Lehrstuhl für integrierte Systeme der Signalverarbeitung
Rheinisch-Westfälische Technische Hochschule Aachen
Prof. Dr. H. Meyr

Diplomarbeit D 305

Scheduling von datenflußorientierten Multiraten-Programmen auf Multi-DSP-Architekturen

von

Rainer Schoenen

Matrikel-Nr. 180583

Dezember 2007

Betreuung durch:
Univ. Prof. Dr. H. Meyr
Dipl. Ing. Vojin Zivojnovic

Eidesstattliche Erklärung

Ich versichere, daß die vorliegende Arbeit - bis auf die offizielle Betreuung durch den Lehrstuhl - ohne fremde Hilfe von mir durchgeführt wurde. Die verwendete Literatur ist im Anhang vollständig angegeben.

Aachen, den 28.12.2007

(Rainer Schoenen)

Inhaltsverzeichnis

1 Einführung **1**
1.1 Einleitung 1
1.2 Vorschau 2
2 Zielarchitekturen **3**
2.1 Entwicklungszyklus 3
2.2 Mikroprozessoren und Transputer 4
2.3 Digitale Signalprozessoren 5
2.4 Parallelrechner 7
2.5 VLSI 8
3 Datenflußorientierte Programme **9**
3.1 Voraussetzungen 9
3.2 Digitale Signalverarbeitung 10
3.3 Andere Anwendungen 10
4 Modelle und Eigenschaften **11**
4.1 Motivation 11
4.2 Definitionen 12
4.2.1 Datenflußgraph 12
4.2.2 Pfad 13
4.2.3 Schleife 13
4.2.4 Transitive Kanten 13
4.2.5 Einheitsraten-DFG 13
4.2.6 Singleraten-DFG 13
4.2.7 Multiraten-DFG 13
4.2.8 Zusammenhängende Komponente 14
4.2.9 Maximal stark zusammenhängende Komponente 14
4.2.10 Ausführungszeit 14
4.2.11 Iterationsperiode 14
4.2.12 Iterationsperiodengrenze 14
4.2.13 Latenz 15
4.2.14 Schedul 15
4.3 Die Inzidenz-Matrix 15
4.3.1 Der q-Vektor 16
4.3.2 Die C-Matrix 17
4.4 Die Zustandsgleichung 19
4.5 Lebendigkeit, Beschränktheit, Konsistenz 20
4.6 Marked Graphs als Spezialfall von Petri-Netzen 21
4.6.1 Petri-Netze 21
4.6.2 Endliche Automaten 22
4.6.3 Marked Graphs 22
4.6.4 Weitere Unterklassen von Petri-Netzen 22
4.7 Präzedenzbeziehungen, Quellen möglicher Parallelität 23
4.8 Gewichtete Delaysumme 24
4.8.1 "Weighted Sum of Tokens" 24
4.8.2 "Sum of Weighted Tokens" 25
4.8.3 "Heaviest Dead Marking" 25

4.8.4 "Lightest Live Marking" 25
4.8.5 "Comfortable Marking" 26
4.9 Erreichbarkeit und Steuerbarkeit 27
4.9.1 Der Erreichbarkeitsgraph 27
5 Transformationen von Datenflußgraphen 30
5.1 Umwandlung von Multi-Raten-Graphen in Einheitsraten-Graphen 30
5.2 Azyklischer Präzedenz-Graph 32
5.3 Spannender Baum 33
5.4 Hierarchisierung 33
5.5 Strongly Connecting 34
5.6 Unfolding 35
5.7 Blocking 36
5.8 Retiming 37
5.9 Vektorisierung 37
5.10 Einheitsverstärkungs-Transformation (Normierung) 38
5.10.1 Normschleife 40
5.10.2 Normgraph 40
5.10.3 Invarianz der WST 40
5.10.4 Eigenschaften von coprimen Normschleifen 40
6 Scheduling von datenflußorientierten Programmen 42
6.1 Mathematische Hilfsmittel 42
6.1.1 Lineare Programmierung 42
6.1.2 ILP-Problem 43
6.1.3 Komplexität 44
6.1.4 NP-Vollständigkeit 45
6.2 Schedultheorie 46
6.2.1 Übersicht 46
6.2.2 Ein-Prozessor-Schedul 47
6.2.3 Klasse-S-Algorithmus 47
6.2.4 Multiprozessor-Schedul 48
6.3 Schedultechniken und -ergebnisse 48
6.3.1 Präemptives - nicht präemptives Scheduling 48
6.3.2 Dynamisch - statisch 48
6.3.3 Überlappendes - Nicht-Überlappendes Schedul 49
6.3.4 Heuristiken 50
6.3.5 Raten-optimales Scheduling 51
6.3.6 Inter-Prozessor-Kommunikation 51
7 Retiming bei Multiraten-Datenflußgraphen 52
7.1 Retiming für Einheitsraten-Graphen 52
7.2 Multiraten-Retiming 53
7.2.1 Berechnung des Retiming-Vektors 54
7.2.2 Neue Erreichbarkeits-Bedingung 56
7.3 Unbewegliche Delays 60
7.3.1 Kanten-NMD's (Lokale NMD’s, arc-NMD’s) 60
7.3.2 Pfad NMD's (Path-NMD) 61
7.3.3 Keine NMD's vorhanden 62
7.4 Retiming von Schleifen 63
7.5 Retiming von azyklischen Graphen 64
7.6 Retiming von zyklischen Graphen 64

7.7 Retiming durch Lineare Optimierung ... 65
7.8 Cutting Planes und Branch & Bound ... 67
7.9 Schnittebenen im Zustandsraum ... 68
7.10 Minimaler retimingäquivalenter Einheitsraten-Graph ... 69
7.11 Retiming durch 'Shortest Path' ... 73
7.12 Anfangsbedingungen ... 74
7.13 Anwendungen von Retiming ... 75
7.13.1 Verbesserung des Durchsatzes ... 75
7.13.2 Minimierung der Speicherelemente ... 76
7.13.3 Separatoren zwischen Clustern ... 76
8 Der Durchsatz in Multiratensystemen ... 77
8.1 Die Iterationsperiode und deren Schranke ... 77
8.2 Die Iteration Bound für Einheitsraten-Graphen ... 78
8.3 Die Iterationsperiode für einen Prozessor ... 78
8.4 Die 'Processor-Bound' ... 78
8.5 Auslastung ... 79
8.6 Schwierigkeiten im Multiratenfall ... 79
8.7 Maximale Anzahl von Prozessoren ... 79
8.8 Schranken für den Durchsatz ... 80
8.8.1 Stand der Forschung ... 81
8.8.2 Erweiterungen ... 82
8.8.3 Normierung ... 82
8.8.4 Neue Ansätze ... 83
8.9 Anwendungen ... 89
9 Implementationen ... 90
9.1 Objektorientierte Programmierung in C++ ... 90
9.2 Klassenbibliotheken ... 90
9.3 Die LEDA-Klassenbibliothek ... 91
9.4 Eigene Klassen ... 92
9.5 Methoden der Klasse DFG ... 93
9.6 Weitere implementierte Module ... 93
9.7 Interaktive Bedienoberfläche ... 93
9.8 Automatische Tests ... 95
10 Zusammenfassung und Ausblick ... 97
11 Anhang ... 98
11.1 Programme ... 98
12 Glossar ... 105
12.1 Literatur ... 105
12.2 Index ... 112
13 Literaturverzeichnis ... 115
13.1 Verwendete Literatur ... 115
13.2 Ausklappbares Kurzverzeichnis ... 120

1 Einführung

1.1 Einleitung

Der ansteigende Bedarf an rechenzeitintensiver Echtzeit-Signalverarbeitung überschreitet die Verbesserungen der Verarbeitungsleistung jeder neuen Generation von Digitalen Signalprozessoren. Immer mehr Anwendungen erfordern enorme Rechenleistungen, die ein einzelner Prozessor heutzutage nicht aufbringen kann. Die durch Zusammenschaltung individueller Prozessoren durchführbare Parallelverarbeitung bietet die benötigte Rechenleistung an, die von diesen Anwendungen gefordert wird. Solche Systeme haben bessere Flexibilität, Skalierbarkeit, Fehlertoleranz und Erweiterbarkeit.

DSP-Applikationen sind zur Parallelverarbeitung gut geeignet, da sie genügend potentielle Parallelität besitzen, gut strukturiert sind und häufig periodisch ablaufen. Um diese Parallelität offenzulegen, muß das System auf ein Modell reduziert werden, das Teile der Software zu Blöcken zusammenfaßt und deren Abhängigkeiten berücksichtigt. Umwandlungen (Transformationen) dieses Modells können verschiedenen Zwecken dienen, hauptsächlich soll aber der Durchsatz maximiert werden durch optimale Ausnutzung der Parallelität.

Als am besten geeignetes Modell hat sich eine spezielle Klasse der Petri-Netze herausgestellt, das mit anderen Modellen zusammen zur Theorie beiträgt. Die Modelle existierten bisher parallel und ohne großen gegenseitigen Einfluß. Für die zu behandelnden Systeme sind sie äquivalent und ineinander überführbar.

Die geeignetste Transformation zur Verbesserung der Ausnutzung der Parallelität ist Retiming[1], das sich für nicht-zyklische Strukturen zum Pipelining reduziert. Weitere Probleme, wie die Minimierung benötigter Speicherelemente, lassen sich erfolgreich mit Retiming behandeln. Der Aufwand zur Durchführung eines legalen Retimings wächst jedoch bei Multiraten-Systemen exponentiell mit der Größe des Problems.

In dieser Arbeit wurden Ansätze untersucht, die diese Komplexität nicht aufweisen sollen, wobei festzustellen war, welche Probleme zu der angenommenen Komplexität führen. Es stellten sich gewisse Anomalien der Multiraten-Datenflußgraphen als Quelle der Probleme heraus, die in Systemen mit einheitlichen Raten nicht auftreten können. Daraus entwickelte sich ein Konzept, das zum Begriff ‘Non-Movable-Delay’ führte, wenngleich die Namenswahl nicht ganz treffend ist. Aufbauend auf diesem Konzept galt es Möglichkeiten herauszufinden, um die Komplexität polynomial zu halten. Die Ansätze werden in späteren Kapiteln präsentiert.

Der Durchsatz in rückgekoppelten (zyklischen) Systemen der Signalverarbeitung ist prinzipiell beschränkt, auch wenn beliebig viele Prozessoren zur Verfügung stehen. Während in Systemen mit gleichen Raten der maximale Durchsatz ermittelt werden kann, gibt es wieder Unregelmäßigkeiten im Fall multipler Raten. In dieser Arbeit werden anhand von Beispielen die Probleme demonstriert und erklärt. Im Anschluß werden Verfahren präsentiert, mit denen sich eine Grenze berechnen läßt, die nie überschritten wird, aber nahe genug an der realen Grenze liegt, um als gute Abschätzung zu dienen. In vielen Fällen stimmen die Grenzen überein.

Zur Auswertung und Visualisierung der Modelle und Ergebnisse wurde ein Programm für UNIX-Systeme entwickelt, mit dem sich der Entwurf von Systemen im Datenfluß-Modell interaktiv (mit Mausbedienung) durchführen läßt. Die im Menü enthaltenen Kommandos ermöglichen neben Dateifunktionen einige Transformationen, Auswertungen und Scheduling. Zur

[1] In dieser Arbeit werden oft die in der Literatur üblichen englischen Begriffe beibehalten. Wenn eine naheliegende deutsche Bezeichnung existiert, wird bei der ersten Erwähnung der englische Begriff ergänzt: {so}

Programmierung wurde C++ mit einer Klassenbibliothek „LEDA“ eingesetzt. Die ausgearbeiteten Module werden in der Arbeit vorgestellt und sind im Anhang oder einem separaten Heft teilweise ausgedruckt.

1.2 Vorschau

Die vorliegende Arbeit beschäftigt sich mit den grundlegenden Problemen bei der Implementation von Programmen auf mehreren Prozessoren unter besonderer Berücksichtigung ungleicher Raten. In der folgenden Kapiteln werden Schwerpunkte gesetzt, die sich mit den Stationen des Designs, als auch der Flexibilität der existierenden Methodik auseinandersetzen.

Im Kapitel „Zielarchitekturen“ wird die Bandbreite der Hardware erörtert, die für datenflußorientierte Programme geeignet sind und sich mit den vorgestellten Methoden behandeln lassen.

Das Kapitel „Datenflußorientierte Programme“ stellt eine Einführung in die Softwareseite der Datenfluß-Abstraktion dar. Hier wird dargelegt, für welche Anwendungsgebiete die Theorie genutzt werden kann.

Im nächsten Kapitel „Modelle und Eigenschaften“ werden die theoretischen Grundlagen behandelt, die als Werkzeug für die folgenden Untersuchungen dienen. Insbesondere ist die Ausdehnung von Konzepten auf den Multiraten-Fall von Interesse. Für diesen ergeben sich eine Reihe von interessanten Konsequenzen, die zu andersartigen Problemen führen.

Durch die Umwandlung von Graphen sollen sich markante Größen, wie zum Beispiel die Schleifenmatrix, vereinfachen. Im Kapitel „Transformationen“ werden Methoden vorgestellt, die zu einer Äquivalenzklasse von Graphen führen, die die gesamte oder einen Teil der ursprünglichen Information enthalten.

Im Kapitel „Scheduling“ wird die Festlegung der Ausführungszeitpunkte auf einem und mehreren Prozessoren behandelt. Die Komplexität der verwendbaren Algorithmen ist von besonderem Interesse, insbesondere werden NP-vollständige Probleme identifiziert und die zugrundeliegenden Schwierigkeiten erläutert.

Die wichtigste Graphentransformation zum Erhalt eines optimalen Scheduls wird im Kapitel „Retiming“ behandelt. Die im Multiratenfall veränderten Bedingungen nehmen hier einen breiten Raum ein. Es werden Sonderfälle klassifiziert, deren Lösung in polynomialer Zeit möglich ist. Außerdem werden verschiedene Lösungsansätze und -methoden vorgeschlagen, die alternativ zur Lösung führen.

Der „Durchsatz in Multiratensystemen“ und Grenzen für diesen werden im gleichnamigen Kapitel vorgestellt und mit Beispielen erläutert.

Im Kapitel „Implementationen“ wird die während dieser Arbeit entstandene Software vorgestellt. Die verwendeten Werkzeuge und Bibliotheken finden hier ebenso Erwähnung wie die Konzepte moderner Softwaretechnik.

Der „Anhang“ ist für Ergänzungen zum Text vorgesehen. Hier sind auch einige Programmlistings der entstandenen Software abgedruckt.

Das am Ende befindliche „Glossar“ dient zur Erklärung der in der Literatur verwendeten Fachbegriffe. Zum Teil werden verschiedene Bezeichnungen für dieselbe Sache verwendet, zum Beispiel im Vergleich zwischen der Theorie der Petri-Netze und der des regulären Datenflusses. Eine Ergänzung findet das Kapitel durch die Angabe einiger der in dieser Arbeit verwendeten Fachbegriffe.

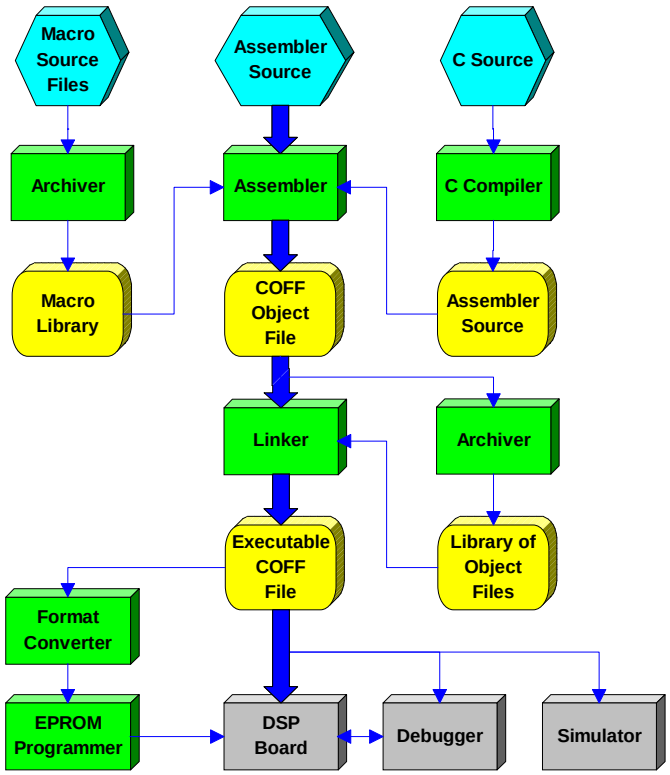
Die letzte Seite ist herausklappbar und enthält in Kurzform die verwendete Literatur, zur übersichtlichen Betrachtung während des Lesens.

2 Zielarchitekturen

2.1 Entwicklungszyklus

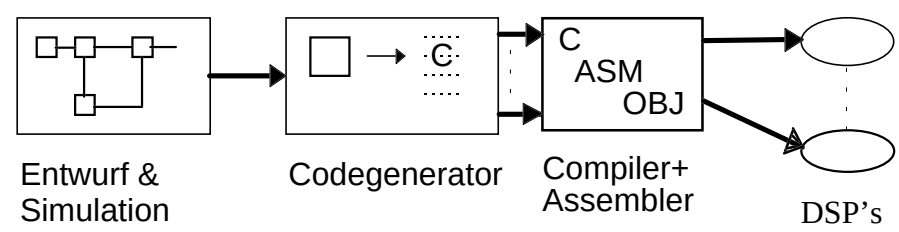
Eine Applikation, die ein gegebenes Problem mit bekannten Algorithmen lösen soll, wird entweder dadurch implementiert, daß eine flexible Software für eine ausgewählte Hardwarezusammenstellung geschrieben wird, oder indem der Algorithmus direkt in Hardware realisiert wird. Für den letzteren Fall existieren Hardware-Beschreibungssprachen wie z.B. VHDL, die eine Realisierung auf ASICs, FPGAs o.ä. ermöglichen. Der Realisierungsaufwand ist jedoch enorm, wenn sämtliche Funktionsblöcke auf einem VLSI-Chip synthetisiert werden müssen. Der Entwickler hat zwar wesentlich mehr Einfluß auf die Performance, jedoch lohnen sich der lange Entwicklungszyklus und die hohen Produktionskosten nur für große Stückzahlen bei relativ starrem Design.

Für Anwendungen, die eine Rechenleistung von bis zu 50..100 MIPS pro Prozessor erfordern, sind Software/Hardware-Systeme die flexiblere Lösung, da Änderungen im Programmcode nur den Austausch der Software in einem programmierbaren Festwertspeicher nötig machen. Der Entwicklungszyklus (siehe Bild) für Mikroprozessorimplementationen hängt von der vorhandenen Entwicklungsumgebung und der Strategie ab. Konventionell wird zuerst die Zielhardware ausgewählt (Typ der Prozessoren), danach werden die Tools des Herstellers benutzt, um ein in Assembler oder einer Hochsprache (C, C++, PASCAL, BASIC) geschriebenes Programm zu compilieren und zu debuggen. Der erstellte Objektcode muß noch durch einen Linker zu einem lauffähigen Modul gebunden werden; dabei werden Referenzen auf Adressen aufgelöst {resolved} und den Speicherblöcken Adressen zugewiesen {relocation}. Das Ergebnis ist ein 'ausführbarer' Objektcode.



Eine Entwicklung in einer Hochsprache wie z.B. C ist schneller zu programmieren und zu debuggen als in Assembler, außerdem ist der Sourcecode auf verschiedene Prozessortypen portabel. Der gravierende Nachteil, als tradeoff, ist jedoch die Geschwindigkeitseinbuße zur Laufzeit. Bis heute sind keine Compiler in der Lage, effektiven Assemblercode zu generieren, der an den üblicherweise 200% schnelleren handgeschriebenen Code heranreicht. Die Folge ist, daß in der Regel die Programme für zeitkritische Aufgaben (wie meistens in der Signalverarbeitung) in Assembler geschrieben werden, mit allen Konsequenzen für Personalkosten und Ent-

wicklungszeit. An der Entwicklung leistungsfähigerer Compiler wird in Zukunft gearbeitet werden müssen, aber wahrscheinlich wird die potentielle Parallelität eines DSP nie automatisch vollständig ausgenutzt werden können.



Alternativ können Algorithmen in Form eines Blockdiagramms dargestellt werden (Bild oben). Im System COSSAP werden solche Blöcke zur compilierten Simulation ohnehin zusammengestellt. Ein Codegenerator, z.B. DESCARTES, kann nun aus dem bekannten Zusammenhang der Blöcke und vorprogrammiertem C-Code für jeden Block das Programm für einen oder mehrere DSPs generieren. Der Codegenerator benötigt dann zusätzlich die Information, auf welchem Prozessor welcher Block laufen soll (Assignment). Ein Scheduler sorgt dann für die korrekte zeitliche Anordnung innerhalb eines DSP. Der C-Code jedes Prozessors inclusive Kommunikationsroutinen muß dann, wie im Einzelprozessor-System, compiliert und assembliert werden. Die Schnittstellen zwischen Blöcken, die auf verschiedenen Prozessoren zugeordnet wurden und Daten untereinander austauschen, sind sorgfältig zu wählen; möglichst sollten keine Daten innerhalb einer Iteration übergeben werden. Die Retiming-Transformation kann in diesem Punkt hilfreich eingesetzt werden.

2.2 Mikroprozessoren und Transputer

Heutzutage erhältliche Mikroprozessoren decken ein weites Leistungsspektrum ab. Man unterscheidet grob nach der Art des Instruktionsdecoders: CISC (Complex Instruction Set Coding) oder RISC (Reduced Instruction Set Coding). Von Bedeutung ist weiterhin die Busarchitektur: von-Neumann- oder Harvard-Architektur (Daten- und Programmspeicher gemeinsam oder getrennt), externe und interne Datenwortbreite (4,8,16,32,64 Bit), integrierte Fließkommazahlenunterstützung {floating point support}, Taktrate, Datentransferbandbreite, internes oder externes ROM und RAM. Es können weiterhin Peripherieeinheiten auf dem Chip integriert sein, z.B. DMA-Controller, Interrupt-Controller, Serielle Ports etc. Der bedeutende Unterschied zu DSP's ist, daß kein schneller Hardware-Multiplizierer vorhanden ist, sondern Multiplikationen mittels Mikrocode aus Additionen zusammengesetzt werden.

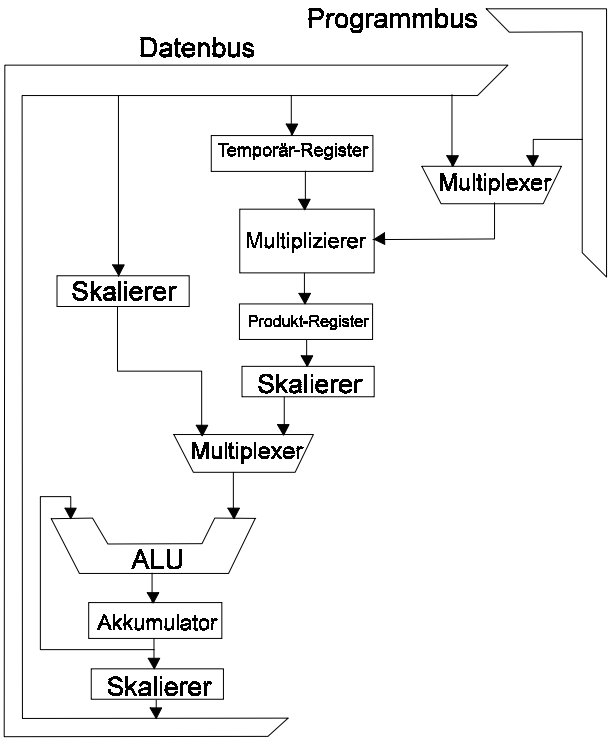
Transputer sind spezialisierte Prozessoren, die mehrere Kommunikationsports hoher Übertragungsbandbreite besitzen, sogenannte Transputer-Links [7]. Für parallel ausgeführte Prozesse mit hohem Kommunikationsbedarf werden diese bevorzugt eingesetzt. Mit vernetzten Transputern können relativ einfach skalierbare Systeme aufgebaut werden, d.h. die Anzahl der Transputer ist in weitem Rahmen anpaßbar. Für Hochleistungsparallelrechner wird häufig ein Transputer mit einem gewöhnlichen RISC-Prozessor zu einem (eng gekoppelten) Cluster zusammengefügt, wobei der Prozessor die Datenverarbeitung durchführt und der Transputer für den Datenaustausch mit den anderen Clustern zuständig ist [13].

2.3 Digitale Signalprozessoren

Für den Einsatz in der digitalen Signalverarbeitung werden bevorzugt Digitale Signalprozessoren eingesetzt. Sie ermöglichen eine schnelle Echtzeitverarbeitung, z.B. für Zwecke der Sprachcodierung (Quellen- und Kanalcodierung), digitalen Modulation (Trellis-codierte Modulation), Sprachanalyse und -synthese usw. Übliche DSP's haben einen Befehlsdurchsatz bis zu 100 MIPS {million instructions per second} respektive MFLOPS {million floating point operations} bei Gleitkomma-DSP's.

Charakteristisch für DSP's sind:

- integrierter schneller Hardware-Multiplizierer (Ergebnis innerhalb eines Zyklus)
- Harvard-Architektur (getrennter Befehls- und Datenspeicher-Bus)
- Befehls-Ausführungszeit ist ein Maschinenzyklus
- Befehle zur Skalierung von Operanden und Ergebnissen ohne Extrazeitverlust
- Intensive Nutzung der Instruktions-Pipeline
- Festkomma- oder Gleitkommaformat für die Zahlendarstellung
- Bit-reversed Adressierung durch 'Reverse-Carry-Propagation'
- Serielle Kommunikationsports



Die Multiplizierer-ALU-Einheit (siehe Bild: vereinfachte Darstellung des Rechenwerks eines Festkomma-DSP) bildet den Kern für arithmetische Operationen. Es gibt für jeden DSP Befehle, die eine Multiplikation und eine Addition ein einem Zyklus ausführen, beim TMS320C50 [4] beispielsweise:

MAC *Multiplikator*, *Multiplikand*

Daneben existieren Befehle, die gleichzeitig noch Daten verschieben können oder Adreßzeiger manipulieren. Der kritische Pfad Multiplizierer-ALU ist der begrenzende Faktor für die Zykluszeit. Daher ist in diesem Pfad bei diesem Prozessor ein Register eingefügt worden (Pipelining). Die ALU kann deshalb nur das Ergebnis einer Multiplikation aus einer vorausgegangenen Iteration benutzen. An der Leistungsfähigkeit dieser Einheiten mißt sich der Durchsatz, zum Beispiel sind für digitale Filter hauptsächlich Operationen wie Multiplikation+Addition und Verschiebungen nötig. Aus diesem Grund sollten diese Operationen in möglichst einem Zyklus abgearbeitet werden. Wenn ein Argument aus dem Programmspeicher entnommen werden muß, dann sind, wie bei obigem Befehl, zwei Zyklen nötig. Der Programmierer hat die mühsame Aufgabe, für jede Adressierungsart und jeden angesprochenen Speicher die resultierenden Zyklen zu berücksichtigen.

Die Pipeline unterstützt die schnelle Ausführung, indem die Bearbeitungsschritte *instruction fetch, instruction decode, operand fetch, instruction execute* zeitlich versetzt aber parallel für aufeinanderfolgende Befehle ausgeführt werden. Darstellung der Pipeline [4]:

Zyklus:	0	1	2	3
	fetch 0	fetch 1	fetch 2	fetch 3
	decode -1	decode 0	decode 1	decode 2
	operand -2	operand -1	operand 0	operand 1
	execute -3	execute -2	execute -1	execute 0

Der Inhalt der Pipeline ist nicht immer gültig, denn erst nach der Ausführung eines Sprungbefehls steht die nächste Programmadresse fest. Alle nachfolgenden Befehle in der Pipeline müssen dann entfernt werden und die Pipeline muß neu gefüllt werden. Als Folge dauert ein Sprungbefehl statt einem vier Zyklen. Eine Zeitverbesserung bringen die verzögerten Sprungbefehle {delayed branch}. Der Instruktionszeiger ändert sich dabei erst, nachdem der nachfolgende Befehl ausgeführt wurde.

Der *Interlocking*-Mechanismus [2] (bei den Prozessoren von Texas Instruments anzutreffen) sorgt dafür, daß keine Pipelinekonflikte auftreten, wenn z.B. ein Operand benutzt wird, der erst vom vorherigen Befehl gesetzt wird. In der Regel hat jedoch der Programmierer dafür Sorge zu tragen, daß keine Argumente benutzt werden, die noch nicht stabil sind, weil sie erst im vorherigen Befehl berechnet wurden. Andere Programmiermodelle berücksichtigen in der Assembler-Schreibweise direkt die Pipeline. Time-Stationary Coding (üblich bei den DSP's von Motorola, NEC und einigen von AT&T) erfordert, daß in jeder Assemblerzeile der Pipelinezustand erkennbar wird [2]. In einer Zeile ist die durchzuführende Operation angegeben (execute) und die Argumente für den nächsten Befehl (fetch). Im Gegensatz dazu werden bei Data-Stationary Coding (AT&T DSP32) in jeder Zeile die Operationen und benötigten Daten angegeben (diese Operation kann dann mehrere Zyklen dauern), aber die nachfolgenden Operationen laufen zeitversetzt parallel in der Pipeline ab, so daß der Durchsatz bei einer Instruktion pro Zyklus liegt.

Die Harvardarchitektur ermöglicht es, im Gegensatz zur von-Neumann-Architektur, gleichzeitig ein neues Instruktionswort und einen Operanden aus dem jeweiligen Speicher zu laden, da jeweils für Daten- und Programmspeicher getrennte Daten- und Adreßbusse vorhanden sind. Einige DSP's haben auf dem Chip ein DARAM [4] {dual access RAM}, auf das in einem Zyklus zweimal zugegriffen werden kann; dadurch können zwei Operanden gleichzeitig geladen werden. In der Regel werden beim Zugriff auf externe Speicher- oder Peripheriebausteine zusätzliche Wartezyklen {waitstates} benötigt, da das Timing der herausgeführten Busleitungen für langsamere Komponenten verzögert werden muß.

Zur Kommunikation mit Peripherie und anderen Prozessoren stehen folgende Möglichkeiten zur Verfügung:

- Serieller Port
- Time Division Multiplex Serial Port
- Shared Memory bzw. gemeinsamer Adreßraum

Ein serieller Port erlaubt nur den Anschluß einer Datenendeinrichtung (DEE), der Transfer kann dabei synchron oder asynchron verlaufen. Der serielle TDM {time division multiplex}-Port [4] kann mehrere Geräte an einem seriellen Bus verbinden. In jedem Zeitschlitz darf nur

eine Einheit senden, die anderen Sender müssen währenddessen hochohmig bleiben. Der Transfer geschieht hier synchron zu einem Datentakt, zur Rahmensynchronisation steht ebenfalls ein Signal zur Verfügung. Shared Memory ist RAM-Speicher, der im Adreßraum aller angeschlossenen DSP's liegt. Er läßt sich als globaler Speicher betrachten, auf den alle Prozessoren zugreifen können, im Gegensatz zum lokalen (meist internen) Speicher. Zum automatischen Datentransfer, während das Programm parallel weiterläuft, kann man bei vielen DSP's eine DMA-Einheit benutzen {direct memory access}, die unabhängig vom Prozessorkern arbeitet (asynchroner Transfer).

2.4 Parallelrechner

Mit mehreren Standard-Mikroprozessoren, Transputern oder DSP's aufgebaute Rechnersysteme werden als Parallelrechner bezeichnet. Je nach Anzahl der Prozessoren und deren Leistung (reziproke Zykluszeit) unterscheidet man [13] :

- Massiv parallele Rechner (hohe Prozessoranzahl)
- Multi-Vektorrechner (wenige Hochleistungs-Prozessoren)
- Workstation-Cluster (über LAN o.ä. lose gekoppelte Rechner)

Eine andere Klassifikation bezieht sich auf die Unterschiede in Anweisungs- und Datenströmen nach Flynn, wie in [14],[16] beschrieben:

- SISD = Single Instruction Single Data = Ein-Prozessor-System
- MISD = Multiple Instruction Single Data (Unpraktisch und bedeutungslos)
- SIMD = Single Instruction Multiple Data
- MIMD = Multiple Instruction Multiple Data

SIMD-Architekturen haben in der Regel einen zentralen Controller, mehrere Prozessoren und ein Verbindungsnetzwerk zum Datenaustausch, wobei der Controller einzelne Befehle an alle Prozessoren verschickt. Typische Verwendung z.B. für assoziative Speicher (inhaltsadressiert).

MIMD-Architekturen enthalten mehrere Prozessoren, die unabhängige Datenströme bearbeiten können und dadurch weitgehend autonom arbeiten. Sie sind asynchrone Computer, charakterisiert durch dezentralisierte Hardware-Steuerung. Die Synchronisation der Daten wird durch Nachrichten {messages} oder Semaphore abgewickelt. Dies ist die passende Klassifikation für Multi-DSP-Systeme.

Die Kommunikationsleistung im Verhältnis zur Rechenleistung erlaubt eine weitere Klassifizierung. Es können mehrere Prozessoren einen gemeinsamen Speicher besitzen {shared memory} und über diesen sehr schnell Daten austauschen. Wegen der begrenzten Speicherzugriffszeit sind solche Systeme nicht unbegrenzt skalierbar. Alternativ kann die Kommunikation (bei Systemen mit verteiltem Speicher {distributed memory}) über Transfer-Kanäle der Prozessoren geschehen (serielle oder parallele Ports), oder über Peripheriebausteine (serieller Transfer, Netzwerktreiber). Die Kommunikationsrate ist sehr viel geringer als bei geteiltem Speicher. Die Klassifikation hierzu heißt [13]:

- Eng gekoppelte Systeme (gemeinsamer Speicher)
- Lose gekoppelte Systeme (verteilter Speicher)

Für eng gekoppelte Systeme lassen sich die Prozesse effektiv sehr fein granular aufteilen, da die nötige Kommunikationsleistung zum Datenaustausch bereitsteht.

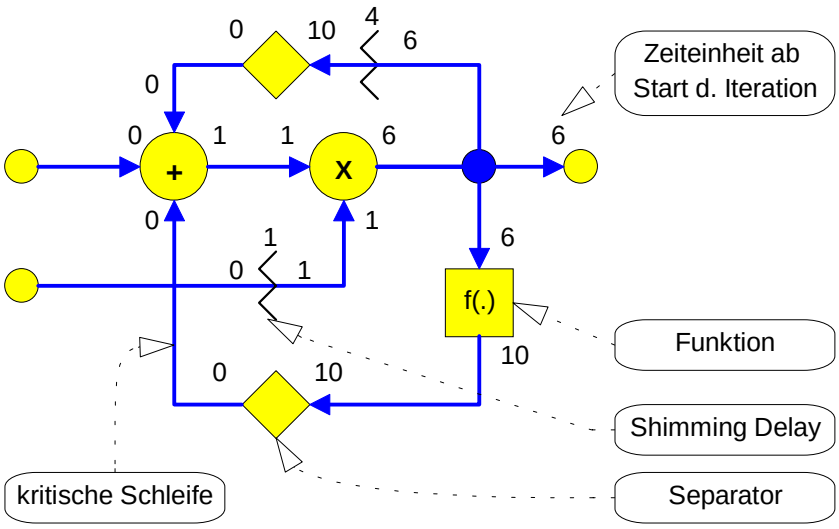
Lose gekoppelte Systeme sollten größere Programmblöcke bearbeiten, die dann sporadisch Daten mit geringer Bandbreite übertragen.
Für vernetzte Systeme gibt es verschiedene Verbindungstopologien: Tree, ring, mesh, hypercube, butterfly [6],[14],[16]. Diese sind je nach Kommunikationsumfang der Applikation unterschiedlich vorteilhaft.

2.5 VLSI

Für Hardwareimplementationen von Algorithmen oder Schaltungen der digitalen Signalverarbeitung auf VLSI-Basis ist das Datenflußkonzept ebenfalls anwendbar. Synchrone digitale Schaltungen, ob azyklisch oder zyklisch, können durch einen Datenflußgraphen dargestellt werden [33],[34],[36]. Dazu werden Knoten für jeden Verarbeitungsblock (Schaltnetz) erstellt; diese bekommen dann eine Ausführungszeit zugewiesen, die der Signallaufzeit {propagation, processing delay} des Blocks entspricht. Jede Signalverbindung zwischen zwei Blöcken wird zu einer gerichteten Kante. Latches oder FIFO-Schieberegister {first in first out} werden durch ein bzw. mehrere Delays modelliert. Bereits vorhandene Pipelinestufen weisen auf der zugehörigen Kante ebenfalls ein Delay auf. Es ist darauf zu achten, daß Delays Speicherelementen entsprechen und nicht mit Laufzeitgliedern verwechselt werden. Sie haben einerseits die Funktion eines Separators zwischen verschiedenen Iterationen (siehe Bild), als auch ein zusätzlich mögliches ‘shimming delay’, falls Laufzeiten ausgeglichen werden müssen [35].

