinfachter reduzierter Abhängigkeitsgraph	49
Abbildung 4-7: Rechenfeld für die Fuzzy-Inferenz	51
Abbildung 5-1: Blockschaltbild des Prototypenboard 'Sparrow'	55
Abbildung 5-2: Entwurfsablauf der FPGA-Implementierung	57
Abbildung 5-3: Schaltplan-Ausschnitt	61
Abbildung 5-4: Xilinx Verdrahtung	62
Abbildung 5-5: Timing Analyse	62

Draft

Tabellenverzeichnis

Tabelle 3-4 Einordnung von Partitionierungsverfahren38
Tabelle 1: Zyklen zum Lesen der Eingabe42
Tabelle 2: Datenpfad-Verzögerung für FIP42

Draft

Draft

I. Einleitung

1.1 Überblick

Dieser Bericht wurde gemeinsam von den Teilprojekten B1 (Durchgängige automatisierte Synthese massiv paralleler Rechenfelder), A2 (Automatisierte Synthese für parallele Rechnerstrukturen) und B2 (Transformationelle Entwurfsveränderung und Optimierungskriterien-Partitionierung und deren Anwendung auf Hardware/Software-Partitionierung) erstellt und beschreibt am Beispiel eines durchgängigen Beispiels das wechselseitige Ineinandergreifen dieser drei Teilprojekte.

Neben dem Übersichtscharakter der Darstellung eines „idealen“ Entwurfsablaufs, auf den im Kapitel 1.2 näher eingegangen wird, werden durch Verwendung des durchgängigen Beispiels eines Fuzzy-Controllers eine Reihe von Fragestellungen des kombinierten Hardware-Software-Entwurfs diskutiert und sowohl im allgemeinen wie auch im speziellen behandelt.

Ausgehend von einer Designspezifikation in der Sprache „C“ wird eine kombinierte Hardware/Software-Implementierung abgeleitet und anhand eines Prototypen, der aus einer SUN-Workstation und einem FPGA-Prototypenboard besteht, demonstriert. Der hierzu entwickelte Entwurfsablauf dient als Basis für die weiteren im SFB 350 geplanten gemeinsamen Hardware/Software-Codierungen der Teilprojekte A1, A2 und B3.

1.2 Entwurfsvarianten und Entwurfsabläufe

Komplexe Systeme bestehen zunehmend aus einem Geflecht von Hardware- und Software-Komponenten. Die Komponenten des Systems sollten so weit wie möglich automatisch aus einer algorithmischen Spezifikation generiert werden.

Für einzelne Komponenten eines Hardware/Software-Systems existieren bereits Entwurfswerkzeuge. Das Zusammenfügen der einzelnen Entwurfshilfsmittel wird im Folgenden untersucht..

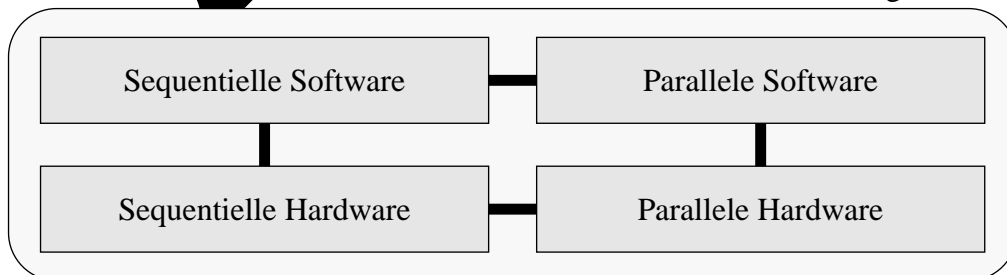


Abbildung 1-1 Zielarchitekturen

Dabei werden vier Implementierungsparadigmen gezielt untersucht, denn komplexe Hardware/Software-Systeme verbinden diese in ihrer Zielarchitektur (Abb.).

Das zu entwerfende Hardware/Software-System sei in einer algorithmischen Spezifikation beschrieben. Als Spezifikationsprache wurde hier eine prozedurale Programmiersprache verwendet („C“). Die Spezifikation enthält weder Hinweise welche Teile in Hardware implementiert werden sollen noch werden Angaben gemacht, welche Teile parallel zur Ausführung kommen. Damit werden fast keine Designentscheidungen vorgegeben und eine automatisierte Untersuchung verschiedener Varianten kann zu wesentlichen Entwurfsverbesserungen führen.

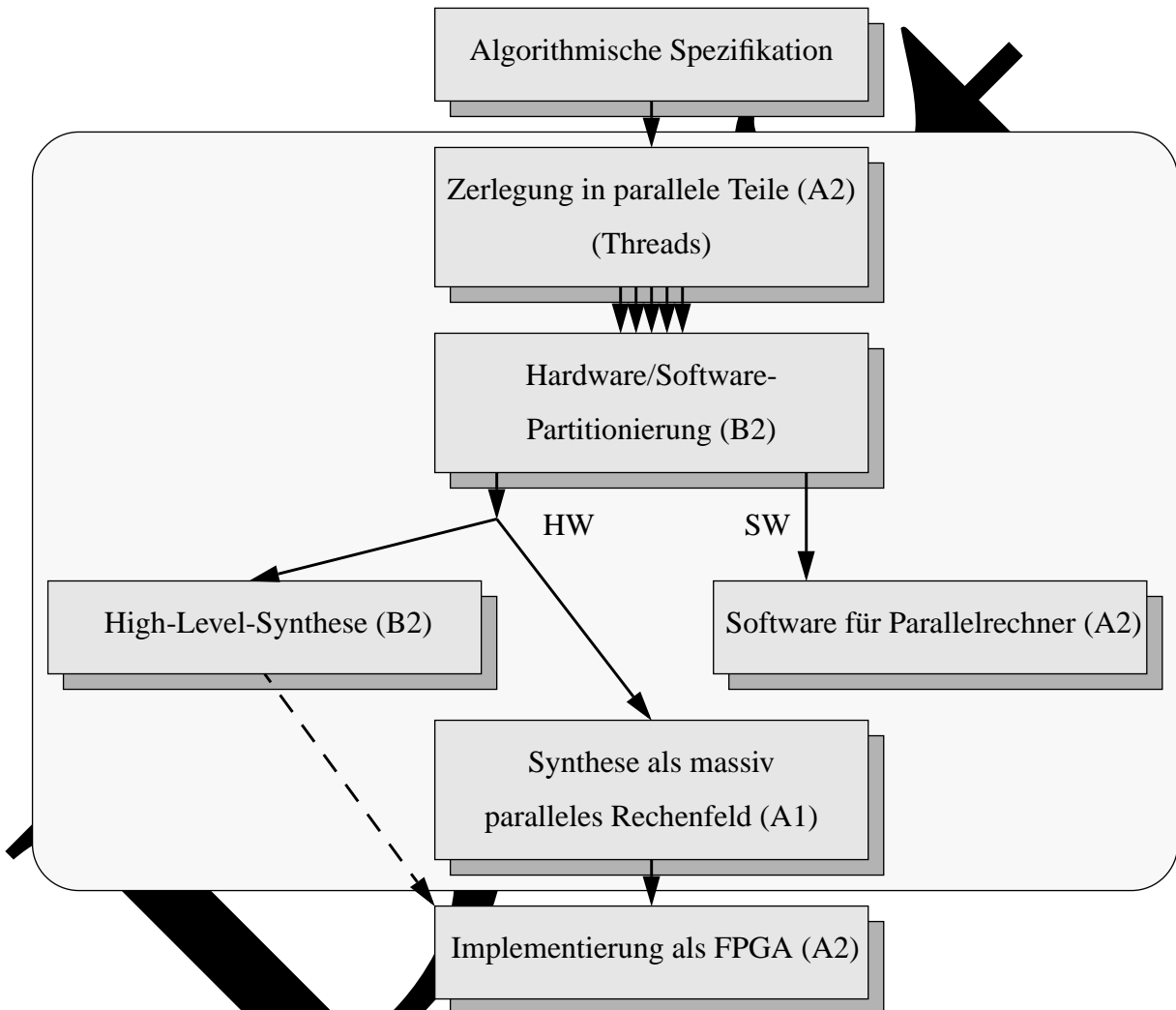


Abbildung 1-2 Entwurfsabläufe

Zur Aufteilung der monolithischen algorithmischen Spezifikation in mehrere parallel ausführbare Teile wurde ein Software-Parallelisierer entwickelt (Kapitel 3.1). Die so isolierten parallel ausführbaren Codestücke sind auf einem Workstation-Cluster direkt ausführbar. Um eine weitere Verkürzung der Ausführungszeiten zu ermöglichen werden die einzelnen Segmente daraufhin untersucht ob sie Teile enthalten, die sich für eine Implementierung in Hardware eignen.

Diese Hardware/Software-Partitionierung wird durch die Arbeiten im Teilprojekt B2 erreicht. Die dabei notwendigen Untersuchungen sind in Kapitel 3.3 beschrieben. Die in Software reali-

sierbaren Teile können direkt von einer Workstation abgearbeitet werden. Für die in Hardware zu realisierenden Teile bieten sich zwei Alternativen an.

1. Weiterverarbeitung durch ein High-Level Synthesensystem
2. Implementierung als ein massiv paralleles Rechenfeld

Alternative eins ist eine sequentielle Lösung während Alternative zwei bei geeigneten Problemen eine höhere Verarbeitungsleistung erwarten läßt.

Als High-Level-Synthesensystem wird das in Paderborn entwickelte System PMOSS eingesetzt. Die hiermit erzielten Ergebnisse werden in Kapitel 4.1 vorgestellt.

Die Synthese massiv paralleler Rechenfelder ist das Arbeitsgebiet von Teilprojekt A1. Enthalten die als Hardware zu synthetisierenden Teile als massiv paralleles Rechenfeld beschreibbare Teile so werden diese, wie in Kapitel 4.2 beschrieben, in eine VHDL-Spezifikation überführt. Diese VHDL-Spezifikation dient als Eingabe für die Realisierung der Schaltung auf einem FPGA. Die Abbildung wird von Teilprojekt A2 durchgeführt und in Kapitel V. beschrieben.

Draft

II. Beispiel - Anwendung: Fuzzy Inferenz Prozeß

Die Modellierung von Systemen mittels der Theorie der Fuzzy-Mengen besitzt insbesondere in Bereichen Bedeutung, wo herkömmliche Methoden versagen oder zu umfangreich sind. So erfahren Fuzzy-Systeme, die durch eine Anzahl von Regeln charakterisiert sind, vor allem in den Bereichen der Steuerung, aber auch in der Datenanalyse und in Expertensystemen zunehmende Anwendung. Charakteristisch für Fuzzy-Regel-Systeme sind ihre rechenintensiven Algorithmen, die die Verarbeitung großer Datenmengen erfordern. Insbesondere bei Echtzeitanwendungen können derartige Systeme nur dann zur Anwendung gelangen, wenn eine hohe Geschwindigkeit der Datenverarbeitung gewährleistet wird. Bisherige rein Softwarelösungen, die in der Lage sind, hohe Rechengeschwindigkeiten von Fuzzy-Algorithmen zu realisieren, bleiben auf einfache Konzepte der Fuzzy-Logik beschränkt [1].

Der Fuzzy-Inferenzprozeß ist ein rechenintensives Verfahren zur Logikauswertung in der Fuzzy-Logik. Die Untersuchungen zur Parallelisierung und Hardware-Software-Partitionierung sollen an diesem Benchmark verdeutlicht werden.



Abbildung 2-1 N-Regeln Fuzzy-System

Der Fuzzy-Inferenzprozeß umfaßt die Fuzzyifizierung der Eingangswerte, die Fuzzy-Inferenz zur Berechnung von Schlußfolgerungen auf der Basis eines Regelwerks und die Defuzzyifizierung der Ausgangswerte.

Während der Fuzzyifizierung werden den scharfen Eingangswerten $x \in U$ (U Trägermenge) Fuzzy-Mengen X zugeordnet. Eine Fuzzy-Menge X stellt sich als Menge von geordneten Paaren $X = \{ (u, \mu_X(u)) | u \in U \}$ beschreiben, wobei $\mu_X(u)$ die Zugehörigkeitsfunktion mit $\mu_X(u) \in [0, 1]$ darstellt. In unserem Beispiel werden als Zugehörigkeitsfunktionen für die scharfen Eingangswerte normierte symmetrische Dreiecksfunktionen mit einer fest vorgegebenen Unschärfe von 5% zum Wert x angenommen.

Zwecks Anschaulichkeit wird im Weiteren ein Fuzzy-System mit einem Ein- und einem Ausgang (Abbildung 2-1) betrachtet. Die Fuzzy-Inferenz wertet im Rahmen des Fuzzy-Inferenzprozesses ein Regelwerk in der folgenden Form aus:

Tabelle 2-1 Fuzzy - Regel - System

Regel	Regel- verknüpfung	Bedingung	Schlußfolgerung
R_1 :		IF X IS A_1	THEN Y IS B_1
R_2 :	AND	IF X IS A_2	THEN Y IS B_2
...
R_N :	AND	IF X IS A_N	THEN Y IS B_N

X bzw. A_i sind Fuzzy-Mengen auf der Trägermenge U mit den Zugehörigkeitsfunktionen $\mu_X(u)$ bzw. $\mu_{A_i}(u)$, $u \in U$, ($i = 1, \dots, N$) und Y bzw. B_i sind die Fuzzy-Mengen auf der Trägermenge V mit den Zugehörigkeitsfunktionen $\mu_Y(v)$ bzw. $\mu_{B_i}(v)$, $v \in V$, ($i = 1, \dots, N$) .

In der Literatur findet man im wesentlichen zwei Ansätze für die Realisierung der Fuzzy-Inferenz: die Methode der Aktivierungsgrade und der Relationalgleichungsansatz [Ban93].

Ersterer ist auf Grund seiner einfachen Realisierungsmöglichkeit weit verbreitet. Bei dieser Methode wird für jede Regel i ein Aktivierungsgrad α_i mit

$$\alpha_i = \max_u (\min(\mu_{A_i}(u), \mu_X(u))) \quad (1)$$

berechnet, der angibt, zu welchem Grad diese Regel zutrifft. Die Schlußfolgerungen werden mit den Aktivierungsgraden gewichtet und disjunktiv verknüpft. Die Fuzzy-Inferenz nach der Methode der Aktivierungsgrade lautet somit:

$$\mu_Y(v) = \max_i (\min(\mu_{B_i}(v), \alpha_i)) \quad (2)$$

Diese Methode besitzt jedoch Nachteile bezüglich der Genauigkeit der Ergebnisse. Auch ist dieser Ansatz nicht modular, da die Ergebnisse nachfolgender Stufen ohne eine dazwischenliegende Defuzzifizierung einen für ihre Verwertung zu hohen Grad an Unbestimmtheit aufweisen.

Der Ansatz der Relationalgleichungen [2] vermeidet diese Nachteile, ist jedoch im Vergleich zu der Methode der Aktivierungsgrade extrem rechenintensiv, so daß eine traditionelle Implementierung zu hohen Laufzeiten führt. Der Relationalgleichungsansatz beschreibt die Regeln durch Fuzzy-Relationen $R_i: A_i \rightarrow B_i$, ($R_i \subset (U \times V)$). Die Zugehörigkeitsfunktion der Relation R des Gesamtsystems kann man punktweise mit Hilfe eines Implikationsoperator \otimes aus den Zugehörigkeitsfunktionen der Fuzzy-Mengen A_i und B_i und einer konjunktiven oder disjunktiven Verknüpfung der Regeln bestimmen. Als den Implikationsoperator kann z.B. der Goedel- oder der Minimum-Operator gewählt werden. Für die Zugehörigkeitsfunktion der Relation R ergibt sich damit als eine Möglichkeit:

$$\mu_R(u,v) = \min(\text{Goedel}(\mu_{A_i}(u), \mu_{B_i}(v))) \quad \text{mit} \quad (3a)$$

$$\text{Goedel}(\mu_{A_i}(u), \mu_{B_i}(v)) = \begin{cases} \mu_{A_i}(u) & \text{wenn } \mu_{A_i}(u) \leq \mu_{B_i}(v) \\ \mu_{B_i}(v) & \text{sonst} \end{cases}$$

bzw.

$$\mu_R(u,v) = \min(\mu_{A_i}(u), \mu_{B_i}(v)) \quad (3b)$$

Der Inferenzalgorithmus lautet dann mit (3a) bzw.(3b)

$$\mu_Y(v) = \max_u (\min(\mu_X(u), \mu_R(u,v))) \quad (4)$$

Die Defuzzifizierung ermittelt aus der in der Fuzzy-Inferenz ermittelten Fuzzy-Menge Y einen scharfen Wert $y \in V$. Zur Bestimmung dieses Wertes verwenden wir die Methode des Flächenschwerpunktes [2], die durch die Gleichung realisiert wird.

$$y = \frac{\sum_{v \in V} v \cdot \mu_y(v)}{\sum_{v \in V} \mu_y(v)}$$

Um Vergleiche der verschiedenen Inferenzverfahren zu ermöglichen, aus denen gegebenenfalls auch Aussagen zum Einsatz der verschiedenen Verfahren ableitbar sind, wurde ein C-Programm geschrieben. Dieses C-Programm berechnet den Fuzzy-Inferenzprozeß zu Beginn mit der Methode der Aktivierungsgrade und zum anderen mit dem Relationalgleichungsansatz sowohl unter Anwendung des Goedel- als auch des Minimum-Operators (Abb. 2-2).

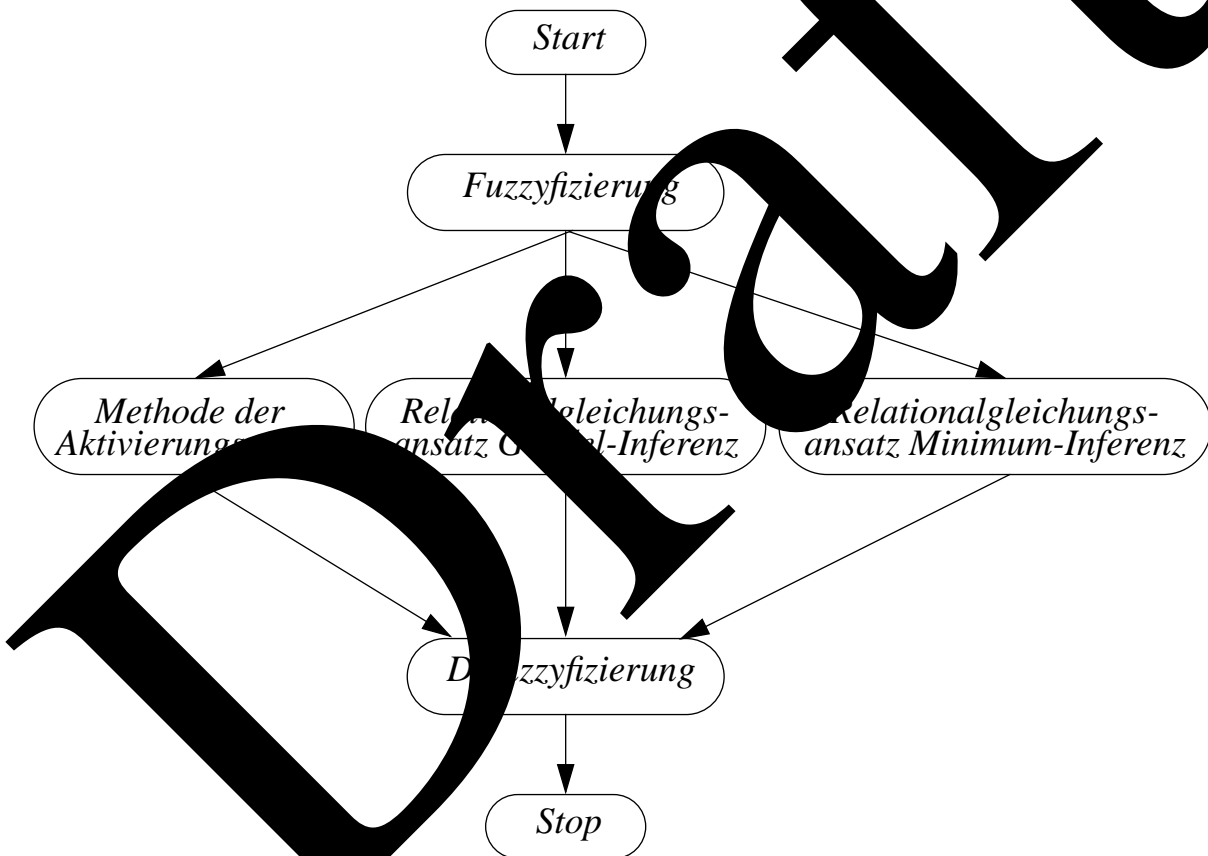


Abbildung 2-2 Ablaufplan des C - Programms für den Fuzzy-Inferenzprozeß

Mit diesem Programm sind Vergleiche zwischen den verschiedenen Ansätzen möglich und gegebenenfalls Aussagen zu erwarten, für welche Problemklassen die einfachere Methode der Aktivierungsgrade ausreichend ist. Das C-Programm ist im Anhang angefügt.