Mit Methoden, die für datenflußorientierte Systeme benutzt werden, können Aufgaben wie die Datensynchronisierung oder Retiming gelöst werden. Zur Datensynchronisierung [34] müssen Schieberegister in den Signalpfaden eingefügt werden, die ihr Ergebnis früher zu einem gemeinsamen Zielpunkt liefern als ein paralleler Pfad mit längerer Durchlaufzeit. Retiming kann bei synchronen Schaltungen angewendet werden, um die Anzahl der Speicherelemente zu minimieren oder den Durchsatz zu erhöhen [33]. Bei azyklischen Schaltungen entspricht dies dem *Pipelining*.

Der Multiratenfall ist denkbar, wenn in einer Schaltung Daten mit unterschiedlicher Rate ausgetauscht werden (die Raten von Sender und Empfänger auf einer Kante sind dabei identisch), wenn eine Wandlung seriell-parallel oder umgekehrt durchgeführt wird, oder wenn ein Multiplexer eingesetzt wird (z.B. für Ressource Sharing).


0
10
4
6
0
0
1
1
6
6
+
X
0
1
1
0
1
6
f(.)
10
0
10
Zeiteinheit ab
Start d. Iteration
Funktion
Shimming Delay
Separator
kritische Schleife

3 Datenflußorientierte Programme

3.1 Voraussetzungen

Das Datenfluß-Modell eines Programms fördert direkt die potentielle Parallelität zutage, die in einer Multiprozessorimplementation ausgenutzt werden kann. Die Methoden dazu werden in den folgenden Kapiteln behandelt. Nicht jedes Programm kann jedoch im Datenfluß-konzept betrachtet werden. Die Voraussetzungen dafür sind [17],[22]:

- Das Programm muß sich in Blöcke (Knoten) aufteilen lassen, die in ihrer Ausführungs-rangordnung festliegen. Die sich daraus ergebenden Präzedenzbeziehungen {precedence constraints} werden durch Kanten im Datenflußgraphen dargestellt.
- Das Modell ist datengesteuert {data-driven}. Das bedeutet, daß ein Block nur dann gestartet werden darf, wenn genügend Eingangswerte vorhanden sind. Die Abhängigkeit von Daten, die nicht über datenflußgetriggerte Eingänge auf den Block wirken, ist nicht erlaubt.
- Es darf keine bedingten Ausführungen von Blöcken {if-then-else-clauses} geben, außer innerhalb eines Blocks (und daher von außen unsichtbar). Daher ist ein datenflußorientier-tes Programm deterministisch.
- Die Anzahl der eingelesenen und ausgegebenen Daten pro Aktivierung eines Knotens muß feststehen (Regulärer Datenfluß = Lee's Synchroner Datenfluß)
- Die Datenkanäle (Kanten im Graph) dürfen nur abzählbare Objekte (durchaus auch im objektorientierten Sinn) transportieren und speichern. Unquantisierte und nicht abgetastete Größen wie analoge Spannungsverläufe etc. sind nicht modellierbar.

Ereignisgesteuerte Programme, wie sie in modernen Dialogsystemen eingesetzt werden, erfüllen diese Bedingungen nicht (in den Ursachen liegt auch ihre Stärke bzw. Flexibilität). Es lassen sich auch reichlich andere Probleme finden, die durch ihre Art des Datenaustauschs Schwierigkeiten machen.

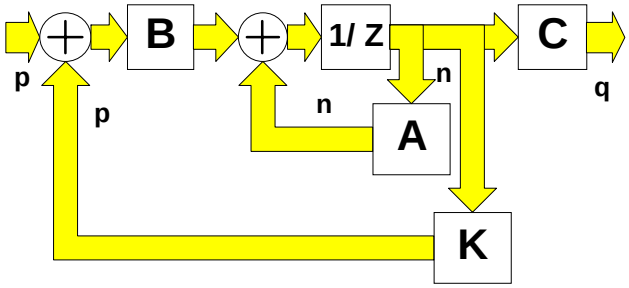
Aufgrund der nur statistisch beschreibbaren Eigenschaften stochastischer Netze werden im folgenden nur Systeme mit bekannten Raten und Ausführungszeiten betrachtet. Die statistische Behandlung stochastischer Systeme liegt jenseits der Thematik dieser Arbeit, näheres ist bei Murata et al. zu finden [37-S.570].

Die Anwendungen sind nicht auf die digitale Signalverarbeitung beschränkt. Mit Daten-flußgraphen oder der Obermenge, den Petri-Netzen [37], lassen sich zahlreiche Probleme beschreiben, die z.B. aus dem Gebiet „Operations Research" (Handhabungstechnik) stammen.

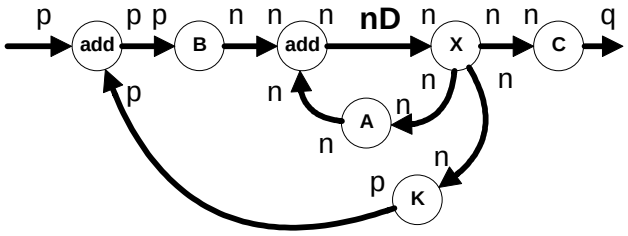
3.2 Digitale Signalverarbeitung

In der digitalen Signalverarbeitung werden Signale in abgetasteter und quantisierter Form verarbeitet. Die daraus resultierenden Datenströme liefern im eingeschwungenen Zustand (hier: kurz nach dem Einschalten) Abtastwerte in regelmäßigen Abständen. Ein synchrones System vorausgesetzt, stehen die im System anzutreffenden Abtastraten in festem rationalen Verhältnis.

Ein nachrichtentechnisches oder regelungstechnisches Blockdiagramm (wie in der Systemtheorie behandelt) läßt sich für solche Abtastsysteme in einen Datenflußgraphen umwandeln.



Hieraus resultieren Programme, die dann auf einen oder mehrere DSP's portiert werden können. Je nach Komplexität der verwendeten Blöcke kann ein Knoten im Datenflußgraph einem Assemblerbefehl (ADD, MUL o.ä.) entsprechen, oder einer größeren Einheit (Filter, Transcoder o.ä.). Letztere kann als hierarchischer Knoten angesehen werden, der seinerseits ein Netz aus Knoten und Kanten enthält. Assemblerbefehle sind *atomare* Knoten.



Die Bilder zeigen als Beispiel die Transformation zu einem Datenflußgraphen für einen digitalen Regelkreis mit mehrdimensionalem Zustand. Speicherelemente werden durch Delays (nD) modelliert. Der Knoten 'X' ist ein Verteilerknoten ohne Ausführungszeit. Der Interpretation eines solchen Graphen ist das nächste Kapitel gewidmet.

Der Datenaustausch mit unterschiedlichen Raten (Multiratenfall) tritt in der Praxis häufig auf. Ein Bitstrom hat z.B. vor und nach Quellcodierung oder Kanalcodierung unterschiediche Bitrate. Interpolations- oder Dezimierungsfilter haben verschiedene Abtastraten an Ein- und Ausgang. An Multiplexern treten ebenfalls multiple Raten auf. Filterbänke wie QMF weisen stufenweise Ratenreduktionen auf.

3.3 Andere Anwendungen

Zur Steuerung eines Betriebsablaufs oder einer Fertigungsstraße werden Methoden des Operations Research herangezogen. Die graphische Repräsentation der Prioritäten bzw. der Reihenfolge der durchzuführenden Aktivitäten ist ein gerichteter Graph. Viele Fragestellungen, die man in der Signalverarbeitung hat, treten hier genauso auf. Dazu gehören Fragen wie die nach der maximalen Zeit, innerhalb der ein Projekt beendet werden kann, und wieviele Ressourcen dafür benötigt werden. Wie soll der Ablauf organisiert werden? Wie groß ist der maximale Produktions-Durchsatz? Auf diesem Gebiet gibt es eine Menge an Foschungsergebnissen, nicht zuletzt weil die Behandlung mit Flußgraphen oder Petri-Netzen interdisziplinär von Mathematikern, Informatikern, Wirtschaftswissenschaftlern und Ingenieuren vorangetrieben wurde.

4 Modelle und Eigenschaften

4.1 Motivation

Zur Beschreibung der strukturellen Eigenschaften eines datenflußorientierten Programms existieren Modelle, die von den eigentlichen Berechnungen abstrahieren und nur noch Informationen über die Zusammenhänge innerhalb des Programms enthalten.

Es sind in der Literatur verschiedene Modelle anzutreffen, die alle geeignet sind, dieselben Sachverhalte zu beschreiben. Das Konzept der Petri-Netze [37] aus dem Jahr 1962 ist nicht nur das älteste Modell, sondern hat auch die weitesten Möglichkeiten zur Modellierung. Unabhängig davon sind die von Karp und Miller [43] 1966 eingeführten „Computation Graphs" entwickelt worden, die etwas allgemeiner sind als das von Lee [20] 1986 speziell für DSP-Anwendungen konzipierte Modell des „synchronen Datenflusses" {synchronous[1] data flow}.

Im Folgenden wird als Modell ein Datenflußgraph (DFG) {data flow graph} benutzt, der eine graphische Repräsentation eines zugrundeliegenden Algorithmus ist. Zu der Verwandschaft mit den „Nonordinary Marked Graphs", einer Unterklasse der allgemeinen Petri-Netze, ist später mehr zu erfahren.

Das Prinzip des datenflußorientierten[2] Programmkonzeptes beruht auf der Tatsache, daß zu einer gegebenen Menge von Eingangswerten eine Menge von Ausgangswerten berechnet werden soll. Die Berechnungsvorschrift besteht in der Regel aus mehreren aufeinanderfolgenden Teilaufgaben oder läßt sich in Teilaufgaben zerlegen. Jeder Teilaufgabe wird im Datenflußgraph durch einen Knoten[3] (engl. node, vertex-vertices) dargestellt. Der jeder Teilaufgabe gewidmete Programmteil kann erst dann sinnvoll arbeiten, wenn seine Eingangswerte vollständig vorliegen. Die benötigten Werte werden von diesem Teil (Knoten) aus einem Zwischenspeicher eingelesen, verarbeitet, und nach einer gewissen Bearbeitungszeit werden die Ausgangswerte in dafür vorgesehene Speicher geschrieben, die von weiteren Knoten gelesen werden können. Die Datenpfade, entlang deren Daten bewegt werden, werden im DFG durch gerichtete Kanten[4] (engl. arc, edge) dargestellt. Der Speicher, der zu jeder Kante gehört, muß als FIFO[5] implementiert sein, um die zeitliche Reihenfolge der Daten zu berücksichtigen. Falls bei einem speziellen Schedul 'auf dieser Kante' nur maximal ein Wert gespeichert wird, sind natürlich einfachere Implementationen möglich.

Weil die Programmausführung von der Verfügbarkeit von Daten abhängt, nennt man solche Datenflußprogramme datengesteuert [17] {data-driven}. Der Spezialfall des *regulären (synchronen)* Datenflusses (SDF) ergibt sich durch die genaue Kenntnis über die Anzahl der von allen Knoten pro Ausführung produzierten und konsumierten Dateneinheiten (*token*). Diese als *output-rate* und *input-rate* bezeichneten Werte müssen zur Compilationszeit bekannt sein und dürfen nicht von den Daten abhängen; außerdem müssen sie untereinander in ganzzahligem Verhältnis stehen[6] [17]. Die Knotenoperationen müssen frei von Seiteneffekten sein; der einzige Einfuß eines Knotens auf einen anderen ist die Übergabe von Daten

[1] Der Datenfluß ist nicht synchron im Sinne eines festen Taktes, daher ist der Begriff 'regulär' angebrachter.
[2] Das datenflußorientierte Konzept steht im Gegensatz zum objektorientierten Konzept und der historischen sequentiellen Programmierung.
[3] Der Begriff Knoten wurde aus der Graphentheorie übernommen, und wird durch einen Kreis dargestellt.
[4] Eine Kante verbindet immer genau zwei Knoten; Sie wird durch eine Linie mit Pfeilspitze dargestellt
[5] First In First Out; Praxis: Schieberegister oder Ringpuffer.
[6] Daher verwendet man nur ganzzahlige Werte als (normierte) Raten

über Kanten. In diesem Zusammenhang muß darauf hingewiesen werden, daß der sogenannte Kontrollfluß nicht mit dem herkömmlichen Konzept des regulären Datenflusses vereinbar ist, da dieser die Ausführung von Knoten zusätzlich beeinflussen kann. Mit den 'erweiterten Petri-Netzen' wird in [37-S.546] ein dazu hilfreiches Konzept beschrieben.

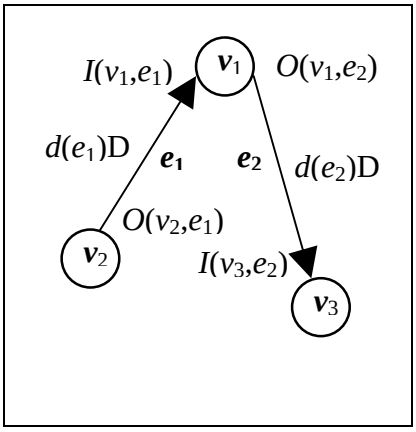
Knoten ohne zulaufende Kanten nennt man Eingangsknoten (Quellen), Knoten ohne abgehende Kanten sind Ausgangsknoten (Senken). In der Signalverarbeitung geht man normalerweise von einem unendlichen Datenstrom am Eingang und Ausgang aus, dabei werden die Daten i.d.R. taktsynchron geliefert und abgeholt. Die Werte 'input/output-rate' haben daher tatsächliche Bedeutung als Rate, bestimmen sie doch zusammen mit der Anzahl der Knotenaufrufe pro Periode und der Periodendauer T die tatsächliche Datenabtastrate an dem entsprechenden Kanten-Knoten-Übergang.

Im Datenfluß-Konzept werden gewichtete[7] Graphen benutzt. Jede Kante hat ein Gewicht in Form einer natürlichen Zahl d, die angibt, wieviel Dateneinheiten sich zur Zeit im Zwischenspeicher befinden. Diese Dateneinheiten werden in der Literatur *delay* oder *token* genannt. Man kennzeichnet die Anzahl der Delays durch Angabe 'dD', d ist dabei eine natürliche Zahl. Die Verteilung der Delays ändert sich während des Programmablaufs; bei konsistenten DFG sollte zu Anfang jeder Periode wieder dieselbe Delayverteilung auftreten. Die Delayverteilung {marking} ist eine Art Zustand des Graphen (ähnlich wie Zustände bei zeitdiskreten Systemen). Diese Zustände können, im Zustandsraum betrachtet, Aufschluß über Eigenschaften des Systems geben (siehe Erreichbarkeit). Desweiteren gibt die Anfangsdelay-verteilung einen zeitlichen Versatz zwischen Daten an, ähnlich wie $\mathbf{z}^{-1}$-Elemente in Blockdiagrammen der digitalen Signalverarbeitung. Eigenschaften, die nicht von der Delayverteilung, sondern nur von der zugrundeliegenden Struktur abhängen, werden *strukturelle* Eigenschaften genannt.

Viele Fragestellungen über DFG beschäftigen sich mit der Korrektheit eines Graphen (Deadlockfreiheit, begrenzter Speicherinhalt, Ratenkonsistenz), der Performance (Durchsatz, iteration period), Transformationen zur Verbesserung von Eigenschaften und Scheduling des Programmes auf einem oder mehreren Prozessoren.

4.2 Definitionen

4.2.1 Datenflußgraph

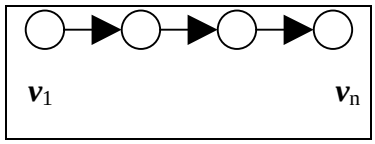


Ein (regulärer) Datenflußgraph (SDFG) [17] ist ein gerichteter und gewichteter Graph [58]: $G=\{V(T),E(I,O,d)\}$. Die Knoten $v \in V$ modellieren Datenoperationen, sie sind mit der Ausführungszeit $T(v)$ gewichtet; die Kanten $e \in E$ modellieren Signalverbindungen zwischen den Funktionselementen. Bei jeder Aktivierung von Knoten v werden $I(v,e_i)$ Werte von jeder Eingangskante e_i konsumiert (eingelesen und aus dem Speicher entfernt) und $O(v,e_o)$ Werte auf den Ausgangskanten e_o ausgegeben. Ein Knoten darf nur dann aktiviert werden, wenn dadurch keine negativen Delays entstehen. Die Raten I und O werden als konstant vorausgesetzt (Lee's Synchronität). Eine gerichtete Kante von Knoten u nach Knoten v wird mit $e(u,v)$ bezeichnet. Jede Kante besitzt außerdem $d(e) \geq 0$ Delays

[7] Gewichte sind mit Kanten e oder Knoten v verknüpfte Werte. Gewichtsfunktion $f_e(e)$, $f_v(v)$

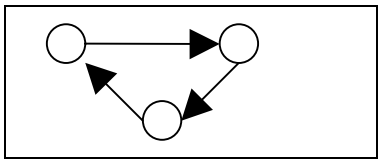
(Anfangsbedingung; initial samples). Ein einziges Delay entspricht einer ‘Phasenverschiebung’ um ein Sample.

4.2.2 Pfad {path}



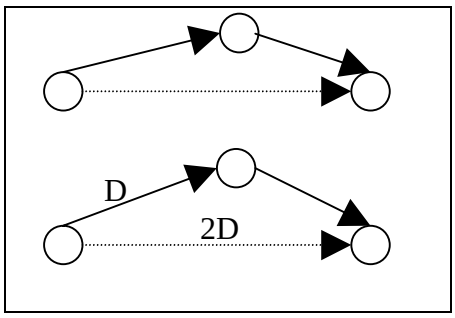
Ein Pfad $p(v_1,...,v_n)$ von Knoten v_1 nach Knoten v_n liegt dann vor, wenn ein Weg (Menge von gerichteten Kanten) von v_1 über Zwischenknoten zum Knoten v_n existiert, wenn also alle Kanten $(v_i,v_{i+1}) \in E$ für $i \in \{1,..,n\text{-}1\}$. Anfangs- und Endknoten dürfen nicht auch Zwischenknoten sein.

4.2.3 Schleife {loop, circuit}



Eine *gerichtete Schleife* ist ein Pfad, bei dem Anfangs- und Endknoten identisch sind und mindestens eine Kante zum Pfad gehört. Die Betrachtung von Schleifen ist zur Betrachtung von Lebendigkeit und Iterationsgrenze von großer Bedeutung. *Ungerichtete Schleifen* erfordern nicht, daß die Kanten in dieselbe Richtung orientert sind.

4.2.4 Transitive Kanten



Kanten, die für die Präzedenzordnung keine Rolle spielen, weil andere Kanten schon dieselbe Präzedenz festlegen (dieselbe Summe von Delays), sind *transitive Kanten* [19]. Sie geben zwar an, daß ein Datenaustausch zwischen zwei Knoten stattfindet, sind aber für Zwecke des Schedulings überflüssig und können entfernt werden (Bild oben). *Erweiterte* transitive Kanten {extended transitive arcs} enthalten mehr Delays als ein anderer Pfad zwischen denselben Knoten und beschreiben daher keine neuen Präzedenzen (Bild unten).

4.2.5 Einheitsraten-DFG {unit-rate DFG}

Ein Einheitsraten-Datenflußgraph besitzt an jeder Kante dieselbe konstante Ein- und Ausgangsrate Eins. Dieser Typ wird auch als homogener DFG bezeichnet [17] und ist identisch mit den Marked Graphs der Petri-Netze. Es ist ein Spezialfall des Singleraten-DFG.

4.2.6 Singleraten-DFG

Im Gegensatz zum Einheitsraten-DFG dürfen die Raten auch größer als Eins sein, allerdings muß an jeder Kante mit derselben Rate produziert und konsumiert werden. Der q-Vektor[8] besteht bei Singleraten-DFG aus Einsen in jeder Komponente.

[8] Der q-Vektor gibt an, wie oft jeder Knoten während einer Periode aktiviert werden muß

4.2.7 Multiraten-DFG

Ein Multiraten-Datenfluß-Graph kennt keine Einschränkungen der Raten, allerdings besteht hier das Problem der Ratenkonsistenz (das bedeutet, daß nur bestimmte Konstellationen der Raten einen gültigen Datenflußgraphen ergeben, siehe *Inzidenz-Matrix*). Sämtliche Fragestellungen sind in diesem allgemeinen Fall komplizierter und teilweise bis heute noch nicht vollständig gelöst. Ihr Petri-Netz-Äquivalent heißt 'Nonordinary Marked Graph'.

4.2.8 Zusammenhängende Komponente {connected component}

Das ist ein Teilgraph, bei dem alle Knoten so miteinander verbunden sind, daß jeder Knoten über ungerichtete (beliebig gerichtete) Kanten von jedem anderen erreicht werden kann. Hier werden nur Graphen mit einer zusammenhängenden Komponente betrachtet, da zwei oder mehr Komponenten wieder eigenständige DFG's darstellen.

4.2.9 Maximal stark zusammenhängende Komponente {strongly connected component}

Das ist ein Teilgraph, bei dem für alle Knotenpaare (u,v) ein Pfad von $u \rightarrow v$ und von $v \rightarrow u$ existiert. Ein solcher Graph hat keine separaten Anfangs- oder Endknoten, jeder Knoten hat also (mindestens) einen Vorgänger und Nachfolger. Desweiteren liegt jeder Knoten und jede Kante in mindestens einer (fundamentalen) Schleife. Knoten, die nicht zu einer maximal stark zusammenhängenden Komponente aus mindestes zwei Knoten gehören, liegen auf einem azyklischen Pfad.

4.2.10 Ausführungszeit

Die Ausführungszeit eines Knotens $T(v)$, $v \in V$, ist die Zeit zwischen der Aktivierung (und sofortigem Auslesen der Eingangsdaten) und der Erzeugung der Ausgangswerte auf den auslaufenden Kanten. Hier wird keine partielle Laufzeit betrachtet, bei der verschiedene Ausgangskanten eines Knotens zu unterschiedlichen Zeiten fertig werden können [20],[27]; ein solches Verhalten könnte durch einen hierarchischen Knoten modelliert werden. Für die Ausführungszeit gibt es drei Fälle:

- Die Ausführungszeiten sind konstant und für jeden Knoten gleich
- Die Ausführungszeiten sind konstant aber u.U. für jeden Knoten verschieden
- Die Ausführungszeiten sind nicht bekannt oder hängen von den eingelesenen Daten ab

Die Komplexität und Qualität eines Schedulalgorithmus wird maßgeblich durch die drei Fälle bestimmt, z.B. kann im letzten Fall nur eine geschätzte Zeit angenommen werden; daher kann das erhaltene Schedul i.d.R. nicht *optimal*[9] sein.

4.2.11 Iterationsperiode

Die Iterationsperiode {iteration period} ist die Zeit, innerhalb der jeder Knoten v_i genau q_i mal (eine Iteration) ausgeführt wurde (siehe später). In einem Schedul wiederholt sich nach jeder Periode (ganzzahliges Vielfaches einer Iteration) die Ausführungssequenz der Knoten. Diese Zeit ist das Reziproke des Durchsatzes (meßbar in Iterationen pro Zeit).

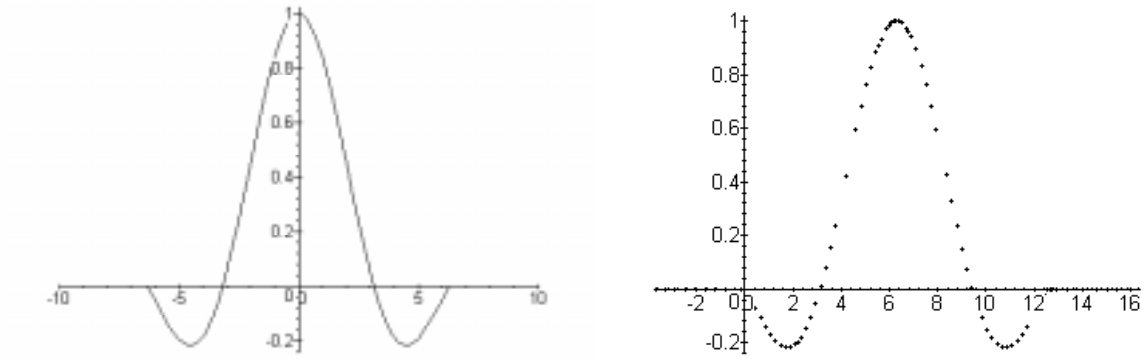
[9] Bei einem optimalen Schedul wird die iteration period bound erreicht (höchstmöglicher Durchsatz)

4.2.12 Iterationsperiodengrenze {iteration period bound}

Das ist die untere Grenze für die Iterationsperiode bei unbegrenzten Ressourcen. Diese Grenze ist durch den Graphen und die Delayverteilung eindeutig bestimmt und kann nicht unterschritten werden (auch nicht bei unendlich vielen Prozessoren). Durch geeignete Neuverteilung der Delays (Retiming) und i.A. überlappendes Schedul kann diese Grenze erreicht werden. Sie kann für Singleraten-Graphen bestimmt werden, wenn $T(v) \ \forall \ v \in V$ bekannt ist. Näheres zu den Grenzen der Iterationsperiode ist im eigenständigen Kapitel „Durchsatz“ zu finden.

4.2.13 Latenz {latency}

Die Latenz ist die Zeit zwischen dem Eintreffen eines Eingangswertes und der Ausgabe des ersten zu diesem Eingangswert korrespondierenden Ausgangswertes. Wegen der Kausalität eines realen Systems kann der zugehörige Ausgangswert (eines LTI-Systems) immer erst später ausgegeben werden. Bei der FIR-Filter-Approximation eines idealen Tiefpasses wird beispielsweise die Sprungantwort dadurch kausal gemacht, daß die mit einer *rect*-Funktion multiplizierte *si*-Funktion so weit nach rechts verschoben wird, bis die Werte vor t=0 zu Null werden. Die Latenz könnte dann als die Zeit interpretiert werden, zu der das Maximum der Stoßantwort auftritt (Bild).



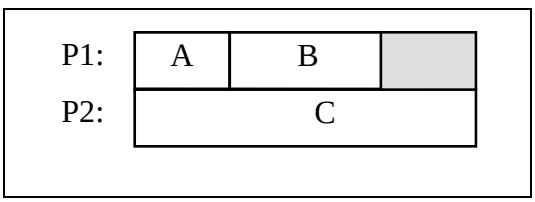
Eine andere Interpretation sieht schon den Wert der verschobenen Stoßantwort bei t=0 als ersten korrespondierenden Ausgangswert an, dann wäre die Latenz im rechten Bild Null und im linken Bild negativ. Erst eine weitere Rechtsverschiebung führt zu positiver Latenz.

Die Latenz von Datenflußprogrammen wird durch die Zeit bestimmt, die vom Startzeitpunkt des Eingangsknotens einer Iteration bis zur Beendigung der Ausführung des Ausgangsknotens derselben Iteration vergeht. In der Gantt-Chart des Scheduls laßt sie sich leicht ablesen.

4.2.14 Schedul {schedule}

Ein Schedul ist eine Zuordnung von Knotenprozessen zu Prozessoren und Festlegung von Startzeitpunkten (Eine Art von ‘Fahrplan’ für den Prozessor). Dargestellt wird das Schedul in der sog. Gantt-Chart (Bild).

Jede Zeile zeigt die Prozesse (A, B, C, ...) eines Prozessors (P1...Pn) an, dabei kann man sich von links nach rechts eine Zeitachse vorstellen. In der Regel stellt man eine Periode des Scheduls dar, jede weitere kann man rechts anschließen. Die Initialprozesse (jedes periodische Schedul beginnt irgend-



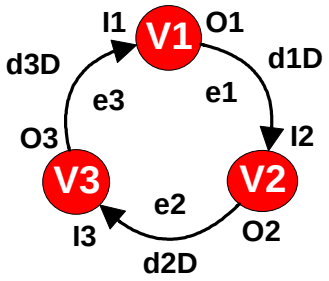
wann mit einer Initialisierung) werden i.d.R. nicht dargestellt.

4.3 Die Inzidenz-Matrix

Ein Datenfluß-Graph kann mathematisch durch eine Inzidenz-Matrix (Topologie-Matrix) beschrieben werden, ähnlich der *Inzidenzmatrix* aus der Graphentheorie (nicht zu verwechseln mit der Adjazenzmatrix [58]). Dazu müssen die Knoten von 1...*n* (*n*=|*V*|=Knotenanzahl) und die Kanten von 1...*m* (*m*=|*E*|=Kantenanzahl) durchnumeriert werden. Die Matrix $\Gamma \in \mathbf{Z}^{m \times n}$ besteht nun aus m Zeilen und n Spalten (Kanten-Knoten-Matrix) und enthält im (*i*,*j*). Eintrag die Anzahl Daten, die Knoten *j* auf der Kante *i* pro Aufruf produziert. Für die konsumierten Daten ist diese Zahl negativ. Wenn keine Verbindung besteht, ist der Eintrag Null [17].

Die Inzidenzmatrix kann auch aus zwei Matrizen zusammengesetzt werden: $g_{ij}=g_{ij}^{+}-g_{ij}^{-}$ (bzw. $\Gamma=\Gamma^{+}-\Gamma^{-}$). Dabei ist $g_{ij}^{+}=O(v_j,e_i)$ und $g_{ij}^{-}=I(v_j,e_i)$. Im Zusammenhang mit Petri-Netzen [37] ist die Transponierte häufiger anzutreffen: $A=\Gamma^{T}$ (Knoten-Kanten-Matrix).

Beispiel:



$$\Gamma^{+}=\begin{pmatrix} O_1 & 0 & 0 \\ 0 & O_2 & 0 \\ 0 & 0 & O_3 \end{pmatrix} \quad \Gamma^{-}=\begin{pmatrix} 0 & I_2 & 0 \\ 0 & 0 & I_3 \\ I_2 & 0 & 0 \end{pmatrix}$$

$$\Rightarrow \Gamma=\begin{pmatrix} O_1 & -I_2 & 0 \\ 0 & O_2 & -I_3 \\ -I_2 & 0 & O_3 \end{pmatrix} \quad , \quad \vec{d}_0=\begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

Ein Problem ist, daß *Selbstschleifen* (self-loops) in der Matrix nur eine Nullzeile hervorrufen und daher nicht erkennbar wird, welcher Knoten eine solche Schleife besitzt. Wenn dieser Tatbestand von Bedeutung ist, dann sollte man die Kante durch einen neuen Knoten (mit Ausführungszeit Null) und zwei Kanten ersetzen, die diesen mit jenem Knoten verbinden. Die gesamten Delays der früheren Kante werden dann auf nur eine der beiden neuen Kanten gesetzt und die Eingangs- und Ausgangsrate des neuen Knotens sollte der Rate auf der alten Kante entsprechen.

4.3.1 Der q-Vektor

Bei einem ‘anständigen’ (ratenkonsistenten) Graph hat Γ den Rang *n*-1=|*V*|-1 (Dies ist auch eine notwendige Bedingung für die Existenz eines Scheduls). Es ist also immer eine Spalte von anderen linear abhängig. Daraus folgt, daß Vektoren $\vec{q}$ existieren mit $\Gamma\vec{q}=\vec{0}$;

$$\vec{q} \in \eta(\Gamma) \qquad \Gamma \cdot \vec{q} = \vec{0} \tag{4.1}$$

$\vec{q}$ ist ein Lösungsvektor des homogenen Gleichungssystems [18] (nullspace). Im Zusammenhang mit Petri-Netzen wird dieser Vektor auch „T-Flow“ oder „T-Invariant“ genannt (T=Transition) [37],[41]. Der Vektor ist in diesem Fall stets positiv [17] und läßt sich zu durch einen Faktor zu einem Vektor aus natürlichen Zahlen (incl. Null) machen (homogenes diophantinisches Gleichungssystem [45]). Der Lösungsraum für $\vec{q}$ hat die Dimension 1

(Spalten-Dimension minus Rang von Γ), daher ist jeder Vektor $\lambda*\vec{q}$ ($\lambda>0$) auch ein Lösungsvektor. Von Interesse sind allerdings immer die ganzzahligen $\vec{q}$ (positive integer vector).

Der kleinste ganzzahlige Vektor $\vec{q}_{\mathrm{m}}$ hat besondere Bedeutung: Seine Komponenten q_j geben an, wie oft ein Knoten v_j innerhalb einer Iteration aufgerufen werden muß (Für Single-raten-Graphen besteht dieser Vektor nur aus Einsen). Weitere Vielfache $J*\vec{q}_{\mathrm{m}}$, $J\in|\mathbf{N}$ geben weitere Aufrufhäufigkeiten für eine Periode (mit Blocking-Faktor J) an. Eine praktische Anwendung ist z.B. die Berechnung der Länge eines Scheduls für einen Prozessor (PASS [18]), die sich mit $\vec{t}\cdot\vec{q}$ (Skalarprodukt) berechnen läßt; dabei ist $\vec{t}$ der Vektor, der alle Ausführungszeiten der Knoten t_j enthält.

Der folgende Algorithmus wurde zur Ermittlung des $\vec{q}$-Vektors ohne Lösung des homogenen Gleichungssystems entwickelt:

Algorithmus zur Bestimmung des $\vec{q}$-Vektors im Multiratenfall:

Der $\vec{q}$-Vektor läßt sich aus dem Graph bestimmen, indem zunächst ein Vektor $q'\in\mathrm{Q}^{|V|}$ ermittelt wird, der hinterher durch Multiplikation mit dem Hauptnenner der Einträge zu $q\in\mathrm{Z}^{|V|}$ gebildet wird:

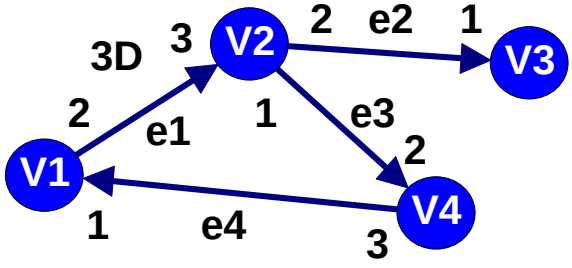
$$\vec{q}=\vec{q}\,'*HN\ ;\ HN=\mathrm{kgV}(\mathrm{Nenner}(q_i'))$$

Die Berechnung zur Bestimmung von q' geschieht rekursiv nach der Methode einer Tiefensuche {depth first search} in Graphen. Beginnend mit einem Startknoten (q_i' dieses Knotens wird willkürlich zu $q_i'=1$ gesetzt) werden über seine angrenzenden Kanten die Nachbarknoten ermittelt. Alle Nachbarknoten, die noch keinen Wert q' zugewiesen bekommen haben, erhalten im ersten Rekursionsschritt ein q' entsprechend der Ratenverhältnisse auf der Verbindungskante e (Bild). Die Nachbarknoten, die schon ein q' zugewiesen hatten, werden auf Konsistenz gemäß der gleichen Gleichung geprüft:

$$\frac{q(Quellknoten(e))}{q(Zielknoten(e))}=\frac{Zielrate(e)}{Quellrate(e)}$$

Ist die Gleichung nicht erfüllt, dann liegt an dieser Kante eine Rateninkonsistenz vor. Der Algorithmus kann dann abbrechen oder die Raten dann so anpassen, daß wieder Konsistenz vorliegt. In einer Schleife werden dann alle Nachbarknoten der Reihe nach rekursiv besucht, wobei die Rekursion zuerst in die Tiefe steigt. Die Rekursions-Abbruchbedingung ist, daß alle Nachbarknoten und -kanten schon besucht wurden. Im Anhang ist der Algorithmus in C++ unter „`DFG::get_q`“ zu finden. Der Aufwand zur Bestimmung ist $O(m+n)=O(|V|+|E|)$.

Beispiel für die Ermittlung von q (Bild):
Startknoten v_1, q_1'=1;
Teste Kante e_1: q_2'=2/3;
Teste Kante e_4: q_4'=1/3;
Nächster Knoten v_2, q_2'=2/3;
Teste Kante e_2: q_3'=4/3;
Teste Kante e_3: q_4'=1/3; konsistent.
Nächster Knoten v_3, q_2'=4/3;
Nächster Knoten v_4, q_4'=1/3;



$$\Rightarrow \vec{q}' = \begin{pmatrix} 1 \\ 2/3 \\ 4/3 \\ 1/3 \end{pmatrix} \quad \Rightarrow \quad \vec{q} = \begin{pmatrix} 3 \\ 2 \\ 4 \\ 1 \end{pmatrix}$$

Bemerkung: Der Algorithmus könnte genauso effektiv auch eine Breitensuche (BFS) benutzen.

4.3.2 Die C-Matrix

Von gleich großer Bedeutung ist der Vektorraum, der von den Vektoren $\vec{y}^T$ aufgespannt wird, die die Gleichung

$$\vec{y}^T \cdot \Gamma = \vec{0}^T = (0,\ldots,0) \quad \Leftrightarrow \quad \Gamma^T \cdot \vec{y} = \vec{0} \tag{4.2}$$

erfüllen. Dabei spielt ein konstanter Streckungsfaktor keine Rolle. Allgemein sind die Komponenten y_i rationale Zahlen (Brüche). Man wird in der Regel die Vektoren so normieren, daß sie nur aus den minimalen ganzzahligen Komponenten bestehen. Die Dimension des Raumes ist m-n+1=$|E|$-$|V|$+1 (Zeilendimension minus Rang von Γ) [28]. Die '*fundamental circuit matrix*' *C* ist aus den (linear unabhängigen) Vektoren $\vec{y}^T$ zusammengesetzt, so daß gilt:

$$\mathbf{C}^T \cdot \Gamma = \underline{\underline{0}} \quad ; \quad C = \left[\vec{y}_1 \quad \cdots \quad \vec{y}_{m-n+1}\right] \tag{4.3}$$

Jeder unabhängigen Schleife (loop) in einem Graph entspricht ein l.u. Vektor $\vec{y}^T$, dessen Komponenten (eine positive rationale Zahl für jede Kante) nur an den Stellen größer als Null sind, die den Kanten dieser Schleife entsprechen. Es existieren eventuell noch weitere Schleifen im Graph, diese enthalten aber keine neuen Kanten, die zugehörigen $\vec{y}^T$-Vektoren sind von den anderen linear abhängig.

Zur genauen Festlegung der l.u. Schleifen betrachte man den dem Graphen zugrundeliegenden *Spannbaum* (dieser enthält keine gerichteten Schleifen und besitzt $|E'|=|V|-1$ Kanten). Jede aus dem Graphen herausgenommene Kante führt bei Einfügung in den Baum zu genau einer neuen l.u. Schleife. Das Verfahren führt genau zu den $|E|-|E'|=m-n+1$ unabhängigen Schleifen. Bei Petri-Netzen werden diese Vektoren „S-Invariant", „P-Invariant" und „P-Flow" genannt (P=Place).

Die **Bestimmung der C-Matrix** kann in der Praxis folgendermaßen durchgeführt werden:

Bestimme den *q*-Vektor (ein O($|V|+|E|$)-Algorithmus ist oben beschrieben).
Finde alle fundamentalen Schleifen oder, falls das nicht möglich ist, alle Schleifen, die mindestens die fundamentalen Schleifen enthalten. Aufwand: O((|*C*|+1)*(|*V*|+|*E*|)) [64]
Berechne den *n*-Vektor durch Besuch aller Kanten:

$$\vec{n} = \Gamma^+ \cdot \vec{q} = \Gamma^- \cdot \vec{q} \tag{4.4}$$

Aufwand: O(|*E*|), wenn der *q*-Vektor bekannt ist. Berechne dann

$$N = \underset{\text{alle } e \in E}{\mathbf{kgV}}\ (n_e) \equiv \mathbf{kgV}(\vec{n}) \tag{4.5}$$

Berechne Vektor $\vec{w}$ gemäß (" $\vec{w} = \frac{N}{\vec{n}}$ " {komponentenweise Division})

$$w_e = \frac{N}{n_e} \qquad \forall\ e \in E \tag{4.6}$$

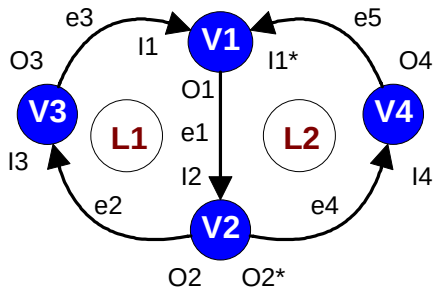
Erzeuge für jede (fundamentale) Schleife eine Zeile in der *C*-Matrix und durchlaufe dann alle Kanten, die zu jeweils einer Schleife gehören. Daraus folgt ein Aufwand von O(|*V*|*|*E*|). Die Spalteneinträge sind dann die passenden Komponenten des oben berechneten *w*-Vektors. Soll die minimale *C*-Matrix bestimmt werden, dann muß jede Zeile durch ihren größten gemeinsamen Teiler geteilt werden und jede linear abhängige Zeile muß entfernt werden. Teile der obigen Rechnung werden auch für die Einheitsverstärkungstransformation benötigt.

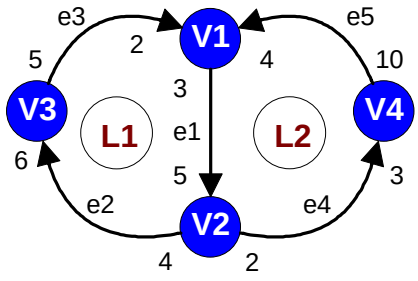
Bei zyklischen Singleratengraphen ergeben sich die Komponenten von $\vec{y}^T$ zu Null oder Eins; eine Eins entspricht einer zur Schleife gehörenden Kante.
Bei Multiratengraphen bestehen die nichtverschwindenden Komponenten aus Produkten von Knotenverstärkungen (rate gains) [29]. Die Knotenverstärkung eines Knotens *v* auf dem Pfad *p* ist definiert als Verhältnis von Ausgangs- zu Eingangsrate:

$$g(v, p) = \frac{o(v, p)}{i(v, p)} \tag{4.7}$$

Bei konsistenten Graphen ist das Produkt der Verstärkungen einer Schleife gleich Eins [38].

Beispiel:





Kante: e_1 e_2 e_3 e_4 e_5 (Schleife)

$$\underline{\underline{\mathbf{C}}} = \left(\begin{array}{c|c|c|c|c} 1 & g_1 g_3 = \frac{1}{g_2} & g_1 = \frac{1}{g_2 g_3} & 0 & 0 \\ \hline 1 & 0 & 0 & g_1^* g_4 = \frac{1}{g_2^*} & g_1^* = \frac{1}{g_2^* g_4} \end{array}\right) \begin{array}{l} (L1) \\ (L2) \end{array} \qquad \underline{\underline{\mathbf{C}}} = \left(\begin{array}{c|c|c|c|c} 1 & \frac{3 \cdot 4}{2 \cdot 5} & \frac{3}{2} & 0 & 0 \\ \hline 1 & 0 & 0 & \frac{3 \cdot 10}{4 \cdot 3} & \frac{3}{4} \end{array}\right)$$

mit $g_1 = \frac{O_1}{I_1}$, $g_1^* = \frac{O_1}{I_1^*}$ usw.

Zu beachten ist noch folgender Umstand:

Auch bei azyklischen Graphen kann die Dimension des Nullraums größer Null sein, d.h. es läßt sich eine ganzzahlige fundamental-circuit Matrix finden, die Zeileneinträge können hier aber nicht gerichteten Schleifen entsprechen!

Die Zeilen-Dimension (=Rang) von *C* ist *m*-*n*+1, also Knotenanzahl minus Kantenanzahl plus Eins. Wenn dieser Wert größer als Null ist, dann müssen mindestens so viele Kanten wie Knoten vorhanden sein. Der Graph kann in diesem Fall kein Baum mehr sein, also müssen bei azyklischen Graphen Kanten oder Pfade in gleicher Richtung parallel verlaufen. Dann existieren aber echte *ungerichtete* Schleifen (mindestens eine Kante gegen die Zählrichtung). Das führt in der *C*-Matrix einer Zeile pro ungerichtete Schleife, die für die entgegenlaufenden Kanten negative Einträge hat.

4.4 Die Zustandsgleichung

Zur optimalen mathematischen Beschreibung des Aktivierungszyklus in einem regulären Datenflußgraphen werden Methoden der Vektoralgebra genutzt, um eine Zustandsgleichung aufzustellen [18]. Diese beschreiben ähnlich den Systemgleichungen der digitalen Regelungstechnik die Zustandsübergänge aufgrund von Eingangswerten.
Der Zustandsvektor $\vec{d}$ repräsentiert die aktuelle Delayverteilung auf den Kanten des Graphen; $\vec{d}_0$ ist der Anfangszustand (die Anfangsbedingung), der zu Beginn jeder Periode auf den Kanten anliegt. Der zur aktuellen Zeit aktivierte Knoten v_i führt zu einem Aktivierungsvektor $\vec{v}(n)$ zum Zeitpunkt *n*, der eine Eins in der Komponente, die dem aktivierten Knoten entspricht, enthält, und Nullen für jeden Block, der nicht aktiviert wurde. Die Zustandsgleichung lautet nun

$$\vec{d}(n+1) = \vec{d}(n) + \Gamma \cdot \vec{v}(n) \tag{4.8}$$

Da der Rang der Matrix bei konsistenten Graphen (in der Praxis gegeben) genau |*V*|-1 beträgt, ist der Zustand nur für Dim($\vec{d}$)=|*E*|=|*V*|-1 (im Einheitsratenfall) vollständig steuerbar. Dieser Fall ist genau bei einem azyklischen Graphen im Form eines Baums gegeben. Jede hinzugefügte Kante erhöht den Rang von Γ nicht, daher ist nicht jeder ganzzahlige Vektor $\vec{d}$ im Zustandsraum erreichbar.

Die Ebenengleichungen $\vec{y}^T\vec{d}$=const schneiden aus dem |*E*|-dimensionalen Zustandsraum jeweils Hyperebenen heraus, deren Schnitthyperebenen die Dimension |*E*|-|*L*|=|*V*| haben, wobei |*L*| die Anzahl der fundamentalen Schleifen angibt (siehe „Erreichbarkeit“).
Die Summe der Aktivierungsvektoren ergibt in einer Periode:

$$\vec{q} = \sum_{n=0}^{p-1} \vec{v}(n) \tag{4.9}$$

Daher folgt, daß nach jeder Periode *p* der Anfangszustand wieder angenommen wird:

$$\vec{d}(n \cdot p) = \vec{d}(0) + n \cdot \Gamma \cdot \vec{q} = \vec{d}(0) \tag{4.10}$$

4.5 Lebendigkeit, Beschränktheit, Konsistenz

Lebendigkeit
Zu jedem Zeitpunkt während eines Scheduls muß garantiert sein, daß die Anzahl der Delays auf jeder Kante größer oder gleich Null ist, da negative Delays aus Gründen der Kausalität nicht sinnvoll sind. Ausführungen von Knoten sind nur dann zulässig, wenn an der Eingangskanten genügend Delays vorhanden sind. Wird durch eine zulässige Aktivierungssequenz ein Zustand erreicht, von dem aus keine weiteren Knoten aktiviert werden dürfen, so liegt eine *Verklemmung* {deadlock} vor. Der Graph ist in diesem Fall nicht lebendig {not (a)live}.

Über die Lebendigkeit gibt es nur für Singleraten-Graphen eine brauchbare Aussage: hinreichende Bedingung ist, daß sich in jeder Schleife mindestens ein Delay befinden muß.

Zur Lebendigkeit bei Multiratengraphen, die nicht so leicht festzustellen ist, wird auf den Abschnitt „Gewichtete Delaysumme“ verwiesen.

Beschränktheit

Es sind auch Fälle denkbar, in denen keine Verklemmung entsteht, jedoch die Delayanzahl an einigen Kanten über alle Grenzen wächst, wenn nur zulässige Aktivierungen durchgeführt werden. Solche Graphen sind *nicht beschränkt* {not bounded}. Eine praktische Realisierung mit einer solchen Aktivierungssequenz ist undenkbar wegen der benötigten unbegrenzt großen Speicher für die betreffenden Kanten.

Berechenbarkeit

Ein Graph, der lebendig und beschränkt ist {live & bounded}, ist *berechenbar* {computable}.

Die Existenz eines Scheduls für einen Prozessor {sequential schedule} ist im Fall der Berechenbarkeit gesichert [17]. Wenn ein Ein-Prozessor-Schedul existiert, dann existiert ebenso ein Multi-Prozessor-Schedul [18] (In jedem Fall kann ein Schedul nur auf einem der Prozessoren ablaufen). Ein Multiprozessor-Schedul kann nicht zu einem Deadlock führen, wenn der Graph berechenbar ist [12].

Konsistenz

Voraussetzung für einen berechenbaren Datenflußgraphen ist die *Ratenkonsistenz*. Ein Graph ist genau dann konsistent, wenn der Rang der Topologiematrix gleich |V|-1 ist [18] (dies ist nur eine notwendige Bedingung für die Berechenbarkeit!). Die Prüfung der Ratenkonsistenz kann zusammen mit der Berechnung der *q*-Vektors geschehen (siehe Abschnitt Topologie-Matrix).

Ein Einheitsratengraph (alle Raten Eins) ist immer konsistent. Er ist dann lebendig, wenn sich in jeder Schleife mindestens ein Delay befindet.

Die Konsistenz eines Multiraten-Datenflußgraphen äußert sich auch darin, daß die Produkte der Verstärkungen jeder gerichteten Schleife genau gleich Eins sind. Ist das Verstärkungsprodukt einer Schleife kleiner als Eins, dann ist der Graph nicht lebendig. Ist es größer als Eins, dann ist diese Schleife unbeschränkt.

4.6 Marked Graphs als Spezialfall von Petri-Netzen

4.6.1 Petri-Netze

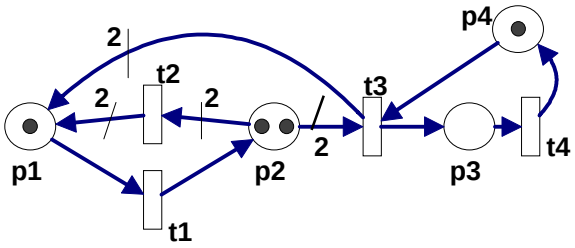
Petri-Netze [37] sind ein Mittel zur Modellierung von konkurrierenden, parallel arbeitenden, nichtdeterministischen und/oder stochastischen Systemen. Als graphische Darstellung enthalten sie eine übersichtliche Struktur in Form eines bipartiten Graphen, d.h. es existieren zwei Typen von Knoten: Transitions und Places; diese sind jeweils durch gerichtete und gewichtete Kanten miteinander verbunden. Das Konzept stammt aus dem Jahr 1962 von Carl Adam Petri.

Ein Petri-Netz ist ein 5-Tupel $PN=(P, T, F, W, M_0)$, wobei gilt:

$P=\{p_1,p_2,...,p_m\}$	eine endliche Menge von 'places'
$T=\{t_1,t_2,...,t_n\}$	eine endliche Menge von 'transitions'

$F \subseteq \{P \times T\} \cup \{T \times P\}$	eine Menge von Kanten zwischen P und T
W: $F \rightarrow \{1,2,3,...\}$	eine Gewichtsfunktion (Eingangs-, Ausgangsraten)
M_0: $P \rightarrow \{0,1,2,3,...\}$	Anfangsdelayverteilung {initial marking}

Bild rechts: Petri-Netz, das kein endlicher Automat und kein Marked Graph ist.

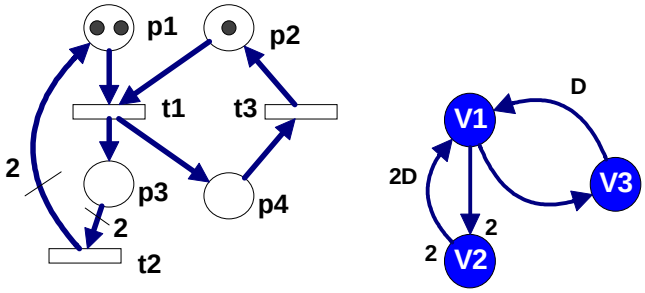


Places werden in üblicher Darstellung als Kreis gezeichnet, der M 'tokens' (Delays) in Form von Punkten enthält. Transitions sind gefüllte oder ungefüllte Rechtecke. Die Kanten zwischen Places und Transitions sind Pfeile, die eventuell mit einer Zahl als Gewicht versehen sind. Diese Gewichte geben an, wieviele Tokens bei Aktivierung einer Transition aus einem Place entfernt werden bzw. in ein Place hineingeschrieben werden. Ein gewöhnliches {ordinary} Petri-Netz hat nur Einsen als Gewichte (entspricht den Singleraten-Datenflußgraphen). Petri-Netze mit von Eins verschiedenen Raten werden auch als 'Weighted P/T-System' [39] oder 'non-ordinary Petri-Net' bezeichnet.
Die Aktivierungsregel für Transitions entspricht genau denen der Datenflußgraphen; eine Transition darf also nur gestartet werden, wenn an allen Eingängen mehr Tokens in den Places sind als das entsprechende Gewicht angibt (Aktivierungsschwelle). Der wichtigste Unterschied zwischen Petri-Netzen und Datenflußgraphen liegt in der Möglichkeit der Petri-Netze, auf einen Datenspeicher (Place) von mehreren Transitions aus zuzugreifen (Abbildung). Die dabei auftretende Konkurrenz ist mit Datenflußgraphen nicht modellierbar. Die Entscheidung, eine Transition von mehreren, die eingangsseitig mit demselben Place verbunden sind, zu starten ist nicht durch das Netzmodell selbst gegeben. Daher sind Netze, bei denen solche Fälle auftreten können, nichtdeterministisch.

4.6.2 Endliche Automaten {Finite State Machines}

Diese Unterklasse der Petri-Netze modelliert exakt die Zustände und Übergänge in einer Form ähnlich einem Zustandsdiagramm. In dieser Klasse können von und zu einem Place (entspricht Zustand) mehrere Kanten laufen, jede Transition (entspricht Zustandsübergang) darf aber nur eine Eingangs- und eine Ausgangskante haben.

4.6.3 Marked Graphs



Marked Graphs entprechen genau den Einheitsraten-Datenflußgraphen. Als Erweiterung entsprechen die Multiple-Weighted Marked Graphs (=Weighted T-Systems) den Multiraten-Datenfluß-graphen (siehe Bild: Petri-Netz und DFG-Äquivalenz). Als Spezialfall der Petri-Netze sind hier nicht mehr als eine Eingangs- und eine Ausgangs-kante zu einem Place erlaubt. Man bezeichnet deshalb die Marked Graphs auch als entscheidungsfrei {decision-free} bzw. deterministisch. Dieser Eigenschaft hat man es zu verdanken, daß viele Probleme der allgemeinen Petri-Netze, wie z.B. Erreichbarkeit,

Lebendigkeit und Begrenztheit, einfacher darstellbar sind und auch bekannte Lösungen haben [39],[40]. Die für Marked Graphs entwickelten Theorien sind direkt für Datenflußgraphen anwendbar und bereichern die dazu vorhandene Literatur. Die für Multiraten-Graphen besonders wichtige 'Weighted Sum of Tokens' und das Konzept der T- und P-Invarianten in [37],[41] ausführlich behandelt.

Die Frage nach der Erreichbarkeit von Zuständen (hinreichende Bedingung) ist für allgemeine Petri-Netze bis heute noch ungeklärt [37]. Ebenso gibt es keine befriedigende Bedingung zur Feststellung von Lebendigkeit [40],[39].
Anzumerken ist zur Nomenklatur, daß ‘tokens’ den einzelnen Delays entsprechen; mit ‘marking’ ist jedoch die gesamte Delayverteilung gemeint.

4.6.4 Weitere Unterklassen von Petri-Netzen

Neben den oben erwähnten ‘State Machines’ (SM) und ‘Marked Graphs’ (MG) gibt es zu Petri-Netzen (PN) die Unterklassen ‘Free-Choice Net’ (FC), ‘Extended Free-Choice Net’ (EFC) und ‘Asymmetric Choice Net’ (AC). Die Menge der Netze FC enthält SM und MG; ihr Merkmal ist, daß jede Kante von einem Place entweder als einzige zu einer Transition führt oder daß jedes Place nur eine Ausgangskante hat, wenn diese zu einer Transition führt, die mehrere Eingangskanten hat. Weitere Erklärungen siehe [37].
Es gilt:

$$SM \neq MG;\ SM \cap MG \neq \emptyset;\ \ SM \cup MG \subset FC \subset EFC \subset AC \subset PN. \qquad (4.11)$$

4.7 Präzedenzbeziehungen, Quellen möglicher Parallelität

Primäres Ziel einer Multiprozessorimplementation ist die Steigerung des Durchsatzes. Dazu ist es nötig, möglichst viele Teile des Programms parallel auszuführen. Die benötigte Parallelität muß schon im Algorithmus bzw. im Datenflußgraph offenliegen, da ein nicht-preemptives[10] Scheduling eingesetzt werden soll.

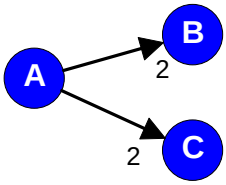
Die Beziehungen, die durch Kanten des Graphen zwischen zwei Knoten ausgedrückt werden, nennt man Präzedenz: Ein Knoten muß vor einem anderen ausgeführt werden. Dadurch können diese Knoten nur sequentiell ablaufen, nicht aber parallel. Der Azyklische Präzedenzgraph (siehe Kapitel Transformationen) beschreibt diese wichtigen Beziehungen für eine begrenzte Anzahl von Perioden. Die Parallelität ergibt sich daraus, daß zwischen manchen Knoten keine Präzedenzbeziehungen {precedence constraints} bestehen.

Parallelität kann sich *räumlich* oder *zeitlich* äußern:

- Räumliche Parallelität: Knoten aus derselben Periode können gleichzeitig ausgeführt werden. Dies ist durch parallele Pfade im Datenflußgraph möglich, bei denen zwischen Knoten keine Präzedenzbeziehungen bestehen. Räumliche Parallelität kann nicht künstlich eingefügt werden, sondern muß schon in der Struktur des Datenflußgraphen verankert sein.
- Zeitliche Parallelität: Knoten aus verschiedenen Iterationen können gleichzeitig ausgeführt werden. Dies ist durch Delays auf Kanten charakterisiert. Für azyklische Graphen entspricht dies genau der Pipelineverarbeitung.. Durch Retiming kann zeitliche Parallelität besser ausgenutzt werden.

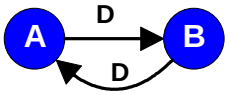
[10] Preemptives Scheduling: Prozesse können vor Beendigung unterbrochen und später fortgesetzt werden

Beispiel für räumliche Parallelität:



P1:	A	B
P2:	A	C

Beispiel für zeitliche Parallelität:

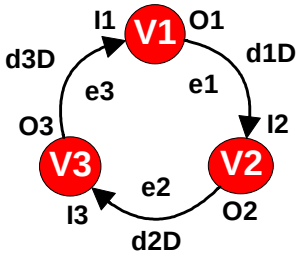


P1:	A1	A2
P2:	B0	B1

Anmerkung: Die Numerierung der Perioden für Knoten *B* könnte auch bei Eins oder Zwei beginnen, wenn die zugehörigen Iterationen anders interpretiert werden.

4.8 Gewichtete Delaysumme

Auf die im nebenstehenden Graph angegebenen Bezeichnungen wird im folgenden eingegangen. Insbesondere sind die Indizes an jedem Knoten und der folgenden Kante identisch.



$$\vec{d} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

Die wichtigste Kenngröße neben der Struktur des Graphen ist die gewichtete Summe der Delays (Tokens). Diese bestimmt unter anderem, ob der Graph lebendig ist und wie groß der maximale Durchsatz sein kann. Zur Erläuterung folgen nun einige Definitionen und Beschreibungen:

4.8.1 „Weighted Sum of Tokens“ (WST)

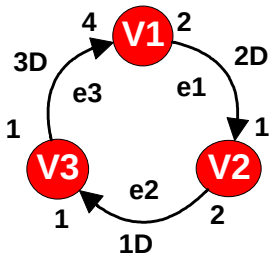
Definition [37]: $WST = \vec{y}^T \cdot \vec{d}$ (4.12)

$\vec{y}^T$ ist der kleinste ganzzahlige Vektor (P-Flow), $\vec{d}$ der Delay-Vektor.
Die gewichtete Delaysumme ist für jede Schleife konstant, sowohl während eines Ausführungszyklus, als auch unter Retiming. Dieser für jede Schleife charakteristische ganzzahlige Wert bestimmt entscheidend die Eigenschaften des DFG, u.a. Lebendigkeit u. Iterationsgrenzwert. Die Gleichung läßt sich auch interpretieren als Ebenengleichung in Hessescher Normalenform (siehe dazu auch Abschnitt Erreichbarkeitsgraph):

$$\vec{y}^T \cdot \vec{d} - WST = 0 \qquad (4.13)$$

Im Zustandsraum, der aus der Delayverteilung gebildet wird (Dimension = Kantenanzahl), stellt $\vec{y}^T$ den Normalenvektor auf der (Hyper-)Ebene dar, auf der sämtliche Zustände einer Schleife liegen. Unter der Zusatzbedingung $d_i>0$ wird die Ebene noch eingeschränkt durch die Grenzebenen $\vec{d} \cdot \vec{\mathbf{e}}_i$ (Wände zwischen Koordinatenachsen)..
Nicht zu jeder ganzzahligen WST existiert eine gültige Delayverteilung, z.B. für $\vec{y}^T$=(3,5,7) ist *WST*=8 unmöglich (Ebene geht nicht durch Punkte mit ganzzahligem Ortsvektor $\vec{d}$).

Für nebenstehenden Graphen ergeben sich folgende Werte:



$$\vec{y}^T = (2,1,1)$$
$$\vec{d}^T = (2,1,3)$$
$$WST = \vec{y}^T\vec{d} = 8$$

Es ist noch anzumerken, daß für Graphen mit mehr als einer Schleife der Wert *WST* für jede (fundamentale) Schleife zu berechnen ist. Das Resultat ist ein Vektor $\overrightarrow{WST} = \underline{\underline{C}} \cdot \vec{d}$. Alle im Folgenden definierten Maße müssen dann genauso vektoriell beschrieben werden.
Wenn zur Gewichtung nicht der minimale Vektor $\vec{y}^T$ benutzt wird, sondern ein ganzzahliges Vielfaches, so ändern sich die Eigenschaften nicht, wenn andere Größen auch mit dem erweiterten Vektor berechnet werden (HDM, CM, NMD).

4.8.2 „Sum of weighted tokens“ (SWT)

In einem Artikel [38] wird statt der WST die SWT verwendet, die beiden Maße unterscheiden sich aber nur durch eine (für jeden Graphen spezielle) Konstante. Der Vorteil der SWT ist, daß man an *SWT*≥1 erkennen kann, daß die Knotenprozesse bequem geschedult werden können (s.u.: CM). Nachteilig ist der unpraktische Gewichtsvektor und die Nichtganzzahligkeit von *SWT*.

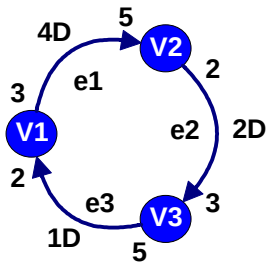
$$SWT = \sum_{i=1}^{K} \frac{d_i}{I_{i\oplus1} \cdot q_{i\oplus1}} = \sum_{i=1}^{K} \frac{d_i}{O_i \cdot q_i} = \sum_{i=1}^{K} \frac{d_i}{n_i} = \vec{d} \cdot \begin{pmatrix} 1/n_1 \\ \vdots \\ 1/n_K \end{pmatrix} = \vec{d} \cdot \frac{\vec{y}}{\mathbf{kgV}(\vec{n})} = \vec{d} \cdot \frac{\vec{y}}{N} = \vec{d} \cdot \frac{\vec{y}}{CM} = \frac{WST}{CM} \tag{4.14}$$

Für der Graphen im vorigen Abschnitt ergibt sich *SWT*=2, da *CM*=4 ist.

4.8.3 „Heaviest Dead Marking“ (HDM) [39-S.363]

Definition: *HDM* ist die größte WST, die zu einem toten Graph gehören kann; d.h. für jede *WST*>*HDM* ist der Graph auf jeden Fall lebendig.

$$HDM = \sum_{e \in Loop} [O(e) - 1] \cdot y_e \qquad (4.15)$$



Der nebenstehende Graph zeigt, daß es für *WST*=*HDM* immer mindestens eine Delayverteilung gibt, für die der Graph nicht lebendig ist. Die Formel legt nahe, dafür auf jede Kante *e* genau *Zielrate*(*e*)-1 Delays zu setzen, damit kein Knoten aktiviert werden kann. Für das Beispiel ist *HDM*=21 und $\vec{y}^T$=(3,2,5).

4.8.4 „Lightest Live Marking“ (LLM)

Für die Lebendigkeit existiert nur folgende allgemeine Aussage [39]:
Wenn die Gleichung $\vec{y}^T \cdot \vec{d} = \vec{y}^T \cdot (\vec{d}_M - \vec{\mu})$ keine Lösung $\vec{\mu}$ bestehend aus natürlichen Zahlen hat, dann ist der Graph mit $\vec{d}$ live.

Die Feststellung der Lebendigkeit ist in der Praxis bisher nur durch einen Klasse-S-Algorithmus [18] oder eine Vereinfachung davon [39] möglich. Dies entspricht der empirischen Bestimmung eines Ein-Prozessor-Scheduls mit damit verbundenem Aufwand O(Σq_v).

Definition: *LLM* ist die kleinste WST, für die ein Graph lebendig sein kann.

Für obiges Beispiel mit *HDM*=21 ist *LLM*=20.

4.8.5 „Comfortable Marking“

Definition: Die komfortable gewichtete Delaysumme CM ist die kleinste WST für die, wenn sich sämtliche Delays einer Schleife vor einem Knoten befinden, dieser und alle folgenden Knoten sich sofort q_v mal gleichzeitig ausführen lassen.

$$CM = Konsumrate(e) \cdot q(ziel(e)) \cdot y_e = n_e \cdot y_e = \mathbf{kgV}(\vec{n}) = N \qquad (4.16)$$

(dabei kann eine beliebige Kante *e* einer Schleife betrachtet werden)

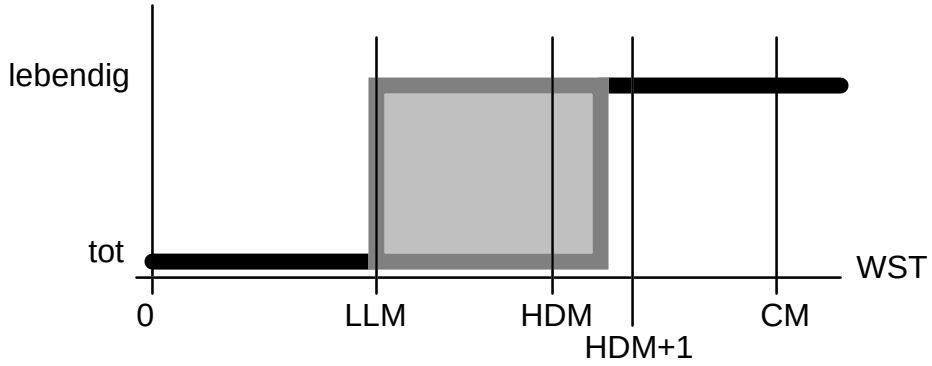
Für *WST*=*CM* können, wenn alle Delays auf der Eingangskante von Knoten v_i konzentriert werden, alle q_i Instanzen des Knotens (also eine vollständige Iteration) gleichzeitig aufgerufen werden (auf q_i Prozessoren), dasselbe ist dann auch mit den folgenden Knoten möglich. Daher wurde diese Delaysumme *bequem* {comfortable} genannt. Für diesen sehr speziellen Fall existiert in der Literatur, weil *SWT*=1 ist, eine Formel für die iteration bound [38]. Der Vektor

$$\vec{n} = \Gamma^+ \cdot \vec{q} = \Gamma^- \cdot \vec{q}\ , \quad \text{„Traffic Vector“} \qquad (4.17)$$

gibt in seinen Komponenten genau an, wieviele Delays auf einer Kante sein müßten, um den nachfolgenden Knoten qi mal zu aktivieren. Dies ist genau die Anzahl der Delays, die während einer Periode "über die Kante fließen".

Es gilt: $1 \le LLM \le HDM+1$; $CM \ge LLM$ (4.18)

Die Lebendigkeit ist nicht monoton über der *WST*, d.h. derselbe Graph kann mit höherer *WST* tot sein und mit niedrigerer *WST* lebendig, je nach der Verteilung der Delays. Für die Zone zwischen *LLM* und *HDM*+1 kann man keine allgemeine Aussage über die Lebendigkeit machen. Das folgende Schaubild soll dir Zusammenhänge verdeutlichen:



Der Bereich zwischen *LLM* und *HDM*+1 ist pathologische Grauzone. Für einige Graphen kann *LLM* mit *HDM*+1 zusammenfallen (bei allen Coprimeraten-Graphen). Auch kann *LLM=CM* sein (bei allen Singleraten-Graphen).

Graphen, in denen für jede Schleife *WST>HDM* gilt, sind immer lebendig; Wenn in einer Schleife eines Graphen *WST<LLM* gilt, ist der gesamte Graph tot.

Für *WST=CM* existieren keine unbeweglichen Delays (NMD's, siehe Kapitel „Retiming") in der Schleife.

Die Darstellung im Bild kann mit den Durchsatz-Graphen im Kapitel „Durchsatz in Multiraten-Systemen" besser interpretiert werden.

4.9 Erreichbarkeit und Steuerbarkeit

Die Erreichbarkeit {reachability} hat große Bedeutung für die Analyse von Datenflußgraphen, insbesondere für die Lebendigkeit, Retiming und Scheduling.

Die Menge der erreichbaren Zustände eines gegebenen Datenflußgraphen wird als $R(\vec{d}_0)$ bezeichnet [37]; dies sind alle Zustände $\vec{d}$, die durch gültige Aktivierungen von Knoten angenommen werden können. Im Fall der lebendigen Graphen gibt es keinen ausgezeichneten Anfangs- oder Endzustand, bei toten Graphen jedoch gibt es genau einen expliziten Anfangs- und einen Endzustand.

Potentiell erreichbare Zustände {state-equation based} sind die, für die ein ganzzahliger 'Parikh-Vektor' $\vec{r}$ ($\vec{\sigma}$) existiert, der die Zustandsgleichung erfüllt:

$$\vec{d} = \vec{d}_0 + \Gamma \cdot \vec{r} \tag{4.19}$$

Die Menge der potentiell erreichbaren Zustände nach der P-Flow-Charakterisierung [39], $PR^B(\vec{d}_0)$, umfaßt genau die Zustände $\vec{d}$, für die die Konstanz der gewichteten Delaysumme gilt:

$$C \cdot \vec{d} = C \cdot \vec{d}_0 \tag{4.20}$$

Daraus folgt, daß *R* eine Teilmenge von *PR* ist: $R \subseteq PR \subseteq PR^B$. Es wurde gezeigt [39], daß *R=PR* für 'Weighted T-Systems', also auch für Multiraten-Graphen, gilt. Die vollständige Gleichheit gilt nur bei Singleratengraphen und coprimen Multiratengraphen.

Die Steuerbarkeit {controllability} ist die Eigenschaft, daß ein beliebiger Zustand durch eine Aktivierungsfolge erreicht werden kann [37]. Dies ist nur möglich, wenn Rang(Γ)=*m* ist, nämlich für Einheitsraten-Graphen in Form eines Baums.

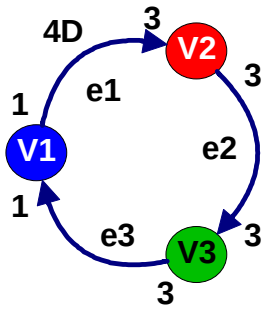
4.9.1 Der Erreichbarkeitsgraph

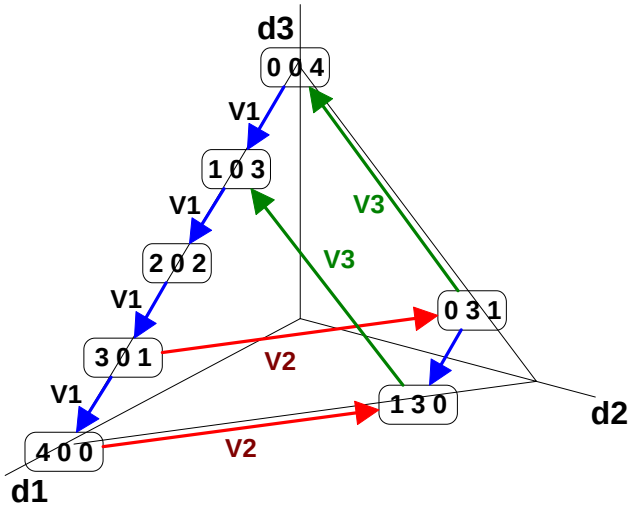
In [37] wurde eine Darstellung für die Menge der erreichbaren Zustände vorgeschlagen, die für Zustandsdiagramme üblicherweise benutzt wird. Diese Darstellung wird für begrenzte {bounded} Graphen aus den allgemeineren ‘coverability graphs’ [37] abgeleitet. Es ergibt sich ein planarer gerichteter Graph, in dessen Knoten die Komponenten des Zustandsvektors geschrieben sind. Die gerichteten Verbindungskanten weisen als Label den Knoten auf, der aktiviert werden muß, um vom alten zum neuen Zustand zu gelangen.