Für das Beispiel wurde aus Gründen der Übersichtlichkeit und Handhabbarkeit angenommen, daß die Fuzzy-Mengen aus 4 Wertepaaren bestehen, d.h. $k = l = 4$, und daß die Regelanzahl auf $N = 10$ beschränkt ist. Der Wertebereich der Zugehörigkeitsfunktionen soll 16 Werte um-

fassen. Damit ist eine 4 Bit-Repräsentation dieser Werte möglich. Diese Restriktionen sind prinzipiell jedoch nicht notwendig.

Referenzen

- [1] Altrock,C.v.: Über den Daumen gepeilt. c't 1991, Heft 3, S.188-200
- [2] Bandemer,H.; Gottwald,S.: Einführung in Fuzzy-Methoden, Akademie-Verlag Berlin, 4.Aufl., 1993

Draft

Draft

III. Partitionierung

Sequentielle Anwendung von Multikriterien-Partitionierung

Ein entscheidender Aspekt bei der Implementation komplexer Systeme ist die Partitionierung in Teilkomponenten. Dabei können verschiedene, von einander unabhängige Kriterien angewandt werden. Die resultierenden Partitionen sind somit verschieden und es ergibt sich Implementationsvarianten für das Gesamtsystem. In diesem Kapitel wird nun untersucht, welche Partitionierungskriterien und -methoden miteinander kombiniert werden können. Dabei werden Kriterien und Methoden für die Partitionierungsaspekte sequentiell/parallel und Hardware/Software entwickelt. Diese werden an einem Beispiel demonstriert. Die resultierende Systemimplementation ergibt sich aus der Festlegung der Partitionierungsreihenfolge. Es wird aufgezeigt, daß der Partitionierungsreihenfolge grundlegende Bedeutung zukommt. Zunächst werden die Partitionierungskriterien und die entwickelten Partitionierungsmethoden vorgestellt. Im Anschluß daran wird das Problem der Reihenfolge erörtert.

3.1 Partitionierung für Parallelrechner

Bei der System-Partitionierung für parallelrechner handelt es sich um die Partitionierung von Software. Ziel dieser Partitionierung ist es, den Einsatz von mehreren Prozessoren eine schnellere Abarbeitung zu erreichen. Dabei spielen mehrere Faktoren eine Rolle. Zum einen ist der Parallelitätsgrad der Anwendung von entscheidender Bedeutung, da er letztendlich darüber entscheidet, in wieviele parallele Partitionen (Prozesse) die jeweilige Applikation zerlegt werden kann. Dies steht die reale Anzahl an Prozessoren des Parallelrechners gegenüber, auf die die einzelnen Partitionen (Prozesse) abgebildet werden müssen. Diese Abbildung wird insbesondere von der Art des Parallelrechners und seiner Prozessor-Topologie bestimmt. In den folgenden Abschnitten wird die Software-Partitionierung für eine parallele Rechnerarchitektur, einen Workstation-Cluster, erörtert und am Beispiel der Fuzzy-Inferenz-Applikation erläutert.

3.1.1 Partitionierung für einen Workstation Cluster

Als parallele Zielarchitektur wurde ein Workstation Cluster ausgewählt, weil diese weitverbreitet sind. Die Vernetzung durch Hochleistungsnetze ermöglicht einen effizienten Datenaustausch und die Prozeßverwaltung durch die Betriebssysteme ist hinreichend ausgebaut. Von besonderem Vorteil ist, daß Testläufe mit expliziten Messungen vorgenommen werden können. Trotzdem lassen sich die entwickelten Kostenfunktionen auch auf andere parallele Architekturen

übertragen. Dazu ist lediglich eine neue Personalisierung der Parameter der Kostenfunktion notwendig.

Die erarbeiteten Kriterien legen insbesondere fest, wann sich die Parallelisierung von Programmteilen auf einem Rechnernetz gegenüber einer sequentiellen Lösung lohnt. Dabei wurde vom LogP-Modell [Culler93] ausgegangen, das auf dem BSP-Modell beruht [Valiant 90]. Beide werden kurz skizziert.

Das **Brückenmodell BSP** unterscheidet sich von herkömmlichen Modellen wie PRAM dadurch, daß es sich um asynchron arbeitende Prozessoren handelt, bei dem die einzelnen Prozessoren über Vermittlungseinheiten miteinander verbunden sind, es werden also insbesondere auch die Kommunikationskosten in dieses Modell mit einbezogen. Da es sich um asynchron arbeitende Prozessoren handelt, werden bei diesem Modell weiterhin die entstehenden Synchronisationskosten berücksichtigt. Bei der Verwendung dieses Modells können bei der Entwicklung von parallelen Algorithmen maschinenspezifische Informationen einbezogen werden, wobei insbesondere Informationen über das Speichermodell verwendet werden können, z. B. ob es sich um "shared memory" oder "message passing" Systeme handelt. Als Weiterentwicklung von BSP wurde das sogenannte **LogP-Modell** vorgeschlagen [Culler93]. Auch hier werden wie bei BSP die Kommunikationskosten berücksichtigt. So bezieht mit der Parameter **L** die Latenzzeit bei der Übertragung einer Nachricht. Die Parameter **o**, **g** und **P** geben weiterhin den **Overhead** beim Übertragen oder Empfangen einer Nachricht, die "gap" die minimale Zeit bei der Übertragung aufeinanderfolgender Nachrichten und die Anzahl **P** der Prozessoren an. Dabei werden die Parameter **L**, **o** und **g** jeweils in Vielfachen von Prozessorzyklen angegeben.

3.1.2 Partitionierungskriterium

Da bei der Parallelisierung für Rechenwerke das Erzeugen neuer Prozesse eine wesentliche Rolle spielt, wurden die Parameter **L**, **o** und **g** in die Beschreibung der Kosten bei der Prozeßerzeugung verwendet. Die Latenzzeit in einem Rechnernetz ist als sehr hoch einzustufen, genau so wie der Parameter "Overhead" bedingt durch das Erzeugen von Prozessen. Die Anzahl der Prozessoren (**P**) sollte um den Faktor 2 bis 3 höher angesiedelt werden, als in Wirklichkeit Workstations zur Verfügung stehen. Diese Workstations auch beim Warten auf Synchronisationsereignisse bedingt durch Kommunikation im allgemeinen weiterhin in der Lage sind, Prozesse auszuführen. Um nun das Problem sinnvoll zu partitionieren, müssen die Partitionsgrößen so gewählt werden, daß

$$\max(\tau(p_n), \dots, \tau(p_1)) + \sum_{i=1}^n (L + o + g + \chi(P_i)) < \sum_{i=1}^n \tau(p_i)$$

gilt. Dabei sei

- **L** die **Latenzzeit** zum Erzeugen eines Prozesses,
- **o** der **Overhead** bedingt durch das Erzeugen eines Prozesses,
- **g** die **minimale Zeit** zum Erzeugen eines Prozesses,
- **p_i** eine **Partition**. Dieser Parameter sollte nicht mit dem Parameter **P** aus dem logP-Modell verwechselt werden, der die Anzahl der Prozessoren angibt.

- $\tau(p_i)$ die **Rechenzeit** für die Partition i , unter der Voraussetzung, daß die Partitionen p_1 bis p_n parallel ausgeführt werden können.
- $\chi(p_i)$ die **Zeit des Parameterraustausches**, d.h. die benötigt wird, durch das Versorgen der Prozessoren mit den zur Berechnung der Partition p_i nötigen Daten und der Rückgabe der berechneten Werte.

Obige Formel besagt also, daß sich eine parallele Ausführung der Partitionen p_1 bis p_n nur dann lohnt, wenn der Rechenaufwand für die größte Partition plus den Kosten für das Starten der n einzelnen Partitionen bedingt durch die Latenzzeit L , den Overhead O , der "gap" g und die Zeit zum Übertragen der Parameter (χ) kleiner ist als die Summe der einzelnen Rechenzeiten, also die parallele Ausführung weniger Zeit benötigt als eine sequentielle Ausführung. Der Abschätzung der Rechenzeit für die einzelnen Partitionen kommt also eine bedeutende Rolle zu.

Um die Rechenzeit einer Partition abzuschätzen ist die Analyse der Problem spezifikation notwendig. Der hier zugrunde gelegte Ansatz sieht eine Zerlegung des Gesamtproblems bis auf die Ebene von Basisblöcken vor. Innerhalb dieser Basisblöcke ist die a-priori Abschätzung der Rechenzeit sehr einfach. Sie kann anhand der zugrunde gelegten Kosten für die einzelnen Operationen konkret ausgerechnet werden. Diese statische Ablaufplanung ist zur Überbrückungszeit möglich, da ein Basisblock keinen Kontrollfluß enthält. Allerdings ist die durchschnittliche Größe eines Basisblocks sehr gering. Daher müssen die Blöcke auch über unbekannte Sprunggrenzen hinweg zusammengefaßt werden, was zu einem dynamischen Scheduling führt. Bei dynamischem Scheduling ist auch das Beenden eines Prozesses mit einem Synchronisationsereignis verbunden. Dies wird in der folgenden Formel berücksichtigt, bei der das Starten eines Prozesses *und* das Beenden eines Prozesses in die Berechnung der Gesamtkosten eingehen, wobei für das Starten und das Beenden jeweils die gleichen Kosten angenommen werden:

$$\max(\tau(p_1), \dots, \tau(p_n)) + \sum_{i=1}^n ((L + o + g) + \chi(P_i)) < \sum_{i=1}^n \tau(p_i)$$

Der Faktor des Erzeugens von n neuen Prozessen ($(L+o+g)$), der bei realen MPP-Rechnern nur eine untergeordnete Rolle spielt, muß beim Einsatz von vernetzten Unix-Workstations neu bewertet

Während der bekannte Betriebssystemaufruf "fork" im Unixbetriebssystem einen Großteil der Ressourcen des Elterprozesses kopiert, und mit diesen dann den Kindprozeß startet, entfällt damit zeitraubende Kopieren bei den sogenannten "light weight processes" (leichtgewichtige Prozesse/threads). Das Erzeugen eines neuen *lokalen* Prozesses kann daher als nahezu kostenlos angenommen werden. Jedoch muß beim Starten eines Prozesses über das Netzwerk mit erheblichen Kosten gerechnet werden, da neben den Daten auch der Programmcode der aufgerufenen Partition mindestens bei ihrem ersten Ausführen auf den Zielprozessor durchgeführt werden muß. Dies kann entweder zur Startzeit des Gesamtprozesses geschehen, hier müssen alle Codesequenzen aller Partitionen geladen werden. Es kann allerdings auch auf Anforderung durchgeführt werden, d.h., wenn die Partition auf einer anderen Workstation gestartet werden soll, muß beim ersten Start der Code mit als Nachricht übermittelt werden. Weiterhin ist beim Starten der Prozesse auf anderen Workstations weiterhin ein *fork*-Systemaufruf mit all seinen

Konsequenzen durchzuführen. Daher ist für den Einsatz von vernetzten Workstations der zusätzliche Parameter σ einzuführen:

$$\max(\tau(p_1), \dots, \tau(p_n)) + \sigma + \sum_{i=1}^n ((L + o + g) \cdot 2 + \chi(P_i)) < \sum_{i=1}^n \tau(p_i)$$

Dabei bezeichnet der Parameter

- σ sämtliche Kosten, die durch das Starten eines Prozesses auf einer Workstation und das Initialisieren der Netzwerkverbindung entstehen.

Dieser Parameter ist in der Regel nur dann von Bedeutung, wenn relativ große Datenmengen zwischen den parallelen Prozessen ausgetauscht werden müssen.

3.1.3 Partitionierungsmethode.

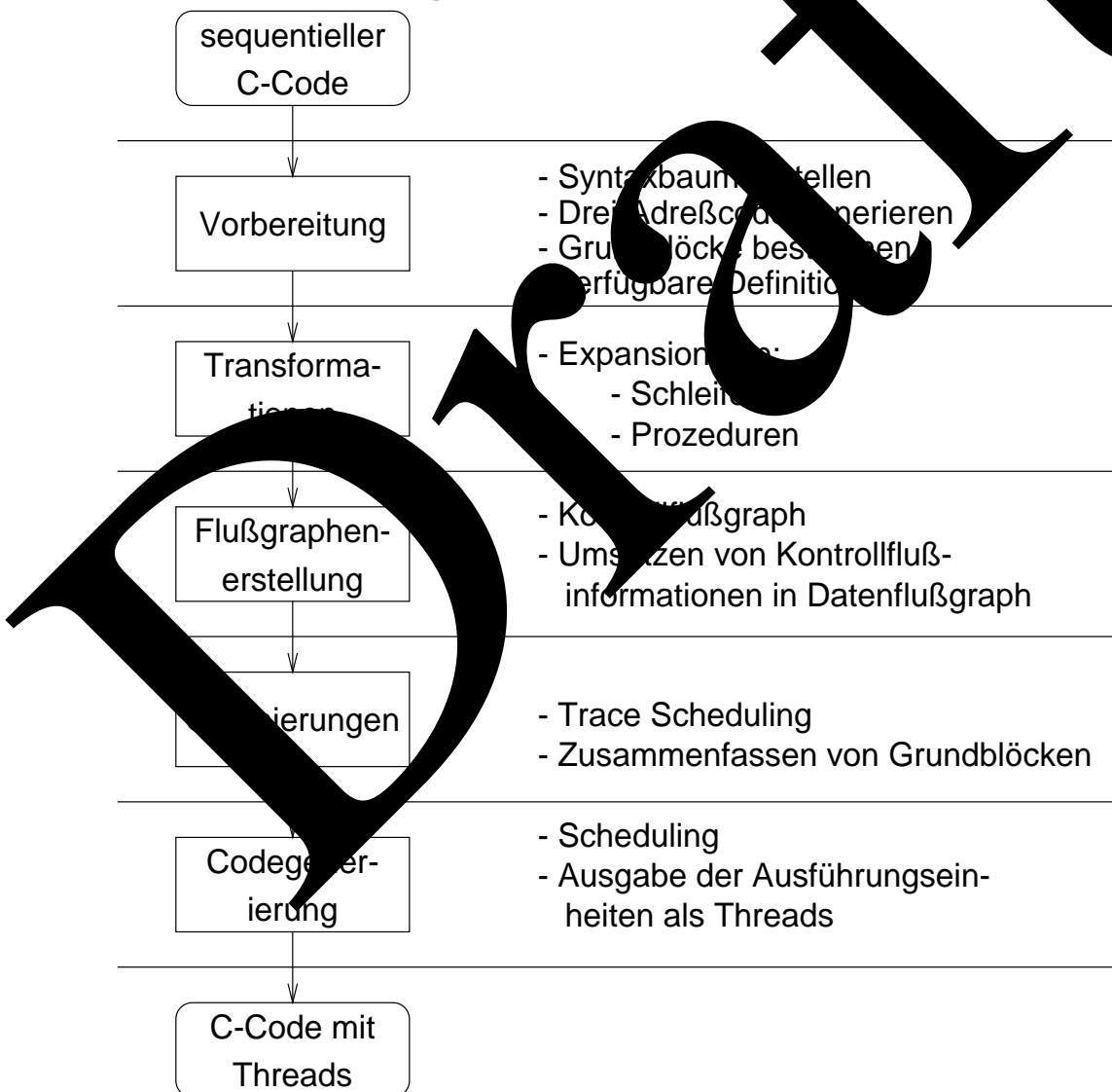


Abbildung 3-1 Überblick über den Parallelisierere

Verschiedene Untersuchungen haben gezeigt, daß auch in sequentiellen Programmen ein gewisser Anteil an Parallelität zu finden ist, der auch mit Hilfe einer automatischen Parallelisierung zumindest teilweise aufgedeckt werden kann [Wall91,WeRo92]. Da bezüglich der Parallelisierung von Schleifen schon umfangreiche Vorarbeiten existieren [Pugh91], wurde im Rahmen des Teilprojektes A2 ein Prototyp eines Parallelisierers entwickelt, der insbesondere die nichtregulären Strukturen von Programmen analysiert

Der prinzipielle Ablauf des Parallelisierers ist Bild 1 zu entnehmen. Nach dem Syntax- und Semantikcheck wird das Eingabeprogramm in eine Dreiadreßdarstellung transformiert, bevor es nach einer Schleifenexpansion (wenn möglich) schließlich in seine einzelnen Grundblöcke zerlegt wird. Danach werden die "verfügbaren Definitionen" berechnet, und anhand dieser Datenflußinformationen wird ein Datenflußgraph erstellt, dessen Kanten die Abhängigkeiten der Basisblöcke untereinander beschreibt. Entlang dieser Kanten werden möglichst voneinander abhängige Partitionen wieder miteinander verknüpft, um so größere Partitionen des Gesamtprogramms zu erhalten. Dieser Schritt wird so lange wiederholt, bis "optimale" Partitionsgrößen erhalten wurden. Danach wird das so parallelisierte Programm als C-Code ausgeben, der dann mit einem nativen C-Compiler für die Zielmaschine übersetzt und mit der PVM verbunden werden kann. Im Rest dieses Unterkapitels wird anhand des Beispiels die Vorgehensweise näher erläutert. Der eigentliche Algorithmus wird exemplarisch die Goedel-Implikation, ein Teil der Fuzzy-Inferenz-Applikation diskutiert. Die Goedel-Implikation ist in Abb. 3-2 dargestellt. Die Funktionen "goe", "min" und "max" beschränken sich jeweils auf die Ausführung eines Vergleichs gefolgt von einer Zuweisung als return-Anweisung.

```

for (v=0;v<SIZE;v++)
    for (u=0;u<SIZE;u++) {
        for (r=0;r<NMF;r++) {
            g=goe(amf[r][u],bmf[r][v]);
            if (r==0) c=g;
            else c=min(c,g); }
        y=min(xmf[u],c);
        if (u==0) ymf[v]=y; else ymf[v]=max(ymf[v],y);
    }

```

Abbildung 3-2 Beispiel Goedel-Implikation

Während der syntaktischen und semantischen Analyse wird zunächst ein Strukturbaum erstellt. Aus ihm wird nachher eine Dreiadreßcodedarstellung generiert, die insbesondere die nachfolgende Analyse der Grundblöcke vereinfachen soll. Insbesondere Konstrukte wie $f(f(a))$, bei denen auf Anhieb die Grundblöcke nicht identifiziert werden können, werden durch die Dreiadreßcodegenerierung vereinfacht, z.B. in die Form:

```

_h3 = f(a);
f(_h3);

```

Abbildung 3-3 Dreiadreßcodefragment

Im Gegensatz zu vielen anderen Compilern werden hier aber keine weiteren Annahmen bezüglich des realen Maschinenmodells der Zielmaschinen getroffen. Insbesondere werden hier keine Informationen über die vorhandene Anzahl von Registern verwendet. Das Ausnutzen dieser Informationen und der dadurch möglichen Optimierungsmaßnahmen bleibt den auf den unterschiedlichen Architekturen erhältlichen nativen C-Compilern vorbehalten. Nach dieser Transformation können auch komplexe Konstrukte, z.B. mehrfache Prozeduraufrufe, auf eine einfache Weise in ihre Grundblöcke zerlegt werden.

Trifft der Parallelisierer auf Schleifen, deren Iterations-Anzahl schon zur Übersetzungszeit bekannt, also konstant ist, kann an dieser Stelle eine Schleifenexpansion durchgeführt werden. Dabei ist zu beachten, daß die Anzahl der Iterationen klein sein muß, da sonst eine Explosion der Codegröße zu erwarten ist. Um eine sinnvolle Anzahl der expandierten Iterationen zu bestimmen, ist der Codeumfang des Schleifenrumpfes zu beachten: Ein Schleifenkopf, der nahezu das gesamte Programm enthält, kann schon zu groß sein, um 5 mal expandiert zu werden, während eine Schleife zum Initialisieren eines einfachen Array schon bei 100 Elementen sinnvoll expandiert werden kann. Das Expandieren von Schleifen zur Bestimmung der Parallelität ist ein primitives Mittel, geeigneter sind z.B. die Bildung von Datenflussabhängigkeiten mit Hilfe des Omega-Tests [10]. Solche Verfahren könnten an dieser Stelle eingesetzt werden, jedoch ist dann für den hier beschriebene Ansatz zur Entdeckung von innerer Parallelität mit relativ schlechten Ergebnissen zu rechnen.