Wenn man sich den Zustandsraum aber als Vektorraum der Dimension |*E*| vorstellt, in dem sich die erlaubten Zustände auf ganz bestimmten Gitterpunkten (mit Ortsvektor $\vec{d}$) befinden, dann erschließt sich dadurch eine periodische Gitterstruktur ähnlich einem Kristall aus chemischer Sicht. Die Länge der Verbindungsvektoren (Richtungsvektoren $\vec{a}$) ergibt sich aus den Raten an dem zugehörigen Knoten. Mit dem zugehörigen Vektor $\vec{v}$ (siehe Abschnitt Zustandsgleichung) und der Inzidenzmatrix Γ ergibt sich für $\vec{a}$:

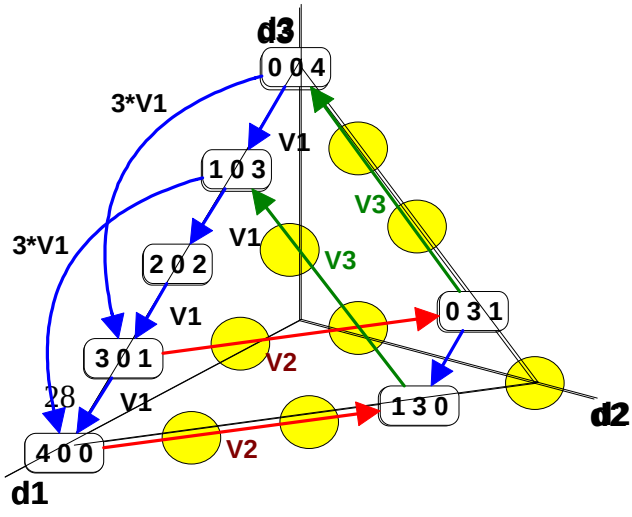
$$\vec{a} = \Gamma \cdot \vec{v} \tag{4.21}$$

Anhand eines Beispielgraphen werden im Folgenden die Zusammenhänge verdeutlicht, da im Kapitel „Retiming“ auf Erreichbarkeitsgraphen zurückgegriffen wird:





Für zwei- oder dreidimensionale Zustandsräume kann man leichter eine Vorstellung von den Zusammenhängen gewinnen, da man die Hyperebenen hier als normale Ebenen direkt vor Augen hat. Am Beispiel des im Bild gezeigten Erreichbarkeitsgraphen soll dies verdeutlicht werden: Der Anfangszustand $\vec{d}=(1,0,3)^T$ liegt in der Ebene $(1,1,1)\cdot\vec{d}=4$ genau wie alle erreichbaren Zustände. Durch Aktivierung von Knoten 1 (v_1), der einzigen Ausführungsmöglichkeit, geht man zum Zustand (2,0,2) über. Der Richtungs-



vektor dazu ist (1,0,-1), was genau der ersten Spalte (dem ersten Knoten zugeordnet) der Inzidenzmatrix entspricht. Da Knoten 1 nur die Rate Eins hat, hat der Richtungsvektor die Länge $\sqrt{2}$ (Norm des Vektors). Für die anderen Knoten ergeben sich längere Vektoren. Da hier im Multiratenfall wegen der höheren Raten die Abstände zwischen benachbarten Zuständen größer sind, ergeben sich in der „Kristall"-Struktur Gitterebenen[11], die einen von Eins verschiedenen Abstand haben. Aus diesem Grund können im Multiratenfall im allgemeinen nicht alle potentiell erreichbaren Zustände PR^B (mittleres Bild) auch wirklich erreichbar sein. Ein periodisches Schedul kann man im Erreichbarkeitsgraph an einer gerichteten Schleife der Kanten ausmachen. Die Aktivierungssequenz (V1, V1, V1, V2, V3) oder (V1, V1, V2, V3, V1) führt zu einem erlaubten Schedul. Es sind, als Erweiterung, auch Richtungsvektoren denkbar, die der parallelen Ausführung mehrerer Knoten gleichzeitig entsprechen. Der Graph wird dadurch zwar unübersichtlicher, aber ein Multiprozessor-Schedul kann durch Ermittlung des kürzesten Pfades im Erreichbarkeitsgraph gefunden werden. Die Länge des kürzesten Pfades ist beschränkt durch $|V|(|V|+1)/2$ [49].

Ein Algorithmus zur Ermittlung des Erreichbarkeitsgraphen geht von einem bekannten Zustand aus und ruft rekursiv in einer Schleife für jeden aktivierbaren Knoten eine Funktion auf, die den durch die Aktivierung neu entstandenen Zustand der Menge R hinzufügt und wiederum weitere Knoten aktiviert, falls der Zustand vorher noch nicht in R war. Der Aufwand zur Berechnung ist allerdings leider nicht polynomial in $|V|$ oder $|E|$, so daß die Berechnung nur für kleine Graphen möglich ist. Die Komplexität ist $O(Z^2)$, wobei Z die Anzahl der Zustände ist. Die Anzahl der Zustände Z ist mit $O(WST^{|E|-1})$ abschätzbar.

[11] Gitterebenen sind parallele Ebenen, auf denen sich die Zentren aller Atomkerne eines Kristalls befinden

5 Transformationen von Datenflußgraphen

5.1 Umwandlung von Multiraten- in Einheitsraten-Graphen

Ein Multiraten-Graph läßt sich immer eindeutig in einen Einheitsraten-Graph umwandeln. Dadurch werden sämtliche im allgemeinen Fall komplizierten Methoden auch für Multiraten-Graphen erschlossen. Die Rückumwandlung vom Einheitsraten-Graph zum Multiraten-Graph ist ebenfalls eindeutig, jedoch nicht eineindeutig, d.h. es gibt zu mehreren äquivalenten Einheitsraten-Graphen denselben zurücktransformierten Multiraten-Graphen. Dieser liefert aber durch die Transformation nur einen speziellen Einheitsraten-Graph. Die Transformation berücksichtigt Struktur (Topologie) des Multiraten-Graphen und seine Delayverteilung! Die Struktur des resultierenden Graphen hängt von diesen beiden Faktoren ab, nicht nur die neue Delayverteilung. Der Einheitsraten-Graph ist dem Ursprungsgraph äquivalent im Sinne des Scheduls und der Erreichbarkeit, und damit auch bzgl. der Lebendigkeit und der Grenze der Iterationsperiode.

Der angegebene Algorithmus zur Hintransformation von E.A.Lee [20] läßt sich in Kürze so beschreiben: Erzeuge für jeden Ursprungsknoten *v q(v)* neue Knoten (Diese entsprechen den *q(v)* Aktivierungen innerhalb einer Periode). Erzeuge dann für jede Ursprungskante *e Quellrate(e)*q(Quellknoten(e))= n*(*e*) neue Kanten[1] und verbinde damit in spezieller Weise die den inzidenten (MR-)Knoten äquivalenten (SR-)Knoten. Die Art der Verbindungen hängt von der Anzahl der Delays ab, die gleichmäßig auf den neuen Kanten verteilt werden; und zwar so, daß die auf den ersten Zielknoten zulaufenden Kanten die ersten Datenwerte bekommen

Umwandlung eines Multiraten-DFG in einen äquivalenten Einheitsraten-DFG

Jeder Knoten, der mehr als einen Ausgangswert erzeugt, wird ersetzt durch einen Knoten, der je ein Sample auf mehreren parallelen Kanten ausgibt.

Jeder Knoten, der mehr als einen Ausgangswert aufnimmt, wird ersetzt durch einen Knoten, der je ein Sample von mehreren parallelen Kanten konsumiert.

Jeder Knoten *K*, der q_i mal aufgerufen wird ($\vec{q}$ ist der kleinste Vektor aus den natürlichen Zahlen, der zum Nullraum der Topologiematrix gehört), wird ersetzt durch q_i Knoten $K_1..K_{qi}$, die jeweils nur einmal in einem Zyklus aufgerufen werden.

Die Reihenfolge der Samples auf einer Kante *U->V* muß eingehalten werden. Daher erzeugt Knoten U_1 die Samples $1..a_U$, U_2 die Samples $(a_U+1)..(2a_U)$, usw. bis Sample q_U*a_U. Insgesamt sind also aus einer Kante des Multiraten-DFG q_U*a_U Kanten geworden.

Sind auf einer ursprüngliche Kante *n* Delays vorhanden, so werden diese auf die resultierenden Kanten so verteilt, daß die Summe der Delays aller resultierenden Kanten konstant und gleich *n* ist.

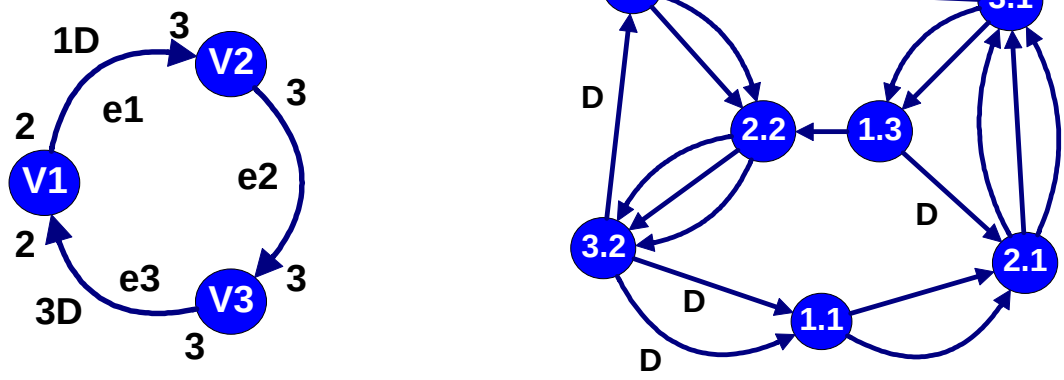
Jedes Delay der Vielfachheit *M* (*M*D) des Multiraten-DFG verursacht eine zirkulare Permutation der *N* parallelen Kanten, d.h. die unteren *M* mod *N* (letzten) Kanten überschneiden die anderen und werden an den Zielknoten V_i die obersten (ersten) Kanten. Die Delays werden an den Zielknoten von oben nach unten jeweils einzeln verteilt. Sind

[1] Zur Berechnung des *n*-Vektors wird auf den Abschnitt „Comfortable Marking“ verwiesen.

mehr Delays als Kanten vorhanden, so verteilt man die restlichen wieder einzeln von oben nach unten.

Aus ‘self-loops’ U->U werden Verkettungen U_1->U_2->...U_{qu}->U_1, die enthaltenen Delays werden wieder jeweils einzeln verteilt auf die entsprechenden Kanten vor den Knoten U_1..U_{qu}, beginnend mit einem Delay ‘vor’ U_1.

Am folgenden Beispiel wurde die Umwandlung exemplarisch durchgeführt:

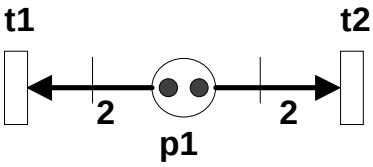


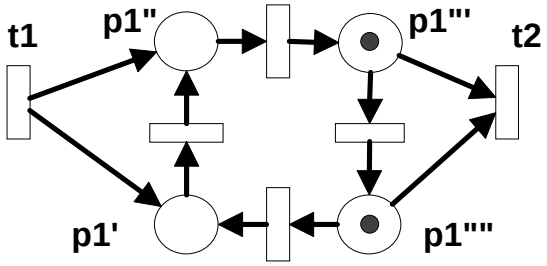
Die gesamte Anzahl aller Delays verändert sich bei der Umwandlung nicht.

Zur Rückwandlung in einen Multiraten-Graphen werden alle Knoten wieder zu einem zusammengefaßt, ebenso alle Kanten. Die Delays auf einer Kante ermitteln sich aus der Summe der Delays auf den äquivalenten vervielfachten Kanten.

Der vollständige äquivalente Einheitsraten-Graph hat insgesamt $n'=\Sigma q(v)$ Knoten und $m'=\Sigma n(e)$ Kanten. Der Aufwand sowohl zur Transformation als auch zur Durchführung von Berechnungen im äquivalenten Einheitsraten-Graph ist damit nicht mehr echt polynomial in der Anzahl der Knoten n und der Anzahl der Kanten m. Der Aufwand sieht zwar mit $O(m'n')$ polynomial aus, muß aber als pseudopolynomial ([63],[70]) bezeichnet werden, da der q-Vektor, genauso wie der n-Vektor nicht notwendigerweise polynomial in n und m sein muß. Für Graphen, deren Raten coprim (und größer Eins) sind, hat schon der q-Vektor Komponenten, die stärker als faktoriell ($n!$) von der Knotenanzahl abhängen. Es ist anzunehmen, daß das Problem dann NP-vollständig ist.

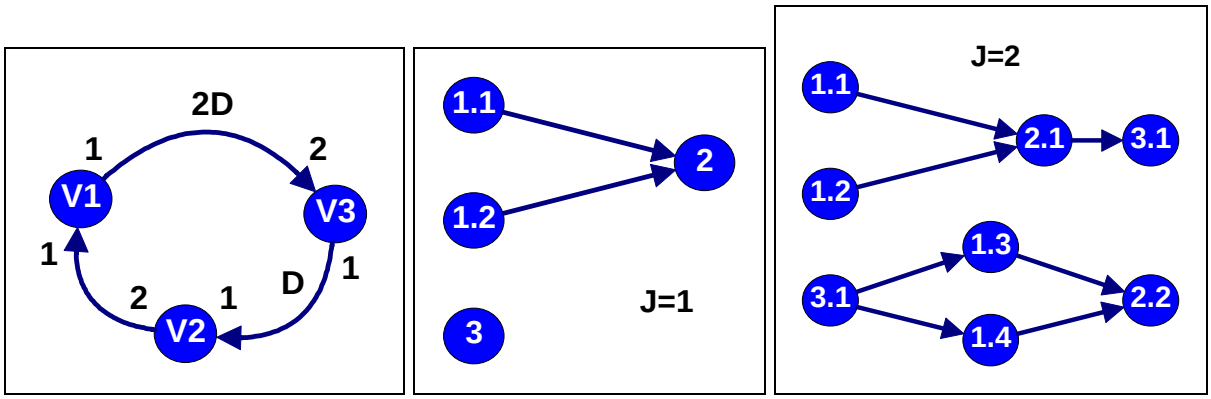
Neben dieser Transformation existiert für ‘non-ordinary’ Petri-Netze eine Umwandlungsvorschrift [46], die aus jedem Place einen Ring von Ersatzplaces macht. Nachteilig ist, daß die Anzahl der Transitions sich erhöht und die neuen Transitions geeignet aktiviert werden müssen (Schedule). Dieses Transformations-Modell ist für unsere Zwecke nicht brauchbar, da es keinen ‘Marked Graph’ darstellt, denn es ist nicht entscheidungsfrei (deterministisch) wegen multipler Kanten, die auf ein Place zu- bzw. davon weglaufen:





5.2 Azyklischer Präzedenz-Graph

Für viele Scheduling-Algorithmen wird der Azyklische Präzedenzgraph {acyclic precedence graph} benötigt. Dieser enthält nur noch die innerhalb einer Iteration nötigen Präzedenzbeziehungen. Die Kanten, die Präzedenzen zwischen verschiedenen Iterationen beschreiben (weil sie Delays enthalten), treten nicht mehr auf. Daher ist der APG ein Singleraten-Graph ohne Zyklen. Die Konstruktion des APG aus Singleraten-Graphen ist besonders einfach: Man entferne einfach alle Kanten, die Delays enthalten.



Ursprungsgraph APG (J=1) APG (J=2)

Der Algorithmus von Lee [18] bestimmt zu einem beliebigen Multiraten-Graphen den APG, indem ein Scheduling durchgeführt wird und währenddessen die Abhängigkeiten zwischen den Knoten festgestellt werden. Aus einem Multiraten-Knoten werden hier, wie beim äquivalenten Singleraten-Graph, $q(Knoten)$ neue Knoten. Die Komplexität des Algorithmus ist aber nicht ganz so hoch, wie die Umwandlung zum Singleraten-Graphen mit nachträglicher Entfernung der Kanten mit Delays. Daher sollte für einen Scheduler, wenn überhaupt, nur der APG-Algorithmus benutzt werden.

In diesem Algorithmus kann ein Blocking-Faktor J berücksichtigt werden, indem der q-Vektor mit diesem J multipliziert wird. Ein solcher Blocking-Faktor kann die zeitliche Parallelität besser ausnutzen helfen, siehe Abschnitt „Unfolding“.

Algorithmus zur Bestimmung des APG:
Anfangswerte: $b(e)=d_0(e)$; $a(k)=0$
Solange Knoten $k \in V$ ausführbar sind (genügend Eingangsdelays und $a(k)<q(k)$)
    Für jeden Knoten $k \in V$
        Wenn k ausführbar ist
            Erzeuge die Instanz $a(k)+1$ des Knotens

Für jede Eingangskante e von k
Es sei n der Vorgängerknoten von k
Berechne $d = \left\lceil \frac{-j \cdot \Gamma_{ek} - b_e}{\Gamma_{en}} \right\rceil$
Wenn $d<0$, dann $d=0$
Erstelle Verbindungskanten mit den ersten d Instanzen von n.
Aktualisiere Delayvektor b entsprechend dem aktivierten Knoten k.
Aktualisiere den Aktivierungsvektor $a(k)=a(k)+1$
Wenn $a(k)=q(k)$, dann war Prozedur erfolgreich, sonst ist Graph dead.

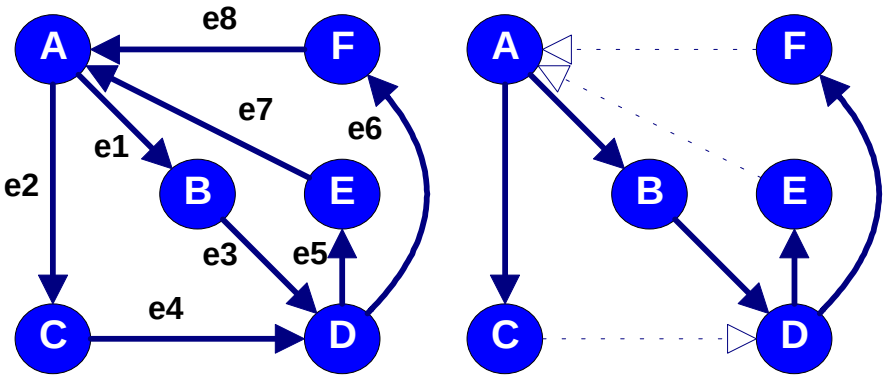
5.3 Spannender Baum

Ein spannender Baum ist für Multiraten-Graphen gleich wie bei Singleraten-Graphen (nämlich nur eine Graphenstruktur ohne Rateninformation und Delays). Ein Spannbaum mit $m=|V|$ Knoten hat genau $n=|E|=m-1$ Kanten. Jede dann hinzugefügte Kante schließt einen Kreis, der aber nicht gerichtet sein muß. Zu jedem Kreis gibt es einen Vektor $\vec{y}^T$ bzw. eine Zeile in der Circuit-Matrix (für nicht gerichtete Kreise enthalten diese auch negative Werte). Mit dem Spannbaum läßt sich die C-Matrix komplett bestimmen, indem jede der entfernten Kanten nach und nach wieder hinzugefügt wird und einer der neu entstandenen Kreise für eine weitere Zeile in der Matrix benutzt wird. Als nächstes sollten die negativen Werte durch Zeilenoperationen, die den Rang nicht verändern, (Addition, Skalierung und Vertauschung von Zeilen) eliminiert werden.

Falls es gelingt, mit jedem Hinzufügen einer Kante neue *gerichtete* Kreise zu erzeugen, dann sind die zugehörigen Zeilen schon nicht-negativ.

Eine Kante, die nur zu einer fundamentalen Schleife gehört, wird Sehne {chord, link arc} genannt [28]. Wenn jede fundamentale Schleife eine Sehne besitzt, kann die C-Matrix durch Spaltenvertauschungen so in eine Einheitsmatrix I und eine Restmatrix C'' (Spannbaum) zerlegt werden:

$$C' = \left[I \vdots C'' \right]$$

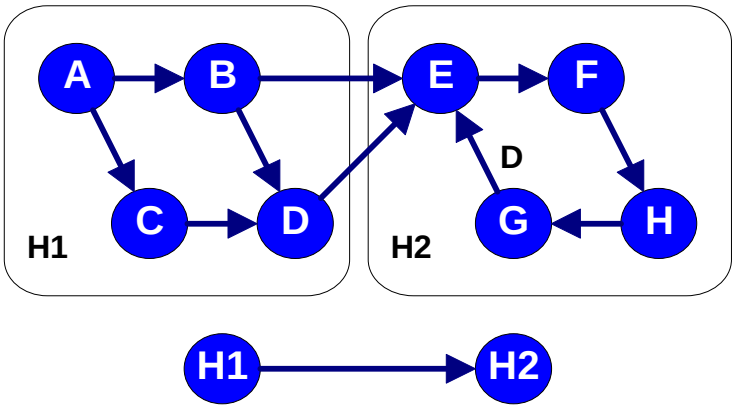


Die zugehörige C-Matrix ist $C = \left(\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & \mathbf{1} & 0 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 & 0 & \mathbf{1} \\ \hline 0 & 1 & 0 & \mathbf{1} & 1 & 0 & 1 & 0 \end{array}\right)$.

5.4 Hierarchisierung

Hierarchisierung ist die Zusammenfassung von mehreren zusammenhängenden Knoten zu einem neuen „Gesamtknoten“, der dann den Teilgraph als Information oder Referenz enthält. Dadurch werden große Graphen übersichtlicher und können schneller geschedult werden. So lohnt es sich eventuell, mehrere Knoten mit sehr kurzer Ausführungszeit im Vergleich zu anderen zusammenzufassen. Bei Bedarf kann der hierarchische Knoten wieder entpackt werden (Auflösung der Hierarchie). Beispiele für Anwendungen befinden sich in [12].

Im folgenden Bild ist Hierarchisierung durchgeführt worden:



5.5 Strongly Connecting

Die „Strongly Connecting Transformation“ ist die Umwandlung eines insgesamt azyklischen Graphen in einen vollständig zyklischen. Die Anzahl der ‘maximal stark zusammenhängenden Komponenten’ {strongly connected components} gibt an, wieviele Einzelknoten oder zyklische Teilgraphen in einem Graphen vorhanden sind. Die Transformation verbindet einige der Knoten so mit einem oder zwei neuen Knoten, daß der resultierende Graph nur noch eine zusammenhängende Komponente hat. Im Multiratenfall müssen die Raten auf den neuen Verbindungskanten so angepaßt werden, daß der Graph ratenkonsistent bleibt (live & bounded). Auf den Verbindungskanten (in der Rückkopplungsschleife) können nun so viele Delays eingeführt werden wie nötig. Die Anwendung dieser Transformation liegt in der Durchführung von Retiming für azyklische Graphen, weil Retiming nur für zyklische Graphen berechenbar ist.

Die erweiterte Vorgehensweise entspricht der für Singleraten-Graphen [12], bis auf die Ratenanpassung und der Option, auch teilweise zyklische Graphen vollständig zyklisch zu machen. Letzeres Problem äußert sich darin, daß u.U. kein expliziter Eingangs- oder Ausgangsknoten vorhanden ist, sondern eine ganze Schleife. Der erste Schritt muß also darin bestehen, einen echt azyklischen Graphen zu erstellen, in dem Anfangs- und Endknoten gefunden werden können. Dazu werden ganz zusammenhängende Komponenten zu einem hierarchischen Knoten sublimiert, der irgendeinen enthaltenen Knoten als Referenz speichert und sämtliche Verbindungen nach außen weiterhin aufweist. Einzelne azyklisch verbundene Knoten bleiben unverändert.

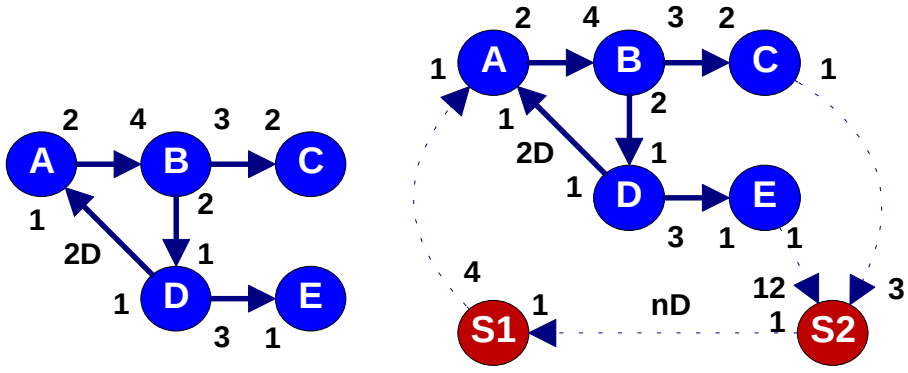
Nun wird ein Verbindungsknoten eingefügt, der so mit den Ausgängen und Eingängen verbunden ist (Raten), daß er nur einmal geschedult werden würde, also der zugehörige Eintrag im *q*-Vektor Eins ist. Da die Komponenten des *q*-Vektors für die Eingangs- und Ausgangs-

knoten bekannt sind, bekommen die Verbindungskanten die Raten so, daß sie einen konsistenten Graph bilden. Der folgende Algorithmus benutzt zwei Verbindungsknoten und eine neue Kante dazwischen, die in einem Zyklus genau einmal durchflossen wird. Ein Delay, das neu auf diese Kante gebracht wird, sorgt nach erfolgtem Retiming für eine komplette zeitliche Trennung der Blöcke vor und nach den Delays, führt also zu echtem Pipelining im azyklischen Ursprungsgraph. Dieses einzelne Delay führt zu (einem Zuwachs von) *WST=CM* in allen Schleifen, die durch die Verbindungskante laufen, und deshalb ist jeder Knoten *v*, vor dem alle Delays angesammelt sind, sofort q_v mal schedulbar.

Algorithmus „Strongly Connecting“:

a) Bestimme den *q*-Vektor des Ursprungsgraphen.
b) Ermittle alle „Strongly Connected Components“ (*SCC*) und die Nummern der SCC’s, zu denen jeder Knoten gehört.
c) Für jede *SCC* erzeuge in einem neuen Singleratengraphen (*SG*) einen hierarchischen Knoten (*HK*), der eine Referenz zu *einem* der enthaltenen Knoten erhält.
d) Für jede Kante des Multiraten-Graphen verbinde die zugehörigen Knoten *HK* in *SG*.
e) Für jeden Knoten *HK* des Graphen *SG* teste, ob Eingangs- oder Ausgangskanten vorhanden sind. Füge die festgestellten Eingangs- und Ausgangsknoten den Mengen *EK* bzw. *AK* zu.
f) Erzeuge zwei neue Knoten (Ausführungszeit Null) *RK1* und *RK2*, verbinde *RK1* mit *RK2* durch eine Kante VK mit Raten Eins.
g) Für jeden Knoten *HK* in *EK* (Eingangsknoten) verbinde *RK2* mit *HK* durch eine Kante, die als Quellrate *q*(*HK*) und als Zielrate Eins erhält.
h) Für jeden Knoten *HK* in *AK* (Eingangsknoten) verbinde *HK* mit *RK1* durch eine Kante, die als Quellrate Eins und als Zielrate *q*(*HK*) erhält.
i) Füge nun die gewünschte Delayanzahl auf die Verbindungskante *VK*.

Im folgenden Bild ist an einem Beispiel „Strongly Connecting“ durchgeführt worden:



5.6 Unfolding

Für Scheduler, die Präzedenzbeziehungen des APG (Azyklischer Präzedenz-Graph) nutzen, um ein nicht-überlappendes Schedul zu erzeugen, ist es möglich, daß kein raten-optimales Schedul erzeugt wird, d.h. daß die Iterationsperiode größer ist als die Iterationsperiodengrenze. Der Grund dafür ist, daß zwar alle Präzedenzen innerhalb einer Iteration ausgenutzt

werden (Ausnutzung räumlicher Parallelität), jedoch Freiheiten innerhalb verschiedener Iterationen nicht genutzt werden (zeitliche Parallelität). Die Unfolding-Transformation [19] legt diese Möglichkeiten bei zyklischen Singleraten-Graphen frei, indem mehrfache Delays in einer Schleife in jeweils ein Delay in vervielfachten Schleifen umgewandelt werden. Die Gesamtanzahl der Delays ändert sich bei der Transformation nicht. Der erhaltene Graph ist weiterhin zyklisch, aber außerdem ratenoptimal, was bedeutet, daß die Iterationsperioden-grenze erreicht wird. *Ratenoptimale* Graphen haben in jeder Schleife genau ein Delay.

Unfolding ist eine Transformation, die aus einem Einheitsraten-Graphen *G*, in dem jeder Knoten pro Iteration genau einmal ausgeführt wird, einen neuen Einheitsraten-Graphen G_U erzeugt, in dem jeder Ursprungsknoten und jede Ursprungskante *J* mal vervielfacht wurde. Die neu erhaltenen Knoten entsprechen jeweils einer aus *J* Iterationen, z.B. führt Knoten U_1 die Aktionen des Ursprungs-Knotens *U* für die Iterationen 1, (*J*+1), (2*J*+1) entsprechend im ersten, zweiten und dritten Zyklus aus. Allgemein führt Knoten U_j die Aufgabe des Knotens *U* für die Iteration (*nJ*-2*J*+*j*) im Zyklus *n* aus.

Die Unfolding-Transformation erhält die Präzedenzen innerhalb und zwischen Iterationen, d.h. jedes für G_U gefundene Schedul ist auch für *G* gültig. Ein Zyklus des Scheduls von G_U entspricht durch die 'Entfaltung' genau *J* Zyklen des Scheduls für den Graph *G*.

Der optimale Unfolding-Faktor J_{opt} eines Einheitsraten-Datenflußgraphen ist das kleinste gemeinsame Vielfache der Anzahl der Delays in einer gerichteten Schleife.
Führe folgende Schritte gemäß dem Algorithmus von Parhi [19] aus:

1) Zeichne für jeden Knoten *U* des originalen DFG im entfalteten DFG *J* entsprechende Knoten und benenne sie U_1, U_2, ... , U_J.
2) Zeichne für jede Kante *U*->*V*, die im Original-DFG kein Delay enthält, *J* Kanten U_k->V_K (*k*=1..*J*) ohne Delay.
3) Für jede Kante *U*->*V* im Original-DFG mit Delays führe einen der folgenden beiden Schritte aus:
   3a) Wenn *i*<*J* (Delay-Anzahl kleiner als Unfolding-Faktor), dann:
   Zeichne Kanten U_{q-i}->V_q (*q*=*i*+1..*J*) ohne Delays.
   Zeichne Kanten $U_{J-(i-q)}$->V_q (*q*=1..*i*) mit einem einzelnen Delay auf jeder Kante (weil U_{J-i+q} vor V_{J+q} Präzedenz hat und V_{J+q} als V_J in nächsten Zyklus ausgeführt wird).
   3b) Wenn *i*>=*J* (Delay-Anzahl größer gleich Unfolding-Faktor), dann:
   Zeichne Kante $U_{\left\lceil \frac{i-q+1}{J} \right\rceil \bullet J-i+q}$ ->V_q mit $\left\lceil \frac{i-q+1}{J} \right\rceil$ Delays (*q*=1..*J*). (Die Ausführung von Knoten V_q erfordert das Ergebnis von Knoten U_{q-i}, der von $U_{\left\lceil \frac{i-q+1}{J} \right\rceil \bullet J-i+q}$ vor $\left\lceil \frac{i-q+1}{J} \right\rceil$ Zyklen aufgerufen wurde.)

(Die oben eckigen Klammern entsprechen der 'ceiling function')

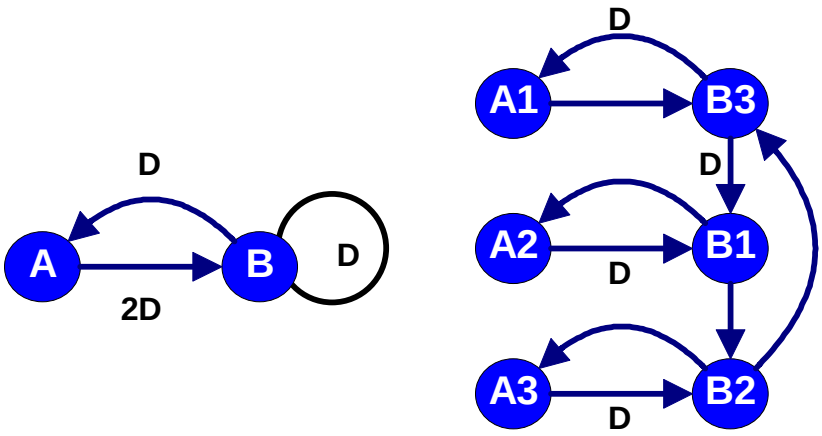
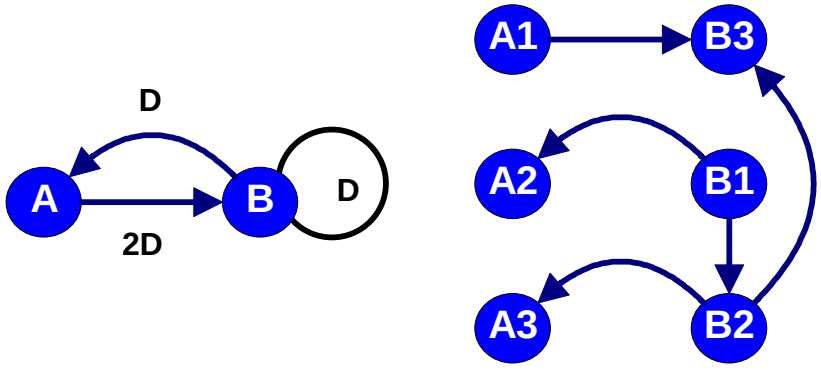


Bild oben: Unfolding mit $J=J_{opt}=3$. Links Original, rechts Unfolded.

5.7 Blocking

Ähnlich wie die Konstruktion des APG erzeugt „Direct Blocking“ [17] einen azyklischen Graphen, der nur noch Präzedenzen beschreibt (Die Knoten eines Multiraten-Graphen werden wie bei der Umwandlung zum äquivalenten Singleratengraphen vervielfältigt). Der Graph umfaßt aber mehrere Perioden, indem Knoten und Kanten vervielfacht werden. Der Blocking-Faktor J gibt dabei die Anzahl der betrachteten Perioden an. Der resultierende Graph kann nicht mehr überlappt geschedult werden, da Informationen über Präzedenzen zwischen verschiedenen Iterationen nicht mehr enthalten sind. Daher kann das Schedul nicht immer optimal sein (also die Iterationsperiodengrenze erreichen). Man kann Blocking interpretieren als Unfolding mit nachfolgender Bildung des APG. Bild: Blocking mit $J=3$.



5.8 Retiming

Zur Verbesserung eines Scheduls durch bessere Ausnutzung der zeitlichen Parallelität kann Retiming benutzt werden. Auch zum Zweck der Minimierung der Speicherelemente oder einer günstigen Clusterbildung[2] ist Retiming das zu nutzende Verfahren.

[2] der Graph wird in Cluster (Segmente) aufgeteilt, wobei jeweils Gruppen von Knoten einen Cluster bilden. Diese werden dann zum Beispiel je auf einem Prozessor (static node assignment) ausgeführt.

Retiming ist eine gültige Neuverteilung der Delays, so daß der neue Delay-Zustand vom alten aus erreichbar ist. Man kann dann immer durch eine Sequenz von Knotenaktivierungen diesen neuen Zustand erreichen. Die Anzahl der dazu nötigen Aktivierungen pro Knoten werden im Retimingvektor $\vec{r}$ angegeben.

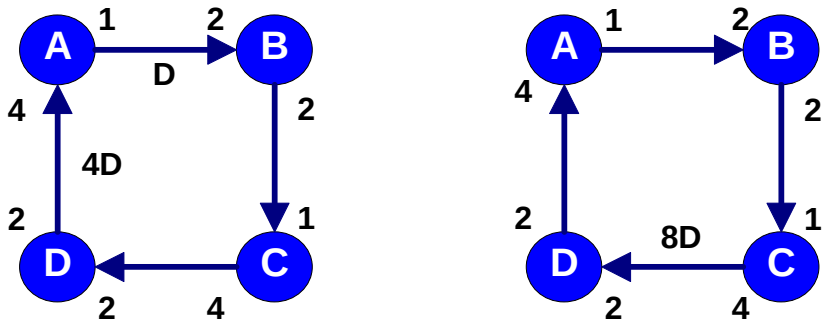
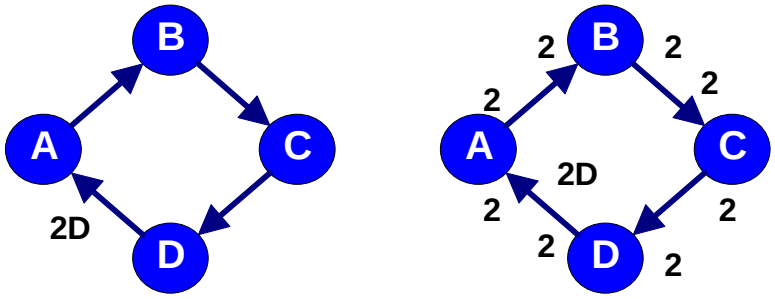


Bild oben: Links: Ursprungsgraph; Rechts: Durch Retiming transformierter Graph.
Im eigenständigen Kapitel „Retiming“ werden sämtliche Aspekte des Retimings behandelt.

5.9 Vektorisierung

Vektorisierung {vectorization} ist eine Transformation, bei der die Raten eines, mehrerer oder aller Knoten eines Graphen um einen Vektorisierungsfaktor erhöht werden, um einen Geschwindigkeitsvorteil durch Parallelverarbeitung auszunutzen [28]. Auf SIMD-Architekturen sind hohe Gewinne erzielbar, aber selbst auf Ein-Prozessor-Systemen kann wegen des verringerten Overheads zum Prozeßwechsel {context switch} eine Verbesserung erreicht werden. Eine Vektorisierung ordnet jedem Knoten v einen Vektorisierungsfaktor $\gamma(v)$ zu, mit dem seine ursprünglichen Raten multipliziert werden. Die Ausführungszeit $T'(v)$ für die vektorisierten Knoten sollte bekannt sein und kleiner als $\gamma(v)*T(v)$ sein, um eine Verbesserung zu bewirken.

Eine *lineare* Vektorisierung bedeutet gleiche Vektorisierungsfaktoren γ für alle Knoten.



Durch eine Vektorisierung kann die Lebendigkeit eines Graphen beeinträchtigen, da die Delays nicht verändert werden, die Größen *LLM*, *HDM*, *CM* aber proportional dem linearen Vektorisierungsfaktor γ anwachsen. Daher wird eine Vektorisierung *legal* genannt, wenn der resultierende Graph berechenbar ist. Für die lineare Vektorisierung im Einheitsratenfall gibt es eine notwendige und hinreichende Bedingung für die Legalität sowie eine obere Grenze für den Vektorisierungsfaktor [28].

Für azyklische Graphen stellt die Vektorisierung auch im Multiratenfall kein Problem dar und kann ohne Komplikationen durchgeführt werden, weil die Lebendigkeit garantiert ist. Für zyklische Graphen ist es nicht garantiert, daß die Vektorisierungsgrenze erreicht werden kann.

Die beste Strategie ist es, zunächst alle Delays einer Schleife durch Retiming auf einer Kante zu sammeln, möglichst auf einer Sehne[3] {chord, link arc}. Dadurch sind dort genügend Delays zur Aktivierung des nachfolgenden Knotens vorhanden, der ja nun eine höhere Aktivierungsschwelle hat als zuvor. Diese Strategie ist auch für Multiraten-Graphen sinnvoll, jedoch ist kein geschlossener Ausdruck für die Legalität oder eine obere Vektorisierungsgrenze bekannt.

5.10 Einheitsverstärkungs-Transformation (Normierung)

{unit gain transformation}

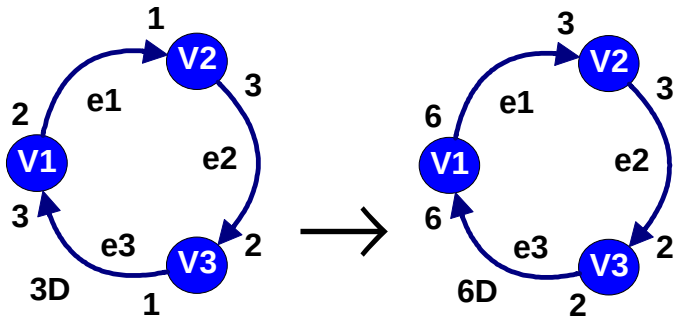


Abbildung: G Abbildung: G'

Diese Transformation $G \rightarrow G'$ verändert die Raten und Delays eines DFG, ohne die Eigenschaften wie Lebendigkeit, gültige Scheduls und andere (s.u.) zu beeinflussen. Der Sinn liegt darin, daß die Raten eines Knotens gleich werden ($I'_v = O'_v$) und somit die Knotenverstärkung $g_v = 1$ wird.

Algorithmus:

Bestimme zum gegebenen DFG G die Vektoren $\vec{q}$ und $\vec{n}$ (siehe auch *CM*)

$$\vec{n} = \Gamma^{+} \cdot \vec{q} = \Gamma^{-} \cdot \vec{q}$$

Berechne

$$N = \underset{\text{alle } e \in E}{\mathbf{kgV}} (n_e) \equiv \mathbf{kgV}(\vec{n})$$

Berechne Vektor $\vec{w}$ gemäß (" $\vec{w} = \frac{N}{\vec{n}}$ " {komponentenweise Division})

$$w_e = \frac{N}{n_e} \qquad \forall\ e \in E$$

Multipliziere dann die Raten und Delays an einer Kante e mit w_e:

[3] damit ist eine Kante gemeint, die ausschließlich zu *einer* fundamentalen Schleife gehört.

Gleichung:

$$\gamma_{ij}' = \gamma_{ij} \cdot w_i \quad \Leftrightarrow \quad \Gamma' = \begin{pmatrix} w_1 & & 0 \\ & \ddots & \\ 0 & & w_K \end{pmatrix} \cdot \Gamma = \underline{\underline{W}} \cdot \Gamma$$

$$d_i' = d_i \cdot w_i \quad \Leftrightarrow \quad \vec{d}' = \underline{\underline{W}} \cdot \vec{d}$$

Eigenschaften:

- G is live $\Leftrightarrow$ G' ist live
- $\vec{q}'=\vec{q}$
- Die Transformation ist äquivalent bezüglich des Scheduls.
- $C \rightarrow C'$ mit $c_{li}>0 \Leftrightarrow c'_{li}=1$, d.h. $c'_{li} \in \{0;1\}$
- $WST'=WST=\sum d'_i$ gilt für jede Menge $M \subseteq E$ von Kanten, also auch speziell für Pfade und Schleifen (Beweis später).
- Die Anzahl der erreichbaren Zustände (=Anzahl Knoten im Erreichbarkeitsgraph) bleibt gleich.
- Position und Gewicht von NMD's bleiben gleich.
- Zu einem normierten Graphen mit ganzzahliger Delayanzahl auf den Kanten existiert nicht immer ein sinnvoller 'zurücktransformierter' Graph. Bei Vorhandensein von NMD's können durch die Rücktransformation nicht-ganzzahlige Delays entstehen (siehe Retiming).
- Die Knotenverstärkung ist $g=O/I=1$.

Der berechnete Vektor $\vec{w}$ eignet sich zur Berechnung von allen Loop-Vektoren $\vec{y}^T$ des Ursprungsgraphen G. Diese werden aus $\vec{w}$ durch Ausblenden (Maskieren) von Komponenten, die nicht zur zu $\vec{y}^T$ gehörenden Schleife passen, gewonnen.

$$y_e = \begin{cases} w_e & e \in Loop \\ 0 \text{ für sonst} \end{cases} \qquad y_e' = \begin{cases} 1 & e \in Loop \\ 0 \text{ für sonst} \end{cases}$$

Der so erhaltene $\vec{y}^T$-Vektor kann, falls mehr als eine Schleife vorhanden ist, ein ganzzahliges Vielfaches (Streckung) des minimalen ganzzahligen $\vec{y}^T$-Vektors für eine Schleife sein. Dies stellt aber kein Problem dar, da er leicht zurückgestaucht werden kann und außerdem die Konstanz der gewichteten Delaysumme auch mit diesem $\vec{y}^T$ gilt.

5.10.1 Normschleife

Def.: Eine Schleife, bei der an jedem Knoten gilt: $g=O/I=1$, heißt Normschleife.

Die Eigenschaft $WST'=\sum d'_i=D_{sum}$ erleichtert einige Untersuchungen, insbesondere kann die notwendige Forderung $\sum d'_i$=const durch Hinsehen überprüft werden.

5.10.2 Normgraph

Def.: Ein Graph, bei der an jedem Knoten und auf jedem Pfad gilt: $g=O/I=1$, heißt Normgraph.

Die Vorteile entsprechen denen der Normschleife, außer daß nun die Delaysumme für jede enthaltene Schleife konstant sein muß.

5.10.3 Invarianz der WST

Es folgt noch der Beweis für die Invarianz der WST:

Satz: Die gewichtete Delaysumme auf einer Kantenmenge K in einer Schleife wird durch die Einheitsverstärkungstransformation nicht verändert.

Beweis:

$$WST=\sum_{e\in K} y_e d_e=\sum_{e\in K} y_e\left(w_e^{-1}w_e\right)d_e=\sum_{e\in K}(y_e w_e^{-1})(w_e d_e)=\sum_{e\in K}1\cdot d'_e=WST'$$

Wenn man die Gewichtung mit w statt mit y vornimmt, dann gilt die Invarianz auch für alle Graphen mit mehr als einer Schleife.

5.10.4 Eigenschaften von coprimen Normschleifen

Def.: Eine Normschleife, deren Knotenraten paarweise prim sind (ggT(I_i , I_k)=1 $\forall$ $i\neq k$), ist eine coprime Normschleife.

Eigenschaften:

- Es sind keine unbeweglichen Delays (NMD's) möglich.
- Alle Delays sind auf eine Kante bewegbar (durch Retiming).
- $LLM=HDM+1$, d.h. die notwendige und hinreichende Bedingung für die Lebendigkeit ist berechenbar.
- Die Bedingung $\underline{\underline{C}}\cdot\Delta\vec{d}=\vec{0},\quad \Delta\vec{d}=\vec{d}_R-\vec{d}_0,\quad \vec{d}_R\in\aleph^{|E|}$ ist notwendig und hinreichend für legales Retiming. Sie reduziert sich hier auf die Forderung, daß die (einfache) Summe der Delays konstant bleiben muß.
- Alle ganzzahligen Zustände $\vec{d}$ mit $\sum d_i=WST$ werden erreicht.
- $\vec{y}^T=(1,\cdots,1)$
- Es gibt für $\vec{y}^T=(1,\cdots,1)$ auf der Fläche $\vec{y}^T\cdot\vec{d}-WST=0$, $d_i>0$ genau

$$Z=\sum_{i=0}^{WST}\binom{N-2}{i}=\binom{N-1+WST}{WST}$$

ganzzahlige Zustände, wobei N die Dimension des Zustandsraumes (= Anzahl der Kanten) ist. Die Summe der Binomialkoeffizienten läßt sich, wie dargestellt, äquivalent in einem Binomialkoeffizient ausdrücken [60].

6 Scheduling von Datenflußprogrammen

Dieses Kapitel behandelt Eigenschaften bekannter Schedulverfahren im Hinblick auf Anwendung für Multiraten-Programme und unter Berücksichtigung der algorithmischen Komplexität.

Zu einem Scheduling gehören die Teilaufgaben Partitionierung der Knotenmenge in Teilmengen, die jeweils einem Prozessor zugeordnet sind {node assignment}, Bestimmung einer Aktivierungsliste {node ordering} und Festlegung der Startzeit jedes Knotens {timing} für jeden Prozessor. Zu einer vollständigen Implementation eines Programms muß dann noch für eine Inter-Prozessor-Kommunikation gesorgt werden, die den Datenaustausch zwischen den Prozessoren durchführt, und es muß Speicher für die übergebenen Daten auf den Kanten vorgesehen werden („Delay Buffer“). Für die Synchronisation der Startzeiten muß ebenfalls gesorgt werden, z.B. durch Verwendung von Semaphoren oder einem exakten Timing der Instruktionsströme.

Im folgenden werden die grundlegenden Begriffe erklärt. Es wird eine Klassifizierung anhand des Schedul-Zeitpunkts vorgestellt, und es wird die Verwandtschaft zu anderen Problemen aufgezeigt. Die Diskussion des Aufwands und der Realisierbarkeit des Schedulings soll das wichtigste Problem beim Scheduling verdeutlichen.

6.1 Mathematische Hilfsmittel

Die hier angesprochenen Verfahren und Theorien habe große Bedeutung für das Scheduling, aber auch für optimierendes Retiming

6.1.1 Lineare Programmierung

Def.: Ein Problem der linearen Programmierung ([60],[61],[62],[63],[68]) ist ein Optimierungsproblem, bei dem

1) eine lineare Funktion in den Entscheidungsvariablen maximiert oder minimiert werden soll. Diese Funktion wird Zielfunktion {objective function} genannt.
2) Die Werte der Entscheidungsvariablen müssen eine Menge von Nebenbedingungen {constraints} erfüllen. Jede Bedingung muß eine lineare Gleichung oder lineare Ungleichung sein.
3) Jede Variable ist einer Vorzeichenbeschränkung {sign restriction} unterworfen. Für jede Variable x_i gibt diese an, daß x_i nicht-negativ ($x_i \geq 0$) oder unbeschränkt ist {unrestricted in sign (urs)}.

Das Problem LP kann in mathematischer Weise so formuliert werden:

Die Entscheidungsvariablen x_i werden in einem Vektor $\vec{x}$ zusammengefaßt.

Maximiere $\vec{c} \cdot \vec{x}$ (Zielfunktion)

unter den Nebenbedingungen {subject to (s.t.)}

$$A \cdot \vec{x} \leq \vec{b} \tag{6.1}$$

Dabei enthält die Matrix A die linearen Koeffizienten und der Vektor $\vec{b}$ die rechte Seite.

Algorithmen zur Lösung verlangen, daß das Problem in Standardform vorliegt:

(6.2)

$$A' \cdot \vec{x}' = \vec{b}$$

In der Standardform sind alle Bedingungen Gleichungen und alle Variablen vorzeichenbeschränkt ($x'_i \geq 0$).

Die Konvertierung von Ungleichungen in Gleichungen geschieht durch Addition einer Zusatzvariablen pro Ungleichung {slack/excess variable}. Aus einer Ungleichung $a_1x_1+a_2x_2 \geq b$ ($a_1x_1+a_2x_2 \leq b$) wird $a_1x_1+a_2x_2-e=b$, $e \geq 0$ ($a_1x_1+a_2x_2+s=b$, $s \geq 0$).

Eine nicht vorzeichenbeschränkte Variable x_i wird durch die Differenz zweier vorzeichenbeschränkter Variablen $x_i'-x_i''$ ($x_i', x_i'' \geq 0$) ersetzt, danach muß ein modifizierter Algorithmus durchgeführt werden (Big M oder Zweiphasen-Simplex)

Durch die Bedingungen wird im Vektorraum für $\vec{x}$ ein konvexer Raum beschrieben {feasible region}, innerhalb dessen eine gültige Lösung vorliegt. Dieses Polyhedron besitzt optimale Lösungen immer am Rand.
Prinzipiell können als Ergebnis verschiedene Fälle auftreten:

- Die erhaltene Lösung ist die einzige optimale Lösung (Ecke des Polyhedrons)
- Es gibt mehrere optimale Lösungen (diese liegen auf der Ebene, die aus den Verbindungsvektoren zwischen alternativ optimalen Eckpunkten aufgespannt wird)
- Es gibt keine optimale Lösung, weil es überhaupt keine Werte gibt, die die Bedingungen erfüllen {infeasible solution}
- Es gibt keine begrenzte optimale Lösung, da der Lösungsraum in Optimalrichtung nicht beschränkt ist {unbounded}.

Zur Lösung von LP-Problemen wird gewöhnlich der Simplex-Algorithmus benutzt. Dieser führt, ausgehend von einer Anfangslösung {basic feasible solution}, Reihenumformungen der Matrizengleichung mit Pivotisierung durch und führt nach endlicher Zeit zur Lösung.

Die Simplex-Methode ist kein echt polynomialer Algorithmus! Für ein LP-Problem der Größe n wird in seltenen Sonderfällen die optimale Lösung in einer Zeit von höchstens $c*2^n$ gefunden. In den meisten praktischen Fällen ist der Simplex-Algorithmus jedoch sehr effektiv und übersteigt nicht die polynomiale Komplexität.

Ein echter polynomialer Algorithmus ist mit Karmarkar's oder Khachiyan's Methode [61],[68] gegeben. Daher kann man lineare Programmierung als Problem polynomialer Komplexität bezeichnen.

Die Dualität hat für theoretische Untersuchungen große Bedeutung.
Das duale LP-Problem zu einem gegebenen {Primal-} Problem

$$\text{Maximiere } z=\vec{c} \cdot \vec{x} \text{ s.t. } A \cdot \vec{x} \leq \vec{b} \tag{6.3}$$

lautet:

$$\text{Minimiere } w=\vec{b} \cdot \vec{y} \text{ s.t. } A^{\mathrm{T}} \cdot \vec{y} \geq \vec{c} \tag{6.4}$$

Dabei sind y_i die neuen Entscheidungsvariablen.
Das Dualitätstheorem [61] sagt aus, daß wenn eine optimale Basis BV für das primale Problem existiert, $\vec{c}_{\mathrm{BV}} B^{-1}$ die optimale Lösung des dualen Problems ist und die optimalen Werte der Zielfunktion gleich sind: $z_{opt} = \vec{c}\vec{x}_{opt} = \vec{b}\vec{y}_{opt} = w_{opt}$

6.1.2 ILP-Aufgabe

An ein Problem der linearen Programmierung kann noch die (nicht lineare) Zusatzbedingung gestellt werden, daß einige oder alle Variablen ganzzahlige Lösungen aufweisen sollen.

$$\text{Maximiere } z=\vec{c}\cdot\vec{x} \text{ s.t. } A\cdot\vec{x}\le\vec{b},\ x_i \text{ integer} \tag{6.5}$$

Dieses Problem der ganzzahligen linearen Programmierung {integer linear programming, ILP} läßt sich konvertieren in ein anderes, in dem die Variablen nur boolesche Werte (0;1) annehmen dürfen. Beide Probleme verlangen die optimale Lösung im beschriebenen Zahlenraum; diese ist im Allgemeinen nicht identisch mit der auf ganze Zahlen gerundeten LP-Lösung! Die Ganzzahligkeit muß daher schon im Algorithmus berücksichtigt werden.

Zu diesem Zweck existieren zwei bekannte Methoden [61],[62],[63],[68]. Beide gehen von der LP-Relaxation des Problems und dessen Lösung aus (das LP-Problem wird so gelöst, als wären die Variablen reellwertig). Die weiteren Schritte werden iterativ durchgeführt, wobei nach jedem Schritt wieder ein (neues) LP-Problem gelöst wird.

Die Methode „Cutting Planes“ fügt in jedem Schritt (nach jeder LP-Lösung) dem Problem eine zusätzliche Ungleichung (Schnittebene) hinzu, die die erhaltene nicht-ganzzahlige Lösung abschneidet und im nächsten Schritt zu einer neuen Lösung findet. Das Polyhedron wird schrittweise auf das ganzzahlige Polyhedron reduziert. Durch die Ungleichungen wird aber nie eine machbare {feasible} ganzzahlige Lösung abgeschnitten. Der Algorithmus ist beendet, wenn die Lösung des letzten LP-Problems eine ganzzahlige Lösung liefert.

Die Methode „Branch and Bound“ verzweigt nach jeder erfolgten LP-Lösung rekursiv zu zwei neuen LP-Problemen {branch}, die durch Aufteilen des (Teil-)Polyhedrons in zwei kleinere Räume entstehen. Diese schließen beide die vorherige nicht-ganzzahlige Lösung aus. Wenn eine erhaltene LP-Lösung ganzzahlig ist, wird an dieser Stelle die Rekursion abgebrochen, der Wert der Zielfunktion festgehalten {candidate solution} und mit den anderen bisher festgehaltenen verglichen. Die Lösung ist ermittelt, wenn keine besserer Zielfunktionswert mehr erzielt werden kann {bound}.

Ganzzahlige lineare Programmierung ist NP-vollständig im strengen Sinn [68],[70-S.245], daher ist allgemein exponentielle Komplexität zu erwarten.

6.1.3 Komplexität

Die Komplexität eines Algorithmus ist die asymptotische Schranke für den Zeitbedarf eines Algorithmus. Für den Speicherbedarf kann ebenfalls eine Schranke angegeben werden. Die Komplexität wird als eine Funktion der Anzahl der Eingangsvariablen/-werte ausgedrückt. Für einen Graph sind dies z.B. die Anzahl der Knoten oder Kanten. Man schreibt, das Problem hat eine Komplexität der Ordnung $f(n)=O(g(n))$, wenn eine Konstante $c>0$ existiert, so daß für genügend große n gilt: $f(n)\le c*g(n)$ [62]. In der Regel sind die Eingangsdaten *Integer*-Zahlen oder lassen sich als solche darstellen. Da die Algorithmen gewöhnlich Operationen wie Addition, Vergleich und Multiplikation durchführen, ist die Komplexität eigentlich auch proportional zu $\log M$, wobei M die größte vorkommende ganze Zahl ist, denn der Zeitaufwand hängt auch von der Zahl der benutzten Bits ab.

- Polynomiale Algorithmen sind durch $O(n^K)$ beschränkt, K=konstant. Auch $O(n^K \log n)$ wird als polynomial bezeichnet.

- Exponentielle Komplexität liegt für Schranken wie $O(k^n)$, $O(n^{\log n})$, $O(n!)$ vor. Auch Binomialkoeffizienten in den Eingangswerten sind nicht mehr polynomial (Simplex).

Bei Problemen mit exponentieller Komplexität bringen auch 100 mal schnellere Computer keine signifikante Verbesserung. Für ein Problem $O(2^n)$ könnten z.B. an einem Tag nur 46 statt 40 freie Variablen berücksichtigt werden.
Die Tabelle soll die Unterschiede der Komplexität verdeutlichen:

Komplexität des Algorithmus	Zeitmultiplikator		
n	10	100	1000
$n\log n$	33	664	9966
n^3	1000	1.000.000	10^9
$10^6 n^8$	10^{14}	10^{22}	10^{30}
2^n	1024	$1{,}27*10^{30}$	$1{,}05*10^{301}$
$n^{\log n}$	2099	$1{,}93*10^{13}$	$7{,}89*10^{29}$
$n!$	3.628.800	10^{158}	$4*10^{2567}$

Unter „pseudo-polynomialer“ Komplexität versteht man einen Aufwand, der polynomial beschränkt in zwei Variablen ist: der Länge (Anzahl) der Eingangsdaten {Length[*I*]} und dem Maximum der Werte der Eingangsvariablen {Max[*I*]}. Es ist keine echt polynomiale Komplexität, die nur polynomial in Length[*I*] beschränkt ist.

6.1.4 NP-Vollständigkeit

Die Theorie der NP-Vollständigkeit beschäftigt sich mit den Problemen, für die kein effizienter (polynomialer) Algorithmus existiert. Dafür werden zwei Klassen definiert [58]:

P: Menge aller Probleme, die mit Hilfe deterministischer Algorithmen in polynomialer Zeit gelöst werden können.
NP: Menge aller Probleme, die mit Hilfe nichtdeterministischer Algorithmen in polynomialer Zeit gelöst werden können.

Die Fähigkeit des Nichtdeterminismus ist die unerfüllbare Annahme, daß bei der Wahl zwischen verschiedenen Varianten der Algorithmus die richtige „errät“. Überhaupt sind die Methoden, die im Zusammenhang mit NP-Vollständigkeit benutzt werden, meistens sehr unanschaulich, wie die Turingmaschine, die im Satz von Cook [70] zum Beweis der NP-Vollständigkeit des Erfüllbarkeitsproblems[1] verwendet wird.
Bis heute ist nicht bewiesen worden, daß $P \neq NP$ ist [58],[62].

Ein Problem, das *NP* angehört, muß als Entscheidungsproblem formuliert werden können, das mit „Ja“ oder „Nein“ beantwortet werden kann. Eine Instanz *x* des Problems muß in polynomialer Zeit verifizierbar sein, d.h. man kann mit polynomialem Aufwand feststellen, ob das Ergebnis des Algorithmus das Problem löst [62-S.348].

NP-vollständige Probleme sind definiert als solche Probleme, die (a) aus *NP* sind und (b) in die sich alle anderen Probleme aus *NP* polynomial transformieren lassen [70]. Für einen Beweis der NP-Vollständigkeit muß das Problem Π als Ja/Nein-Entscheidungsproblem[2]

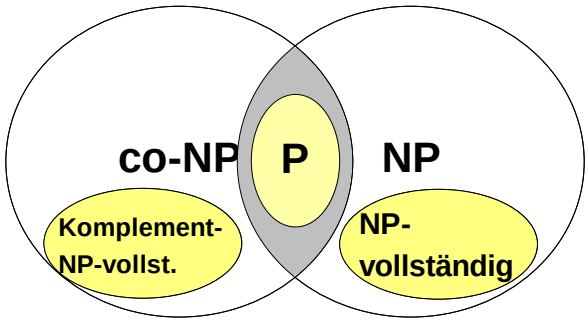
[1] Gegeben ist eine Boolesche Formel; kann sie für irgendwelche Argumente wahr werden ?
[2] Ein Beispiel ist im Kapitel 6.2.1 angegeben

formuliert werden. Eine polynomiale Prozedur muß entscheiden können, ob spezielle Werte der Variablen das Problem lösen (dann gehört es *NP* an). Anschließend muß gezeigt werden, daß sich ein bekanntes NP-vollständiges Problem polynomial in das Problem Π transformieren läßt. Dann lassen sich gemäß Definition nämlich alle NP-vollständigen Probleme dorthin transformieren.
Eine etwas einfachere Methode kann der Beweis durch Einschränkung sein, für den zu zeigen ist, daß das Problem Π eine Verallgemeinerung eines NP-vollständigen Problems ist, was gleichbedeutend damit ist, daß sich Π auf dieses Problem durch Einschränkung reduzieren läßt {restriction} [62-S.391],[70-S.63].

Die Klasse *co-NP* ist die Klasse aller Probleme, die Komplemente der Probleme in *NP* sind. Nicht alle Probleme aus *co-NP* sind in *NP* enthalten. Die Komplemente der NP-vollständigen Probleme stellen eine Unterklasse der Klasse *co-NP* dar und sind nicht in *NP* [62-S.386].

Daneben bezeichnet man als „NP-schwierig“ {NP-hard} die Probleme, auf die sich zwar alle Probleme aus *NP* polynomial reduzieren lassen, deren Zugehörigkeit zu *NP* aber nicht gezeigt werden kann [62-S.397].

Trotz der NP-Vollständigkeit können einige Probleme durch einen Algorithmus in pseudo-polynomialer Zeit gelöst werden [70-S.94]. Die NP-vollständigen Probleme, für die das nicht möglich ist, werden „NP-vollständig im strengen Sinn“ genannt.
Die Einteilung der Klassen nach bisherigem Wissen soll am folgenden Bild erkennbar sein:



6.2 Schedultheorie

6.2.1 Übersicht

Ein ganz generelles Scheduling-Problem kann allgemein durch folgende Beziehungen beschrieben werden [62]:

> Gegeben ist ein System von Aufträgen $V=\{v_1,...,v_n\}$. Für jeden Auftrag $v_i \in V$ ist die Ausführungszeit t_i bekannt und die Quantität R_{ji} der j.ten Ressource, die er benötigt ($j \in \{1,...,r\}$). Die Aufträge sollen auf p Prozessoren ausgeführt werden; ein boolescher Wert a_{ij}, $i \in \{1,...,n\}$, $j \in \{1,...,m\}$, gibt an, ob v_i auf Prozessor j ausgeführt werden kann. Mit jedem v_i kann eine Deadline T_i verbunden sein; v_i muß die Ausführung T_i Zeiteinheiten nach dem Start des ersten Auftrags beendet haben. Zusätzlich existiert eine Präzedenz-

relation (V,E) (Ein azyklischer Graph aus Knoten und Kanten), so daß $(v_i,v_j) \in E$ bedeutet, daß v_i seine Ausführung beendet haben muß bevor v_j seine startet.

Die Frage „gibt es ein Schedul *S*, das die obigen Bedingungen erfüllt?“ ist NP-vollständig [70],[62]. Die Formulierung als ILP-Problem mit der Technik in [62] bestätigt die Komplexität, da ganzzahlige lineare Programmierung NP-vollständig ist.
Auch einige vereinfachte Unterprobleme gehören dieser Klasse an [70]:

- „Multiprocessor Scheduling“ mit unterschiedlichen Ausführungszeiten und einer Gesamt-Deadline; ist zwar ab p=2 Prozessoren NP-vollständig, kann aber für jedes feste p in pseudopolynomialer Zeit gelöst werden. Trivial und polynomial lösbar bei Einheits-Ausführungszeit.
- „Precedence Constrained Scheduling“ mit beliebigem Präzedenzgraph *G* und beliebig viele Prozessoren *p*. Es sind jedoch für Einheits-Ausführungszeit die Unterprobleme $p<3$ oder *G* ein Baum oder *G* leer in polynomialer Zeit lösbar! Die Unterprobleme $m<4$ bis $m<C$, *C* fest, sind noch offen.
- „Ressource Constrained Scheduling“ mit begrenzten Ressourcen und Gesamt-Deadline; kann aber für zwei Prozessoren in pseudopolynomialer Zeit gelöst werden.
- „Scheduling mit individuellen Deadlines und Präzedenzen“; nur polynomial lösbar für $p<3$.
- „Präemptives Scheduling mit Gesamt-Deadline und Präzedenzen“; polynomial für $p<3$.

6.2.2 Ein-Prozessor-Schedul

Ein PASS[3] [18] ist ein Schedul, das die Bedingungen des Datenflußkonzepts (Knoten starten, wenn genug Daten vorhanden sind) berücksichtigt, sich nach Vielfachen einer festen Zeit (der Iterationsperiode) periodisch wiederholt und auf einem Prozessor (sequentiell) läuft. Ein solches Schedul zu konstruieren, stellt im Einheitsraten- wie im Multiratenfall kein Problem dar, weil zu jeder Zeit nur ein lauffähiger Knoten gefunden werden muß, der der Schedul-Liste hinzugefügt wird. Eine solche Liste kann als Sequenz (z.B. Φ={A,B,B,C} für die Knoten A,B und C) geschrieben werden. Nach q_v Aktivierungen jedes Knotens *v* ist eine Periode des Scheduls beendet. Daraus ergibt sich ein pseudo-polynomialer Aufwand $O(|V|*\Sigma q)$. Das Schedul lastet den Prozessor zu 100% aus und benötigt eine Iterationsperiode von $T_{ip} = \vec{q} \cdot \vec{t}$. Der folgende Algorithmus findet dieses Schedul und kann auch zur Feststellung der Lebendigkeit genutzt werden.

6.2.3 Klasse-S-Algorithmus

Def.: Ein Klasse-S-Algorithmus [18] ist ein beliebiger Algorithmus, der einen ausführbaren Knoten der Schedule hinzufügt, den Vektor *b* aktualisiert, und erst dann aufhört, wenn keiner der Knoten mehr ausführbar ist. Dabei ist ein Knoten *i* ausführbar, wenn genügend Eingangs-Delays vorhanden sind und die Anzahl der bisherigen Ausführungen von Knoten *i*, b_i, die maximale Ausführungshäufigkeit q_i nicht überschreitet. Wenn der Algorithmus abbricht, bevor jeder Knoten *i* q_i mal in die Schedule eingefügt wurde (b=q), dann ist der Deadlock-Zustand erreicht.

Notwendige Bedingung für die Existenz eines PASS ist dir Ratenkonsistenz (Rang(Γ)=s-1).
Wenn ein PASS existiert, dann findet der Algorithmus auch einen.

[3] PASS=Periodic Admissible Sequential Schedule

Klasse-S-Algorithmus (zur Konstruktion einer PASS):

1) Löse nach dem kleinsten Vektor aus natürlichen Zahlen auf, der zum Nullraum von Γ gehört:

$$\vec{q} \in \eta(\Gamma) \quad \Gamma \cdot \vec{q} = \vec{0} \; .$$

2) Erstelle eine beliebig geordnete Liste L aller Knoten des Systems
3) Für jeden Knoten α aus Γ: schedule α, wenn er ausführbar ist, probiere dabei jeden Knoten einmal.
4) Wenn jeder Knoten α q_α mal geschedult wurde, beende Algorithmus.
5) Wenn kein Knoten mehr geschedult werden kann, zeigen ein Deadlock an. (Fehler im Graphen)
6) Wiederhole Schritt 3 und folgende.

6.2.4 Multiprozessor-Schedul

Das Ziel eines Multiprozessor-Schedulings ist die Generation eines PAPS[4], das vorgegebene Kriterien optimal erfüllt. Die Existenz eines generellen PAPS ist schon gesichert, wenn ein PASS existiert. Die formale Definition [18] lautet:
Eine periodisches zulässiges paralleles Schedul für *M* Prozessoren ist eine Menge von Listen $\{\Phi_i \mid i=1,..,M\}$, wobei Φ_i das Schedul für einen Prozessor angibt.
Der Scheduler soll nun das optimale Schedul bezüglich des Gesamtdurchsatzes finden. Dazu ist es in der Regel nötig mehrere Iterationen (wie sie durch den *q*-Vektor festgelegt sind) zu einer Periode des PAPS zusammenzufassen, um den maximalen Durchsatz zu erreichen. Dieser Faktor *J* kann schon zur Vorbereitung des Schedulings durch die Transformationen Unfolding und Blocking berücksichtigt werden. Die Iterationsperiode des erhaltenen Scheduls ist die Laufzeit für eine Periode des PAPS dividiert durch diesen Faktor *J*. Der optimale Faktor *J* ist für Einheitsraten-Graphen bekannt [19]. Man kann zur Bestimmung das Optimum berechnen oder sich auf (suboptimale) Heuristiken verlassen. Die Wahl hängt von der zulässigen Komplexität ab. In den folgenden Abschnitten werden übliche Techniken vorgestellt.

6.3 Schedultechniken und -ergebnisse

6.3.1 Präemptives - nicht präemptives Scheduling

Ein ‘nicht-präemptives’ Schedul gewährleistet die Abarbeitung einer Auftragseinheit {task} in einem Stück ohne Unterbrechung bis zum Ende. Das ‘präemptive’ Schedul kann laufende Aufträge vorübergehend unterbrechen, um einen anderen Auftrag (weiter) zu bearbeiten. Häufig wird dabei ein Zeitscheibenverfahren {time-slice} angewendet, das jedem Prozeß genau eine Zeitscheibe fester Länge zuteilt, nach deren Ablauf der nächste Prozeß an der Reihe ist. Der Verwaltungsaufwand zum Umschalten des Kontextes {context-switch} kann insgesamt die Ausführungszeiten verschlechtern, dafür spielt aber die zeitliche Länge eines Blocks keine Rolle und die Parallelität kann i.a. besser ausgenutzt werden.

[4] PAPS=Periodic Admissible Parallel Schedule

Für die digitale Signalverarbeitung spielt nur das ‘nicht-präemptive’ Schedul eine Rolle, da es statisch durchgeführt werden kann.

6.3.2 Dynamisch - statisch

Das Scheduling kann zur Laufzeit oder zusammen mit der Compilation durchgeführt werden. Ersteres bezeichnet man als dynamisch, letzteres als statisch. Nach der Zeit, zu der die Teilaufgaben des Schedulers erledigt werden, kann aber noch genauer unterschieden werden, was zur folgenden Klassifikation [22] führt:

	Zuweisung	Reihenfolge	Timing
voll dynamisch	Laufzeit	Laufzeit	Laufzeit
statische Zuweisung	Compilation	Laufzeit	Laufzeit
selbst-zeitgesteuert	Compilation	Compilation	Laufzeit
voll statisch	Compilation	Compilation	Compilation

Ziel für eine DSP-Implementation sollte ein voll statisches Schedul sein, um jeden Zusatzzeitaufwand zur Laufzeit zu vermeiden. Jeder reguläre Datenflußgraph läßt sich statisch scheduln.

6.3.3 Überlappendes - Nicht-Überlappendes Schedul

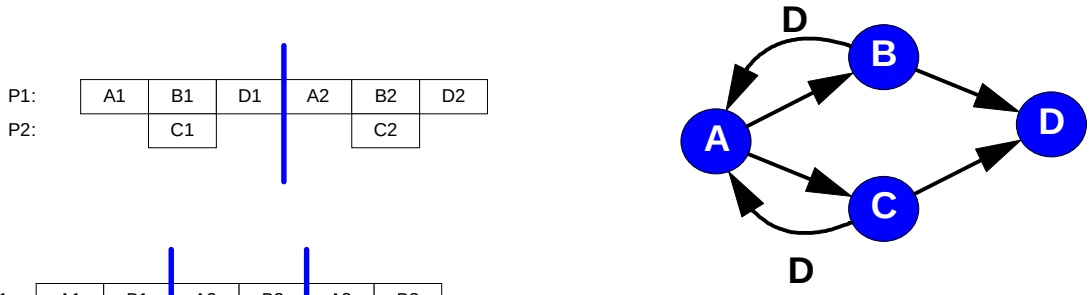
Die für die Signalverarbeitung üblichen iterativen Programme benötigen ein periodisch wiederholtes Schedul. Die Trennung zwischen den Iterationen kann klar zu festen Zeiten entschieden werden, wenn sich die Scheduls verschiedener Iterationen nicht überlappen.

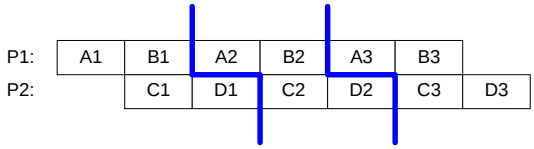
Bei nicht-überlappenden Scheduls beginnt eine Iteration erst, wenn die vorherige Iteration vollständig abgearbeitet wurde. Ein überlappendes Schedul erfordert dies nicht, also existieren Zeitpunkte, an denen gleichzeitig zwei aufeinanderfolgende Iterationen bearbeitet werden.
Nicht immer gelingt es, einen Zeitabschnitt zwischen zwei Übergangszeitpunkten voll auszufüllen und damit den maximalen Durchsatz zu erreichen. Ein überlappendes Schedul kann einige Lücken mit Prozessen aus anderen Iterationen füllen, wenn die zeitliche Parallelität gegeben ist.