Der so erhaltene Kontrollflußgraph wird nun weiterverwendet, um die Datenflußinformation, wie sie vom Programmierer angegeben wurden, zu bestimmen. Hierzu wird durch den gesamten Kontrollflußgraph gewandert, und die Menge der von jedem Basisblock benötigten und produzierten Daten *gen* und *use* bestimmt. Weiterhin wird die Menge der Variablen *kill* bestimmt, die alle Definitionen von Variablen enthält, deren Gültigkeitsbereich im aktuellen Basisblock endet, d.h. die überschrieben wird. Nun kann jedem Basisblock die Menge der Eingangsvariablen *in* und Ausgangsvariablen *out* folgt bestimmt werden:

```

foreach statement in block b:
    foreach definition d[i] in use[b]
        if d[i] not in gen[b]
            in[b] = in[b] + d[i];
        endif
    endfor
endfor

```

Abbildung 3-4 Algorithmus für den Aufbau der Eingangs- und Ausgangsmengen von Variablen

Analog dazu kann die Menge *out* durch Ersetzen von *use* durch *gen* und von *gen* durch *kill* bestimmt werden. In der nachfolgenden Datenflußanalyse muß nun eine Kante von einem Basisblock A zu einem Basisblock B eingefügt werden, wenn folgende Bedingung nicht gilt:

$$(in(A) \cap out(B) \neq \emptyset) \vee (out(A) \cap in(B) \neq \emptyset) \vee (out(A) \cap out(B) \neq \emptyset)$$

Der resultierende Datenflußgraph hat nun das Problem, daß seine Knoten sehr kleine Partitionen sind, die im Mittel ca. 5 Maschineninstruktionen entsprechen. Daher muß nun ein Zusammenfügen dieser Partitionen zu größeren Einheiten erfolgen, um den Overhead nicht so groß werden zu lassen, daß die Laufzeit der Partition inklusive Kommunikation nicht größer wird als eine äquivalente sequentielle Lösung ist. Vergrößerung der Ausführungseinheiten kann einerseits geschehen entlang der Kanten im Datenflußgraph. Hierbei werden nur Partitionen zusammengefügt, die sowieso nur sequentiell ausgeführt werden können. Dabei wird allerdings der sonst existierende Overhead bedingt durch Prozeßerzeugung und Prozeßelimination verringert. Andererseits können auch Partitionen, zwischen denen kein Pfad bzw. Kante im Datenflußgraphen besteht, zusammengefügt werden. Durch das Zusammenfassen solcher datenunabhängiger Partitionen wird eine Zwangssequentialisierung erreicht, die dann sinnvoll ist, wenn der Parallelitätsgrad sehr hoch ist, die Rechenzeit τ gegenüber den Parametern L , o , g und χ relativ klein ist, also der Kommunikationsüberhang den durch die Partitionierung gewonnenen Zeitvorteil eliminieren oder sogar verringern würde. Weiterhin ist eine solche Zwangssequentialisierung sinnvoll, wenn die Anzahl der verfügbaren Prozessoren sehr klein ist gegenüber dem Parallelitätsgrad der voneinander unabhängigen Partitionen.

3.1.4 Partitionierung des Beispiels

Die entwickelte Methode zur Software-Parallelisierung wurde auf die Fuzzy-Inferenz-Applikation angewandt. Zunächst wurde die Goedel-Inferenz versucht. Diese stellte sich jedoch dann als zu feingranular heraus. Daher wurde dann die entwickelte Methode auf die gesamte Applikation angewandt. Diese Ebene besteht aus wesentlich komplexeren Komponenten, im wesentlichen der Minimum-Inferenz, der Goedel-Inferenz, Methoden zur Aktivierungsgrade, der Fuzzyifizierung und der Defuzzyifizierung (vgl. Abb. 7-2). Hier ist eine sinnvolle Partitionierung für einen Workstation-Cluster möglich.

Die Anwendung des Parallelisierers auf die Goedel-Inferenz expandiert nach der Synthese des Kontrollflußgraphen in "kleinen" Schritten. Dies führt dazu, daß vom Parallelisierer alle Schleifen des Beispiels expandiert werden, und zwar beginnend von der äußersten Schleife. Die nachfolgende Erzeugung von Single-Assignment Strukturen hat zur Folge, daß folgende Struktur erzeugt wird. In Abb. 3-5 wird dabei zur Illustration nur von einer Iteration der inneren Schleife ausgegangen, in Wirklichkeit werden für alle Iterationen die Variablen bzw. ihre Partitionen mit eindeutigen Namen versehen. u sei im folgenden beliebig, aber fest.:

```

g_0 = goe (amf[0][u], bmg[0][u]);
g_1 = goe (amf[1][u], bmg[1][u]);
...
g_9 = goe (amf[0][u], bmg[0][u]);
c_0 = g_0;
c_1 = min (c_0, g_1)
...
c_9 = min(c_8, g_9);

```

Abbildung 3-5 Beispiel für generierten Single-Assignment Code

Die *if*-Anweisung kann o.B.d.A. weggelassen werden, sie ist innerhalb aller Schleifendurchläufe konstant, was von einem Compiler i.d.R. auch zur Laufzeit entdeckt wird. Die Aufrufe von *goe* und *min* sind in Wirklichkeit gemischt, sind aber alle datenunabhängig voneinander, so daß eine Umsortierung des Codes stattfinden kann (dies wird ihm Rahmen der DF-Analyse des Parallelisierers durchgeführt). Durch die DF-Analyse erhält man nun folgendes DFG-Fragment:

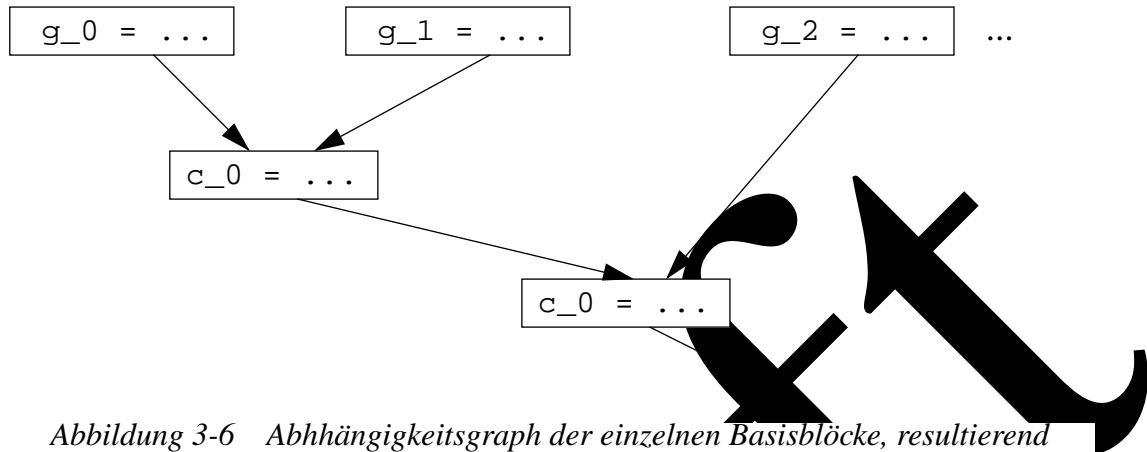


Abbildung 3-6 Abhängigkeitsgraph der einzelnen Basisblöcke, resultierend aus der Schleifenexpansion

Der Aufruf von "goe" kann offensichtlich parallel erfolgen vorausgesetzt, die Eingabeparameter sind zu diesem Zeitpunkt richtig initialisiert. Als nächstes können nun die nächst äußere Schleife betrachtet werden. Das Kürzel "fr0" steht hierbei für die *for*-Schleife mit Parameter *r* und der Initialisierung dieses Parameters auf den Wert 0:

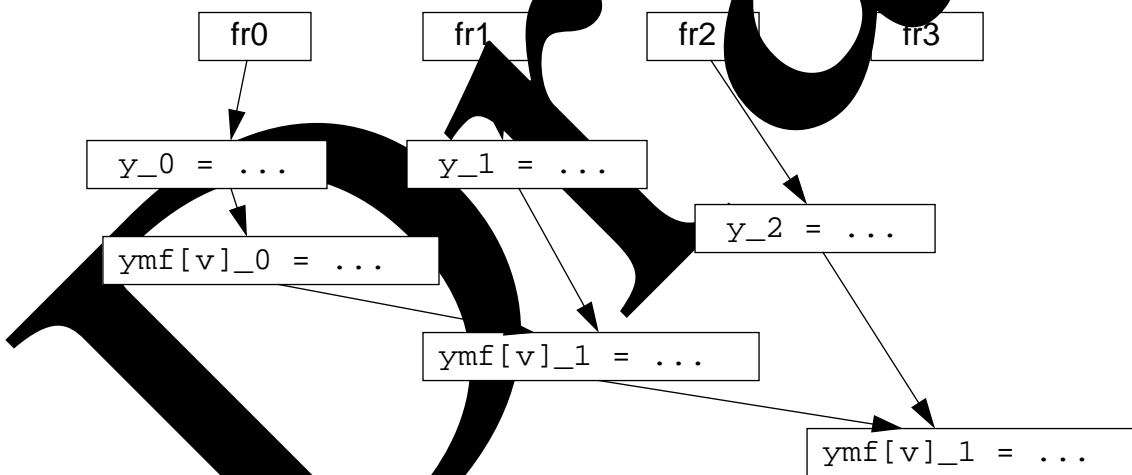


Abbildung 3-7 Abhängigkeitsgraph der einzelnen Instanzen der FOR-Schleife von goe

Auch hier kann am Datenflußgraphen erkannt werden, daß die Anfänge der mittleren Schleifenrumpfe unabhängig voneinander gestartet werden können, die Ausgangsabhängigkeit am Ende erfordert dann eine spätere Sequentialisierung. Wird nun die äußere Schleife betrachtet, ist zu Erkennen, daß alle Iterationen unabhängig voneinander ausgeführt werden können. Durch die Expansion werden also die Anfänge der innersten Schleife zu $4 \cdot 4 \cdot 10 = 160$ unabhängige Partitionen erzeugt. Ohne nachfolgendes Zusammenfügen wäre hiermit eine sehr ineffiziente Ausnutzung eines Workstation-Netzes zu erwarten, zumal die Rechenleistung innerhalb der inneren Schleife auf einen Vergleich und eine Zuweisung beschränkt wäre. Hier ist also eine Zwangsse-

quentialisierung innerhalb der innersten Schleife angebracht, sämtliche sonst benötigten Prozeßerzeugungs- und eliminationselemente können durch diese Sequentialisierung entfallen. Die Betrachtung der nächsthöheren Schleifenebene ergibt ein ähnliches Ergebnis. Zwar können die 4 Schleifenanfänge mit ihren inneren expandierten Schleifen unabhängig voneinander ausgeführt werden, jedoch ergibt sich das Problem der übermäßigen Kommunikation bedingt durch die Ausgangsabhängigkeit verursacht durch Schreibzugriff auf das Array-Element $ymf[v]$. Dies kann gelöst werden, indem der zweite Teil der Schleife für jede Iteration zusammengefaßt wird. Damit ergibt sich folgender Datenflußgraph mit den neuen Ausführungseinheiten als Knoten:

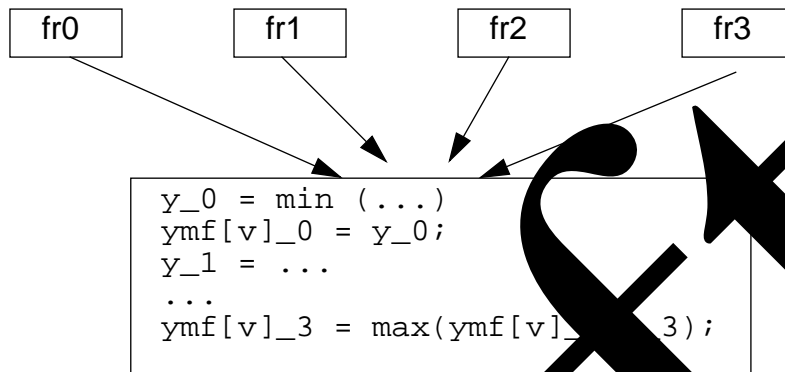


Abbildung 3-8 Modifizierter Datenflußgraph

Dies ist für reale MPP-Rechner eine kritische Partitionierung, da die einzelnen Iterationen, wie weiter unten dargelegt, zu wenig Rechenzeit benötigen im Vergleich zu dem Aufwand, der durch eine Prozeßerzeugung (auch bei leichtgewichtigen Prozessen) auftritt. In diesem Fall würden 16 Partitionen der mittleren *for*-Schleife parallel ausgeführt. Auch auf Workstation Clustern ist dieser Rechenaufwand bzw. die Granularität zu gering. In der innersten Schleife werden im parallel ausführbaren Teil jeweils 2 Vergleiche und 2 Zuweisungen ausgeführt unter der Annahme, "min" und "max" würden expandiert. Durch Expansion der inneren Schleife werden daher für jede Iteration der inneren Schleife 20 Vergleiche und 20 Zuweisungen, sowie weitere 4 Vergleiche und 4 Zuweisungen durchgeführt, wobei die Adreßberechnungen noch nicht berücksichtigt ist. Das Übersetzen der Funktion "fuzmagine" mit dem Gnu C-Compiler Version 2.6.2 mit der Option "-O2" ergibt, daß diese gesamte Funktion mit 44 Maschineninstruktionen realisiert werden kann. Zur Laufzeit werden insgesamt ca. 3200¹ Instruktionen für die innere Schleife durchgeführt, wenn keine Inlinexpansion von Funktionen und keine Optimierung der im Prinzip schlüssigen Vergleiche durchgeführt wurde. Ein ähnliches Ergebnis würde eine 160-fache Expansion (wie sie vom Parallelisierer durchgeführt wird) ergeben, jedoch wäre der Schleifen-Overhead bestehend aus Vergleich und Rücksprung eliminiert, es wären mindestens 320 Befehle weniger. Die Ausführung des mittleren Schleifenrumpfes benötigt weitere 400 Maschinenbefehle, bzw. die Kontrollen der Schleife würde zu 400 Maschinenbefehlen führen. Demgegenüber stehen auf Sparc 10 unter SunOs 4.1.3 jedoch alleine mindestens jeweils 1000 Befehle von `lwp_create` und `lwp_destroy` (bzw. `thr_create` und `thr_kill` unter Solaris 2.3) gegenüber. Für jeden Prozeß wären also mindestens 2000 Maschinebefehle auf dem initiierenden Prozessor bzw. Workstation nötig. Werden nun auch nur 2 Prozesse neu erzeugt,

1. Alle nachfolgenden Zahlenangaben wurden durch Compilierung und nachfolgenden Disassemblierung auf einer Sparc-Station 10 ermittelt und auf die vollen 100 aufgerundet.

bedeutet dies, daß 4000 Instruktionen alleine für das Starten der Prozesse benötigt werden, gegenüber den 3200 Instruktionen der nicht optimierten Version. Wird nun versucht, einen Prozeß über das Netzwerk zu starten, kommen noch ganz erhebliche Verzögerungen durch die Treiber für den Netzwerkadapter und die Kommunikation mit der anderen Workstation hinzu. Der Parallelisierer ist also ungeeignet, Schleifen mit solch feinkörniger Parallelität zu parallelisieren, sowohl auf sogenannten Multiprozessor-Rechnern wie Sparc-Station 10, als auch auf Rechner-netzwerken. Hierfür sollte vielmehr in Zukunft ein Mechanismus entwickelt werden, der solche für eine Spezialhardware gut implementierbare Algorithmen anhand von Äquivalenzklassen identifiziert und geeigneteren Synthesystemen zur Parallelisierung übergibt. Diese Äquivalenzklassen können zum Beispiel für Schleifen mit regulärer Parallelität Matrizen gleichen Ranges sein. Für irreguläre Parallelität können solche Äquivalenzklassen zum Beispiel über Pattern-Matching von ähnlichen Strukturen innerhalb der internen Darstellungsform bestimmt werden. Ein ähnliches Ergebnis ergeben die Betrachtungen für die anderen beiden Teile des Programms "max_min_inference" und "act_deg_inference". Um eine solche Parallelisierung durchzuführen zu können müssen solche Analysen auf einer Ebene höherer Modularität durchgeführt werden. Eine Abhängigkeitsanalyse der gesamten Fuzzy-Inferenz-Applikation (Abb. 3-9) ergibt,

```

1 for( ct = 0; ct < inputs; ct++ ) {
2     /* Fuzzyfizierung des Eingangswertes */
3     fuzzyfy( x[ct], xmf );
4
5     /* Fuzzy Inferenzen */
6     goedel_inference( amf, bmf, xmf, ymfGoedel );
7     max_min_inference( amf, bmf, xmf, ymfMaxMin );
8     act_deg_inference( amf, bmf, xmf, ymfActDeg );
9
10    /* Defuzzyfizierung (Massenschwerpunkt) */
11    GoedelOut = defuzzyfy( &yGoedel[ct], ymfGoedel );
12    MaxMinOut = defuzzyfy( &yMaxMin[ct], ymfMaxMin );
13    ActDegOut = defuzzyfy( &yActDeg[ct], ymfActDeg );
14
15    /* Ausgabe der Ergebnisse */
16    . . .
17 }

```

Abbildung 3-9 Hauptprogramm der Fuzzy-Inferenz-Applikation

daß eine Parallelisierung in drei Stufen möglich ist. Diese sind:

1. Rumpf der Schleife des Hauptprogramms (Zeile 2-16, Abb. 3-9)
2. die datenunabhängigen Teile dieses Rumpfes (Zeile 6, 7, 8)
3. die Berechnung der jeweiligen Inferenz.

Diese drei Stufen werden der Reihe nach betrachtet. Eine Parallelisierung der Stufe 1 bildet jede Instanz der Schleife (Zeile 2 - 16) des Hauptprogramms auf eine Workstation ab. Dies ist möglich, da die einzelnen Aktionen datenunabhängig sind. Damit werden N^1 Workstations notwen-

-
1. Anzahl der zu bearbeitenden Eingabedaten

dig. Diese Partitionierung ist offensichtlich ungünstig, da die Berechnung für ein Eingabedatum nicht komplex genug ist.

In der 2. Parallelisierungs-Stufe wird nun der Rumpf der Schleife in Zeile 1 betrachtet. Die einzelnen Funktionsaufrufe sind datenunabhängig (Zeile 6-8). Jeder Funktionsaufruf kann damit auf eine separate Workstation abgebildet werden. In diesem Fall sind relativ wenig Daten an die einzelnen Prozesse zu übertragen, nämlich genau die Eingabedaten. Dem steht eine recht komplexe Inferenzberechnung gegenüber, und die definierte Kostenfunktion wird erfüllt.

Die 3. Parallelisierungs-Stufe befaßt sich mit der parallelen Berechnung der einzelnen Inferenzen. Diese wurde bereits exemplarisch an der Goedel-Inferenz diskutiert. Die Anzahl der auszuführenden Instruktionen wird durch die Konstanten $SIZE^1$ und $RULES^2$ bestimmt. Die in diesem Beispiel gewählten Werte erlauben keine sinnvolle Parallelisierung auf dieser Ebene, denn die Konstanten $SIZE$ und $RULE$ sind recht klein gewählt. Damit wird der Overhead für das jeweilige parallele Aufrufen der Funktionen im Schleifenrumpf erheblich höher als die benötigte Rechenzeit. Ist jedoch z.B. die Konstante $SIZE$ entsprechend groß gewählt, können auch die Funktionsaufrufe selber parallelisiert werden. Hierbei ist sogar eine Kombination beider Möglichkeiten sinnvoll: Parallelisierung auf dieser Ebene durch ein Rechnernetz mit MIMD-Prozessoren und Parallelisierung der einzelnen Funktionsaufrufe (Ebene 2) auf den einzelnen Prozessoren der MIMD-Rechner. Jedoch steigt im allgemeinen bei der Erhöhung der Berechnungskomplexität auch die Anzahl der zu transferierenden Daten. Das ist auch für dieses Beispiel zu beachten. Durch Erhöhung der Konstante $SIZE$ steigt auch der Umfang der über das Netz zu transferierenden Daten. Da jeweils zwei Arrays mit $SIZE$ Elementen als Parameter übergeben werden, beträgt die Datenkomplexität $O(2 * n)$, wobei n die Anzahl der Elemente der Arrays darstellt. Demgegenüber hat die Goedel-Inferenz in unserem Beispiel die Rechenkomplexität $O(n^2 + r * n)$, wobei r die Anzahl der Fuzzy-Regeln ($RULES$) beschreibt. Die Fuzzy-Inferenz nach der Methode der Aktivierungsgrade hat die Komplexität $O(r * n)$ und die Fuzzy-Inferenz mittels Maximum/Minimum besitzt ebenfalls die Komplexität $O(n^2 + r * n)$. Da in unserem Beispiel $r > 2$ gilt, sind für alle zu parallelisierenden Funktionen die Rechenkomplexitäten größer als die Datenkomplexität, der Rechenaufwand steigt mit wachsender Konstante $SIZE$ stärker an, als die Kosten für den Datentransfer.

Für die Fuzzy-Inferenz-Applikation ergibt sich also durch Anwendung der Parallelisierungsstufe 2 eine Abbildung auf einen Workstation-Cluster von 4 Workstations:

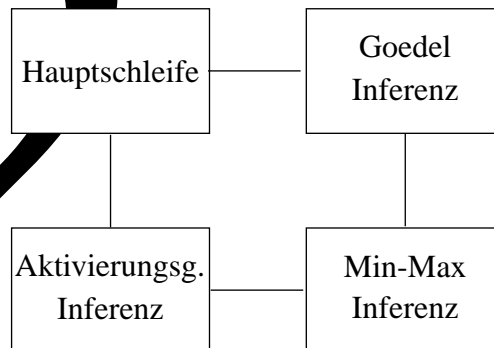


Abbildung 3-10 Workstation-Cluster für Fuzzy-Inferenz-Applikation

1. Größe der Membership-Funktionen
2. Anzahl der berücksichtigten Fuzzy-Regeln

3.2 Partitionierung für Prozessorfelder

Eine weitere Variante der sequentiell/parallel-Partitionierung soll nun betrachtet werden. Neben der beschriebenen Software-Parallelisierung kann auch eine Parallelisierung für eine Hardware-Implementation untersucht werden. Das Entwurfsziel ist dann die Realisierung rechenintensiver Teile durch eine massiv parallele Hardware. Ausgehend von einer algorithmischen Spezifikation sind die Teile der Spezifikation zu ermitteln, die möglichst rechenintensiv sind und auf dieser Hardware realisiert werden können. Ein automatisches Verfahren zur Erkennung dieser Teile existiert bisher noch nicht. Es lassen sich jedoch die Anforderungen, die an die Spezifikation gestellt werden müssen, um mit den vorhandenen Werkzeugen [6] ein entsprechendes Rechenfeld zu realisieren, genau charakterisieren.

3.2.1 Partitionierungskriterien

Abstrakte Verhaltensbeschreibungen, die hier als Spezifikationen in C-Sprache dienen, sind aus verschiedenen Konstrukten, wie z.B. Zuweisungen und Schleifen etc. zusammengesetzt. Rechenintensive Teile sind durch Schleifen-Konstrukte definiert. Um Schleifen durch eine massiv parallele Rechenfeld zu realisieren müssen die folgenden Kriterien erfüllt sein:

- Alle Typen von Schleifen müssen in die C-Notation von *For*-Schleifen konvertiert sein.
- Die Schachtelung von Schleifen ist erlaubt, wenn die oberen und unteren Grenzen der Laufindizes affine Funktionen zu den benutzten Indizes sind.
- Schleifen können „*if/else*“-Konstrukte enthalten, wenn die Bedingung nur aus logischen UND Verknüpfungen von Vergleichsoperatoren von affinen Ausdrücke der Schleifenindizes besteht.
- Die Variable der Schleifenindizes dürfen im Schleifenkörper nicht verändert werden.
- Die Schleifen können Aufrufe von arithmetischen Funktionen mit Parametern beinhalten. Die Identifikation der Parametern als Ein- bzw. Ausgabevariable muß möglich sein.
- Alle Variablen, die innerhalb der Schleifen vorkommen, haben die Form $v(f(I))$, wobei v den Namen der Variablen angibt, I ein Indexvektor ist, der die Schleifenindizes enthält und $f(I)$ eine affine Funktion von I sein kann. Die Funktion $f(I)$ wird Indexfunktion der Variablen v genannt.

Liegt eine Spezifikation aus einem von affinen Rekurrenzgleichungen vor, so kann auch diese auf massiv parallele Rechenfelder abgebildet werden. Einzelheiten sind in [6] zu finden.

Werden diese Kriterien auf die Fuzzy-Inferenz-Applikation angewandt, so erhält man folgende Ergebnisse:

Die Funktionen „*goedel_inference*“, „*max_min_inference*“ und „*act_deg_inference*“ enthalten geschachtelte „*for*“-Schleifen, in denen nur Funktionsaufrufe von arithmetischen Funktionen vorkommen. Auch die anderen oben genannten Kriterien sind jeweils erfüllt. Somit ist es möglich, für diese Funktionen jeweils ein stückweise reguläres massiv paralleles Rechenfeld zu erzeugen. Der verbleibende Rest des C-Programms wird als sequentielle Software realisiert.

3.3 HW/SW-Partitionierung für erweiterte Standard-Architekturen

Als weiterer Aspekt der System-Partitionierung wird nun die HW/SW-Partitionierung untersucht. Die Anzahl der möglichen Partitionierungen ist so groß, daß eine erschöpfende Implementierung nicht praktikabel ist. Verschiedene Partitionierungsansätze sind bereits untersucht worden, z.B. [1, 2, 3]. Dabei stellt sich heraus, daß sowohl der Optimierungsgesichtspunkt als auch die intendierte Zielarchitektur von großer Bedeutung sind. Der hier vorgestellte Ansatz zielt auf Performanz-Steigerung hin. Besonderer Wert wird auf eine effiziente Verknüpfung von Standard und Spezial-Hardware gelegt. Im folgenden wird nun die Erkennung von Hardware-Partitionen beschrieben.

3.3.1 Partitionierungsmethode.

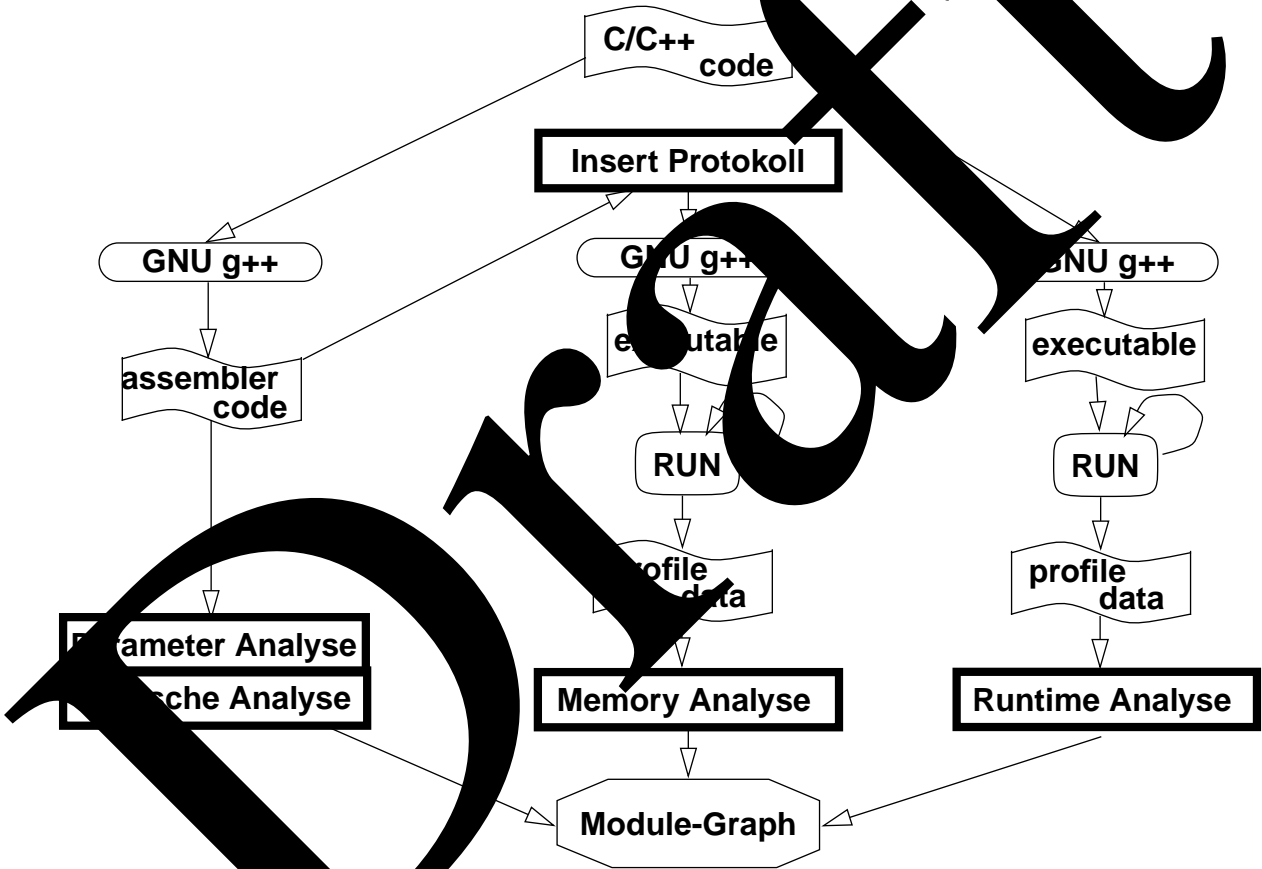


Abbildung 3-11 Spezifikationsanalyse

Nachdem der Optimierungsgesichtspunkt und die Zielarchitektur festgelegt sind stellt sich die Frage nach einer Partitionierungsstrategie. Bisherige Ansätze basieren auf einer initialen Systemimplementierung in einer Technologie¹. Dann werden Teile aufgrund von Simulationsdaten, Entwurfsvorgaben oder Designererfahrungen in die jeweils andere Technologie verschoben. Es kann leicht bemerkt werden, daß Simulationszeit sehr teuer ist und daß viele Systemspezifika-

1. Hardware oder Software

tionen keine speziellen Vorgaben (z.B. bezüglich des Zeitverhaltens) enthalten. Daher wird hier eine **Spezifikationsanalyse** (Abb. 3-11) auf Basis einer SW- Implementation zur Generierung von Partitionierungsinformation verwendet

Dieser erste Codesign-Schritt gliedert sich in vier verschiedene Phasen, die kurz erläutert werden.

- **Statische Spezifikations-Analyse:** Zunächst wird die Systemspezifikation, ein C/ C++ -Programm, für den Standardprozessor der Zielarchitektur in Assemblercode umgesetzt werden. Dazu wird ein Standard-Compiler [4] verwendet. Aus dieser nun Architektur-spezifischen Spezifikation wird eine untere Schranke für die Abarbeitungsdauer einer Software-Implementation berechnet. Basis für die Berechnung ist der Instruktionssatz des Zielprozessors, die Abarbeitungsdauer der einzelnen Instruktionen und das Verhalten der Instruktionspipeline. Diese Berechnung ist wesentlich genauer und damit aussagekräftiger, als eine Aussage über die durchschnittliche Abarbeitungsdauer aller Befehle. Weiterhin werden statistische Informationen über Instruktionen, die sich besonders gut in Hardware implementieren lassen, generiert. Zu diesen Instruktionen gehören Sprung- und Bit-Level-Operationen.
- **Runtime Spezifikations-Analyse:** Nicht alle Aspekte, die von Bedeutung sind, lassen sich durch statische Analyse des Assembler-Codes generieren. Insbesondere die Iterationshäufigkeit von Schleifen ist aufgrund von Datenabhängigkeiten nur während der Systemlaufzeit ermittelbar. Daher wird in dieser Analysephase ein ausführbares Programm erzeugt. Bei der Ausführung können Profiling-Informationen gewonnen werden. Dies geschieht für viele verschiedene Eingabedaten. Aus den ermittelten Ergebnissen wird der Erwartungswert und die Standardabweichung berechnet. Dies führt zu einer ersten Bewertung des durchschnittlichen Laufzeitverhaltens. Der „Worst-Case“ wird betrachtet, da keine Entwurfsvorgaben existieren. Weiterhin ist für Performanz-Steigerung das durchschnittliche Verhalten von besonderer Bedeutung. Auf diese Weise wird absolute, maximale und mittlere Laufzeitverhalten des betrachteten Systems ermittelt. Die Betrachtungsgranularität entspricht der kleinsten Einheit (Modul), die von einer Technologie in die andere verschoben werden kann. Aus technischen Gründen wurden diese Einheit als C-Funktion definiert. Damit bleibt dem Benutzer die Freiheit für manuelle Eingriffe erhalten.
- **Parameteranalyse:** Neben dem Laufzeitverhalten ist auch die Schnittstelle zwischen HW und SW zu betrachten. Einen Aspekt stellen die Parameter einer Funktion (Moduls) dar. Diese lassen über die Schnittstelle transferiert werden. Daher wird die Anzahl und die Größe dieser Parameter in einer weiteren Analysephase ermittelt.
- **Memory Analyse:** Häufig werden Daten über eine Referenz (Pointer) angesprochen. Der Pointer an sich enthält keine Information über die Größe der zu transportierenden Daten und die Häufigkeit der Zugriffe. Dies ist für die Kommunikation zwischen den Implementationsteilen von großer Bedeutung und kann nur zur Ausführungszeit bestimmt werden. Daher wird in dieser Analyse Phase eine modifizierte ausführbare Version des betrachteten Systems erzeugt, die ein Protokoll aller Speicherzugriffe generiert. Daraus läßt sich die Größe des Speichers, auf den zugegriffen wurde und die Häufigkeit der Zugriffe ermitteln.

Die Korrelation dieser Aspekte führt zu einer Bewertung der betrachteten Moduln bezüglich ihrer Eignung zur Implementation in der einen oder anderen Technologie. Die Auswertung dieser Daten ermöglicht die automatische Partitionierung des Entwurfsraumes unter dem Hardware/Software-Gesichtspunkt und damit eine wesentliche Beschleunigung des Entwurfsvorgangs. Eine Formalisierung der Spezifikationsanalyse und weitere Einzelheiten sind in [5] zu finden.

3.3.2 Partitionierung des Beispiels

Beispielhaft sollen die Ergebnisse der Anwendung dieser HW/SW-Partitionierungsmethode anhand des Fuzzy-Inferenz-Applikation (Kapitel II.) demonstriert werden. Zunächst wird die gesamte Fuzzy-Inferenz-Applikation allen vier Phasen der Spezifikationsanalyse unterzogen. Die Statische Analyse (Phase 1) generiert den Modulgraph. Dieser enthält einen Knoten für jedes Modul (Abb. 3-12).

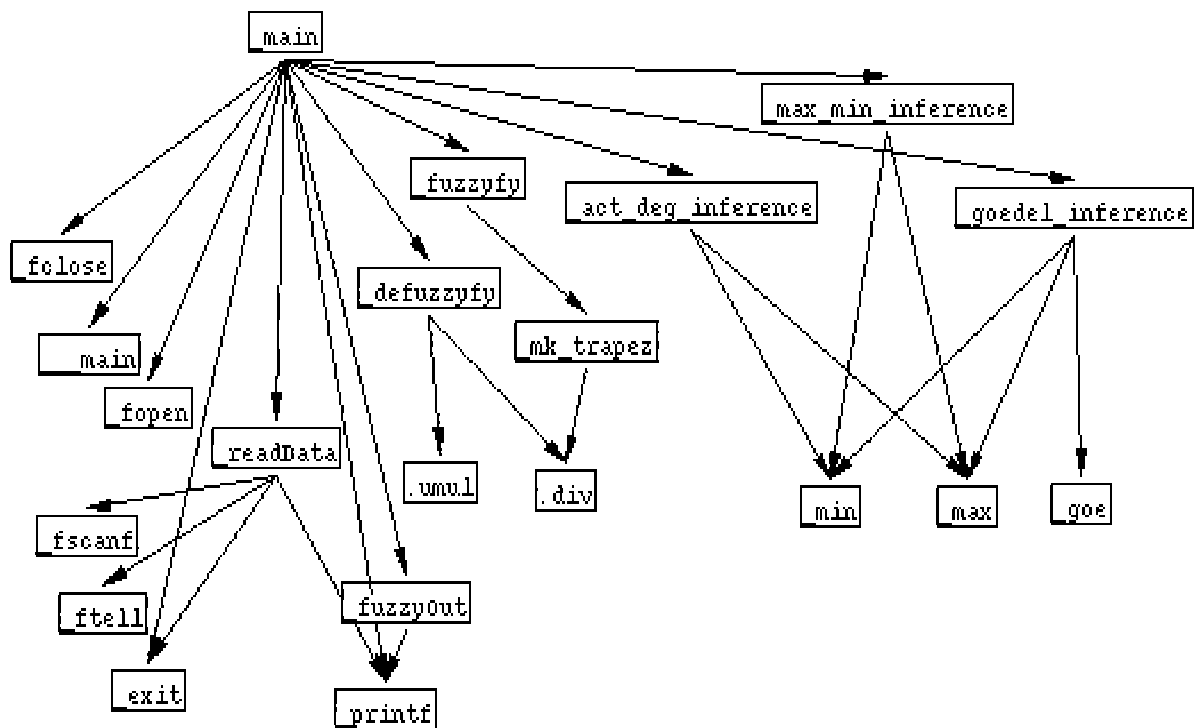


Abbildung 3-12 Modulgraph der Fuzzy-Inferenz-Applikation

Die Analyseergebnisse sind in Tabelle 3-1 zusammengefaßt. Aufgrund der schlechten Ergebnis-

Tabelle 3-1 Spezifikationsanalyse der Fuzzy-Inferenz-Applikation

Analyse-Phase	Module, geeignet für HW-Implementation
SA ^a	12
DA ^b	5
SA && DA	4
PA ^c	0
SA && DA&& PA	0

- a. Statische Spezifikations-Analyse
- b. Dynamische Spezifikations-Analyse
- c. Parameter-Analyse

se der Parameter-Analyse wird eine leere HW-Partition generiert. Offensichtlich liegt der Grund dafür in der hohen Konnektivität des Modulgraphen.

Weiterhin wurde die HW/SW-Partitionierungsmethode im Hinblick auf die Parallelisierung der Fuzzy-Inferenz-Applikation für einen Workstation-Cluster angewandt. Der Modulgraph (Abb. 3-13) weist eine wesentlich geringere Konnektivität auf, insbesondere im dem Bereich der rechenintensiven Module. Für jedes Modul ist die Eignung für eine Implementation in Hardware

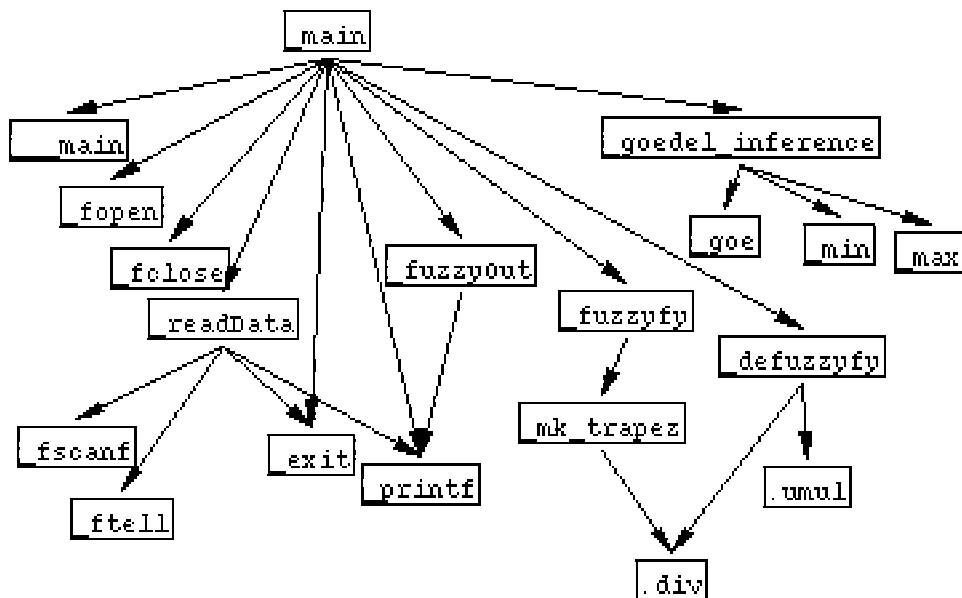


Abbildung 3-13 Modulgraph der Goedel-Inferenz Partition

resultierend aus der jeweiligen Analysephase angegeben. Die Ergebnisse der einzelnen Analyse-

phasen sind in Tabelle 3-2 angegeben. An dieser Stelle ist also eine sinnvolle HW/SW-Partitio-

Tabelle 3-2 Spezifikationsanalyse der Goedel-Inferenz

Analyse-Phase	Module, geeignet für HW-Implementation
SA ^a	12
DA ^b	5
SA && DA	4
PA ^c	3
SA && DA && PA	3

a. Statische Spezifikations-Analyse

b. Dynamische Spezifikations-Analyse

c. Parameter-Analyse

nierung möglich. Eine Bewertung der einzelnen Module gemäß der verwendeten Kostenfunktion ist in Tabelle 3-3 aufgelistet. Bei den nicht explizit angegebenen Modulen handelt es sich um Bibliotheksfunktionen, die nicht im Goedel-Code verfügbar sind. Da diese in hohem Maße Betriebssystem-spezifisch sind, ist ihre Eignung für Hardware-Implementation nicht gegeben. Die für Hardware-Implementation geeigneten Module bilden eine in sich nahezu abge-

Tabelle 3-3 Ergebnisse der Spezifikationsanalyse

Modul	SA ^a	DA ^b	PA ^c	Ergebnis
mk_datafl	0.86	9.4	26	0
_trapez	4	9	4	0
min	1	5	2	123.665
max	4.6	7.75	1	26.795
goe	5.5	104.58	1	579.373
goedel_inferenz	2.8	310.64	5	178.307
main	18	18.02	47	0

a. Statische Spezifikations-Analyse

b. Dynamische Spezifikations-Analyse

c. Zusammenfassung von Parameter-Analyse und Memory-Analyse unter dem Gesichtspunkt des Modul Interfaces.

schlossene Einheit. Eine nichthierarchische Implementation des Moduls “goedel_inferenz” würde die Module “min”, “max” und “goe” expandieren und direkt einschließen. Somit wird die Berücksichtigung des Modul Interfaces sehr deutlich. Manuelle Analysen haben gezeigt, daß das Modul “goedel_inferenz” ein wesentlicher Rechenkern dieser Applikation ist. Damit stimmt

das Ergebnis der automatischen und manuellen Hardware/Software Partitionierung in bemerkenswerter Weise überein.