Nicht immer kann ein einfaches überlappendes Schedul konstruiert werden. Das Schedulmuster {iteration tile} jeder Iteration muß aber nicht immer identisch aussehen, sondern kann auf eine der zwei Arten verändert sein:

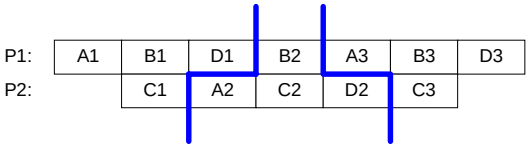
(1) Das Muster ist in der Prozessorzuweisung {assignment} permutiert, wobei zwischen Iterationen eine Zeitverschiebung {time displacement} und eine Prozessorverschiebung {processor displacement} die Variation beschreiben. Diese Scheduls werden zyklostatisch {cyclo-static} genannt [19].
(2) Durch Zusammenfassung von *J* Iterationen zu einer Periode, deren internes Gesamt-Schedulmuster nicht regulär strukturiert sein muß, kann diese Periode wieder nicht-überlappend an die vorherige angehängt werden. Techniken dazu sind Unfolding und Blocking. Das Gesamt-Schedul ist dann wieder voll statisch {fully-static} [19].

Retiming kann häufig dazu benutzt werden, den Graphen nicht-überlappend schedulbar zu machen. Dies ist leider nicht in allen Fällen möglich.
Die folgenden Bilder zeigen einen Datenflußgraph und ein nicht-überlappendes, überlappendes und cyclo-statisches Schedul (von oben nach unten; man beachte die unterschiedliche Iterationsperiode):





(überl
appendes Schedul)



(cyclo-statisches Schedul)

6.3.4 Heuristiken

Praktische Schedulverfahren verwenden häufig die Methode des kritischen Pfades (CPM[5]).

Der kritische Pfad ist der Pfad zwischen einem Ein- und Ausgangsknoten, der die höchste aufsummierte Ausführungszeit hat. Er bestimmt die Zeit ‘Makespan’ und den Durchsatz bei nicht-überlappenden Scheduln.

Der erste Schritt ist die Konstruktion des azyklischen Präzedenzgraphen (APG, siehe Kapitel „Transformationen“) für *J* Iterationen. Dieser beinhaltet die volle Information über die Prioritäten der Knotenprozesse innerhalb der *J* Iterationen.

Der nächste Schritt ist die Bestimmung eines Levels für jeden Knoten, je nachdem wie weit er zeitlich von den Endknoten des APG entfernt ist. Das Level ist also die maximale Summe der Ausführungszeiten aller Knoten vom aktuellen bis zum Endknoten, ermittelt über alle Pfade, die zu einem Endknoten führen. Der Knoten, der sich am Anfang des kritischen Pfades befindet, hat dann das höchste Level.

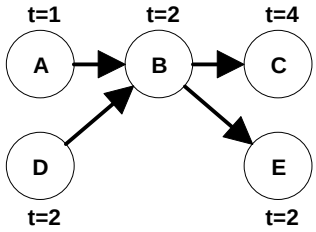
Im letzten Schritt werden die Knoten so geschedult, daß zuerst die Knoten mit dem höchsten Level an die Reihe kommen. Der Algorithmus wird daher „Highest-Level-First“ (HLF) genannt [18]. Er ist der Klasse der List-Scheduler zuzuordnen, bei der immer zuerst der Knoten ausgewählt wird, der ein Kriterium maximiert.

Andere Heuristiken lassen in die Berechnung des Levels noch andere Faktoren einfließen [12], die beispielsweise bei gegebener Prozessorverteilung der Knoten {a priori node assignment} die zur Kommunikation verwendeten Kanten besonders berücksichtigen. Als günstig wurde es dort angesehen, Sendeknoten möglichst früh und Empfangsknoten möglichst spät zu scheduln.

Beispiel für ein Scheduling mit HLF: Der kritische Pfad des gegebenen APG ist D-B-C, das höchste Level hat Knoten D mit L=8.

[5] CPM=Critical Path Method

Knoten	Level
A	7
B	6
C	4
D	8
E	2



Die Methode PERT[6] wird manchmal eingesetzt, wenn die Ausführungszeiten nicht genau, sondern nur statistisch mit Erwartungswert und Varianz bekannt sind [61].

6.3.5 Raten-optimales Scheduling

Ein Schedul, dessen Iterationsperiode minimal ist und dadurch die Iterationsperiodengrenze erreicht, ist das unbestrittene Ziel des Schedulings. Ein solches raten-optimales Scheduling [21] kann für perfekt[7]-ratige Datenflußprogramme (Einheitsraten) immer durchgeführt werden. Die Überführung in einen solchen Graphen ist durch Unfolding möglich. Natürlich kann nur optimal geschedult werden, wenn keine Beschränkung der Prozessoranzahl zugelassen wird.

Die optimale Lösung kann im allgemeinen nur mit kombinatorischer Optimierung gefunden werden. Dazu muß das Schedulproblem in ein Problem der ganzzahligen linearen Programmierung konvertiert werden. Das Verfahren ist in [62] und [12] beschrieben. Die ILP-Formulierung bestätigt die NP-vollständige Komplexität.

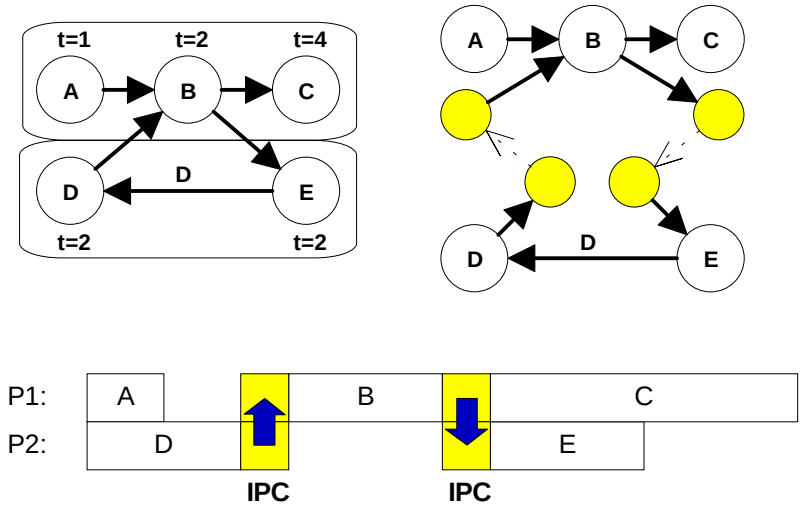
[6] PERT=Program Evaluation and Review Technique
[7] perfect-rate DFG's sind Einheitsratengraphen, die in jeder Schleife genau ein Delay haben

6.3.6 Inter-Prozessor-Kommunikation

Bei der Implementation von Datenflußprogrammen auf mehreren Prozessoren ist die Kommunikation zwischen Blöcken zu berücksichtigen, deren Knoten im Datenflußgraph durch eine Kante verbunden sind und die verschiedenen Prozessoren zugewiesen sind. Zur Inter-Prozessor-Kommunikation (IPC) können die Daten entweder synchron übergeben werden, dazu muß ein Prozeß auf jedem der beiden Prozessoren zum Senden und Empfangen gleichzeitig laufen, oder es wird ein asynchroner Transfer mit Hilfe der DMA-Controller oder über gemeinsamen Speicher durchgeführt. Im letzteren Fall können andere Prozesse weiterlaufen.

Zur Synchronisation müssen geeignete Maßnahmen durchgeführt werden, zum Beispiel Nachrichten {messages} zum Starten der Übertragung oder Semaphore zum Verhindern von Überschreiben des Zwischenspeichers bzw. Verwendung nicht-aktueller Werte.

Die Zeit zur Kommunikation ist im Schedul einzuplanen. Praktisch ist es, Kommunikations-Knoten, denen die Transfer-Zeit zugewiesen wurde, an die Teilgraphen anzufügen.



7 Retiming bei Multiraten-Datenflußgraphen

Die Retiming-Transformation ist die vielseitigste Transformation, die ohne die Funktionalität zu beeinträchtigen, die Verteilung der Delays verändert. Das wird genutzt, um z.B. eine minimale Anzahl von benötigten Speicherelementen zu erreichen oder um ein besseres Schedul zu erhalten.

Für Einheitsraten-Graphen ist Retiming mit relativ geringem Aufwand durchführbar. Für Multiraten-Graphen ergibt sich wegen spezifischer Anomalien ein im allgemeinen NP-vollständiger Aufwand. Für Sonderfälle ist jedoch, wie im Folgenden gezeigt, eine Lösung in polynomialer Zeit ermittelbar.

7.1 Retiming für Einheitsraten-Graphen

Retiming [33],[21],[25] ist eine Transformation $G \rightarrow G_r$ des ursprünglichen Einheitsraten-Graphen $G=\{V,E(d)\}$ zu dem 'retimed' Graphen $G_r=\{V,E(d_r)\}$, wobei für jede Kante e, die die Knoten u und v verbindet, die Anzahl der Delays d_r im transformierten Graph diese Gleichung erfüllen:

$$d_r(e) = d(e) + r(u) - r(v) \tag{7.1}$$

Die Transformationsfunktion $r(v):V \rightarrow \mathbf{Z}$ ordnet jedem Knoten $v \in V$ einen ganzzahligen Wert r zu, der als Vektor $\vec{r}$ zusammengefaßt dem Aktivierungsvektor (Parikh-Vektor) entspricht, der den (Delay-) Zustand $\vec{d}$ in den Zustand $\vec{d}_r$ überführt:

$$\vec{d}_r - \vec{d} = \Gamma \cdot \vec{r} \tag{7.2}$$

Diese Zustandsgleichung in Matrixform macht erkennbar, daß Retiming nicht den dem Graphen zugrundeliegenden Algorithmus verändert, denn der neue Zustand ist durch ein Initial-Schedul erreichbar. Die Äquivalenz zwischen der Bedingungen für legales Retiming und denen für Erreichbarkeit erlaubt es, einen Graphen durch Retiming in einen erreichbaren Zustand $\vec{d}_r$ zu überführen. Es können aber auch nur erreichbare Zustände $\vec{d}_r$ Ergebnis einer Retiming-Transformation sein. In jedem Fall muß gewährleistet sein, daß an jeder Kante nur eine nicht-negative Anzahl von Delays vorhanden ist:

$$d(e) \geq 0 \ \ \forall\, e \in E \text{ , anders ausgedrückt: } \vec{d}_r \geq \vec{0} \tag{7.3}$$

Die notwendige und hinreichende Bedingung für legales Einheitsraten-Retiming ist in mehreren alternativen Formulierungen verwendbar [37],[39],[28]:

1) $\vec{d}_r \in R(\vec{d}) = PR(\vec{d})$
2) Es existiert ein ganzzahliger Vektor $\vec{r} \in \mathbf{Z}^{|V|}$, der die Zustandsgleichung erfüllt.
3) $\vec{d}_r \in PR^B(\vec{d}) \Leftrightarrow$ Die Summe der Delays in jeder fundamentalen Schleife bleibt konstant.

Die dritte Form der Bedingung ergibt sich aus der Tatsache, daß (nur im Einheitsratenfall) jeder ganzzahlige Zustand erreichbar ist, der die Gleichung

$$C \cdot \vec{d}_r = C \cdot \vec{d} = \vec{WST} \Leftrightarrow C \cdot \Delta\vec{d} = \vec{0} \tag{7.4}$$

erfüllt. Diese verlangt, daß in jeder fundamentalen Schleife (beschrieben durch die fundamental-circuit-Matrix) die (einfache) Summe der Delays konstant und gleich $WST=D_{sum}$

ist [37],[28]. Nur die zweite und dritte Bedingung kann in echt polynomialer Zeit getestet werden, da die Bestimmung der erreichbaren Zustände NP-vollständig ist.

Die Konstanz der Delaysumme garantiert, daß ein Vektor $\vec{r} \in \mathbf{Q}^{|V|}$ gefunden werden kann, der die Zustandsgleichung erfüllt (partikuläre Lösung des Gleichungssystems). Nur im Einheitsratenfall ist aber dann auch gesichert, daß ein ganzzahliges $\vec{r}$ existiert. Wenn ein (rationaler) $\vec{r}$-Vektor existiert, dann kann dieser durch Addition eines skalaren Vielfachen des $\vec{q}$-Vektors ganzzahlig gemacht werden, da jedes Vielfache von $\vec{q}$ auch eine homogene Lösung der Zustandsgleichung ist.

Retiming verändert nicht die Eigenschaften eines Datenfluß-Graphen wie Lebendigkeit, Beschränktheit und Konsistenz. Dies gilt sowohl für Einheitsraten- als auch für Multiraten-Graphen [29].

7.2 Multiraten-Retiming

Multiraten-Retiming [29] ist eine Transformation $G \rightarrow G_r$ des ursprünglichen Multiraten-Graphen $G=\{V(I,O),E(d)\}$ zum Graphen $G_r=\{V(I,O),E(d_r)\}$, wobei für jede Kante e, die die Knoten u und v verbindet, die Anzahl der Delays d_r im transformierten Graph diese Gleichung erfüllen:

$$d_r(e) = d(e) + O(u,e) \cdot r(u) - I(v,e) \cdot r(v) \tag{7.5}$$

Wie im Einheitsratenfall lautet die vektorielle Schreibweise (Zustandsgleichung):

$$\vec{d}_r - \vec{d} = \Gamma \cdot \vec{r} \tag{7.6}$$

Die Existenz eines ganzzahligen Retiming-Vektors $\vec{r}$ ist notwendige und hinreichende Bedingung für legales Retiming [29], basierend auf der Erkenntnis zur Erreichbarkeit des Zustands $\vec{d}_r \in R(\vec{d}) = PR(\vec{d})$.

Der $\vec{r}$-Vektor kann auch negative Komponenten besitzen. Ein negativer Retiming-Koeffizient $r(v)$ für einen Knoten v bedeutet, daß dieser ‘rückwärts’ aktiviert werden muß, also Werte von der Ausgangskante einliest und auf der Eingangskante ausgibt, je nach der entsprechenden Rate. Jeder ganzzahlige $\vec{r}$-Vektor läßt sich aber nicht-negativ machen durch Addition eines ganzzahligen Vielfachen des $\vec{q}$-Vektors, d.h. für jedes $\vec{r}$ als Lösung ist auch

$$\vec{r} + k\,\vec{q}\ ,\ \mathrm{k} \in \mathbf{Z} \tag{7.7}$$

eine ganzzahlige Lösung des (inhomogenen diophantischen) Gleichungssystems. Negative Werte $r(v)$ können Probleme bei der Berechnung der Anfangswerte bereiten. Dieses Problem ist für die Implementation des durch Retiming modifizierten Algorithmus von Bedeutung und wird im abschließenden Abschnitt behandelt.

Durch Multiplikation der Zustandsgleichung von links mit der ‘fundamental circuit’-Matrix C folgt eine notwendige Bedingung für legales Retiming, denn $C\Gamma$ ist nach Definition eine Nullmatrix:

$$\vec{d}_r - \vec{d} = \Gamma \cdot \vec{r} \quad | \ C^* \tag{7.8}$$

$$\Rightarrow \quad C \cdot (\vec{d}_r - \vec{d}) = C \cdot \Gamma \cdot \vec{r}$$
$$\Leftrightarrow \quad C \cdot \vec{d}_r - C \cdot \vec{d} = \underline{0} \cdot \vec{r} = \vec{0}$$
$$\Leftrightarrow \quad C \cdot \vec{d}_r = C \cdot \vec{d} = \vec{WST}$$

Diese Bedingung ist, im Gegensatz zum Einheitsratenfall, nicht hinreichend für legales Retiming. Sie garantiert lediglich, daß ein rationaler Vektor $\vec{r}$ existiert, der die Zustandsgleichung erfüllt. Für den dazu gehörigen Zustand gilt dann $\vec{d}_r \in PR^B(\vec{d})$. Nur wenn das Retiming legal war läßt sich ein rationales $\vec{r}$ durch Addition eines rationalen Vielfachen von $\vec{q}$ ganzzahlig machen. Dann gilt $\vec{d}_r \in R(\vec{d})$

Eine hinreichende Bedingung, die nicht auf der Ganzzahligkeit von $\vec{r}$ basiert, wird in einem folgenden Abschnitt vorgestellt. Dies hat große Bedeutung für das automatische Retiming, das beispielsweise durch lineare Optimierung durchgeführt wird[1].

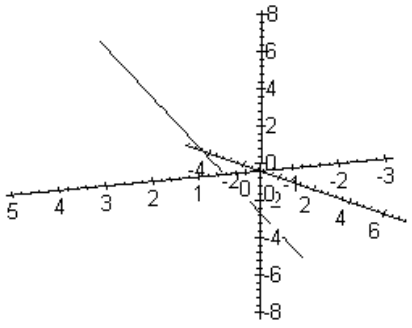
7.2.1 Berechnung des Retiming-Vektors

Die Zustandsgleichung ist ein inhomogenes diophantisches[2] Gleichungssystem [66],[67], dessen Existenz einer Lösung $\vec{r}$ in polynomialer Zeit geprüft werden kann [45]. Dazu wird sie gemäß [45] in ein homogenes System umgewandelt, dessen Lösungsvektor eine zusätzliche Komponente *h* hinzubekommt:

$$\Gamma \cdot \vec{r} = \Delta\vec{d} \tag{7.9}$$
$$\Rightarrow \quad \left[\Gamma \vdots - \Delta\vec{d}\right] \cdot \begin{bmatrix} \vec{r} \\ \hline h \end{bmatrix} = \vec{0}$$

Stellt sich als minimale ganzzahlige Lösung des homogenen Systems *h*=1 heraus, genau dann existiert die Lösung, die in $\vec{r}$ enthalten ist. Das Lösungsverfahren für ein homogenes diophantisches Gleichungssystem ist äußerst aufwendig (u.a. Zerlegung in Hermit'sche oder Smith'sche Normalform), kann aber in polynomialer Zeit gelöst werden [68],[45].

Die gefundene Lösung $\vec{r}$ ist nicht eindeutig, da immer die homogene ganzzahlige Lösung $\vec{q}$ hinzuaddiert werden darf. Der Lösungsraum des rationalen inhomogenen Gleichungssystems, das sich auch mit einer Variante des Gauß-Algorithmus lösen läßt, ist eine Gerade in |*V*|-dimensionalen Raum für $\vec{r}$ (Bild rechts: |*V*|=3). Die Gerade wird beschrieben durch einen Ortsvektor $\vec{r}$ (partikuläre Lösung) und einen Richtungvektor $\vec{q}$ (homogene Lösung).



Diese Gerade kann ganzzahlige Gitterpunkte schneiden, d.h. enthalten, wenn eine ganzzahlige Lösung existiert. Andernfalls verläuft sie nur durch echt rationale Punkte. Wenn sie einen ganzzahligen Punkt enthält, dann enthält sie unendlich viele ganzzahlige Punkte, die sich im Abstand $|1*\vec{q}|$ wiederholen. Wird eine rationale partikuläre Lösung $\vec{r}_p$ gefunden, dann muß eine ganzzahlige Lösung $\vec{r}_g$ im

[1] siehe Abschnitt „Retiming durch lineare Optimierung".
[2] diophantische Gleichungen erfordern die Ganzzahligkeit der Lösungsvariablen. Sie sind keineswegs so trivial wie lineare Gleichungssysteme lösbar. Für die Existenz der Lösung gibt es keine geschlossene Bedingung.

Intervall[3] $k \in [0;1[$ zu finden sein, wenn überhaupt eine existiert, wobei gilt:

$$\vec{r}_g = \vec{r}_p + k * \vec{q} \tag{7.10}$$

Eine Alternative zur Berechnung ist die Formulierung als Problem der ganzzahligen linearen Optimierung, deren asymptotische Komplexität allerdings exponentiell ist. Ein ILP-Lösungsprogramm stellt fest, ob eine ganzzahlige Lösung existiert und liefert diese gegebenenfalls:

$$\min 1*\vec{r} \quad \text{s.t.} \quad \Gamma \cdot \vec{r} = \Delta\vec{d}, \quad \vec{r} \text{ integer} \tag{7.11}$$

Dieselbe Formulierung als LP-Problem liefert den kürzesten Ortsvektor $\vec{r}$, der vom Koordinatenursprung auf die Gerade zeigt (Lot).

$$\min 1*\vec{r} \quad \text{s.t.} \quad \Gamma \cdot \vec{r} = \Delta\vec{d} \tag{7.12}$$

Der folgende neu entwickelte Algorithmus soll den Vektor $\vec{r}$ bei gegebener Ursprungs- und Zielverteilung berechnen ohne ein diophantisches Gleichungssystem oder ILP-Problem lösen zu müssen, das in der Praxis lange Rechenzeit beansprucht. Die Chancen zur Verbesserung sind naheliegend, denn die Matrix Γ enthält nur zwei von Null verschiedene Einträge pro Zeile. Ein auf der Tiefensuche basierender Algorithmus, ähnlich dem zur Bestimmung von $\vec{q}$, nutzt die Tatsache aus, daß sich die Lösung für den ersten Knoten $r(v)$ im Intervall $[0;q(v)[$ befinden muß. Zu jedem ganzzahligen $r'(v)$ aus diesem Intervall lassen sich die anderen $r'(u)$, $u \neq v$, durch DFS bestimmen. Sind diese ganzzahlig, dann ist die Lösung gefunden. Gibt es zu keinem $r'(v)$ vollständig ganzzahlige Lösungen, dann existiert überhaupt keine ganzzahlige Lösung für $\vec{r}$.

Algorithmus zur Bestimmung des $\vec{r}$ -Vektors im Multiratenfall:

Gegeben: $\vec{d}$, $\vec{d}_r$, $\vec{q}$

Berechne $\Delta\vec{d} = \vec{d}_r - \vec{d}$

Bestimme Knoten v_{Start}, für den $qs := q(v_{Start})$ minimal ist.

Für jedes ganzzahlige m {entspricht $r'(v_{Start})$} aus $[0;qs[$ durchlaufe

- Setze den Vektor $\vec{r}_{temp}$ zu Null
- Setze Menge der besuchten Kanten E_B und Knoten V_B zu $\varnothing$ „Leere Menge“
- Starte eine DFS {depth first search} im Knoten $v = v_{Start}$: OK=VISIT(v, m)
- Wenn Ergebnis OK, dann existiert ein $\vec{r} := \vec{r}_{Temp}$; breche Schleife ab.

Wenn OK=wahr, dann ist das zuletzt bestimmte $\vec{r}$ gültig, sonst existiert kein ganzzahliges $\vec{r}$.

Funktion VISIT(Knoten v, Integer m); Rückgabewert: OK

Setze $r_{temp}(v) := m$

$V_B := V_B \cup \{v\}$

Für jede adjazente Kante e von v, die nicht besucht wurde: $e \in (E \setminus E_B) \cap \text{Adj}(v)$ durchlaufe

- $E_B := E_B \cup \{e\}$
- Ermittle den mit v über e verbundenen Knoten u
- Bestimme $\rho_1 := \text{Rate}(v,e)$ und $\rho_2 := \text{Rate}(u,e)$; kehre Vorzeichen um, wenn Zielknoten
- Berechne, welches ganzzahlige $r'(u)$ anderer Knoten haben müßte:
- $r'(u) = \lfloor (\Delta d(e) - \rho_1 * m)/r_2 \rfloor$ {Berechne neues r'}
- OK:=wahr, wenn $(\rho_1 * m + \rho_2 * r'(u)) = \Delta d$, sonst falsch und Abbruch der Funktion

[3] Andere Schreibweise für halboffenes Intervall: [0;1)

Wenn $u \notin V_B$, dann OK:=$OK \wedge$ VISIT(u, $r'(u)$)	{Rekursion}
sonst OK:=$OK \wedge (r'(u)=r_{temp}(u))$	{stimmen die r' überein ?}
Wenn OK=falsch, breche Schleife ab.	
Rückgabe: OK	

In Experimenten hat sich der Algorithmus als drei bis zwanzig mal schneller als die ILP-Berechnung erwiesen, bei Graphen zwischen drei und zehn Knoten.
Die asymptotische Komplexität liegt bei $O(q_{\min} \cdot (|V|+|E|))$.

7.2.2 Neue Erreichbarkeits-Bedingung

In den folgenden Abschnitten werden die Besonderheiten der erreichbaren Zustände beschrieben, die sich während der Analyse bemerkbar machten. Als Vorgriff wird hier aber nun die Theorie vorgestellt, die für die Erreichbarkeit eine notwendige und hinreichende Bedingung darstellt und daher festlegt, welche Zustände durch legales Retiming angenommen werden können. Diese Bedingung basiert auf der Invarianz der gewichteten Delaysumme in jeder Schleife (notwendige Bedingung) und einer Zusatzbedingung, die die gewichtete Delaysumme auf Pfaden und die Raten der Endknoten betrifft.

Es wird im folgenden angenommen, daß die Graphen normalisiert wurden. Dies kann mit der Einheits-Verstärkungs-Transformation aus dem Kapitel „Transformationen“ durchgeführt werden. Die Gewichtsfaktoren $y(e)$ ($\vec{y}^{\mathbf{T}}$) für die WST sind dann alle Eins. Die Auswirkungen der Normierung lassen sich auch in die folgenden formelmäßigen Ausdrücke integrieren, würden aber die Anschaulichkeit verhindern.

Satz: In einem Multiraten-Datenflußgraphen ist der Zustand $\vec{d}_r$ genau dann erreichbar von $\vec{d}_0$, wenn

$$C \cdot \vec{d}_R = C \cdot \vec{d}_0 \qquad \text{und} \tag{7.13}$$

$$\Delta d(p) \bmod \mu(p) = 0 \qquad \forall\ p \subset G \tag{7.14}$$

Dabei ist C die ‘fundamental circuit’-Matrix, $\Delta d(p)=d_R(p)-d_0(p)$ die Differenz der Delaysummen auf dem Pfad p und $\mu(p)$ der größte gemeinsame Teiler (ggT) der Raten am Anfangs- und Endknoten des Pfades. C besteht aufgrund der Normierung nur aus Nullen und Einsen und entspricht daher der C-Matrix des Topologie-äquivalenten Einheitsraten-Graphen (bei diesem sind alle Raten durch Eins ersetzt).

Eine alternative Formulierung benutzt die Inzidenzmatrix Γ_E des Topologie-äquivalenten Einheitsraten-Graphen (Γ_E besteht nur aus Einträgen 0,1.-1):

Satz: In einem Multiraten-Datenflußgraphen ist der Zustand $\vec{d}_r$ genau dann erreichbar von $\vec{d}_0$, wenn

$$\vec{d}_r - \vec{d}_0 = \Gamma_E \cdot \vec{r}_E \tag{7.15}$$

eine Lösung für $\vec{r}_E$ hat und

$$\Delta d(p) \bmod \mu(p) = 0 \qquad \forall\ p \subset G \tag{7.16}$$

Die beiden Sätze [71] zeigen, daß die Erreichbarkeits-Bedingung für Multiraten-Graphen aus zwei Teilen zusammengesetzt ist: Der erste Teil ist die Erreichbarkeits-Bedingung für den topologisch äquivalenten Einheitsraten-Graphen (diese Bedingungen sind im Abschnitt „Einheitsraten-Retiming“ erläutert worden). Der in beiden Sätzen gleiche zweite Teil beschränkt die Summe der Delays in jedem Pfad auf diskrete ganzzahlige Werte.

Der Einheitsraten-Fall ist offensichtlich enthalten, denn wenn alle Raten Eins sind ergibt sich $\mu(p)$ immer zu Eins und die zweite Bedingung ist automatisch erfüllt (Ganzzahligkeit der Delays natürlich vorausgesetzt). Dieselbe Vereinfachung tritt jedoch auch auf, wenn $\mu(p)$ aus anderen Gründen immer Eins ist; nämlich dann, wenn der ggT aller Paare von Knotenraten[4] jeweils Eins ist. Dies bedeutet, daß alle Raten prim untereinander sind und keinen Primfaktor gemeinsam haben. In diesem Fall spricht man von coprimen Raten.

Die zweite Bedingung bedarf noch genauerer Erläuterung. Für diese Bedingung ist jeder gerichtete einfache Pfad zwischen allen Paaren von Knoten zu betrachten. Die Rate des Anfangsknotens v_i des k.ten Pfades $p_{ij}^{(k)}$ (von v_i nach v_j) sei m_i und die des Endknotens v_j sei m_j. Der größte gemeinsame Teiler (ggT {gcd}) und damit größter gemeinsamer Primfaktor der Raten m_i und m_j sei

$$\mu(p_{ij}^{(k)})=\mu_{ij}=\text{ggT}(m_i\,,\,m_j) \neq f(k) \tag{7.17}$$

Im normierten Ursprungsgraph G_0 sei $d_0(p_{ij}^{(k)})$ die Summe der Delays auf dem Pfad $p_{ij}^{(k)}$, im neuen Graph G_R sei $d_R(p_{ij}^{(k)})$ die Summe der Delays auf dem Pfad $p_{ij}^{(k)}$. Die Bedingung verlangt nun, daß sich die Reste des Quotienten (Summe der Delays)/(μ_{ij}) nicht unterscheiden:

$$\begin{aligned} & d_0(p_{ij}^{(k)}) \;\mathbf{mod}\; \mu_{ij} \;=\; d_R(p_{ij}^{(k)}) \;\mathbf{mod}\; \mu_{ij} \quad \forall\, i,j \in \{1;..;n\} \;\; i\neq j \,,\, \forall\, k(i,j) \\ \Leftrightarrow\; & (\, d_0(p_{ij}^{(k)}) - d_R(p_{ij}^{(k)}) \,) \;\mathbf{mod}\; \mu_{ij} = 0 \quad \forall\, i,j \in \{1;..;n\} \;\; i\neq j \,,\, \forall\, k(i,j) \\ \Leftrightarrow\; & (\, \Delta d\,(p_{ij}^{(k)}) \,) \;\mathbf{mod}\; \mu_{ij} = 0 \qquad \forall\, i,j \in \{1;..;n\} \;\; i\neq j \,,\, \forall\, k(i,j) \end{aligned} \tag{7.18}$$

Beweis: Es sei $\Gamma_E = \Gamma \cdot \underline{\underline{M}}^{-1}$ **(**$\Gamma = \Gamma_E \cdot \underline{\underline{M}}$**)** und $\vec{r}_E = \underline{\underline{M}} \cdot \vec{r}$ **(**$\vec{r} = \underline{\underline{M}}^{-1} \cdot \vec{r}_E$**)** mit der Diagonalmatrix

$$\underline{\underline{M}} = \mathbf{diag}(m_i, \ldots, m_n) = \begin{pmatrix} m_i & 0 \cdots & 0 \\ 0 & \ddots & \vdots \\ \vdots & & 0 \\ 0 & \cdots 0 & m_n \end{pmatrix} \tag{7.19}$$

Die Zustandsgleichung für G kann dann so transformiert werden, daß sie für den topologisch äquivalenten Einheitsraten-Graph G_E gilt:

$$\begin{aligned} & \Gamma \cdot \vec{r} \;=\; \vec{d}_R - \vec{d}_0 \;=\; \Delta\vec{d} \\ \Leftrightarrow\; & \Gamma \cdot \underline{\underline{M}}^{-1} \cdot \underline{\underline{M}} \cdot \vec{r} \;=\; \Delta\vec{d} \\ \Leftrightarrow\; & \Gamma_E \cdot \vec{r}_E \;=\; \Delta\vec{d} \end{aligned} \tag{7.20}$$

Die Inzidenz-Matrix Γ_E ist total unimodular, da sie nur die Struktur des Graphen mit Einheitsraten beschreibt (Netzwerk-Matrizen sind unimodular [61],[63]). Wegen der

[4] Durch die Normierung hat jede Kante an einem Knoten dieselbe Ein- bzw. Ausgangsrate, die Knotenrate.

Unimodularität kann immer wenn eine Lösung existiert (erste Bedingung im Satz) eine ganzzahlige Lösung $\vec{r}_E$ gefunden werden [63],[68] (Retiming im Einheitsraten-Graph). Diese Lösung ist nicht eindeutig, denn ein u-faches ($u \in \mathbf{Z}$) der homogenen Lösung $\vec{q}_E$ (Einsvektor) kann immer hinzuaddiert werden und führt zu einer anderen ganzzahligen Lösung. Daraus ergib sich die Lösung im Multiratenfall:

$$\vec{r} = \underline{\underline{M}}^{-1} \cdot (\vec{r}_E + u \cdot \vec{1}) \tag{7.21}$$

Diese ist i.a. nicht ganzzahlig. Das „Verallgemeinerte Chinesische Restwert-Theorem“ {Generalized Chinese Remainder Theorem [69],[65]} garantiert eine ganzzahlige Lösung genau dann, wenn

$$(r_{E\,i} - r_{E\,j}) \ \mathbf{mod}\ \mathbf{ggT}(m_i, m_j) = 0 \quad \forall\, i,j \in \{1;..;n\} \ \ i \neq j \tag{7.22}$$

$$\text{mit } (r_{E\,i} - r_{E\,j}) = \Delta d(p_{ij}^{(k)}) \ \forall\, k \text{ folgt}$$

$$\Leftrightarrow \Delta d(p_{ij}^{(k)}) \ \mathbf{mod}\ \mathbf{ggT}(m_i, m_j) = 0 \quad \forall\, i,j \in \{1;..;n\} \ \ i \neq j, \forall\, k$$

Dabei ist $r_{E\,i}$ die Komponente des Vektors $\vec{r}_E$ für Knoten v_i. Wegen der Symmetrie brauchen nicht alle Kombinationen von i und j getestet zu werden; es reichen die Werte $1 \le i < j \le n$ aus (Werte der $\Xi = (\mu_{ij})$ Matrix oberhalb der Diagonalen).

Erklärungen:

(a) Das Verallgemeinerte Chinesische Restwert-Theorem [69] stellt fest, daß eine Zahl u, die

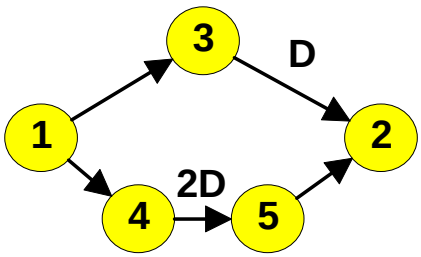
$$u = r_{E\,j} \ \mathbf{mod}\ m_j \quad \forall\, i \text{ mit } 1 \le i \le n \tag{7.23}$$

erfüllt, genau dann existiert, wenn

$$r_{E\,i} \ \mathbf{mod}\ \mathbf{ggT}(m_i, m_j) = r_{E\,j} \ \mathbf{mod}\ \mathbf{ggT}(m_i, m_j) \quad \forall\, i,j \text{ mit } 1 \le i < j \le n$$

$$\Rightarrow \quad (r_{E\,i} - r_{E\,j}) \ \mathbf{mod}\ \mathbf{ggT}(m_i, m_j) = 0 \quad \forall\, i,j \in \{1;..;n\} \ \ i \neq j$$

(b) Eine Vorstellung von der Aussage mit $(r_{E\,i} - r_{E\,j}) = \Delta d(p_{ij}^{(k)})$ gewinnt man, wenn man die Pfade zwischen zwei Knoten betrachtet und sich erinnert, daß die Werte $r_{E\,i}$ angeben, wie oft der Knoten v_i aktiviert werden muß um zur neuen Delayverteilung zu gelangen. Im Bild wird erkennbar, daß eine $r_{E\,1}$-fache Aktivierung des Eingangsknotens v_1 jedem Pfad genau $r_{E\,i}$ Delays hinzufügt. Aktivierungen des Ausgangsknotens v_2 bewirken eine entsprechende Abnahme. Die Aktivierung irgendwelcher anderer Knoten, innerhalb oder außerhalb der Pfade, bewirkt keine Änderung der Summe der Delays auf den Pfaden.



(c) Die Bestimmung aller Pfade ist mit einem Aufwand $O(|E|^2)$ anzusetzen (eine Schleife enthält beispielsweise $|E|*(|E|-1)$ unterschiedliche Pfade). Die hier betrachteten einfachen

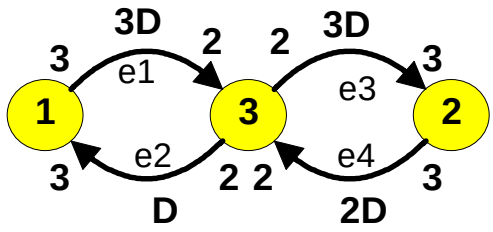
Pfade dürfen sich nicht selbst überkreuzen, also eine Kante oder einen Knoten mehrmals enthalten. Dadurch wird auch der Aufwand zur Bestimmung reduziert. Ein Algorithmus dazu beginnt jeweils mit einem anderen Startknoten und ermittelt von dort aus alle Pfade, indem zunächst eine Kante, dann die nachfolgenden Kanten jeweils zu einer Liste von Pfaden hinzugefügt werden. Die Rekursion endet, wenn ein schon besuchter Knoten als nächstes erreicht werden würde.

(d) Die Modulo-Bedingungen sind nur für einen der parallelen Pfade zu testen, wenn die erste Bedingung in den Sätzen erfüllt ist. Diese garantiert nämlich, daß sich die Delaysumme in allen Schleifen nicht verändert (auch in ungerichteten). Daher sind, wenn ein Pfad korrekt ist, auch alle parallelen Pfade korrekt.

Beispiel:

Zu einem gegebenen Graphen, der aus zwei Schleifen besteht (Bild), werden zwei Neuverteilungen der Delays ($\vec{d}_1$, $\vec{d}_2$) bestimmt, die die Bedingung der Konstanz der (gewichteten) Delaysumme erfüllen (erste Bedingung des Satzes). Die Modulo-Bedingung wird aber nur von einer der neuen Zustandsvektoren erfüllt. Nur die dazu gehörige Verteilung der Delays ist durch legales Retiming aus der Anfangsverteilung erreichbar. Gegeben sind die Daten des Graphen und eine Tabelle, die erkennen läßt, daß die erste Verteilung dir Modulo-Bedingung erfüllt und die zweite nicht.

$$\Gamma = \begin{pmatrix} 3 & 0 & -2 \\ -3 & 0 & 2 \\ 0 & -3 & 2 \\ 0 & 3 & -2 \end{pmatrix} \quad \vec{d}_0 = \begin{pmatrix} 3 \\ 1 \\ 3 \\ 2 \end{pmatrix}$$



$$C = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad \vec{WST} = \begin{pmatrix} 4 \\ 5 \end{pmatrix} \quad \vec{q} = \begin{pmatrix} 2 \\ 3 \\ 2 \end{pmatrix}$$

erste Verteilung: $\vec{d}_1 = \begin{pmatrix} 4 \\ 0 \\ 2 \\ 3 \end{pmatrix} \quad \Delta\vec{d}_1 = \begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix}$,

zweite: $\vec{d}_2 = \begin{pmatrix} 4 \\ 0 \\ 0 \\ 5 \end{pmatrix} \quad \Delta\vec{d}_1 = \begin{pmatrix} 1 \\ -1 \\ -3 \\ 3 \end{pmatrix}$

	ggT(m_i , m_j)			$\Delta d_1(p_{ij})$			$\Delta d_2(p_{ij})$		
i \ j	1	2	3	1	2	3	1	2	3
1	3	3	1	0	0	1	0	-2	1
2	3	3	1	0	0	1	2	0	3
3	1	1	2	-1	-1	0	-1	-3	0

Der topologisch äquivalente Einheitsraten-Graph wird durch diese Matrizen beschrieben:

$$M = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{pmatrix} \quad \Gamma_E = \Gamma \cdot M^{-1} = \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

Dieser erlaubt zwei ganzzahlige Lösungen für $\Gamma_E \cdot \vec{r}_E = \Delta\vec{d}$:

$$\vec{r}_{E1}=(1, 1, 0)^T \text{ und } \vec{r}_{E2}=(1, 3, 0)^T.$$

Die Berechnung des Multiraten-Retiming-Vektors gemäß $\vec{r} = \underline{\underline{M}}^{-1} \cdot (\vec{r}_E + u \cdot \vec{1})$ offenbart, daß nur der erste Vektor $\vec{r}_{E1}$ mit u=2 ganzzahlig gewonnen werden kann. Dies erklärt sich gemäß der Theorie daraus, daß die Modulo-Bedingung im zweiten Fall nicht eingehalten wurde. Das Ergebnis ist

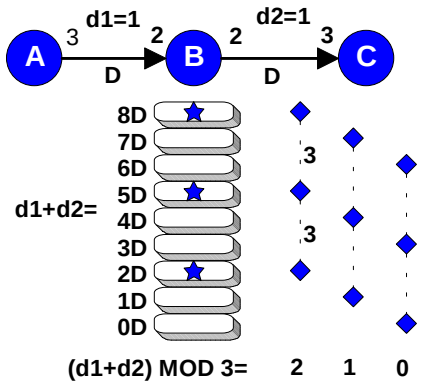
$$\vec{r}_1=(1, 1, 1)^T.$$

Konsequenzen:

Die Erfüllung der Modulo-Bedingungen hat sich als Kriterium für die Gültigkeit des Multiraten-Retiming bzw. für die Erreichbarkeit des Zielzustands herausgestellt. Die Optimierung der Delayverteilung durch lineare Programmierung ist daher nicht wie im Einheitsratenfall durchführbar. Alternativen werden in später folgenden Abschnitten diskutiert. Die Interpretation der Besonderheiten, die durch die Modulo-Bedingung entstehen, ist im folgenden Abschnitt beschrieben.

7.3 Unbewegliche Delays

Unbewegliche Delays {non-movable delays = NMD's} sind Delays, die (bezüglich ihrer Häufung {marking} und nicht bezogen auf den Dateninhalt des Samples {token}) durch kein Retiming von ihrem Platz verschoben werden können. Das bedeutet unter anderem, daß die gewichtete Summe der Delays (WST) auf dieser Kantenmenge nicht Null werden kann. Es gibt Positionen, an denen sich NMD's potentiell befinden können, und solche, an denen kein NMD auftreten kann. Diese Positionen können mit einer Ratenbedingung ermittelt werden. Ob tatsächlich dort NMD's auftreten, muß anhand der Anfangs-Delayverteilung entschieden werden. Natürlich kann auch an keiner Stelle durch legales Retiming ein NMD auftreten, wo vorher keines vorhanden war.



Die Modulo-Bedingung der Sätze aus Kapitel 7.2 erzwingt, daß die (gewichtete) Delaysumme auf jedem Pfad MODULO einer Konstanten sich nicht verändert. Für den rechts im Bild gezeigten Beispiel-Pfad, dessen Modulo-Konstante μ_{AC}=3 ist, gibt es nur drei verschiedene Möglichkeiten die Delaysumme zu verteilen: So daß der Rest 0, 1 oder 2 ist. Im Abstand 3 wiederholen sich dann die erlaubten Delaysummen.

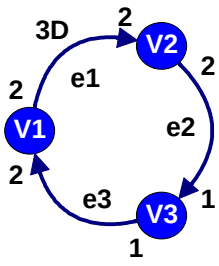
In Schleifen oder Graphen kann man drei Fälle unterscheiden:

7.3.1 Kanten-NMD's (lokale NMD's, arc-NMD's)

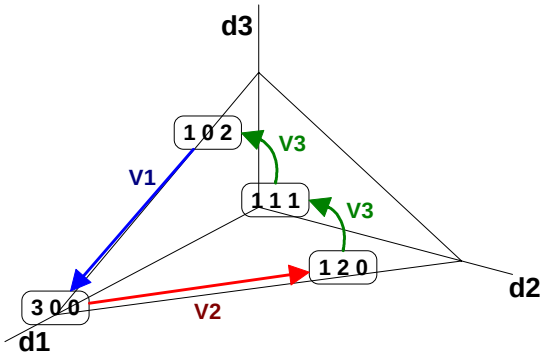
Kanten-NMD's sind auf einer Kante e unbeweglich, d.h. $d(e)>0$ für alle gültigen Retiming-Transformationen.

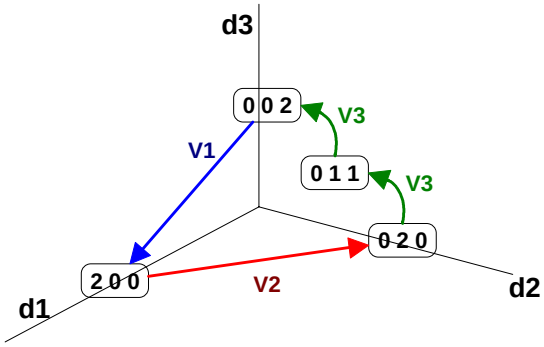
Sie können dann und nur dann existieren, wenn ggT$\{I(e),O(e)\}$ >1 (Ratenbedingung) ist. Ein lokales NMD existiert genau dann auf Kante e, wenn gilt

$$s = \mathbf{ggT}\{I(e),O(e)\} >1 \quad \text{und} \quad [d(e) \ \mathbf{mod}\ s] >0 \tag{7.24}$$



Die Delays sind in diesem Fall nicht auf jeder Kante sammelbar. Die Entfernung der Delays auf dieser Kante, die zu dem Modulo-Rest führen (Kanten-NMD's), verändert nicht die Anzahl der erreichbaren Zustände und damit die Lebendigkeit. Zum Zweck des Retimings können diese NMD's entfernt und später wieder an gleicher Position hinzugefügt werden. Im Erreichbarkeitsgraphen äußert sich ein Kanten-NMD auf Kante e durch eine "Lücke" zwischen dem Zustandsgitternetz und der Ursprungsebene senkrecht zum Einheitsvektor in e-Richtung (Koordinaten-Wand). Die Entfernung dieser NMD's verschiebt das Gitter entgegen der e-Richtung. Ein Beispiel ist im Bild rechts zu sehen: Auf Kante e_1 bleibt immer ein Delay zurück. Dieses kann für Retiming vorübergehend entfernt werden und verursacht eine Verschiebung des Erreichbarkeitsgraphen um eine Einheit in $-d_1$-Richtung.
Die Bilder unten zeigen die Erreichbarkeitsgraphen vor und nach Entfernen des NMD's:





7.3.2 Pfad NMD's {Path-NMD}

Diese sind auf einem Pfad (mit mindestens zwei Kanten) gefangen, d.h. $d(e)=0$ ist für jede einzelne Kante $e \in Pfad$ möglich, aber für alle Zustände nach Retiming gilt

$$\sum_{e \in Pfad} y_e d_e >0. \tag{7.25}$$

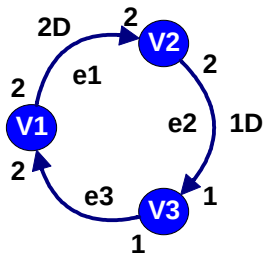
Zur Ermittlung von Pfad-NMD's muß man die Modulo-(Raten-)bedingung für jeden Teilpfad des zu untersuchenden Pfades (mit N Kanten) prüfen (insgesamt $N(N+1)/2$ Pfade). Will man feststellen, ob in einer Schleife NMD's sind, dann muß man die Ratenbedingung in allen möglichen Pfaden außer der Schleife selbst prüfen (insgesamt $N(N-1)$ Pfade).
Die Ratenbedingung heißt bei Normschleifen und -graphen:

$$s = \mathbf{ggT}\{I(p),O(p)\} >1 \quad \text{und} \quad [d_{sum}(p) \ \mathbf{mod}\ s]>0, \tag{7.26}$$

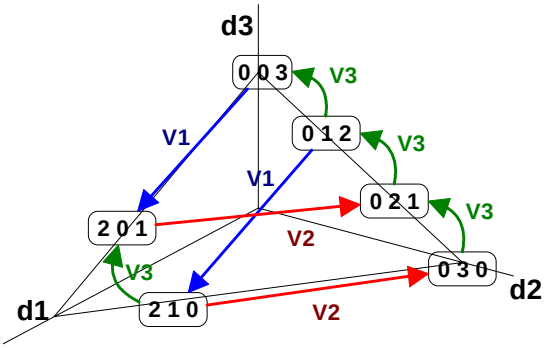
wobei $d_{sum}(p)=WST(p)$ die Delaysumme auf dem Pfad, I(p) die Eingangsrate des Endknotens und O(p) die Ausgangsrate des Anfangsknotens auf dem Pfad ist.
Bei unnormierten Schleifen läßt sich die Normierung in die Bedingung einfügen (man beachte, daß $WST'=WST$ invariant ist bei Normierung, siehe Kap. „Transformationen“):

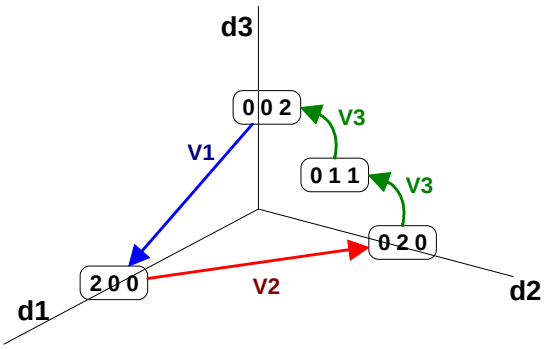
$$s = \mathbf{ggT}\{I(p)\cdot w(e_{End}),\ O(p)\cdot w(e_{Start})\} >1 \quad \text{und} \quad [WST\ \mathbf{mod}\ s] >0 \tag{7.27}$$

Die Delays sind bei Existenz dieser NMD's nicht auf jeder Kante sammelbar, nämlich nicht auf der Kante $e \in Loop$ mit $e \notin Path$, wobei $Path \subset Loop$. Das NMD verursacht im Erreichbarkeitsraum (Zustandsraum) neue Zustände, da z.B. bei Bewegung vor oder hinter einen Knoten des Pfades zwischen zwei Zuständen gewechselt wird. Die Entfernung dieser NMD’s („Modulo-Delays“) verringert nur die Anzahl der Zustände, kann aber in besonderen Fällen zu Deadlocks führen, da manche Pfad-NMD's zur Überschreitung der Aktivierungsschwelle (*LLM*) benötigt werden (diese wurden „Hocker“ genannt, da sie wie eine Art Steighilfe wirken, damit nachfolgende Delays die Aktivierungsschwelle {threshold} überschreiten können und der nachfolgende Knoten aktiviert werden kann).



Der Beispielgraph rechts im Bild hat ein Pfad-NMD mit Gewicht Eins auf dem Pfad $\{e_2;e_3\}$. Dieses kann nicht durch Retiming auf die Kante e_1 gebracht werden, um alle Delays dort zu konzentrieren. Die Entfernung dieses NMD’s bewirkt zunächst eine Verschiebung des Erreichbarkeits-Raums in $-d_2$-Richtung. Dadurch werden zwei frühere Zustände ungültig, da für diese $d_2<0$ wird. Der restliche Erreichbarkeitsgraph enthält noch eine Schleife, daher ist der Graph weiterhin lebendig. Die unteren Bilder zeigen diese Graphen vor und nach Entfernung:





Das nächste Beispiel im Bild rechts zeigt eine Schleife, die wieder auf dem Pfad $\{e_2;e_3\}$ ein NMD enthält. Dieses ist allerdings für die Lebendigkeit von Bedeutung, denn im Zustand $\vec{d}=(0,3,1)^T$ reicht eine Aktivierung von Knoten v_3 aus, um zusammen mit dem auf Kante e_3 befindlichen (NMD-) Delay die zur Aktivierung von Knoten v_1 benötigten drei Delays bereitzustellen. Im Erreichbarkeitsgraph würde die Entfernung des NMD-Delays zum Verschwinden der Zustände (0,0,4) und (3,0,1) führen. Es wäre dann keine Schleife mehr vorhanden, d.h. es existierte kein lebendiges Schedul.

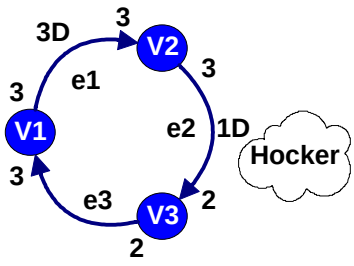
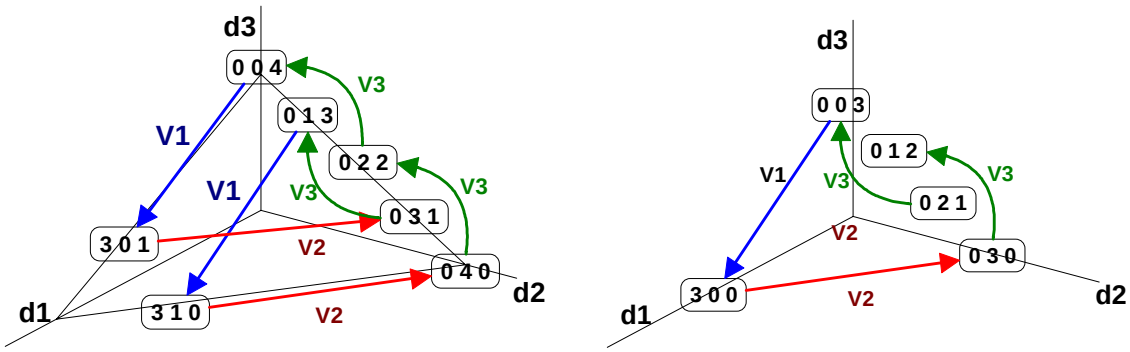


Bild unten: links Original-Erreichbarkeitsgraph, rechts: nach Entfernen des Pfad-NMD’s



7.3.3 Keine NMD's vorhanden

In diesem Fall sind alle Delays eines Graphen mit nur einer Schleife auf jeder Kante $e \in Loop$ sammelbar, es ist also möglich, daß

$$d_i = \delta_{i,e} \cdot n_e \;\forall\; i \in \text{Loop}. \tag{7.28}$$

Ein Problem der linearen Optimierung (LP) kann im Zustandsraum gelöst werden und ergibt einen gültigen Lösungsvektor aus natürlichen Zahlen, es muß also nicht das speziellere (und wesentlich aufwendigere) ILP-Problem gelöst werden.
Eine notwendige Bedingung für die NMD-Freiheit einer Schleife ist:

$$WST = K \cdot \underset{e \in Loop}{\mathbf{kgV}}\left(y_e\right) \quad , \quad K \in \mathbf{N} \;\Leftarrow\; \text{keine NMD's.} \tag{7.29}$$

Dies ist darin begründet, daß auf jeder Kante e (WST **mod** y_e)=0 gelten muß, um die Delays dort zu konzentrieren.

Satz: Es sind keine NMD's auf einer einzelnen Schleife L vorhanden $\Leftrightarrow$ die Delays sind auf jeder Kante sammelbar.

Beweis:

$\Rightarrow$ per Definition sind keine Pfad-NMD's vorhanden. Betrachte (für alle Kanten e_i) alle Pfade $p_i = L \backslash e_i \;\; \forall \; e_i \in L$. Dort sind keine NMD's vorhanden. In diesen Fällen p_i müssen sich alle Delays komplett auf den Kanten e_i befinden (sammelbar).

$\Leftarrow$ Betrachte alle Kanten $e_i \in L$. Dort seien jeweils die Delays konzentriert. Auf dem Pfad $p_i = L \backslash e_i$ kann daher kein NMD sein. Auf jedem Teilpfad $p_{ij} \subset p_i$ kann ebenfalls kein NMD sein. Daher existieren überhaupt keine NMD's.

7.4 Retiming von Schleifen

Retiming als Veränderung der Delayverteilung vom Ausgangszustand $\vec{d}_0$ aus entspricht der Aktivierung von Knoten (entsprechend der Komponenten von $\vec{r}$) solange, bis sich die gewünschte Delayverteilung $\vec{d}_R$ einstellt. Daher gilt $\vec{d}_R = \Gamma\vec{r} + \vec{d}_0 \;\; \Leftrightarrow \;\; \Delta\vec{d} = \Gamma\vec{r}$
Daraus folgt durch Multiplikation von links mit der Circuit-Matrix C die bekannte Bedingung:

$$\Rightarrow \; \underline{\underline{C}} \cdot \Delta\vec{d} = \underline{\underline{C}} \cdot \Gamma\vec{r} = \underline{\underline{0}} \cdot \vec{r} = \vec{0} \tag{7.30}$$

Die notwendige Bedingung für ein legales Retiming ist also die Konstanz der gewichteten Delaysumme. Gesucht wird aber auch eine hinreichende Bedingung, um bei vorgegebenem $\vec{d}_R$ die Gültigkeit zu prüfen. Diese liefert folgender

Satz: $G \xleftrightarrow{R} G'$ ist legales Retiming einer Schleife genau dann, wenn $\vec{y}^T \Delta\vec{d} = \vec{0}$, $d_i \geq 0$ ist, sofern keine NMD's in *G* und *G'* vorhanden sind und die Graphen lebendig sind.

Beweis: Wegen $\vec{y}^T \Delta\vec{d} = \vec{0}$ ist die gewichtete Delaysumme in *G* und *G'* konstant gleich *WST*. Da keine NMD's vorhanden sind, lassen sich alle Delays auf einer beliebigen Kante ansammeln. Auf dieser Kante *e* ist für *G* $d(e)=WST/y(e)$, ebenso wie für *G'* $d(e)=WST/y(e)$ ist. Für beide Graphen ist also derselbe Zustand $\vec{d}$ erreichbar. Wenn beide *einen* erreichbaren Zustand gemeinsam haben, dann haben sie *alle* Zustände gemeinsam; *G* und *G'* sind also durch ein legales Retiming überführbar.

Lemma: Zwei lebendige Graphen sind durch Retiming überführbar ⇔ Die zugehörigen Anfangszustände $\vec{d}_R$ und $\vec{d}_0$ liegen auf demselben Zustands-(Erreichbarkeits-)graph *RG*. Der Zustandsraum enthält alle $\vec{d}$, die von $\vec{d}_0$ aus durch Aktivierung von ausführbaren Knoten erreicht werden können (und umgekehrt bei lebendigen Graphen). Eines dieser $\vec{d}$ muß gleich $\vec{d}_R$ sein, da jeder durch Retiming gewonnene Zustand auch durch geeignete Aktivierungen erreicht werden kann. Da von $\vec{d}_R$ aus auch jeder Zustand $\vec{d}$ erreicht werden kann, reicht ein gemeinsamer Zustand $\vec{d}$ aus, damit $\vec{d}_R$ und $\vec{d}_0$ auf demselben Erreichbarkeitsgraphen liegen. Damit haben diese alle erreichbaren Zustände gemeinsam.

Diese Aussagen sind ein Spezialfall der allgemeinen Bedingung aus Kapitel 7.2.

7.5 Retiming von azyklischen Graphen

Für das legale Retiming azyklischer Graphen gelten dieselben Bedingungen aus Kapitel 7.2 wie für zyklische Graphen. Die ‘fundamental circuit’-Matrix C enthält für azyklische Graphen allerdings keine Zeile oder nur Zeilen, die mindestens einen negativen Eintrag haben (ungerichtete Schleife). Betrachtet man diese Matrix, dann ist die notwendige Bedingung für legales Retiming weiterhin

$$C\,\vec{d} = C\,\vec{d}_0. \tag{7.31}$$

Ist die Matrix leer, d.h. existiert keine ungerichtete Schleife, dann ist die Existenz eines rationalen $\vec{r}$-Vektors in jedem Fall gesichert. Die Modulo-Bedingungen garantieren dann darüber hinaus die Ganzzahligkeit dieses Vektors.

7.6 Retiming von zyklischen Graphen

Die Untersuchungen zum Retiming von allgemeinen Datenflußgraphen, die i.a. mehr als eine Schleife enthalten, müssen anders als bei Einzelschleifen durchgeführt werden, da sich einige Eigenschaften verändern, wenn Schleifen miteinander gekoppelt sind. Das Konzept der unbeweglichen Delays wird insofern übernommen, daß die Ratenbedingung weiterhin das

Kriterium für Existenz oder Nichtvorhandensein von NMD's darstellt. Die Sammelbarkeit von Delays auf Kanten ist jedoch bei NMD-Freiheit nicht ohne weiteres gegeben. Im Falle gekoppelter Schleifen müssen immer die Bedingungen $WST = \vec{y}^T \cdot \vec{d}$ für jede Schleife gelten. Dadurch werden in einigen Fällen Rest-Delays auf Kanten erzwungen, obwohl sie gemäß der Ratenbedingung keine NMD's sind. Diese sollen im Folgenden nicht als NMD's bezeichnet werden. Zu den Eigenschaften betrachte man folgenden Satz:

Satz: $G \overset{R}{\longleftrightarrow} G'$ ist legales Retiming eines Graphen genau dann, wenn $\underline{\underline{C}} \cdot \Delta\vec{d} = \vec{0}$, $d_i \geq 0$ ist, sofern keine NMD's in G und G' vorhanden sind und die Graphen lebendig sind.

Beweis: Wenn keine NMD's in G und G' vorhanden sind bedeutet dies, daß in beiden Graphen

$$d(p_{ij}) \textbf{ mod } \mu_{ij} = 0 \quad \text{mit} \quad \mu_{ij} = \textbf{ggT}(m_i , m_j) \tag{7.32}$$

Damit sind die Modulo-Bedingungen des Satzes über die Erreichbarkeit aus Kapitel 7.2 erfüllt und mit der notwendigen Bedingung ergibt sich ein legales Retiming.

7.7 Retiming durch Lineare Optimierung

Das Problem eine lineare Funktion über $\vec{d}$ zu maximieren oder minimieren wird im Einheitsratenfall mit linearer Programmierung (LP) gelöst. Im Multiratenfall führt dieses Vorgehen im Allgemeinen nicht zu legalem Retiming, denn LP garantiert nur die Einhaltung der notwendigen Bedingung der Konstanz der gewichteten Delaysumme. Die LP-Formulierung

$$\textbf{max } \vec{c} \cdot \vec{d} \quad \text{s.t. } C \cdot \vec{d} = C \cdot \vec{d}_0 \tag{7.33}$$

oder die äquivalente Formulierung

$$\textbf{max } \vec{c} \cdot \vec{d} \quad \text{s.t. } \vec{d} - \Gamma \cdot \vec{r} = \vec{d}_0 \tag{7.34}$$

liefern als Lösung häufig keinen Zustand $\vec{d}$, der auch die Modulo-Bedingung erfüllt. Nur für einzelne Schleifen, die keine NMD's enthalten, kann ein legales Retiming garantiert werden, da die lineare Programmierung eine Lösung in den Eckpunkten des Polyhedrons findet. Das Polyhedron wird gebildet durch die Ebene im Zustandsraum, in der sich die erreichbaren Zustände befinden. Bei NMD-Freiheit befinden sich in allen Eckpunkten erreichbare Zustände.

Die Bedingungen $d_i \geq 0$ und $r_i \geq 0$ werden implizit angenommen und automatisch durch jeden Lösungsalgorithmus erfüllt.

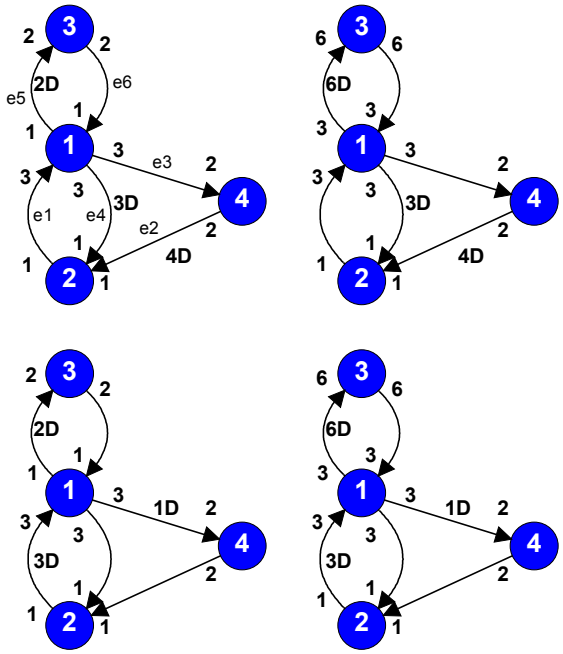
Im allgemeinen Fall ist der nach einer Optimierungsaufgabe erhaltene Graph nicht, wie bei einfachen Schleifen, frei von NMD's, wenn der Ursprungsgraph NMD-frei war, sondern die lineare Programmierung kann u.U. zu Delayverteilungen führen, die nicht vom Anfangszustand erreichbar sind. Der Graph im folgenden Beispiel ergab sich bei der automatischen Suche mit Zufallsgraphen ohne NMD's als einziger unter 5000 Graphen, der nach der Optimierung NMD's aufwies und dessen Zustand daher nicht vom Ursprungsgraph aus erreichbar war.

Es bestätigt sich, daß die NMD-Freiheit nicht legales Retiming durch lineare Programmierung erlaubt, obwohl die Wahrscheinlichkeit für einen unzulässigen Zustand relativ gering ist.

Beispiel im Bild rechts: Oben ist der Ursprungsgraph (enthält keine NMD's) abgebildet, links im Original, rechts der normierte Graph.

Unten sieht man den Graphen, den man durch lineare Optimierung der Delayverteilung erhält. Dabei war die Zielfunktion gegeben durch:

Max (100, 2, 4, 7, 100, 5)*$\vec{d}_r$,
wobei $C^*(\vec{d}_r - \vec{d}_0)=0$

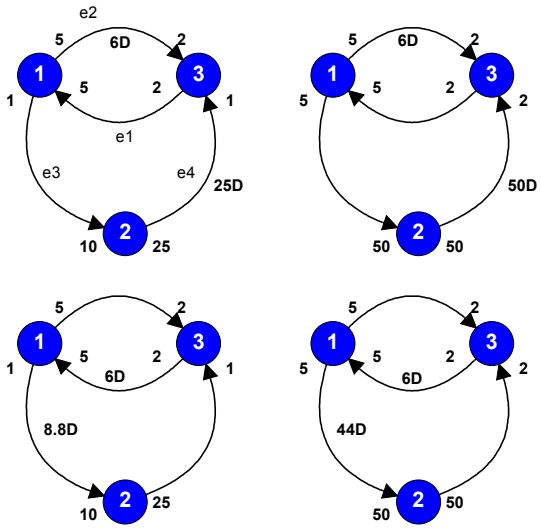


Der neu erhaltene Graph besitzt nun NMD's auf den Pfaden (e_3-e_6) und (e_1-e_2-e_5). Der neue Zustand ist daher nicht vom Ursprungszustand aus erreichbar.

Es gibt Graphen mit einfacherer Topologie als dieses zufällig gefundene Beispiel, die trotz-

dem solche Probleme aufweisen. Sie lassen sich sogar konstruieren. Ein sehr einfaches Beispiel ist schon im Kapitel 7.2 vorgestellt worden.

Es kann weiterhin passieren, daß die Optimierung durch lineare Programmierung keine ganzzahlige Delayverteilung findet. Im normierten Graphen (Einheitsverstärkung) kann dies nicht passieren, allerdings ist die Rücktransformation dann nicht ganzzahlig möglich. Dieser Fall wird aber ausgeschlossen, wenn die Neuverteilung als ungültig bezeichnet wird, weil sie NMD's enthält. Im unnormierten Fall muß man dann nicht-ganzzahlige Delays als NMD's qualifizieren (es gilt $[d \bmod s]>0$ für $d \notin \mathbf{Z}$ und $s \in \mathbf{Z}$).



Beispiel im Bild rechts: Oben ist der Ursprungsgraph (ohne NMD's), links unnormiert, rechts normiert.
Die Optimierung wurde in beiden Fällen mit folgender Zielfunktion durchgeführt:
Max (100,1,10,8)*$\vec{d}_r$.

Bemerkungen:

Die lineare Optimierungsaufgabe

$$\mathbf{max}\ \vec{c} \cdot \vec{d} \quad \text{s.t.}\ \vec{d} - \Gamma \cdot \vec{r} = \vec{d}_0$$

liefert im Multiratenfall allgemein nicht-ganzzahlige Lösungen für die Komponenten von $\vec{d}$ und $\vec{r}$. Im Fall normierter Graphen ist eine ganzzahlige Lösung für $\vec{d}$ garantiert (nicht jedoch ein legales Retiming). Die Ganzzahligkeit ist gesichert, weil das LP-Problem dem des topologisch äquivalenten Einheitsraten-Graphen entspricht (bis auf eine Skalierung von $\vec{r}$). Für Graphen mit coprimen Raten ist die erhaltene Lösung für $\vec{d}$ legal, da keine Modulo-Bedingungen einzuhalten sind. Der Vektor $\vec{r}$ wird sich aber allgemein nicht-ganzzahlig ergeben. Dieser Vektor bestimmt durch $\Gamma^*\vec{r}$ die kürzeste Verbindung zwischen Ursprungszustand $\vec{d}_0$ und neuem Zustand $\vec{d}_r$. Durch Addition eines Bruchteils von $\vec{q}$ kann der Retimingvektor $\vec{r}$ aber ganzzahlig gemacht werden, was der realen summarischen Aktivierungssequenz entspricht, die zur Überführung notwendig ist.

In der obigen Optimierungsaufgabe kann man die Komponenten von $\vec{d}$ als Slack-Variablen auffassen, die die LP-Aufgabe

$$\mathbf{max}\ \vec{c}_r \cdot \vec{r} \quad \text{s.t.}\ \Gamma \cdot \vec{r} \geq -\vec{d}_0 \qquad (7.35)$$

in Standardform bringen. Diese Formulierung hat die direkte Bedeutung „Es dürfen nur so viele Delays von einer Kante abgezogen werden, wie vorher dort vorhanden waren ($d(e)\geq 0$)“.

Eine Maximierungsaufgabe, die die Komponenten von $\vec{r}$ mit in die Maximierung einschließt ist allgemein nicht zulässig, da durch die Mehrdeutigkeit von $\vec{r}$ jedes Vielfache von $\vec{q}$ zu $\vec{r}$

addiert ebenfalls eine Lösung ist. Die lineare Optimierung würde nicht zur Lösung führen und das Problem mit ‘unbounded solution’ beantworten.
Der Grund für die Problematik liegt auch darin, daß die Optimierungsrichtung (beschrieben durch den *c*-Vektor) senkrecht auf einem Richtungsvektor steht, der den $\vec{q}$-Vektor enthält. Die in einer Ecke des Polyhedrons gefundene optimale Lösung hat daher noch weitere alternative optimale Lösungen, die sich auf einer Kante des Polyhedrons befinden. Diese Kante beginnt im gefundenen Optimalpunkt und endet im Unendlichen.

$$\textbf{max}\ \vec{c}\cdot\vec{d}\ \ \text{s.t.}\ \ \vec{d}-\Gamma\cdot\vec{r}=\vec{d}_0$$

$$\Leftrightarrow\ \textbf{max}\begin{pmatrix}\vec{c}\\ \hline \vec{0}\end{pmatrix}\cdot\begin{pmatrix}\vec{d}\\ \hline \vec{r}\end{pmatrix}\ \text{s.t.}\ \ \vec{d}-\Gamma\cdot\vec{r}=\vec{d}_0 \qquad (7.36)$$

$$\begin{pmatrix}\vec{c}\\ \hline \vec{0}\end{pmatrix}\perp\begin{pmatrix}\vec{0}\\ \hline \vec{q}\end{pmatrix}*k\ \text{(alternative optimale Lösung)}$$

7.8 Cutting Planes und Branch & Bound

Es wurde gezeigt, daß sich das Problem des optimierenden Multiraten-Retimings allgemein nicht mit linearer Programmierung lösen läßt. Die zusätzliche Beschränkung der Entscheidungsvariablen auf ganzzahlige Werte garantiert aber ein legales Retiming, denn die Existenz eines ganzzahligen Retiming-Vektors $\vec{r}$ (interpretierbar als Parikh-Vektor) ist schon notwendige und hinreichende Bedingung für Erreichbarkeit des durch Retiming gewonnenen Zustands (Kap. 4.9). Die ganzzahlige Beschränkung führt zu einem ILP-Problem

$$\textbf{max}\ \vec{c}\cdot\vec{d}\ \ \text{s.t.}\ \ \vec{d}-\Gamma\cdot\vec{r}=\vec{d}_0\ ;\ \vec{d},\vec{r}\ \text{integer.} \qquad (7.37)$$

Die Formulierung $\textbf{max}\ \vec{c}\cdot\vec{d}\ \ \text{s.t.}\ \ C\cdot\vec{d}=C\cdot\vec{d}_0$ ist wegen fehlender *r*-Komponenten nicht geeignet um als ILP-Problem legales Retiming hervorzurufen.

Standardalgorithmen zur ganzzahligen linearen Programmierung sind entweder durch die Methoden „Cutting Planes“ oder „Branch & Bound“ implementiert (Kap. 6.1.2). Beide Verfahren sind von exponentieller Komplexität, da das ILP-Problem NP-vollständig ist [70].

Die von „Cutting Planes“ in jedem Lösungsschritt neu hinzugefügte Schnittebene (sie begrenzt ein Stück des Polyhedrons auf ganzzahlige Randpunkte durch Einfügung einer neuen Facette) könnte möglicherweise schon in die Problemstellung integriert werden. Ein Ziel wäre es, schon ein ganzzahliges Polyhedron [68] bereitzustellen, das dann durch einen LP-Löser in polynomialer Zeit zur Lösung führt. Ein ganzzahliges Polyhedron (alle Eckpunkte sind ganzzahlige Ortsvektoren) garantiert, daß immer eine ganzzahlige Lösung gefunden wird.

Die Analyse des „Cutting Planes“-Verfahrens offenbarte allerdings folgende Probleme:

- Das Problem, das ganzzahlige Polyhedron zu bestimmen ist NP-vollständig [68] und von höherer Komplexität als der Algorithmus „Cutting Planes“
- Der Algorithmus bestimmt nur Schnittebenen, die in der Richtung der Optimierung das Polyhedron einschränken [61].
- Alle ganzzahligen Vektoren, die die gegebenen Ungleichungen erfüllen, bleiben auch weiterhin im Lösungsraum. Die Schnittebenen schneiden also nie das ‘integer polyhedron’ an [61].
- Die Anzahl der Schnittebenen steigt exponentiell mit der Problemdimension.