3.4 Schlußfolgerung

Es wurden die Partitionierung von

- Software in sequentielle/parallele Teile,
- Spezifikationen in Hardware/Software Teile und
- Hardware in sequentielle/parallele Teile

untersucht. Es konnte gezeigt werden, daß eine Kombination aller genannten Partitionierung innerhalb des Designflusses möglich und sinnvoll ist. Dabei stellt sich die Frage nach Automatisierbarkeit einer solchen Kombination von Partitionierung zu einer universellen Implementationsmethode:

Die entwickelten Partitionierungsmethoden lassen sich durch die verwendeten Datenstrukturen und die intendierten Zielarchitekturen klassifizieren. Die Datenstrukturen lassen man verschiedenen Ebenen des Entwurfsprozesses zuordnen. Die Zielarchitekturen sind unterschiedlich komplex und variieren in ihrer Leistungsfähigkeit: Eine Gegenüberstellung ist in Tabelle 3-4 angegeben.

Tabelle 3-4 Einordnung von Partitionierungsmethoden

Partitionierung	Datenstruktur	Entwurfsebene	Zielarchitektur
SW: seq./par.	Datenflußgraph	Verhalten	Workstation-Cluster
HW/SW	Spezifikation	System	Workstation + ASIC
HW: seq./par.	Gleichungen	Struktur	ASIC

Ordnung der Partitionierungen gemäß dem Abstraktionsgrad der verwendeten Datenstruktur, so ergibt sich die folgende Anwendungsreihenfolge:

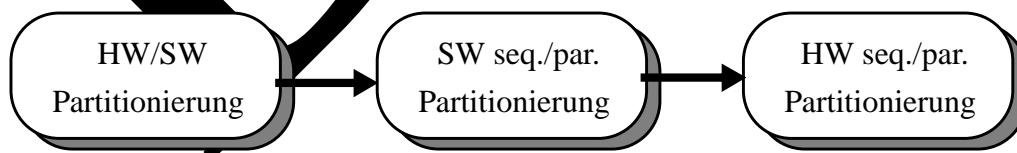


Abbildung 3-14 Anordnung der Partitionierungen gemäß der verwendeten Datenstrukturen

Betrachtet man die erzielten Ergebnisse nach der Anwendung der Partitionierungen in dieser Reihenfolge auf die Fuzzy-Inferenz-Applikation, so stellt sich heraus, daß

- die automatische HW/SW-Partitionierung zu einer leeren HW-Partition führt,

- die SW seq./par. Partitionierung keine sinnvolle Anwendung aufgrund zu geringer Granularität findet und
- die HW seq./par. Partitionierung unbeeinflusst bleibt.

Ordnet man jedoch die Partitionierungen gemäß der Komplexität der Zielarchitekturen, so ergibt sich die folgende Anwendungsreihenfolge:

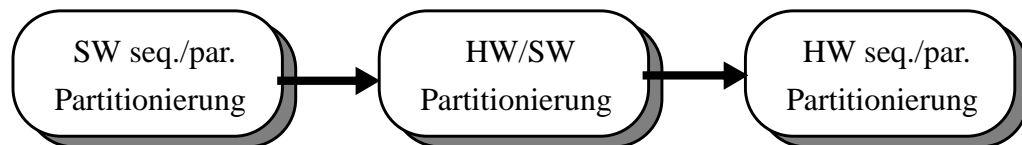


Abbildung 3-15 Anordnung der Partitionierungen gemäß der intendierten Zielarchitekturen

Die nun erzielten Ergebnisse können zu einer sinnvollen Systemimplementierung zusammengefügt werden. Eine Automatisierung der Multikriterien-Partitionierung sollte dabei mit Rücksicht auf die Komplexität der gerade betrachteten Systemarchitekturen durchgeführt werden. Dabei ist die komplexeste Ebene zuerst betrachten, was einem klassischen „Top-Down-Entwurf“ entspricht.

Es fällt auf, daß der Komplexitätsgrad der verwendeten Datenstruktur in der intendierten Zielarchitektur für jeweils eine Partitionierungsmethode zu groß und ungleichmäßig ist zu prüfen, ob entsprechende Informationen nicht auch auf höheren Datenstruktur-Ebenen gewonnen bzw. verwaltet werden können.

Dieser Automatisierungsansatz ist nur auf weitere, signifikante Anwendungsbeispiele anzuwenden. Bestätigen sich die hier gewonnenen Erkenntnisse, können sie in die weitere Automatisierung des Systementwurfes einbezogen werden.

Referenzen

- [1] R. Gupta and G. DeMicheli. System synthesis via hardware - software co - design. Technical Report Techn. Rep. SFB - 358 - B2 - 5/94, Stanford, 1992.
- [2] J. Holtman and R. Ernst. Speculative computation for coprocessor synthesis. In *Proc. of the 1st Workshop on CAD*, 1993.
- [3] F. Hörsing and N. Mellergård and J. Stranstrup. *The Priority Queue as an Example of Hardware/Software Codesign*. In *Proc. of the 3rd International Workshop on HW/SW Codesign*, Cambridge, 1994.
- [4] R. Stallmann. *Compiling and Porting GNU CC*. Free Software Foundation Inc., December 1992.
- [5] W. Hardt and R. Camposano. Specification analysis for hw/sw- partitioning. Technical Report Techn. Rep. SFB - 358 - B2 - 5/94, University of Paderborn, University of Dresden, 1994.
- [6] Schubert,A.; Merker,R.; Schreiber,H.: Systematic Generation of a Variety of Processor Arrays. Jesshope,Chr.; Jossifov,V.; Wilhelmi,W. (Herausgeber), *Mathematical Research, Parcella '94*,Volume 81, Akademie-Verlag 1994,S.267-276.

- [7] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company Inc., 1990.
- [8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Forth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 1–12, 1993.
- [9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.
- [10] W. Pugh. The Omega Test: a fast and practical integer programming algorithm test for dependence analysis. In *Proceedings Supercomputing '91*, pages 4–17, 1991.
- [11] L. G. Valiant. A Bridging Model for Parallal Computation. In *Communications of the ACM, Vol.33, No.8*, pages 103–111, 1990.

Draft

IV. Synthese

In der vorangegangenen Spezifikations-Analyse wurde von der einer Workstation zugeordneten Programmspezifikation der rechenintensive und damit auch die Hardwareimplementierung geeignete Teil bestimmt. In diesem Kapitel wird diese Teilspezifikation synthetisiert zum einen mit dem Ziel der Erzeugung einer sequentiellen Hardware und zum anderen mit dem Ziel der Erzeugung eines massiv parallelen Rechenfeldes. An Hand des Beispiels der Prozy-Inferenz sollen die Lösungen einem Vergleich unterzogen und gegebenenfalls Schlußfolgerungen für eine allgemeine Herangehensweise bei der Entscheidung für eine der Lösungen entwickelt werden.

4.1 Synthese als sequentielle Hardware

Die mittels Spezifikations-Analyse ermittelte Hardware-Partition wird zunächst synthetisiert und dann über eine Interface-HW mit vorgegebene Systemarchitektur angebunden. Generell gibt es viele Wege der Hardware-Synthese. Unterschiede sind in der Implementationsstrategie festzustellen. Verfahren verwenden zum Aufbau der Kommunikationswege Multiplexer, andere Verfahren basieren auf VLIW oder parallelisierenden Ansätzen. Diese Verfahren sind in verschiedenen Systemen automatisiert und können ggf. gegeneinander ausgetauscht werden. PMOSS (Paderborner Modularer Synthese System) [1] generiert ausgehend von C/C++ oder einer VHDL Spezifikation eine Multiplexer-basierte sequentielle Hardware. Zunächst wird die Systemspezifikation in eine Graph-basierte interne Darstellung (CDFG) konvertiert. Danach werden die klassischen Algorithmen der High-Level-Synthese angewendet: List-Scheduling, Register-Allokation und Functional-Unit-Binding. Die gesamte Syntheseinformation steht ab dieser abstrakten CDFG-Ebene zur Verfügung und ermöglicht den gezielten Einsatz von Optimierungstransformationen. Als nächster Entwurfsschritt findet eine Konvertierung der gewonnenen Daten in eine Steuerwerks/ Datenpfad Struktur (CDP) statt. In diesem Schritt werden sowohl die notwendigen Verbindungen zwischen Registern und Operations-Einheiten eingefügt, als auch die Signale zur Steuerung der Multiplexer generiert. Durch eine VHDL-Ausgabe der Synthese-Ergebnisse können kommerzielle Logik-Optimierer (Synopsys) angewandt werden. Einzelheiten über diesen Synthese-Ablauf sind in [2] zu finden. Auf das Beispielsystem angewandt generiert dieser Hardware-Synthese Prozeß ein Steuerwerk und einen Datenpfad. Diese sind über Steuer- und Statusleitungen verbunden.

Die Integration der Hardware- und der Software- Partition in ein System erfordert ein leistungsfähiges Interface. Fallstudien [RepInterf] haben gezeigt, daß unter Performanz- Gesichtspunkt

ten eine enge Kopplung von Standard- und Spezialhardware von großem Vorteil ist. Viele Prozessoren verfügen über eine oder mehrere Schnittstellen zur Adaption von Coprozessoren (Intel X86, MC68000, MC88000, Sparc, etc.). Da der letztgenannte Prozessor über zwei Schnittstellen verfügt, wurde er für die weiteren Implementationen ausgewählt. Zur Anbindung der synthetisierten Spezial-Hardware wurde für den Hardwareteil des Interface eine dreistufige Instruktions-Pipeline aufgebaut. Diese Pipeline basiert auf der ersten Pipeline-Stufe des Sparc-Prozessors. Dieser übernimmt alle Berechnungen zum Zugriff auf den dynamischen Speicher. Diese Besonderheit ist von hoher Wichtigkeit. Zum einen berechnet der Prozessor sehr effizient die virtuellen Adressen, da eine spezielle Berechnungs-Unit im Prozessor integriert ist, zum anderen ermöglicht diese Verzahnung der Instruktions-Pipeline eine parallele Aktivierung von Prozessor und Spezial-Hardware.

Weiterhin wurde eine gezielte Aufteilung der Spezial-Hardware in Teilkomponenten, die separat angesprochen werden können, ermöglicht. Für dieses Interface wurde von uns eine VHDL-Spezifikation entworfen und ein separater (Logik-) Synthesewerkzeug durchgeführt. Dieses Interface ist von der zu integrierenden Spezial-Hardware unabhängig, so daß dieser Entwurf nur einmal pro Prozessor durchzuführen ist.

Für eine Bewertung des generierten Designs können drei Aspekte unterschieden werden:

- Lesen der Eingabe-Daten
- Dauer eines einzelnen Berechnungsdurchlaufs und
- Abarbeitungsdauer insgesamt.

Eine (nicht optimierte) Abschätzung des Ladevorgangs der Eingabedaten ergibt sich aus der Anzahl der zu ladenden Daten und der Dauer eines einzelnen Ladevorgangs.

Tabelle 1: Daten zum Lesen der Eingabe

Anzahl Eingabe-Daten	CPU-Zyklen pro Ladevorgang	Gesamt
54	2	108

Eine Optimierung kann entsprechend der Datenabhängigkeiten durchgeführt werden. Die Dauer eines einzelnen Berechnungsdurchlaufs entspricht der Ausführung einer Schleifen-Iteration. An dieser Stelle können verschiedene Schwerpunkte betont werden, z.B. wenige versus schnelle Taktzyklen. In der Tabelle sind verschiedene Varianten, die sich durch HL-Transformationen ergeben, angegeben.

Tabelle 4-1 Syntheseergebnisse bei sequentieller Entwurfsstruktur

Version	Datenpfad			Steuerwerk			Gesamt		
	Comb ^a .	Ncomb ^b .	Geamt	Com.	Ncomb.	Gesamt	Com.	Ncomb.	Gesamt
drei geschachtelte Schleifen	287	280	567	166	367	533	455	647	1102
äußerste Schleife expandiert	297	240	537	445	398	843	720	638	1358
alle Schleifen expandiert	102	120	220	106	120	-	-	-	-

- a. Fläche für Kombinatorik
- b. Fläche für nicht kombinatorische Module

Die durchgeführte Synthese als Multiplexer-basierter Entwurf erfordert Operationseinheiten, die mehrere Zyklen pro Operationsabarbeitung benötigt und für Ein-Zyklus-Operationseinheiten die direkte Hintereinanderausführung (chaining). Dabei lässt sich heraus, dass der Datenpfad deutlich verkleinert werden kann, indem alle Schleifen expandiert werden. Dies ist zu erklären, da gerade dann Multicycling und Chaining extensiv eingesetzt werden. Die Größe des Steuerwerks erscheint hingegen sehr groß. Hier ist zu prüfen, inwieweit ungünstige Codierungen des Steuerwerksautomaten diese Größe verursachen. In Tabelle 4-2 sind einige abstraktere Entwurfskriterien zusammengestellt. Es wird sehr deutlich, daß die Expansion der Schleifen zu einer enorm großen Anzahl von Zuständen für die geringe Verzögerungszeit der ersten beiden Implementationsvarianten zu einer sehr hohen Taktrate (100 MHz).

Tabelle 4-2 Abstrakte Entwurfsergebnisse

Version	Zustände des Steuerwerks	Verzögerungszeit des Datenpfades	Verzögerungszeit des Steuerwerks	Verzögerungszeit des Geamtentwurfs
drei geschachtelte Schleifen	18	7.45	2.62	8.29
äußerste Schleife expandiert	64	6.90	2.62	10.15
alle Schleifen expandiert	738	6.90	7.21	-

Referenzen

[1] R. Genevriere and A. Hoffmann. Pmoss - a modular synthesis and hw/sw-codesign system. Technical Report SFB - 358 - B2 - 2/94, March 1994.

- [2] H.-J. Eikerling, R. Genevriere, W. Hardt, A. Hoffmann, K. Feske, G. Franke, M. Koegst, and H.-G. Martin. Flexible hw synthesis and optimization by incremental design modification. Technical Report Techn. Rep. SFB - 358 - B2 - 6/94, University of Paderborn, University of Dresden, 1994.

4.2 Synthese als paralleles Rechenfeld

Die durch die vorangegangene Spezifikationsanalyse ermittelte Hardware-Partition wird im weiteren bezüglich ihrer Realisierbarkeit auf massiv parallelen Rechenfeldern untersucht. Ist eine Realisierung möglich, erfolgt mittels eines Entwurfssystems [1, 5] der Rechenfeldentwurf für die vorgegebene Algorithmusspezifikation.

Als initiale Algorithmusspezifikationen für das Entwurfssystem können Systeme von affinen Rekurrenzgleichungen der Form $y_i(f_i(I)) = F_i^r(\dots, v_k(I), \dots), I \in \mathbf{I}_r \subseteq \mathbb{R}^n$ behandelt werden. Hierbei repräsentiert $\mathbf{I}_r (r = 1, \dots, m)$ die Menge der ganzzahligen Indizespunkte eines konvexen n -dimensionalen Rechnungsraumes, die Indizes $v_k(I)$ bzw. $g_k(I)$ der Variablen v_k und y_i sind affine Funktionen von I , und F_i^r beschreibt eine beliebige Funktion, die auch in Form von eingebetteten globalen Operatoren (z.B. \sum, \prod) vorliegen kann. Daten, die auf der rechten und linken Seite der Rekurrenzgleichungen auftreten, werden als iterative Daten bezeichnet, und Daten, die entweder auf der linken oder auf der rechten Seite der Gleichungen auftreten, als globale Daten. Das System von Rekurrenzgleichungen wird durch einen auf dem konvexen Indexbereich $\mathbf{I} = \cup \mathbf{I}_r$ definierten Abhängigkeitsgraphen beschrieben, der die Abhängigkeiten der Daten in Form von Kanten explizit darstellt. Die Rechnungsräume \mathbf{I}_r sind dichte Polyeder und müssen nicht disjunkt sein.

Dem Ziel der Erzeugung einer Variantenvielfalt folgend wird hier im Vergleich zu existierenden Systemen eine modifizierte Vorgehensweise gewählt. Iterative Daten werden in einen gemeinsamen Index eingebettet und realisiert. Die Verbreitungsrichtungen der globalen Daten werden in dieser Hinsicht nicht vorgelegt, um die Freiheitsgrade bei der Lokalisierung der globalen Daten im weiteren Entwurfsschritt zur Erzeugung einer Variantenvielfalt ausnutzen zu können. Die globalen Daten sind damit den Knoten im Abhängigkeitsgraphen zugeordnet, zu deren Rechnung sie benötigt werden (als globale Eingabedaten) bzw. infolge deren Berechnung sie entstehen (als globale Ausgabedaten). Es entsteht ein Abhängigkeitsgraph, der nur die Datenabhängigkeitsvektoren der iterativen Daten und damit die gesamte Klasse von Abhängigkeitsgraphen mit lokalen Datenabhängigkeitsvektoren der betrachteten Algorithmusspezifikation repräsentiert.

Zur Erzeugung von Rechenfeldern voller Größe (d.h., das Rechenfeld ist abhängig von der Größe und Dimension des Indexraumes des Algorithmus) der Dimension $n-1$ werden, wie in derartigen Systemen üblich, Allokierungs- und Scheduling-Funktionen eingesetzt, die den Rechnungen der Knoten des Abhängigkeitsgraphen Ort und Zeit der Ausführung im Rechenfeld zuordnen. Wir verwenden affine Allokierungsfunktionen, bestehend aus einer linearen Transformation \mathbf{S} und einem Verschiebungsvektor \mathbf{g}_δ . Um den Suchraum für die lineare Transformation \mathbf{S} einzuschränken und gleichzeitig dem Ziel der Variantenvielfalt Rechnung zu tragen, wird die Methode von [7] verwendet und zulässige Verbindungstopologien des Rechenfeldes einschließlich der Richtungen der einzelnen Verbindungen vorgegeben. Da wir im Unterschied zu [7] affine Allokierungsfunktionen anstelle linearer verwenden, können wir so für jeden Rechnungsraum individuelle Verschiebungsvektoren einführen. Durch dieses Vorgehen erhöht sich die Anzahl von möglichen Verbindungstopologien, für die eine Transformation \mathbf{S} bestimmt werden kann, und Rechenfeldtopologien, die gegebenenfalls mit linearen

Allokierungsfunktionen nicht ermittelt worden wären, sind eingeschlossen.

Für jede Allokierungsfunktion wird eine affine Schedulingfunktion analog zu [6] erzeugt, bestehend aus einem Schedulingvektor und einem für jeden Rechenraum separaten Verschiebungsvektor. Zur Berechnung der Schedulingfunktion wird ein lineares Optimierungsproblem gelöst.

An dieser Stelle können nun, bedingt durch unseren Ansatz, die Freiheitsgrade für die Propagierungsvektoren der globalen Daten genutzt werden, um verschiedenartige Rechenfeldlösungen zu erzeugen. Unter den möglichen Propagierungsvektoren der Nullraumpropagation werden solche ausgewählt, die der Kausalitätsbedingung und den entsprechenden Topologieforderungen der Allokierungsfunktion genügen.

Um nach Anwendung der Allokierungsfunktion stets eine Initialisierung der Daten an den Rändern des Rechenfeldes vornehmen zu können, entwickelten wir eine Methode [10], die im Gegensatz zu [8] davon ausgeht, daß ein Datentyp nur einmal gegebenfalls auch von unterschiedlichen Rändern in das Prozessorfeld ein- bzw. ausgelesen werden kann, wobei die Auswahl der zu diesen Zweck notwendigen Propagierungsvektoren der Initialisierung mit dem Ziel einer minimalen Erhöhung der Taktanzahl unter Einhaltung der Kausalitätsbedingung und der Topologieforderung erfolgt.

Zur Steuerung unterschiedlicher Funktionen in den Prozessoren des Rechenfeldes werden Kontrollsignale eingeführt und entsprechend [9] so ausgewählt, daß die Anzahl minimal ist.

Um das Ziel der Implementierung der entworfenen Rechenfelder voller Größe auf einem Schaltkreis realisieren zu können, ist ein Adaptionssystem entwickelt worden, das das Rechenfeld an vorgegebene Hardwarerestriktionen anpaßt. Das Adaptionssystem enthält Transformationen der entworfenen Struktur mit dem Ziel der Anpassung an die zur Verfügung stehende Fläche zur Integration der Schaltung, an die Anzahl der realisierbaren I/O-Kanäle zwischen Schaltkreis und Peripherie sowie an die Formgebung nach einem ein- oder zweidimensionalen Layout der Schaltung [14].

Die entwickelte Entwurfstrajektorie umfaßt existierende Methoden, die für unsere Herangehensweise modifiziert wurden, sowie darüber hinaus neue Strategien bezüglich der Erzeugung einer Variantenvielfalt, der Initialisierung der Daten an den Randprozessoren des Rechenfeldes und des Adaptionssystems.

Im weiteren wird der Entwurfprozeß für die Algorithmenspezifikation in Form der C-Funktion „`single_inference`“ beschrieben, die während der Spezifikationsanalyse als für eine Realisierung auf der Hardware geeignet erkannt wurde. Diese Funktion ist in Abb. 4-1 dargestellt. Den Ausgangspunkt für den automatisierten Entwurf paralleler Rechenfelder bildet die Beschreibung des zu realisierenden Algorithmens in einer Single Assignment Form (SAF). Diese Beschreibung erfolgt in der Sprache „`DALIS`“, die an die Sprache C angelehnt ist und als Eingabesprache für den Entwurf von Rechenfeldern dient [3].

Im folgenden wird zunächst der Schritt zur Ableitung der Single Assignment Form aus der Algorithmenspezifikation an Hand des Beispieles beschrieben. Diese Ableitung erfolgt bisher heuristisch durch den Benutzer, eine automatische Prozedur hierfür ist bei einer Weiterführung der Arbeiten vorgesehen. Danach wird für das Beispiel die automatisierte Entwurfstrajektorie bis zum massiv parallelen Rechenfeld voller Größe beschrieben.

```

1 void goedel_inference (int amf[RULES][SIZE], int bmf[RULES][SIZE],
2                       int xmf[SIZE], int ymf[SIZE])
3 {
4   int l, k, i;
5   int c;
6
7   for( l = 0; l < SIZE; l++ )
8     for( k = 0; k < SIZE; k++ ) {
9       c = goe( amf[0][k], bmf[0][l] );
10
11      for( i = 1; i < RULES; i++ )
12        c = min( c, goe( amf[i][k], bmf[i][l] ) );
13
14      if( k == 0 )
15        ymf[l] = min( xmf[k], c );
16      else
17        ymf[l] = max( ymf[l], min( xmf[k], c ) );
18    }
19 }

```

Abbildung 4-1 Die C-Funktion „goedel_inference“

Ableitung der Single Assignment Form

Die Funktion „goedel_inference“ enthält drei geschichtete „for“-Schleifen, deren Zuweisungen sequentiell ausgeführt werden. Diese Notation läßt sich in eine äquivalente Notation mit perfekt geschachtelten Schleifen überführen. Zu diesem Zweck werden die Anweisungen der Zeilen 9 sowie 14 bis 17 in die innere „for“-Schleife überführt. Dies geschieht dadurch, daß der Bereich des Index i der inneren Schleife um l erweitert 0 erweitert und somit die Anweisung aus Zeile 14 als if/else-Konstrukt in diese Schleife integriert werden kann. Die Ausführung der Anweisungen der Zeilen 12 bis 17 findet bei der sequentiellen Berechnung zeitlich nach der „for“-Schleife über dem Index k statt. Bei perfekt geschachtelten Schleifen entspricht dies der Einführung der Bedingung $if(i = RULES-1)$.

Anschließend werden die Argumente der Funktion „goedel_inference“ und ihre lokalen Variablen bezüglich ihrer Eigenschaften nach dem Ziel der Erzeugung einer Single Assignment Form untersucht. Variablen amf , bmf und xmf treten nur auf den rechten Seiten von Zuweisungen auf, d.h. beschrieben wird nur gelesen. Sie sind damit Eingabevariable. Im Gegensatz dazu werden die Variablen ymf und c sowohl gelesen als auch beschrieben und müssen deshalb weiter analysiert werden. Für die Variablen ymf und c ist es zur Realisierung einer SAF notwendig, je eine iterative Variable $R_{i,l}$ bzw. $R_{i,k}$ einzuführen mit den Indexfunktionen $f(I) = EI$ (E ist die Einheitsmatrix). Dann ergibt sich z.B. für Zeile 12 der C-Funktion von Abb. 4-1 folgende SAF:

$$R_{i[l,k,i]} = \min (R_{i[l,k,i-1]}, \text{goe} (amf[i,k], bmf[i,l]));$$

Die Abb. 4-2 veranschaulicht das Gesamtergebnis, d.h., die SAF der „goedel_inference“ in der Sprache „DAGS“, die den Ausgangspunkt für den Entwurf massiv paralleler Rechenfelder für die Fuzzy-Inferenz bildet.

```

type int size 4 unsigned;

int max (int a, int b)
  %{ return (a > b)? a : b; %}

int min (int a, int b)
  %{ return (a < b)? a : b; %}

int goe (int a, int b)
  %{ return (a <= b)? 15 : b; %}

parameter RULES = 10;
parameter SIZE = 4;

parallel goedel_inference (
input int amf[2],
input int bmf[2],
input int xmf[1],
output int ymf[1] )
{
forall (l; l >= 0 && l <= SIZE - 1)
  forall (k; k >= 0 && k <= SIZE - 1)
    forall (i; i >= 0 && i <= RULES - 1) {
      int R_k[3];
      int R_i[3];

      if (i == 0)
        R_i[l,k,i] = goe (amf[i,k], bmf[i,l]);
      else
        R_i[l,k,i] = min (R_i[l,k,i-1], goe (amf[i,k], bmf[i,l]));
      if (i == RULES - 1) {
        if (k == 0)
          R_k[l,k,i] = min (xmf[k], R_i[l,k,i]);
        else
          R_k[l,k,i] = max (R_k[l,k-1,i], min (xmf[k], R_i[l,k,i]));
        if (k == SIZE - 1)
          ymf[l] = R_k[l,k,i];
      }
    }
}

```

Abbildung 4-2 Die Gödel Inferenz in der Sprache „DAGS“

Entwurfstrategie

Aus der in der Sprache „DAGS“ spezifizierten *SAF* des Algorithmus wird in einem ersten Schritt ein System von Rekurrenzgleichungen erzeugt (Abb.4-3). In diesem Gleichungssystem ist jeder Gleichung ein Raum \mathbf{I}_0^u zugeordnet bestehend aus einer Menge von Indexpunkten

$$I = (l, k, i)^T \in \mathbb{Z}^3 .$$

$$\mathbf{i}_1: \quad R_1[l, k, i] = \text{goe}(\text{amf}[i, k], \text{bmf}[i, l]) \quad \forall I \in \mathbf{I}_1^1: i = 1 \wedge 1 \leq l, k \leq 4$$

$$\begin{aligned}
 \mathbf{i}_2: \quad R_1[l, k, i] &= \min(\text{goe}(\text{amf}[i, k], \text{bmf}[i, l]), R_1[l, k, i-1]) & \forall I \in \mathbf{I}_1^2: 1 \leq i \leq n \wedge 1 \leq l, k \leq 4 \\
 \mathbf{ii}_1: \quad R_2[l, k, i] &= \min(\text{ymf}[k], R_1[l, k, i-1]) & \forall I \in \mathbf{I}_2^1: i = n+1; k = 1; 1 \leq l \leq 4 \\
 \mathbf{ii}_2: \quad R_2[l, k, i] &= \max(\min(\text{ymf}[k], R_1[l, k, i-1]), R_2[l, k-1, i]) & \forall I \in \mathbf{I}_2^2: i = n+1 \wedge 2 \leq k \leq 4 \wedge 1 \leq l \leq 4 \\
 \mathbf{iii}: \quad \text{ymf}[l] &= R_2[l, k, i] & \forall I \in \mathbf{I}_3: i = n+1; k = 4; 1 \leq l \leq 4
 \end{aligned}$$

Abbildung 4-3 System von Rekurrenzgleichungen

Die Veranschaulichung der einzelnen Räume der Rekurrenzgleichungen aus Abb. 4-3 in einem Punktgitter des dreidimensionalen Euklidischen Raumes erfolgt in Abb. 4-4. Dieses Punktgitter stellt den gemeinsamen Indexbereich der betrachteten Algorithmenverifikation dar.

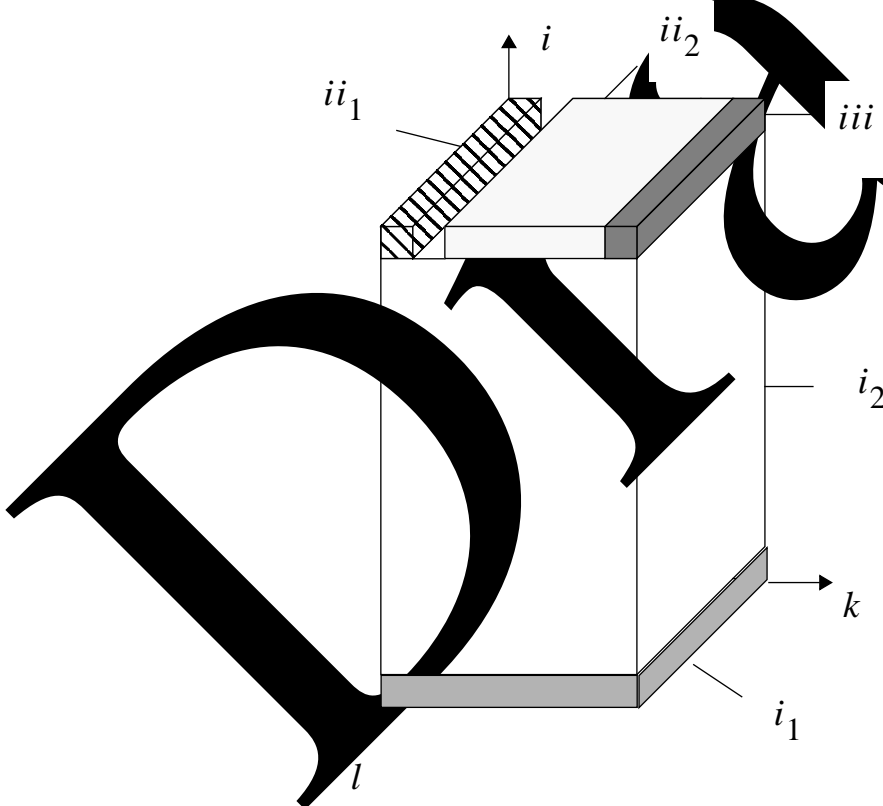
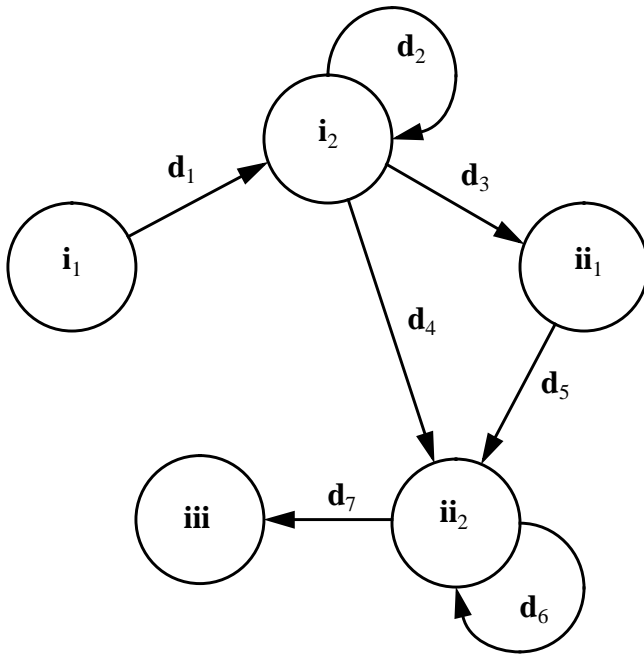


Abbildung 4-4 Gemeinsamer Indexbereich

In einem nächsten Schritt erfolgt für das System von Rekurrenzgleichungen eine Datenabhängigkeitsanalyse nach [12], um die Datenabhängigkeiten innerhalb und zwischen den einzelnen Rekurrenzgleichungen zu ermitteln. Als Ergebnis entsteht ein *reduzierter Abhängigkeitsgraph* (Abb.4-5). Jeder Knoten des reduzierten Abhängigkeitsgraphen repräsentiert eine Rekurrenz-

gleichung; die gerichteten Kanten stellen Datenabhängigkeiten dar und werden mit dem zugehörigen Abhängigkeitsvektor \mathbf{d} bewertet.



$$D = \begin{matrix} & \mathbf{d}_1 & \mathbf{d}_2 & \mathbf{d}_3 & \mathbf{d}_4 & \mathbf{d}_5 & \mathbf{d}_6 & \mathbf{d}_7 \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

$$C = \begin{matrix} \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} & \begin{matrix} \mathbf{i}_1 \\ \mathbf{i}_2 \\ \mathbf{ii}_2 \\ \mathbf{ii}_2 \\ \mathbf{iii} \end{matrix} \end{matrix}$$

Abbildung 4-5 Reduzierter Abhängigkeitsgraph

Die Abhängigkeitsmatrix D und die Inzidenzmatrix C stellen eine algebraische Repräsentation des reduzierten Abhängigkeitsgraphen (Abb. 4-5) dar.

Im weiteren ist es zweckmäßig, nur die *Rechnungsräume* zu betrachten, wobei als Rechenraum die Vereinigung aller Räume mit Rekurrenzgleichungen in denen ein und dieselbe iterative bzw. Ausgabevariable berechnet wird, bezeichnet wird. Für das Beispiel entstehen drei Rechenräume und der Speicher des ursprünglichen Schleifenprogrammes notwendig komplex sind. Man erhält $\mathbf{I}_1 = \mathbf{I}_1^1 \cup \mathbf{I}_1^2$, $\mathbf{I}_2 = \mathbf{I}_2^1 \cup \mathbf{I}_2^2$ und \mathbf{I}_3 . Die Beschränkung auf Rechenräume hat eine Zusammenfassung von Knoten im reduzierten Abhängigkeitsgraphen zur Folge und führt zu einem vereinfachten reduzierten Abhängigkeitsgraphen (Abb.4-6).

Nach Ermittlung der Datenabhängigkeiten kann durch Anwendung von affinen Transformationen ein Prozessor-Zeit-Raum für das massiv parallele Rechenfeld wie folgt bestimmt werden:

Durch eine affine Indextransformation wird jedem Indexpunkt mittels einer Scheduling-Funktion $t_\delta = \tau^T I + h_\delta$ ein Zeitpunkt der Abarbeitung sowie mittels Allokierungsfunktion $\mathbf{p}_\delta = \mathbf{S}I + \mathbf{g}_\delta$ ein Ort im Prozessorraum zugeordnet. Die verwendete Indextransformation besteht aus einer für alle Rechenräume gleichen Transformationsmatrix \mathbf{M} , sowie aus einem Verschiebungsvektors \mathbf{q} , der separat für die Rechenräume gewählt wird:

$$\begin{bmatrix} t_\delta \\ \mathbf{p}_\delta \end{bmatrix} = \mathbf{M}I + \mathbf{q}_\delta = \begin{bmatrix} \tau^T \\ \mathbf{S} \end{bmatrix} I + \begin{bmatrix} h_\delta \\ \mathbf{g}_\delta \end{bmatrix} \quad \begin{matrix} \mathbf{M} \in Z^{n \times n}; \mathbf{S} \in Z^{n-1 \times n}; t_\delta, h_\delta \in Z \\ \mathbf{g}_\delta, \mathbf{p}_\delta \in Z^{n-1}; \mathbf{q}_\delta, \tau \in Z^n \end{matrix} \quad \delta \in \{\mathbf{i}, \mathbf{ii}, \mathbf{iii}\}$$

Die Abbildung ist bijektiv und es gilt $\text{rang}(\mathbf{M}) = n$.

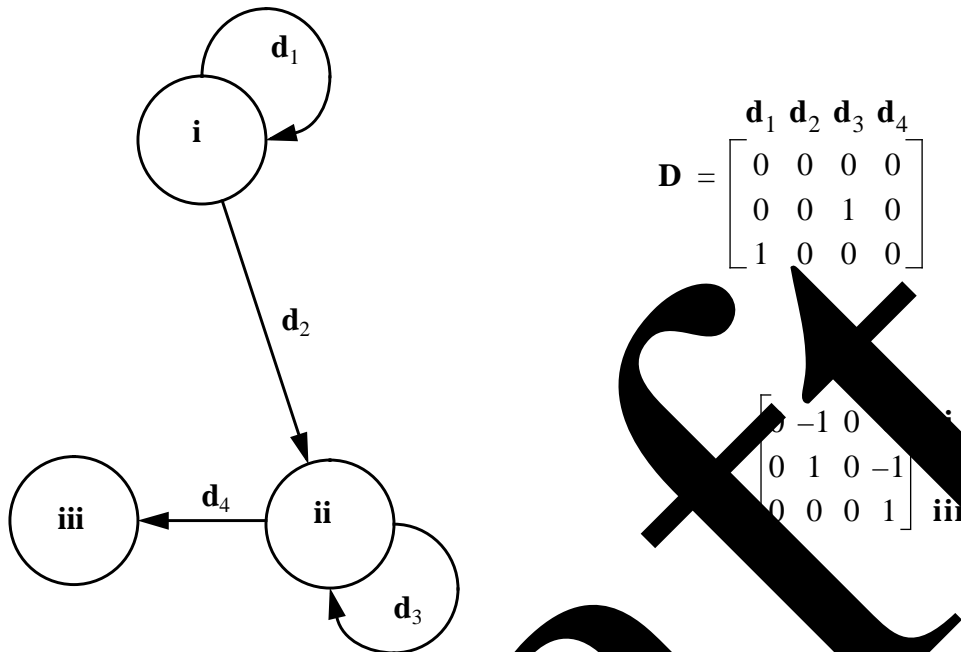


Abbildung 4-6 Vereinfachter reduzierter Abhängigkeitsgraph

Im Prozeß der Anwendung der beschriebenen Affinen Transformationen wird zunächst eine Vielzahl von Allokierungsfunktionen ermittelt. Hierzu dient als Voraussetzung zunächst die Erzeugung einer Menge von möglichen Verbindungstopologien \mathbf{V}_η des parallelen Rechenfeldes. Dabei wird für jeden Abhängigkeitsvektor \mathbf{d} des vereinfachten reduzierten Abhängigkeitsgraphen entsprechende Ebenen-Verbindungstopologie \mathbf{V}_η ein Verschiebungsvektor im Rechenfeld vorgegeben. Die zulässigen Verschiebungsvektoren werden in der Menge \mathfrak{S} zusammengefaßt und sind dadurch gekennzeichnet, daß nur Verbindungen zwischen Prozessoren, die eine vorgegebene Nachbarschaftsbeziehung erfüllen, erlaubt sind. Durch diese Restriktion ergibt man eine systolische Anordnung der Daten im Rechenfeld. Für jede Ebene-Verbindungstopologie wird jetzt das folgende Gleichungssystem gelöst:

$$\mathbf{V}_\eta = \mathbf{S}_\eta \mathbf{F} + \mathbf{G}_\eta \mathbf{C}, \quad (1)$$

wobei $\mathbf{G}_\eta = \begin{bmatrix} \mathbf{g}_1^\eta & \dots & \mathbf{g}_n^\eta \end{bmatrix}$

In unserem Beispiel gehen wir die Menge der zulässigen Verschiebungsvektoren im zweidimensionalen Rechenfeld mit $\mathfrak{S} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$ vor. Über diese Menge wird, unter Beach-

tung weiterer Kriterien, wie beispielsweise des Ausschließens kongruenter Topologien und Minimierung des Abstandes zwischen verschiedenen Rechenräumen, variiert und man

erhält z.B. folgende Verbindungstopologien: $\mathbf{V}_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$, $\mathbf{V}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$. Als Lösung

von (1) erhält man die Allokierungsfunktionen $\mathbf{S}_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $\mathbf{S}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, die nur aus den

Matrizen \mathbf{S}_η bestehen und keinen Verschiebungsvektor \mathbf{g}_δ^η enthalten.

Für jede Allokierungsfunktion wird jetzt mit Hilfe ganzzahliger linearer Programmierung [15] eine Scheduling-Funktion bestimmt, die die Abarbeitungszeit des Algorithmus minimiert, die Kausalitätsbedingung, d.h. $\tau^T \mathbf{d} > 0$, für alle Abhängigkeitsvektoren \mathbf{d} , sowie die Bedingung $\tau^T \mathbf{u} \neq 0$, wobei $\mathbf{S}\mathbf{u} = 0$ gilt, einhält. Resultat dieser Berechnung ist für \mathbf{S}_1 :

$\tau_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$, $\mathbf{h}_1 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$ sowie für \mathbf{S}_2 : $\tau_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$, $\mathbf{h}_2 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$. Aus der Vielzahl von Indextransforma-

tionen, die wir erhalten, wird diejenige ausgewählt, die ein Optimum hinsichtlich minimaler Prozessorzahl und minimaler Verarbeitungszeit darstellt. In unserem Fall ist das die durch $(\mathbf{S}_2, \tau_2, \mathbf{h}_2)$ gebildete Transformation.

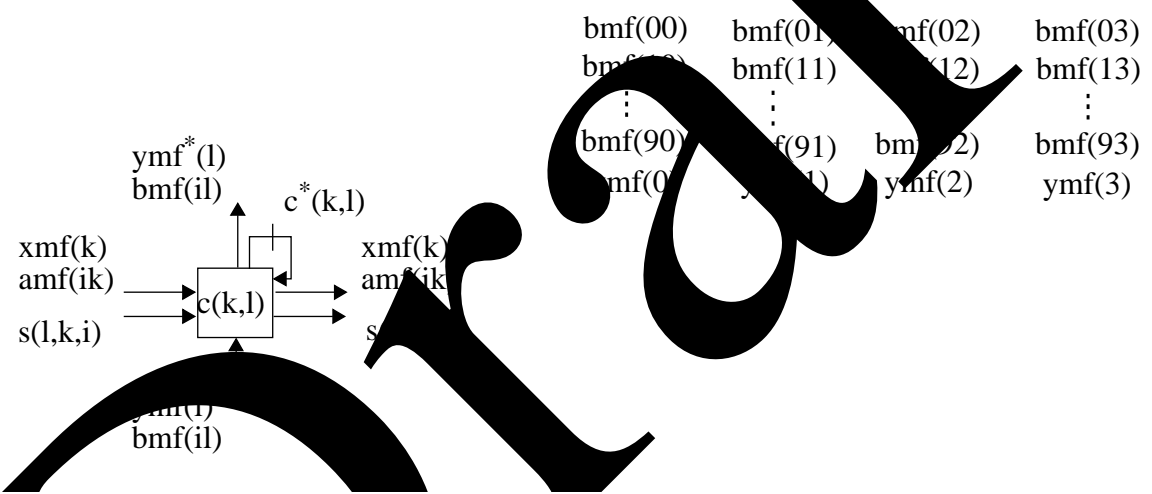
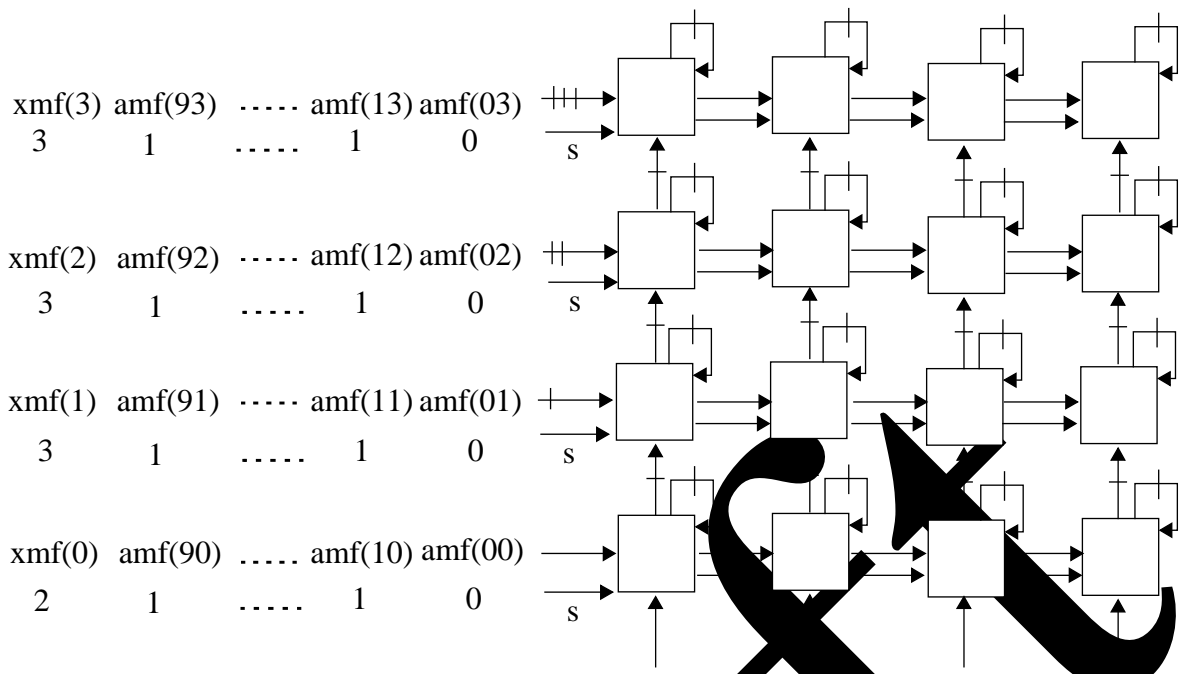
An dieser Stelle erfolgt nun die Nullraum-Propagierung der globalen Daten, d.h., die Propagierungsvektoren der globalen Daten sind Elemente des Nullraumes der Matrix m_g . (Für die Indizes $g_g(I)$ der globalen Daten gilt: $g_g(I) = m_g I + \dots$) Da die Nullräume für das betrachtete Beispiel jeweils eindimensional sind, erfolgt die Auswahl der Propagierungsvektoren \mathbf{d}_v nur unter Beachtung der abgeschwächten Kausalitätsbedingung $\tau^T \mathbf{d}_v \geq 0$, d.h., es ist auch Broadcasting zugelassen. Daraus resultieren folgende Propagierungsvektoren \mathbf{d}_v für die Daten *cmf*, *amf*, *bmf*:

cmf $\rightarrow \mathbf{d}_x = [1 \ 0 \ 0]^T$, *amf* $\rightarrow \mathbf{d}_a = [0 \ 0 \ 0]^T$, *bmf* $\rightarrow \mathbf{d}_b = [0 \ 1 \ 0]^T$.

Als Ergebnis der Indextransformation und der Bestimmung der Propagierungsrichtung für die globalen Daten entsteht das Rechenfeld der Abb. 4-7. Im Vergleich zu existierenden systolischen Lösungen [13] wurde ein gemeinsames Feld für alle Rekurrenzgleichungen der Fuzzy-Implementierung erzeugt, dessen Größe außerdem unabhängig von der Anzahl der Fuzzy-Regeln ist.

Die Anwendung des Tools zur Implementierung der Daten an den Rändern des Rechenfeldes entfällt hier, da bereits eine diebezügliche Lösung erzeugt wurde. Eine Minimierung der Kontrollsignale und die Anpassung an Hardwarerestriktionen zwecks Realisierung auf einem Schaltkreis erfolgte [11].

Draft



$$ymf^*[l] = \begin{cases} \max(ymf[l], \min(xmf[k], c[k, l])) & (s[l, k, i] = 3) \\ \min(xmf[k], c[k, l]) & (s[l, k, i] = 2) \end{cases}$$

$$c^*[k, l] = \begin{cases} \max(c[k, l], \text{goe}(amf[i, k], bmf[i, l])) & (s[l, k, i] = 1) \\ \text{goe}(amf[i, k], bmf[i, l]) & (s[l, k, i] = 0) \end{cases}$$

Abbildung 4-7 : Rechenfeld für die Fuzzy-Inferenz

Für den Schaltkreis der Fuzzy-Inferenz können folgende Leistungsparameter angegeben werden:

Table 2: Leistungsparameter

Anzahl der Wertepaare der Fuzzy-Mengen	Anzahl der Regeln	Anzahl der berechneten Ein- bzw. Ausgabewerte	Anzahl der Takte	Gatter-äquivalente
4	10	1	14	3548

4.3 Schlußfolgerung

Die Fuzzy-Inferenz Applikation konnte in verschiedenen Varianten synthetisiert werden. Vergleicht man den Multiplexer-basierten Entwurf mit der Implementierung auf Basis von systolischen Arrays, so läßt sich feststellen, daß der Mux-basierte Entwurf:

- wesentlich kleiner ist und auf 31% - 38% der Fläche des systolischen Entwurfs untergebracht werden kann.
- etwa fünf mal schneller getaktet werden kann. Die maximale Verarbeitungszeit des Gesamtsystems beträgt 8ns - 10 ns. Damit sind Taktraten von 100MHz - 125MHz möglich.
- eine viel höhere Anzahl von Takten benötigt, um eine Ergebnis zu berechnen.

Die aufgezeigten Entwurfsvarianten sind also interessante Alternativen. Sie unterscheiden sich in der Taktrate, Entwurfsgröße und der erzielten Performanz.

Referenzen

- [1] R. Genevriere and A. Hoffmann. Pmos modular synthesis and hw/sw-codesign system. Technical Report SFB - 358 - B2 - 2/94, March 1994.
- [2] J. Eikerling, R. Genevriere, A. Hardt, A. Hoffmann, K. Feske, G. Franke, M. Kogst, and G. Martin. Flexible hw synthesis and optimization by incremental design modification. Technical Report Tech. Rep. SFB - 358 - B2 - 6/94, University of Paderborn, University of Paderborn, 1994.
- [3] Kortke, M.: VLS - A Language for the Description of Affine Recurrence Equations, *Forschungsberichte*, SFB 358-A-11/94
- [4] Schubert, A.; Merker, R.; Schreiber, H.: Systematic Generation of a Variety of Processor Arrays. Jesshope, C.; Jossifov, V.; Wilhelmi, W. (Herausgeber), *Mathematical Research, Parcella '94*, Volume 81, Akademie-Verlag 1994, S.267-276.
- [5] Köpcke, S.; Kortke, M.; Merker, R.; Schubert, A.; Schreiber, H.; Schüffny, R.; Thiessenhusen, Th.: Erzeugung von VLSI-gerechten Architekturen für rechenintensive Algorithmen. *Tagungsband Anwenderforum des GI/ITG-Workshops 1993*, S.49-53.
- [6] Rao, S.K.: Regular Iterative Algorithms and their Implementation on Processor Arrays, *PhD Thesis*, Stanford University, 1985
- [7] Zhong, X.; Rajopadhye, S. and Wong, I.: Systematic Generation of Linear Functions in Systolic Array Design, *J. of VLSI Processing*, 4, pp. 279-293, 1992

- [8] Baltus, D.G.: Efficient Exploration of affine Space-Time Transformations for Optimal Systolic Array Synthesis, *PhD Theses*, MIT 1994
- [9] Teich, J.; and Thiele, L.: Control Generation in the Design of Processor Arrays, *Int. J. on VLSI and Signal Processing*, Vol.3, pp. 77-92, 1991
- [10] Fimmel, D.: Propagation of Input and Output Data in Massive Parallel Processor Arrays, *Forschungsbericht SFB 358-A1-8/94*.
- [11] Thiessenhusen, Th.; Merker, R.; Schüffny, R.: Flexible Abbildung des approximativen Schließens auf ein massiv paralleles Rechenfeld. Proc. *Workshop Fuzzy-Systeme 93-Management unsicherer Information*, Okt. 1993, Braunschweig, S.51-58.
- [12] Banerjee, U.: Loop Transformation for Restructuring Compilers: The Foundations, Kluwer Academic Publishers, Boston, 1993
- [13] Manzoul, M.; Serrate, H.: Fuzzy Systolic Arrays. In Proc. *18th Ann. Symp. Multiple-Valued Logic*, Spain 1988, S.106-112
- [14] Eckhardt, U.: A System for the Adaptation of Systolic Arrays on VLSI Constraints, *Forschungsbericht SFB 358-A1-6/94*.
- [15] Thiele, L.: Resource Constraint Scheduling of Uniform Algorithms, *Int. Conf. on Application-Specific Processors*, pp. 29-40, 1993

Draft

Draft

V. Abbildung auf FPGAs

Abbildung des in VHDL beschriebenen parallelen Rechenfeldes auf ein FPGA

5.1 Zielarchitektur

Die in den vorangegangenen Kapiteln beschriebene Vorgehensweise wird durch eine Implementierung validiert. Als Ausgangspunkt für die Implementierung dienen Codes in 'C' und 'VHDL'. Als Zielarchitektur wird ein Universalrechner (eine Workstation) mit einer seriellen und parallelen Schnittstelle angeschlossenen FPGA-basierten Prototypenmaschine (*Sparrow*) verwendet. Auf dem Sparrow-Board befindet sich ein FPGA-Baustein der Firma Xilinx. Diese Bausteine besitzen statische RAM-Zellen die die logischen Funktionen und die Verbindungen auf dem Chip steuern. Der verwendete Baustein des Typs XC4000-PC64 besteht aus 484 konfigurierbaren Logikblöcken. In jedem dieser Logikblöcke befinden sich zwei FlipFlops und zwei RAM-Felder mit denen beliebige Funktionen von bis zu vier Variablen aufgebaut werden können. Neben diesem Chip sind auf dem Sparrow noch 64Kb SRAM und ein Interface zur Workstation untergebracht (Abb. 5-1).

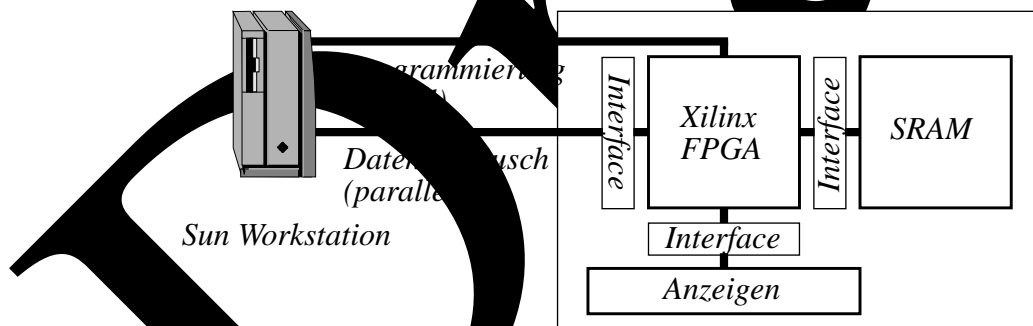


Abbildung 5-1 Blockschaltbild des Prototypenboard 'Sparrow'

5.2 Entwurfsablauf

Ausgangspunkt für die Realisierung einer gemischten Hardware/Software-Implementierung sind getrennte Beschreibungen der beiden Teile. Die Software-Spezifikation liegt dabei in der Sprache 'C' vor, die Hardware-Spezifikation in 'VHDL'.

Zunächst sind beide Spezifikationen so modifiziert, daß sie zusammenarbeiten können. Die Schnittstelle befindet sich an der Grenze eines Funktionsaufrufs. Alle Parameter und Ergebnisse werden als Wert (nicht als Referenz auf einen Wert) übergeben. Daher vereinfacht sich die

Die beschriebenen Werkzeuge werden nicht in ihrem vollen Umfang in allen Internet-Domänen installiert. Somit ist es erforderlich, die einzelnen Übersetzungsphasen über einen Superserver als Dienste den anderen Domänen zur Verfügung zu stellen. So werden z.B. die Installation von Mentor am WSI in Tübingen benutzt, die Xilinx Software am FZI in Karlsruhe und GNU Werkzeuge in Dresden.

Als Eingabe für den Compiler wird eine komprimierte Datei benutzt, die die *VHDL*-Dateien und die Anschlußbelegung des verwendeten Xilinx-Bausteins enthält. Die *VHDL*-Dateien müssen vor der High-Level-Synthese noch von Konstrukten bereinigt werden, die der Synthese Schwierigkeiten bereiten. Für diese Aufgabe wird Votan eingesetzt. Votan benötigt vor der Behandlung von '*VHDL architectures*' die Deklaration der '*VHDL entities*'.

Um die Deklaration der '*entities*' zu erhalten wurde ein Precompiler in '*Perl*' implementiert. Er extrahiert aus der gegebenen *VHDL*-Datei alle *entities* und schreibt sie in eine Ausgabedatei. Diese Datei muß als erste von '*VOTAN*' gelesen werden.

Nun werden die anderen *VHDL*-Texte durch '*VOTAN*' in die Datenstruktur von '*VTIP*' eingelesen. Hiernach werden die passenden Bibliotheken geladen und jede '*unit*' der eingelesenen *VHDL*-Texte durchlaufen. In diesen units wird nach Prozessoren gesucht. In jedem Schritt werden Transformationen durchgeführt:

1. Jeder Ausdruck wird so expandiert, daß jeder Befehl nur einen einfachen Ausdruck enthält. Dabei werden temporäre Variablen benutzt.
2. Die temporären Variablen werden mehrfach verwendet.
3. Unterprogramme werden expandiert.
4. Die '*wait*'-Befehle werden gefaltet. Dann entsteht ein einfacher Automat, dessen Zustände im Anschluß reduziert werden.

Die durch diese Transformationen entstehenden *VHDL*-Beschreibungen werden in neue Dateien abgelegt.

Das Programm *xntor* kann diese Dateien lesen. Zuerst müssen die *entity* Deklarationen eingelesen werden, dann alle restlichen *VHDL* Dateien. Alle Daten liegen nun in einem Mentor internen Format vor. In diesem Format arbeitet das Logiksynthesystem *Autologic* von Mentor. Unter Angabe der Zieltechnologie kann die Synthese durchgeführt werden. Es wird ein 'Rezept' angegeben:

1. Bestimmung der benötigten Fläche
2. Hinzufügen der Ein-/Ausgang-Blöcke
3. Schreiben der Netzliste in Xilinx Format

Dieser Schritt ist sehr zeitintensiv. Es entstehen eine Vielzahl von Dateien mit Netzlisten die die Gesamtschaltung beschreiben. *Autologic* erzeugt diese Dateien allerdings für einen anderen Baustein als den auf dem '*Sparrow*' eingesetzten. Daher muß eine Übersetzung für den dort verwendeten Baustein durchgeführt werden. Die übersetzte Netzliste wird zusammen mit der Beschreibung der gewünschten Anschlüsse zur Generierung der Programmierdatei an das FZI gesandt. Hier sind die Werkzeuge von Xilinx installiert.

Das Programm *xnfmerge* erzeugt aus der Vielzahl der noch hierarchischen Netzlisten eine große flache Netzliste. Diese Netzliste wird dann vom Programm *xnfmap* in die Zieltechnologie abgebildet.

Das Plazieren und Verdrahten der Xilinx Logikblöcke auf dem Chip geschieht mit dem Programm *apr*. Dazu muß jedoch zuerst die Ausgabe von *xnfmap* mit Hilfe des Programms *map2lca* in das richtige Format gebracht werden. Die Plazierung benutzt einen Algorithmus der Klasse 'Simulierte Abkühlung'. Dieser Schritt ist sehr zeitaufwendig.

Die Ausgabe von *apr* kann zur Erzeugung der Programmierdatei benutzt werden. Hierfür dient das Programm *makebits*. Die Programmierdatei wird zurückgeliefert und kann direkt zur Programmierung des Xilinx Chip auf der 'Sparrow'-Platine benutzt werden.

5.2.2 Entwurfsablauf Software

Das in 'C' vorliegende Programm wird zunächst mit einem Übersetzer in ausführbaren Code übersetzt. Die Ausführung selbst ist jedoch nicht auf jedem Rechner möglich und muß speziell vorbereitet werden.

5.3 Ausführung

Es ist heute nicht mehr zumutbar, daß der Benutzer ein Programm sich mit der Hardware begibt, auf der das Programm ausgeführt werden soll. Die Vernetzung bietet die Möglichkeit auch weit entfernte Hardware-Komponenten zu nutzen.

Auch das implementierte Hardware-/Software-System sollte von jedem Rechner im Internet nutzbar sein. Im Gegensatz zu reinen Software-Systemen genügt es nicht, den Code des Programms auf den lokalen Rechnerknoten zu holen, dort eventuell zu übersetzen, und lokal zu nutzen. Das Programm muß immer am Zielknoten ablaufen, indem die spezielle Hardware installiert ist.

Dazu trägt der Benutzer den vollen Namen des gewünschten Rechners an dem sich die Spezialhardware befindet in der Datei ein.

5.3.1 Programmierung der Hardware

Da es sich bei der vorliegenden Spezial-Hardware um Anwenderprogrammierbare Bausteine (FPGA) handelt, kann von der Ausführung des Programms die Programmierung stattfinden. Diese Programmierung erfordert den Aufruf eines Programms auf dem Zielrechner und die Übertragung des Bitstroms der Programmierung. Der Ablauf ist dabei:

1. Komprimierung der Programmierdatei zur Übertragung
2. Feststellen des Zielrechners
3. Bestimmung der Internet-Domäne des Zielrechners
4. Auswahl des zuständigen Superservers für diese Internet-Domäne
5. Beauftragung des Superservers unter Angabe des Zielrechners und Übertragung der Programmierdatei

Auf dem Superserver der Internet-Domäne des Zielrechners geschieht der Ablauf:

1. Annahme der Programmierdatei und des Namens des Zielrechners

2. Übertragung der Programmierdatei auf den Zielrechner
3. Starten des Programms zur Programmierung der Hardware auf dem Zielrechner

Die Ausgaben der auf dem Zielrechner laufenden Programmierung werden dabei auf dem lokalen Rechner angezeigt.

5.3.2 Ausführung der Software

Nach der Programmierung der Hardware kann die Software auf dem Zielrechner gestartet werden. Dazu sind ähnliche Schritte notwendig wie bei der Programmierung der Hardware. Auch die Ausführung kann nur auf dem Zielrechner stattfinden. Der Ablauf ist:

1. Feststellen des Zielrechners
2. Bestimmung der Internet-Domäne des Zielrechners
3. Auswahl des zuständigen Superservers für diese Internet-Domäne
4. Beauftragung des Superservers unter Angabe des Zielrechners und des Programms

Auf dem Superserver der Internet-Domäne des Zielrechners geschieht der Ablauf:

1. Annahme der Namens des Zielrechners und des Programms
2. Starten des Programms auf dem Zielrechner

Die Ausgaben des auf dem Zielrechner laufenden Programms werden dabei auf dem lokalen Rechner angezeigt.

5.3.3 Exklusive Nutzung

Bei der eingesetzten Spezial-Hardware handelt es sich im Sinne eines Betriebssystems um ein 'nicht teilbares Betriebsmittel der Einmaligkeit eins'. Dies ist zu behandeln wie beispielsweise ein Drucker. Es werden Aufträge in eine Menge der zu bearbeitenden Aufgaben eingereiht und bei Freigabe des Betriebsmittels entsprechend einer Auswahlfunktion dieser Menge entnommen und zur Ausführung gebracht.

Diese Funktion ist noch nicht vollständig implementiert. In der aktuellen Implementierung verdrängt ein neuer Auftrag einen gerade in der Ausführung befindlichen.

5.4 Beurteilung der Ergebnisse

5.4.1 Der Übersetzungsvorgang

Der Übersetzungsvorgang wurde durch die anstehende Aufgabe intensiv genutzt. Er wurden zirka 25 mal gestartet und führte in 15 Fällen zu einer Programmierdatei. In den Fällen, in denen keine Programmierdatei entstand mußte in den Logdateien nachgesehen werden warum der Versuch fehlschlug. Der häufigste Grund waren Syntax- oder Semantikfehler im VHDL-Text. Weitere Fehlerquellen waren die Benutzung eines falschen Xilinx-Bausteins oder die Verwendung nicht vorhandener Anschlüsse. Schwieriger zu finden waren die Fehler, die durch Votan in den

VHDL-Text eingebaut wurden. Hier wurde der Quelltext so modifiziert, daß die fehlerhaften Transformationen nicht benutzt werden.

Die Synthese mit Mentor führte zu keinem Abbruch des Übersetzungsvorgangs. Zu bemängeln ist hier aber die lange Laufzeit und die zu geringe Flexibilität. So kann Autologic nur für einen Xilinx-Bausteintyp eine Netzliste schreiben. Es unterstützt auch nicht die Nutzung des auf den Xilinx-Chips vorhandenen Netzes zur Verteilung des Taktsignals. Erst ein nachgeschalteter Editiervorgang in der Netzlistendatei kann diese Defizite beheben. Eine interessante Beobachtung ist, daß die Einstellung des Optimierungsgrads (low, medium, high) keinen Einfluß auf das Ergebnis hat.

Die Nachgeschaltete Technologieabbildung durch die Xilinx Software 'Xact' stellte sich als das Nadelöhr des 'Rapid Prototyping' heraus. Die Plazierung auf dem Chip dauerte circa zehn Stunden. Dabei kam es wiederholt vor, daß der Übersetzungsvorgang bei der nachfolgenden Verdrahtung abgebrochen wurde, da ein Netz nicht gerouted werden konnte. Hier half ein zweiter oder dritter Anlauf (weitere 10-20 Stunden!) da die Plazierung nicht deterministisch ist.

Es wurden 15 erfolgreiche Versuche benötigt um eine funktionierende Schaltung zu erzeugen, da es erhebliche Defizite in der Beschreibung der Anbindung der Spezialhardware an die Workstation gab. Die VHDL-Beschreibung des Schaltungskerns wurde in allen Versionen unverändert gelassen, funktionierte also bereits im ersten Durchlauf.

5.4.2 Die Implementierung

Während des Übersetzungsvorgangs entstehen Dateien in verschiedenen Zwischenformaten. Ausgangspunkt ist eine VHDL-Datei von 952 Zeilen. Die entstehende Programmierdatei hat eine feste Länge von 11958 Bytes. Nach der Vorverarbeitung mit Xact hat der Entwurf eine Länge von 768 Zeilen und besteht aus 20 Dateien. Diese werden von Mentor eingelesen und zur Generierung der (Netlist Format) Dateien benutzt. Die Mentor-internen Daten bestehen aus 120 Dateien mit einer Größe von insgesamt 2586 kByte. Ein Teil des Schaltplans ist in Abbildung 5-3 zu sehen. Es wurden 54 Dateien mit insgesamt 4784 Zeilen von Au-

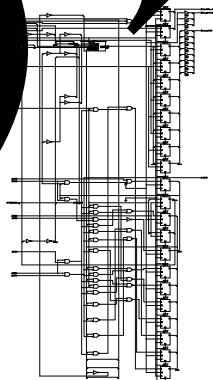


Abbildung 5-3 Schaltplan-Ausschnitt

Autologic ausgegeben. Nach der Technologieabbildung (Abb. 5-4) ergibt sich für einen Baustein

3195PC84-3 eine Ausnutzung von 26% oder 138 CLBs (Configurable Logic Block). Es werden

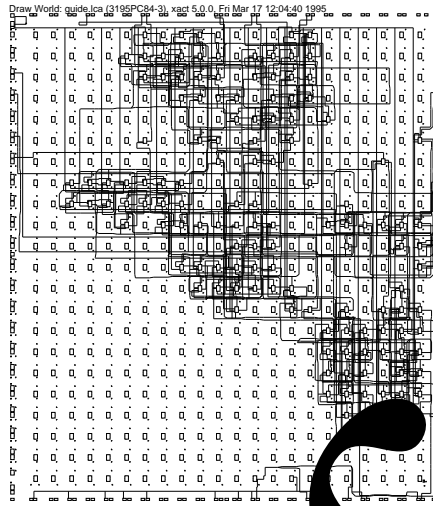


Abbildung 5-4 Xilinx Verdrahtung

86 Flipflops benutzt. Die geschätzte maximale Taktfrequenz beträgt 13,8Mhz (Abb. 5-5)

Clock net "i_x_clk" path delays:

Pad to Setup : 72.6ns (7 block levels)

(Includes an external input margin of 0.0ns.)

Pad "x_reset" (P56) to FF Setup (EC) at "o_x_dw<7>.EC"

Target FFY drives output net "o_x_dw<7>"

Clock to Pad : 31.2ns (0 block levels)

(Includes an external output margin of 0.0ns.)

Clock to Q, net "o_x_opcode<1>" to Pad "x_opcode<1>" (P16.O)

Clock to Setup (same edge) : 72.1ns (7 block levels)

(Includes 0.1ns clock skew compensation.)

Clock to Q, net "ca_state__1<3>" to FF Setup (EC) at "o_x_dw<7>.EC"

Target FFY drives output net "o_x_dw<7>"

Minimum Clock Period : 72.6ns

Estimated Maximum Clock Speed : 13.8Mhz

Abbildung 5-5 Timing Analyse

Referenzen

- [1] T. Bubeck, U. Keschul, W. Rosenstiel. *Sparrow - Ein Mikroprozessor-Entwurfs- und -Ausbildungssystem*. Technical Report WSI-93-7, Universität Tübingen
- [2] G. Koch, U. Keschul, W. Rosenstiel. *A Prototyping Environment for Hardware/Software Codesign in the COBRA Project* CODES/CASHE. Grenoble 1994

- [3] A. Kunzmann. *Modelling a FPGA Design Flow in the JESSI-COMMON-Framework*. Proc. 10th Intern. Conference on CAD/CAM. Robotics and Factories of the Future. Ottawa/Kanada 1994.
- [4] W. Rosenstiel. *Experiences with High-Level Synthesis from VHDL-Specifications*. Proc. of Workshop on Design Methodologies for Microelectronics and Signal Processing. Gliwice/Polen 1993
- [5] E. Schubert, P. Thole, W. Rosenstiel. Exercises with the VOTAN High-Level Synthesis System, EUROCHIP Course on High-Level Digital System Design, Leuven/Belgium, 1994
- [6] H. Speckmann, P. Thole, W. Rosenstiel. Hardware Synthesis for neural networks from a behavioural description with VHDL. IJCNN Nagoya/Japan 1993
- [7] M. Wendling, W. Rosenstiel. A Hardware Environment for Prototyping and Partitioning Based on Multiple FPGAs, EURO-DAC, Grenoble/Frankreich, 1994
- [8] J.K. Ousterhout. *Frequently Asked Questions About TeX*.
<http://playground.Sun.COM:80/~ouster/>
- [9] *GNU software*, <ftp://prep.ai.mit.edu/pub/gnu/>
- [10] *Xilinx Home Page*, <http://www.xilinx.com/>

Draft

VI. Anhang

6.1 Das C-Programm „fuzzy.c“

```
#define GEN_EXAMPLE /* Beispiel-Generator einbinden */
#define SIZE 4 /* Groesse der Membership(MF)-Funktionen */
#define MMVAL 16 /* Wertebereich der MF */
#define NMF 4 /* Anzahl moeglicher Praemissen: SIZE >= NMF >= RULES */
#define RULES 10 /* Anzahl der Regeln */
#define FZ 5 /* Wert in % zur „Verunschaerfung“ der Eingangsfakten

#include <stdio.h>
#include <stdlib.h>

int readData( FILE *fep, int amf[RULES][SIZE], int bmf[RULES][SIZE], int xmf[SIZE] );
void mk_trapez(int a1, int a2, int a3, int a4, int mf[SIZE] );
int goe(int a, int b);
int max(int a, int b);
int min(int a, int b);
void fuzzyfy( int x, int *xmf );
void goedel_inference( int amf[RULES][SIZE], int bmf[RULES][SIZE],
int xmf[SIZE] );
void max_inference( int amf[RULES][SIZE], int bmf[RULES][SIZE],
int xmf[SIZE], int ymf[SIZE] );
void _deg_inference( int amf[RULES][SIZE], int bmf[RULES][SIZE],
int xmf[SIZE], int ymf[SIZE] );
void fuzzyfy( int *y, int *ymf );
void print( int flag, int index, int *ymf, char *string );

static char errors[3]={„\nSpeicher-Allokationsfehler\n“,
„\nDaten-Lesefehler an Position %ld\n“,
„\nSuchueberschreitung an Position %ld (%d)\n“};

#define FSCANF(pformat, p) \
if(!fscanf(fep,pformat,p) || feof(fep)) \
{ printf(errors[1],ftell(fep)); exit(4); }

#define RANGE(ptest,pmin,pmax) \
if((ptest)<(pmin) || (ptest)>(pmax)) \
{ printf(errors[2],ftell(fep),ptest); exit(5); }

#endif random
#define random(num) ((int)(((long)rand()&0X7FFF)*(long)(num))/0X8000L)
#endif
```

```

int readData( FILE *fep, int amf[RULES][SIZE], int bmf[RULES][SIZE], int x[SIZE])
{
    int i, j, inputs;

    /* Einlesen der MF */
    for(j = 0; j < RULES; j++) {
        for(i = 0; i < SIZE; i++) {
            FSCANF( „%d“, &(amf[j][i]) );
            RANGE( amf[j][i], 0, MMVAL-1 );
        }
        for(i = 0; i < SIZE; i++) {
            FSCANF( „%d“, &(bmf[j][i]) );
            RANGE( bmf[j][i], 0, MMVAL-1 );
        }
    }

    /* Einlesen der Anzahl der Eingabewerte */
    FSCANF( „%d“, &inputs );
    RANGE( inputs, 1, SIZE );

    /* Einlesen der Eingabewerte */
    for(i = 0; i < inputs; i++) {
        FSCANF( „%d“, &(x[i]) );
        RANGE( x[i], 0, SIZE-1 );
    }

    return inputs;
}

/* Beispiel-Generator (GEN_EXAMPLE) zugeschaltet */
#ifdef GEN_EXAMPLE
void mk_datafile(char *filename)
{
    int uel[2];
    int w, r, s, t, j, k;
    int mf[SIZE];
    FILE *fep;

    if((fep=fopen(filename, "w"))==NULL)
        { printf(„\n\nKann File nicht öffnen.\n“,filename); exit(1); }

    for(k=0;k<2;k++)
    {
        for(nmf[k]=uel[k]=0;nmf[k]<2 || nmf[k]>RULES || uel[k]<0;)
        {
            if(!k) printf(„Eingang (Praemissen/Regeln) A\n“);
            else printf(„Ausgang (Konklusionen) B\n“);
            nmf[k]=min(NMF,RULES);
            printf(„Ueberlappung in %% (min 0) :“);
            fflush(stdin); scanf(„%d“,uel+k);
        }
        printf(„\n“);
    }
}

```

```

/* Praemissen/Konklusionspaare erzeugen und schreiben */
for(j=0;j<nmf[0];j++)
{
t=random(nmf[1]);
l=(j<nmf[0]-1) ? 1 : RULES-NMF+1;
for(h=0; h < l; h++)
{
for(k=0;k<2;k++)
{
width=SIZE/nmf[k];
delta=(int)(((long)width*(long)uel[k])/100);

if(k!=0) r=t;
else r=j;

s=(SIZE*r)/nmf[k];
mk_trapez(s-delta,s,s+width-1,s+width-1+delta,mf);

for(i=0;i<SIZE;i++)
{
if(!(i%16)) fprintf(fep,“\n“);
fprintf(fep,“%4d „,mf[i]);
}
fprintf(fep,“\n\n“);
}
fprintf(fep,“\n\n“);
}

t=min(NMF,10);
fprintf(fep,“%d „,t);
for(i=0;i<t;i++)
fprintf(fep,“%d „,(i*SIZE)/t);
fprintf(fep,“\n\n“);

fclose(fep);
return 0;
}
#endif

int min(int a, int b)
{ return( a<b ? a : b ); }

int max(int a, int b)
{ return( a>b ? a : b ); }

int goe(int a, int b)
/* Goedel-Implikation */
{ return( a<=b ? MMVAL-1 : b ); }

void mk_trapez(int a1,int a2,int a3,int a4,int mf[SIZE])
/* Erzeugt aus den Parametern eine normalisierte Trapez-MF */
/* a0-3 Trapezparameter, mf zu belegendes MF-Feld */
{

```

```
int i,s;
```

```
for(i=0;i<SIZE;i++)
{
  if(i<a1) s=0;
  else if(i<a2) s=1;
  else if(i<=a3) s=2;
  else if(i<=a4) s=3;
  else s=0;
  switch(s)
  {
    case 0: mf[i]=0; break;
    case 1: mf[i]=(MMVAL*(i-a1+1))/(a2-a1+1); break;
    case 2: mf[i]=MMVAL-1; break;
    case 3: mf[i]=(MMVAL*(a4-i+1))/(a4-a3+1); break;
  }
}
return;
}
```

```
void fuzzyfy( int x, int *xmf )
```

```
{
  int delta;

  /* es wird eine Unschärfe von +-FZ % oder +- 1 Wert angenommen */
  delta = ( SIZE / (100/FZ) != 0 ) ? SIZE / (100/FZ) : 1;
  mk_trapez( x-delta, x, x, x + delta, xmf );
}
```

```
int defuzzyfy( int *y, int *ymf )
```

```
{
  int i;
  long sm = 0, sx = 0L;

  /* Koeffizienten fuer Defuzzyfizierung (Wissenschwerpunkt) */
  for( i = 0; i < SIZE; i++ ) {
    sm += (int)y[i] * (long)ymf[i];
    sx += (int)ymf[i];
  }

  /* Division und Runden */
  *y = ((2L*sm)/sx + 1) / 2;

  /* Test auf spezielle Werte */
  if( sm == SIZE*(MMVAL-1) )
    return 2;
  if( sm == 0 )
    return 1;
  return 0;
}
```

```
/* Fuzzy-Inferenz mittels Goedel - Inferenz */
```

```
void goedel_inference( int amf[RULES][SIZE], int bmf[RULES][SIZE],
  int xmf[SIZE], int ymf[SIZE] )
{
```

```

int l, k, i;
int c;

for( l = 0; l < SIZE; l++ )
  for( k = 0; k < SIZE; k++ ) {
    c = goe( amf[0][k], bmf[0][l] );

    for( i = 1; i < RULES; i++ )
      c = min( c, goe( amf[i][k], bmf[i][l] ) );

    if( k == 0 )
      ymf[l] = min( xmf[k], c );
    else
      ymf[l] = max( ymf[l], min( xmf[k], c ) );
  }
}

```

```

/* Fuzzy-Inferenz mittels Maximum/Minimum - Inferenz */
void max_min_inference( int amf[RULES][SIZE], int bmf[RULES][SIZE],
                        int xmf[SIZE], int ymf[SIZE] )

```

```

{
  int l, k, i;
  int c;

  for( l = 0; l < SIZE; l++ )
    for( k = 0; k < SIZE; k++ ) {
      c = min( amf[0][k], bmf[0][l] );

      for( i = 1; i < RULES; i++ )
        c = max( c, min( amf[i][k], bmf[i][l] ) );

      if( k == 0 )
        ymf[l] = min( xmf[k], c );
      else
        ymf[l] = max( ymf[l], min( xmf[k], c ) );
    }
}

```

```

/* Fuzzy-Inferenz nach der Methode der Aktivierungsgrade */
void activation_inference( int amf[RULES][SIZE], int bmf[RULES][SIZE],
                          int xmf[SIZE], int ymf[SIZE] )

```

```

{
  int l, k, i;
  int actdeg;

  for( i = 0; i < RULES; i++ )
    for( k = 0; k < SIZE; k++ )
      actdeg = ( k == 0 ) ? min( xmf[k], amf[i][k] ) : max ( actdeg, min( xmf[k], amf[i][k] ) );

  for( l = 0; l < SIZE; l++ )
    ymf[l] = ( i == 0 ) ? min( actdeg, bmf[i][l] ) : max( ymf[l], min( actdeg, bmf[i][l] ) );
}
}

```

```

void fuzzyOut( int flag, int index, int *y, int *ymf, char *string )
{
    int i, j;

    printf( „\nDatensatz %d (%s): \n“, index, string );
    switch( flag ) {
        case 0:
            printf(„ Defuzzifizierung: %d\n“,*y );
            for(j = 3; j >= 0; j--) {
                for(i = 0; i < SIZE; i++)
                    printf(„,%c“, (ymf[i]*5)/MMVAL>j ? ,#‘ : ,.‘);
                printf(„\n“);
            }
            break;
        case 1:
            printf(„Ergebnis ist die leere Menge.\n“);
            break;
        case 2:
            printf(„Resultat absolut unbestimmt.\n“);
            break;
    }
}

```

```

int main(int argc, char *argv[]){
    FILE *fep;

    int inputs; /* Anzahl der Ein- und Ausgangswerte */
    int x[SIZE], yGoedel[SIZE], yMaxMin[SIZE], yReg[SIZE]; /* scharfe Eingangswerte */
    int xmf[SIZE], ymfGoedel[SIZE], ymfMaxMin[SIZE], ymfActDe[SIZE]; /* Ausgangs-MF, Ausgangs-MF's */

    int amf[RULES][SIZE];
    int bmf[RULES][SIZE];
    /* Mem. f. Amp-Faktoren der Praemissen und Konklusionen */

    int ct, GoedelOut, MaxMinOut, ActDeOut;

    printf(„** Fuzzy-Inferenz-Beispielprogramm (Thilo Thiessenhusen, Mathias Kortke) **\n\n“);
    if( argc < 2 )
        { printf(„Bitte geben Sie die filename\n“, argv[0] ); exit( 2 ); }

#ifdef GEN_EXAMPLES
    mk_datafile( argv[1] );
#endif

    /* Einlesen des Regelwerks und der scharfen Eingangswerte aus Datei */
    if( ( fep = fopen( argv[1], „r“ ) ) == NULL ){
        printf( „,Kann File %s nicht lesen\n“,argv[1] ); exit( 1 ); }
    inputs = readData( fep, amf, bmf, x );
    fclose( fep );

    /* Haupt-Rechenschleife */
    for( ct = 0; ct < inputs; ct++ ) {
        /* Fuzzifizierung des Eingangswertes */
        fuzzyfy( x[ct], xmf );

        /* Fuzzy Inferenzen */
        goedel_inference( amf, bmf, xmf, ymfGoedel );
    }
}

```

```
max_min_inference( amf, bmf, xmf, ymfMaxMin );
act_deg_inference( amf, bmf, xmf, ymfActDeg );

/* Defuzzifizierung (Massenschwerpunkt) */
GoedelOut = defuzzyfy( &yGoedel[ct], ymfGoedel );
MaxMinOut = defuzzyfy( &yMaxMin[ct], ymfMaxMin );
ActDegOut = defuzzyfy( &yActDeg[ct], ymfActDeg );

/* Ausgabe der Ergebnisse */
fuzzyOut( GoedelOut, ct, &yGoedel[ct], ymfGoedel, „Goedel - Inferenz“ );
fuzzyOut( MaxMinOut, ct, &yMaxMin[ct], ymfMaxMin, „Maximum/Minimum - Inferenz“ );
fuzzyOut( ActDegOut, ct, &yActDeg[ct], ymfActDeg, „Methode der Aktivierungsgrade“ );
}
return( 0 );
}
```

Draft

Draft