- Die Algorithmus startet nicht nach jeder Einfügung einer neuen Schnittebene von neuem mit der Optimierung sondern geht von der zuvor gefundenen (nicht-ganzzahligen) Lösung aus und ermittelt durch dualen Simplex eine neue gültige Lösung.
- Die Schnittebenen können nicht nur in den Variablen d_i ausgedrückt werden, sondern beschränken auch die Variablen r_i, sie lassen sich daher nicht als Beschränkung in den Delays interpretieren.

Die Ergebnisse machen deutlich, daß eine Verbesserung des gegebenen Algorithmus ohne weitere Kenntnisse über Besonderheiten des Graphen nicht möglich ist.

Die Methode „Branch & Bound“ teilt in jedem rekursiven Schritt den Lösungsraum in zwei Teile auf, deren Optimum wieder gesucht wird usw.. Diese Vorgehensweise ist nicht durch Berechnungen im Voraus ersetzbar und sollte daher unverändert angewandt werden. Die Formulierung als gemischtes ganzzahliges Optimierungsproblem (r_i integer, d_i reell) reicht zur Lösung aus und kann die Lösungssuche etwas schneller machen.

7.9 Schnittebenen im Zustandsraum

Der Zustandsraum, den die (Orts-)Vektoren $\vec{d}$ aufspannen, enthält diskrete (ganzzahlige) erreichbare Punkte. Die im Kapitel 4.9.1 vorgestellte Struktur offenbart die Möglichkeit, ähnlich dem Verfahren „Cutting Planes“ Schnittebenen so in den Raum einzufügen, daß sie die erlaubten Zustände in der Kanten des Polyhedrons befinden lassen. Jede schon vorhandene Schleifen-Bedingung

$$\vec{y}^{\mathbf{T}}\vec{d}=WST$$

führt zu einer Hyperebene im Zustandsraum für $\vec{d}$. Aus dem Schnitt mehrerer solcher Ebenen setzt sich das Polyhedron zusammen. Durch zusätzliche Ebenen läßt sich dieses auf ein Polyhedron reduzieren, das nur noch erreichbare Zustände in den Eckpunkten enthält. Dann kann mit gewöhnlicher linearer Programmierung in polynomialer Zeit eine legale Lösung gefunden werden. Zu beachten ist, daß das durch die Schleifen-Bedingungen gebildete Polyhedron (für normierte Graphen) schon ein ganzzahliges ist, d.h. es liefert schon ganzzahlige Lösungswerte, diese sind nur nicht immer erreichbar.

Es stellte sich heraus, daß immer Schnittebenen im $\vec{d}$-Raum ausreichen und nicht, wie bei „Cutting Planes“, noch Bedingungen in $\vec{r}$ enthalten sein müssen. Es läßt sich also eine geeignete konvexe Hülle um die erreichbaren Zustände legen, in der jeder Eckpunkt ein erreichbarer Zustand ist.

Zur allgemeinen Bestimmung der zusätzlichen Ebenen kann „Cutting Planes“ nicht eingesetzt werden, weil es auch die Information aus $\Gamma\vec{r}$ benötigt und schon exponentielle Komplexität hat. Die direkte Bestimmung der konvexen Hülle erfordert die Kenntnis über die erreichbaren Zustände; diese können auch nur mit exponentiellem Aufwand bestimmt werden.

Da diese Ansätze nicht sinnvoll sind, wurde versucht, die Gleichungen der Schnittebenen aus der Struktur des Graphen abzuleiten. Aus der Diskussion der Eigenschaften von NMD-Kanten und -Pfaden (Kapitel 7.3) ergibt sich, daß charakteristische Lücken im Erreichbarkeitsraum zwischen dem Graph und den Wandebenen ($d_i \geq 0$) bei Kanten-NMD’s auftreten. Für den Beispielgraph aus Kapitel 7.2.1 mit einem NMD auf Kante e_1 reicht eine Zusatzbedingung $d_1 \geq 1$ aus, um alle erreichbaren Zustände zu Ecken des Polyhedrons zu machen. Einzelne NMD-Pfade bewirken eine Lücke, die die Summe der Delays auf diesem Pfad nicht Null werden läßt. Der Graph aus Kapitel 7.2.2 enthält ein Pfad-NMD auf Pfad $\{e_2,e_3\}$. Die Bedingung $d_2+d_3 \geq 1$ reicht hier aus um zu verhindern, daß das NMD verschwindet.

Für eine Teilklasse von Graphen hat sich diese Methode bewährt. Dazu muß nur für jeden Pfad $p_{ij}=\{e_i,...,e_{j-1}\}$ (zwischen Knoten v_i und v_j), dessen Modulo-Rest (unnormiert)

$$\rho_{ij} = \sum_{e \in p_{ij}} w_e d_e \ \textbf{mod ggT}[O(v_i)w(e_i), I(v_j)w(e_{j-1})] \tag{7.38}$$

größer als Null ist, die Bedingung

$$\sum_{e \in p_{ij}} w_e d_e \geq \rho_{ij} \tag{7.39}$$

den LP-Ungleichungen hinzugefügt werden.

Diese Zusatzbedingungen für jeden NMD-Pfad verhindern, daß die lineare Optimierung die Delays auf diesem Pfad 'wegoptimiert'. Die gewöhnliche LP-Lösung würde für eine Schleife so aussehen, daß die Kante mit dem höchsten Gewicht in der Zielfunktion alle Delays der Schleife enthält und die anderen Kanten leer sind. Obige Bedingungen gewährleisten, daß die minimale Anzahl Delays auf jedem Pfad genau die Modulo-Bedingung erfüllt. Der Aufwand zur Bestimmung der Bedingungen ist nicht mehr exponentiell. Es müssen lediglich Pfade ermittelt werden mit $O(|E|^2)$ und die NMD-Gewichte ρ_{ij} festgestellt werden mit $O(|E|^3)$.

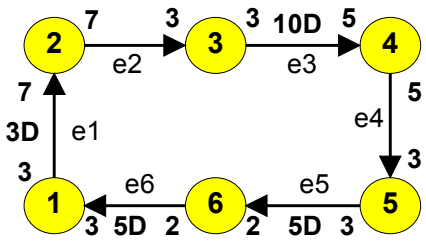
In dieser Form ist dieses Prinzip schon für azyklische Graphen brauchbar, bei denen man durch lineare Optimierung möglichst viele Delays entfernen möchte (Maximierung ist nicht möglich, weil das Optimum 'unbounded' wäre).

In zyklischen Graphen wird auf der Kante mit dem höchsten Gewicht die maximale Anzahl Delays gesammelt. Die obigen Bedingungen sorgen dafür, daß auf allen Pfaden, die diese Kante nicht enthalten, die notwendige Anzahl Delays verbleiben. Für alle Modulo-Pfade ($\mu_{ij}>1$), die diese Kante enthalten, existiert aber keine obere Grenze für die Delaysumme außer *WST*. Daher kann allgemein nicht garantiert werden, daß die Modulo-Bedingung für diese Pfade erfüllt wird. Dies tritt in allen überlappenden Modulo-Pfaden auf, die unterschiedliche Primfaktoren enthalten ($\textbf{ggT}(\mu_1,\mu_2) < \textbf{Min}(\mu_1,\mu_2)$).

Wenn die Kante mit dem höchsten Gewicht nicht Teil einer derart überlappenden Menge von Pfaden ist, sondern nur Teil eines separaten Pfades oder einer Kante mit $\mu_{ij}>1$, dann wird auch auf dieser Kante für die obere Grenze die Modulo-Bedingung eingehalten. Grund dafür ist die garantierte Konstanz der WST in jeder Schleife und die aufsummierten minimalen Delays WST_{Rest} auf den anderen Kanten. Auf der Zielkante einer Schleife bleiben nur WST-WST_{Rest} Delays übrig, die dort die Bedingung erfüllen.

Der Graph im Bild rechts soll als Beispiel dienen. Hier ist *WST*=23. Die Modulo-Bedingungen und die daraus folgenden Ungleichungen lauten:

$d_1+d_2 \bmod 3 = 0$	keine.
$d_3+d_4 \bmod 3 = 1$	$d_3+d_4 \geq 1$
$d_5+d_6 \bmod 3 = 1$	$d_5+d_6 \geq 1$



Jede Neuverteilung, die *WST*=23 und die Ungleichungen erfüllt, ist vom Anfangszustand erreichbar und damit legaler Retiming-Zustand.

7.10 Minimaler retimingäquivalenter Einheitsraten-Graph

Eine alternative Methode um Retiming durch lineare Optimierung durchzuführen ist es, den Multiraten-Graph in einen Einheitsraten-Graph zu konvertieren, die Optimierung mit linearer Programmierung in diesem Graph durchzuführen und danach wieder eine Rückwandlung in einen Multiratengraphen vorzunehmen. Diese Methode garantiert legales Retiming. Die Transformationsvorschriften zur Konvertierung in beiden Richtungen wurden im Kapitel „Transformationen“ (5.1) vorgestellt. Die Transformation in einen Einheitsraten-Graphen hat allerdings eine Komplexität $O(\Sigma q_i+\Sigma n_i)$ und kann damit nur als pseudopolynomial bezeichnet werden.

Die Anzahl der Kanten des echt äquivalenten Einheitsraten-Graphen ist mit Σn enorm groß (immer größer als die Anzahl der Knoten: Σq). Bei einem coprimen Multiraten-Graphen ist der Wert des $\vec{q}$-Vektors eines Knotens das Produkt der Raten aller anderen Knoten. Die Summe aller Einträge des $\vec{q}$-Vektors ist also immer größer als das Produkt der ersten bis zur (k-1).ten Primzahl (k=|V|=Knotenanzahl). Damit ist eine exponentielle Komplexität der Umwandlung für diesen Fall gegeben.

Die lineare Optimierung müßte in einem Raum mit so hoher Dimension wie die Anzahl der Kanten $|E_E|$ des äquivalenten Graphen durchgeführt werden. Bei polynomialer Komplexität in $|E_E|$ ist dies trotzdem wegen der allgemein hohen Kantenanzahl des äquivalenten Graphen kein polynomialer Aufwand in $|V|$ oder $|E|$.

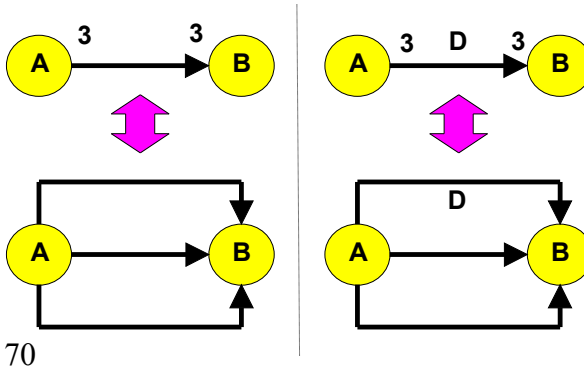
Die Analyse des linearen Gleichungssystems, das dem Simplex-Algorithmus als Nebenbedingung zur Optimierung bereitsteht und aus $\Sigma|E_E|$ Entscheidungsvariablen besteht, offenbarte nach einigen Anläufen die Funktionsweise der Gewinnung ganzzahliger Lösungen für den Multiratenfall: Die Lösung des Problems

$$\begin{aligned} &\textbf{max}\ \vec{c}\cdot\vec{d} \\ \text{s.t.}\quad &\vec{d}-\Gamma\vec{r}=\vec{d}_0 \\ \Leftrightarrow\ &\left[I\ \middle|\ -\Gamma\right]\cdot\begin{bmatrix}\vec{d}\\ \vec{r}\end{bmatrix}=\vec{d}_0 \end{aligned} \tag{7.40}$$

für Einheitsraten liefert ganzzahlige Lösungen für $\vec{d}$ und $\vec{r}$, da die Matrix Γ unimodular ist und die Anfügung einer Einheitsmatrix I die Unimodularität nicht verändert [68].

Die Anzahl der Einheitsraten-Kanten, die einer Kante im Multiraten-Graphen entsprechen, ist ein ganzzahliges Vielfaches der Rest-Bedingungs-Werte μ_{ij}, die für jeden Pfad p_{ij}, der diese Kante enthält, die Bedingung $d(p_{ij})$ **mod** μ_{ij} =const fordern. Wird durch die Optimierung auf einer Einheitsraten-Kante ein Delay abgelegt, dann werden auf allen anderen Einheitsraten-Kanten, die zu derselben Modulo-Bedingung gehören, eine passende Anzahl von Delays vorgesehen, die die Bedingungen insgesamt wieder erfüllen.

Eine Vorstellung von dieser durch die Struktur des Graphen erzwungenen Einschränkung gewinnt man am besten, wenn man nur eine Kante betrachtet, deren End-Raten einen Rest-Bedingungs-Wert von z.B. drei aufweisen (Bild). Die Umwandlung dieser Kante führt zu drei parallelen Kanten, deren summarische Delay-



anzahl nur in Vielfachen von drei geändert werden kann, denn drei durch diese Kanten laufenden (ungerichteten) Schleifen erzwingen die Konstanz der Delaysumme. Auf jede Kante können nur ganzzahlige Delays gebracht werden, daher wird die Bedingung d_e **mod** 3=const immer erfüllt.

Die Erkenntnis über die strukturellen Einschränkungen des normalen äquivalenten Einheitsraten-Graphen und der für legales Retiming notwendigen und hinreichenden Bedingungen (Kapitel 7.2) führte zu Ideen, wie eine andere Äquivalenztransformation funktionieren könnte, die nicht unbedingt bezüglich des Scheduls oder des Datentransfers äquivalent zu sein braucht aber für Zwecke des Retimings die Einhaltung der dafür erforderlichen Bedingungen garantiert.

Erste Ansätze ließen eine einfache Vervielfachung der Kanten vermuten, allerdings stellte sich bald heraus, daß auch andere Bedingungen eingehalten werden müssen um zu korrekten Delayverteilungen zu führen. Ausgehend von einem normierten Graph sind die folgenden Bedingungen einzuhalten:

$$\left(\sum_{i \in \ Pfad} d_i \right) \mathbf{mod}\, \mu_{ij} \qquad \mu_{ij} = \mathbf{ggT}\left(m_i, m_j \right) \qquad \forall \ \ Pfade\ zwischen\ v_i, v_j \tag{7.41}$$

erreicht durch: [Anzahl ER-Kanten für MR-Kante e] = $\mathbf{kgV}(\mu_{ij})$ $\forall$ $p_{ij} \subset \{\, p \subset G \mid e \in p \,\}$

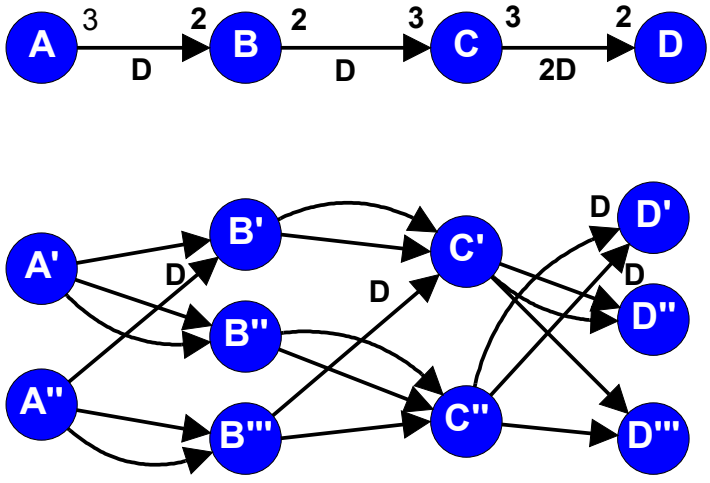
und

$$\sum_{i \in \ Schleife} d_i = const \qquad \forall \ \ Schleifen \subseteq \ Graph \tag{7.42}$$

erreicht durch ΣEingangskanten = ΣAusgangskanten $\forall$ $v_{ER} \in V_{SR}$.

Weiterhin müssen die Kanten in spezieller Weise die Knoten so verbinden, daß nicht bewegliche Delays an manchen Stellen festgehalten werden.

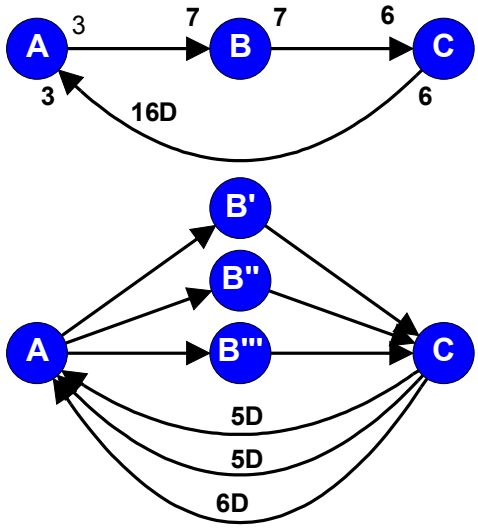
Der folgende Graph verdeutlicht die nötigen Verbindungen für mehrere überlappende Pfade:



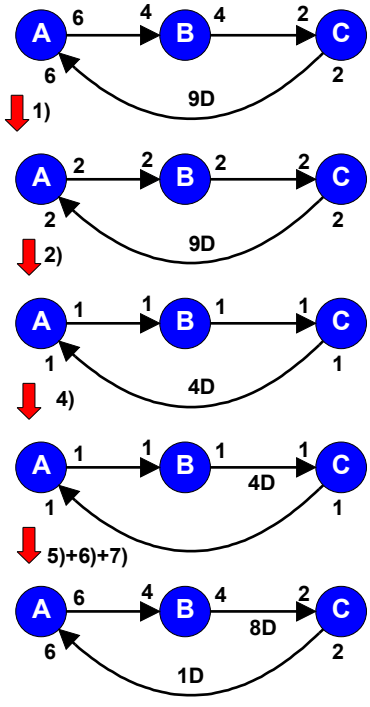
Die Realisation der Bedingungen läßt sich am einfachsten durchführen, wenn von der regulären Transformation Multiraten → Einheitsraten ausgegangen wird, jedoch die überflüssigen Graphenstrukturen, die nicht zur Erfüllung der Modulo-Bedingungen beitragen, entfernt werden. Diese ‘Redundanz’ konnte durch folgende Vorgehensweise (für normierte Multiraten-Graphen) beseitigt werden:

1) Man reduziere die Raten eines Knotens um den Primfaktor, den kein anderer Knoten als Rate enthält. Ein Graph, der z.B. nur einen Knoten mit Rate zwei und einen mit Rate zehn enthält, kann die Rate des zweiten Knotens von zehn auf zwei reduziert bekommen. Die Modulo-Bedingungen werden hierdurch nicht verändert, da für μ_{ij} der ggT nur gemeinsame Primfaktoren der Raten berücksichtigt.
2) Man reduziere danach die Raten aller Knoten gleichmäßig um den größten gemeinsamen Teiler aller vorhandenen Knotenraten, wobei die Anzahl der Delays jeder Kante auch durch diesen Wert dividiert werden. Ein eventuell auftretender Rest (Kanten-NMD) muß gespeichert und später wieder hinzugefügt werden. Ein Graph, dessen Raten z.B. alle acht betragen, kann so direkt zum Einheitsratengraph umgewandelt werden.
3) Die Umwandlung zum Einheitsratengraph (wie in Kapitel 5.1. beschrieben) wird durchgeführt.
4) Retiming wird angewandt.
5) Die Delayverteilung wird auf den reduzierten Multiraten-Graph zurücktransformiert.
6) Raten und Delays werden mit dem Faktor aus Schritt 2 wieder multipliziert. Die gespeicherten Delay-Reste werden anschließend addiert.
1) Die ursprünglichen Raten werden wiederhergestellt.

Die Schritte 1) und 3) wurden im nächsten Beispiel an einer Schleife durchgeführt (Bild rechts). Knoten 'B' enthält den Primfaktor 7, der sonst nicht vorkommt und Knoten 'C' besitzt einen einmaligen Primfaktor 2. Die Redundanz-Reduktion führt zu den Raten 3-1-3 für die Knoten A-B-C, der vereinfachte Graph kann leicht in einen Einheitsratengraph transformiert werden, dessen durch Retiming erreichbaren Zustände denen des Ursprungsgraphen entsprechen. Man beachte die Verschmierung der 16 Delays auf die resultierenden Kanten. Die Modulo-Bedingung für die Kante *e* zwischen C und A ($d(e)$ **mod** 3 = 1) wird dadurch eingehalten. Der gewöhnliche äquivalente Einheitsraten-Graph hätte 126 Kanten haben müssen.



Alle Schritte können im folgenden Bild nachvollzogen werden:



m_A=6=2*3; m_B=4=2*2; m_C=2;
Abspaltung redundanter Primfaktoren:
m'_A=2; m'_B=2; m'_C=2;

Division aller Raten durch 2; 'Merken' des Restwertes 1D auf Kante *C*-*A*.
Umwandlung in äquivalenten Einheitsratengraph (hier nicht mehr nötig).
Retiming durch lineare Optimierung (Ziel: möglichst viele Delays vor Knoten *C*).
Zwischenzustand der Delayverteilung erreicht.
Multiplikation der Raten und Delays mit 2.

Addition der Restwerte (1D auf Kante *C*-*A*).
Wiederherstellung der ursprünglichen Raten.
Endzustand der Delayverteilung; legales Retiming.

7.11 Retiming durch 'Shortest Path'

Für Einheitsraten-Graphen kann die Verwendung graphentheoretischer Eigenschaften und Algorithmen sinnvoll sein. Lautet das Ziel, das durch Retiming erreicht werden soll, die Optimierung der Iterationsperiode, dann ist das in [33] vorgestellte Verfahren dazu geeignet. Es basiert auf der Feststellung, daß die kürzeste machbare Iterationsperiode bei gegebener Delayverteilung in einer synchronen Schaltung (Hardware) sich berechnet aus dem Maximum der summierten Ausführungszeiten aller Pfade, die keine Delays (Separatoren) auf ihren Kanten besitzen:

$$\Phi(G)=\max \{\ T(p) \mid w(p)=0\ \} \tag{7.43}$$

mit $T(p)=\sum t(v_i)$ = Summe der Ausführungszeiten der Knoten des Pfades p.

und $w(p)=\sum d(e_i)$ = Summe der Delays auf den Kanten des Pfades p.

Diese Größen T und w müssen zur Bildung des Maximums für die Berechnung der Iterationsperiode für jeden Pfad bekannt sein. Dazu werden zwei Matrizen W und T bestimmt, die jeweils die obigen Größen für die Pfade zwischen zwei Knoten als Einträge enthalten. Dabei enthält $W(u,v)$ das Minimum der Anzahl der Delays von mehreren parallelen Pfaden zwischen Knoten u und v. $T(u,v)$ ist die summierte Ausführungszeit genau des Pfades, der den Wert $W(u,v)$ liefert.

Die Einträge der Matrizen können durch Anwendung eines 'Shortest Path'-Algorithmus bestimmt werden, der zu zwei Knoten immer den kürzesten Pfad bzw. dessen Maß findet. Ein 'All Pairs Shortest Path' findet dieses Maß für alle Knotenpaare.

Die Einträge der Matrix T geben mögliche Kandidaten für die optimale Iterationsperiode an. Die kurzen Zeiten werden aber wahrscheinlich nicht zu erreichen sein. Der in [33] vorgeschlagene Algorithmus startet eine binäre Suche unter den sortierten Elementen von T und testet für jeden der herausgegriffenen Werte, ob dieser als Iterationsperiode machbar ist.

Der Test der Machbarkeit {feasibility} wird mit einem Bellman-Ford-Algorithmus durchgeführt. Dieser testet, ob ein Satz von Ungleichungen in $\vec{r}$ (Retiming-Vektor) erfüllt werden kann oder nicht. Die Ungleichungen verlangen: 1.) Daß ein Anfangsknoten {host} nicht durch Retiming aktiviert wird; 2.) Daß auf keiner Kante eine negative Anzahl Delays durch Retiming entsteht; 3.) Daß in jeden Pfad, dessen Ausführungszeit $T(p)$ größer als die zu testende Iterationsperiode c ist, ein zusätzliches Delay eingefügt wird (Dieses würde dann nämlich die Ausführungszeit zwischen Iterationen in diesem Pfad verkürzen). Die Bedingungen lauten:

1) $r(v_h)=0$
2) $-r(v) \le w(e)$ für jede Kante e zwischen Knoten $u \rightarrow v$
3) $-r(v) \le W(u,v) - 1$ für alle Knoten $u,v \in V$, für die $T(u,v) > c$ ist.

Die Ungleichungen dieses als Instanz einer LP-Optimierung interpretierbaren Problems sind die 'Bellman-Ford'-Gleichungen eines Graphen, dessen Kantengewichte die rechten Seiten

der Ungleichungen sind. Das Problem, Werte für $\vec{r}$ zu finden, die diese Ungleichungen erfüllen, kann mit dem 'Bellman-Ford-Shortest-Path'-Algorithmus mit Aufwand O($|V||E|$) gelöst werden. Die Elemente von $\vec{r}$ haben hier genau umgekehrtes Vorzeichen im Vergleich zur Definition in Kapitel 7.1; dafür sind die Ungleichungen aber direkt in der geeigneten Form.

Ist die kürzeste machbare Iterationsperiode gefunden, dann sind die dazu ermittelten Werte $r(v)$ die negativen Elemente des Retiming-Vektors $\vec{r}$, der angibt, wie oft ein Knoten aktiviert werden muß.

Der Algorithmus läßt sich so nicht im Multiratenfall verwenden, denn die Iterationsperiode kann nicht so einfach wie im Einheitsratenfall bestimmt werden. Sie ist hier nicht die Summe der Ausführungszeiten der Knoten im längsten Pfad ohne Delays. Die Einträge der Matrix W haben hier keinen Sinn. Auch ist der Bellman-Ford-Algorithmus nicht bei multiplen Raten anwendbar. Das Festhalten eines Knotens v_h darf bei Multiraten-Retiming nicht vorgenommen werden, da für optimierendes Retiming bis zu q_{vh}-1 Aktivierungen dieses Knotens nötig sind.

Für azyklische Graphen ist eine Minimierung der Delays durch Retiming mit einer modifizierten 'Shortest-Path'-Methode nach Dijkstra denkbar. Dafür werden, beginnend mit dem Nachfolgeknoten des Startknotens, von jeder betrachteten Kante nur so viele Delays abgezogen und auf die Ausgangskanten verteilt, wie die Eingangsrate des Zielknotens zuläßt. Dies ist aber wieder nur unter der Nebenbedingung machbar, daß der Startknoten nicht aktiviert wird.

7.12 Anfangsbedingungen

Die Software-Implementation eines durch Retiming veränderten Programms besteht allgemein aus zwei Teilen: Ein Initial-Programm muß die neuen Zustände (Werte) in den Speicherelementen berechnen, die als Delays verschoben wurden; das Hauptprogramm führt die normalen Berechnungen aus, die im periodischen Schedul für die Iterationen festgelegt sind. Das Initial-Programm ist eine Folge von Anweisungsblöcken, die in einem Initial-Schedul festgelegt sind. Dieses enthält jeden Knoten so oft wie es der Retiming-Vektor $\vec{r}$ in seinen Komponenten angibt, falls diese alle positiv sind.

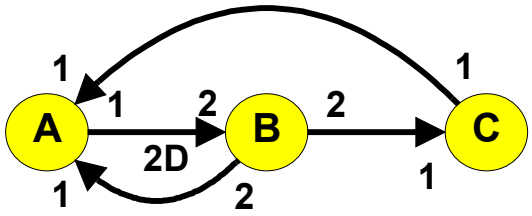
Die Berechnung der Anfangswerte kann, wie erwähnt, in einem Initial-Programm vor den iterativen Berechnungen durchgeführt werden (dynamisch) oder die Anfangswerte werden schon zur Compilationszeit festgelegt (statisch). Im ersten Fall muß die Berechnung mit Vorliegen der notwendigen Eingangswerte starten, was eine zusätzliche Latenz verursachen kann. Der zweite Fall ist nur möglich, wenn sich die Werte überhaupt unabhängig von speziellen Eingangswerten berechnen lassen.

In Fällen iterativer Berechnungen, bei denen in den Knoten $f(0)=0$ gilt, und das System eine hohe Stabilität hat, können die Anfangswerte weggelassen und durch Nullen ersetzt werden. Nach endlicher Zeit ist das System dann so eingeschwungen, daß kein Unterschied bemerkbar ist.

Wenn Elemente des Retiming-Vektors negativ sind (was einer umgekehrten Ausführung entspräche), dann kann die Berechnung der Anfangswerte sehr kompliziert oder sogar unmöglich werden [25]. Soll zum Beispiel ein Delay hinter einem Knoten v vor diesen verschoben werden (r_v= -1), dann ist keine Berechnung des Anfangszustands in diesem Delay möglich, wenn die Funktion, die der Knoten ausführt, nicht umkehrbar ist.

Die Implementation eines durch Retiming veränderten Programms wird nun an einem Beispiel verdeutlicht ($\vec{q}=(2,1,2)^T$; $\vec{r}\,'=(-2,0,0)^T$ bzw. $\vec{r}=(0,1,2)^T$):

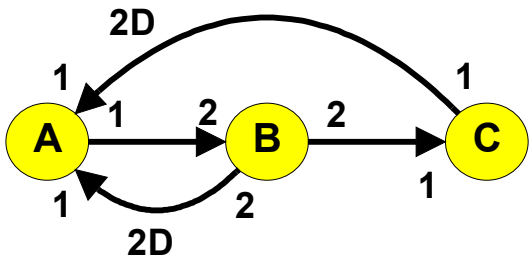
Ursprung



Anfangswerte: $ab(0)$, $ab(1)$

for (n=1 to ∞) {
$bc(2n)=f_{bc1}[ab(2n\text{-}2),ab(2n\text{-}1)]$
$bc(2n+1)=f_{bc2}[ab(2n\text{-}2),ab(2n\text{-}1)]$
$ba(2n)=f_{ba1}[ab(2n\text{-}2),ab(2n\text{-}1)]$
$ba(2n+1)=f_{ba2}[ab(2n\text{-}2),ab(2n\text{-}1)]$
$ca(2n)=f_{ca}[bc(2n)]$
$ca(2n+1)=f_{ca}[bc(2n+1)]$
$ab(2n)=f_{ab}[ca(2n),ba(2n)]$
$ab(2n+1)=f_{ab}[ca(2n+1),ba(2n+1)]$ }

Retimed



Anfangswerte: $ba(2)=f_{ba1}[ab(0),ab(1)]$;
$ba(3)=f_{ba2}[ab(0),ab(1)]$;
$ca(2)=f_{ca}[f_{bc1}[ab(0),ab(1)]]$
$ca(3)=f_{ca}[f_{bc2}[ab(0),ab(1)]]$
for (n=1 to ∞) {
$ab(2n)=f_{ab}[ca(2n),ba(2n)]$
$ab(2n+1)=f_{ab}[ca(2n+1),ba(2n+1)]$
$bc(2n+2)=f_{bc1}[ab(2n),ab(2n+1)]$
$bc(2n+3)=f_{bc2}[ab(2n),ab(2n+1)]$
$ba(2n+2)=f_{ba1}[ab(2n),ab(2n+1)]$
$ba(2n+3)=f_{ba2}[ab(2n),ab(2n+1)]$
$ca(2n+2)=f_{ca}[bc(2n+2)]$
$ca(2n+3)=f_{ca}[bc(2n+3)]$ }

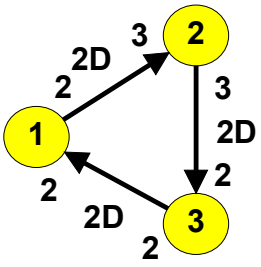
Hierbei ist mit $bc(x)$ der Wert gemeint, der vom Knoten B auf die Kante zwischen Knoten B und C zum Zeitpunkt x geschrieben wird.

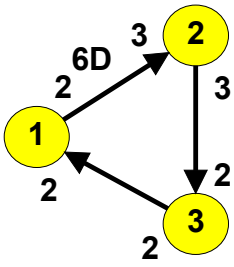
Im durch Retiming veränderten Algorithmus müssen die neuen Anfangswerte vor dem periodischen Schedul berechnet werden. Die Index-Offsets in der Programmschleife sind dort so angepaßt, daß insgesamt dieselben Berechnungen im linken und rechten Algorithmus durchgeführt werden. Wegen der Rate zwei am Knoten B existieren für die Ausgangswerte von B jeweils zwei verschiedene Funktionen, denen aber dieselben Eingangswerte zur Verfügung stehen.

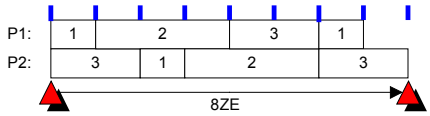
7.13 Anwendungen von Retiming

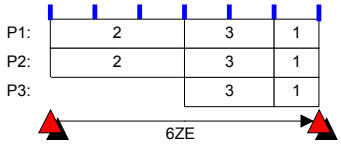
7.13.1 Verbesserung des Durchsatzes

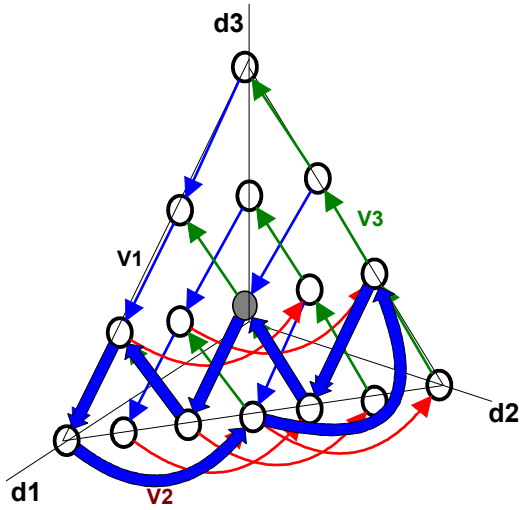
Retiming kann dazu benutzt werden, den Durchsatz zu erhöhen. Es hat sich als zweckmäßig und günstig erwiesen, möglichst alle Delays auf einer Kante zu konzentrieren. Als Beispiel dazu betrachte man den linken Graph im Bild unten. Darunter ist sein Schedul abgebildet und der Erreichbarkeitsgraph mit hervorgehobenem Anfangszustand und eingezeichneter Schedul-Aktivierungsfolge (dicke Pfeile). Im Bild rechts ist derselbe Graph nach Retiming dargestellt. Sein Schedul ist wesentlich schneller, da vom neuen Anfangszustand aus mehr Knoten gleichzeitig aktiviert werden können. Dies ist im Zustandsraum an der langen durchgehenden Kante am Rand der Fläche zu erkennen.

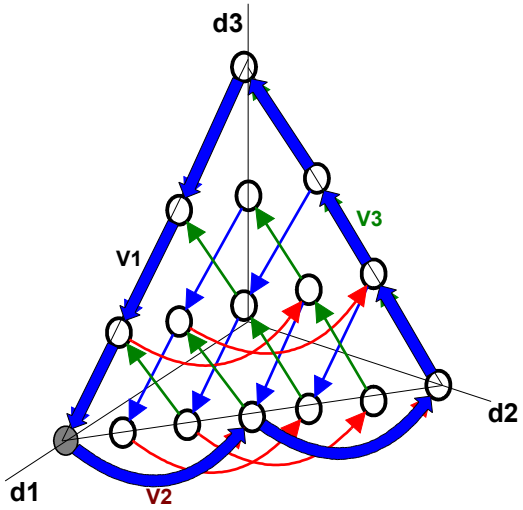












7.13.2 Minimierung der Speicherelemente

Jeder Zwischenspeicher für Tokens erfordert Speicherplatz im Hauptspeicher. Die Gesamtmenge an Speicher kann durch Retiming minimiert werden. Der gesamte Speicherbedarf bestimmt sich, wenn sich Puffer nicht überlappen dürfen, aus der Summe der mit der Wortbreite multiplizierten maximalen Delayanzahl jeder Kante. Die maximale Delayanzahl jeder Kante hängt vom Schedul ab und ist daher nicht einfach berechenbar. Günstig ist es, die Delays möglichst gleichmäßig auf alle Kanten zu verteilen, so daß viele verschiedene Knoten gleichzeitig aktiviert werden können und sich jeweils nur die Delays für eine Aktivierung eines nachfolgenden Knotens auf den Kanten befinden.

7.13.3 Separatoren zwischen Clustern

Bei der Bildung von Clustern aus Knoten, die auf verschiedenen Prozessoren laufen sollen, möchte man möglichst, daß die Prozessoren untereinander nicht auf Daten warten müssen. Durch Bewegung von genügend Delays auf die Übergangskanten zwischen den Clustern läßt sich zeitliche Parallelität schaffen (Pipelining). Die Blöcke hinter diesen Separatoren operieren auf den Daten der vorherigen Iteration, während die vordersten Blöcke bei einer azyklischen Gesamtstruktur die aktuellen Daten bearbeiten. Durch Einfügung von Bedingungen $d(e) \geq s$ für die betreffenden Kanten in eine LP-Optimierung wird Pipelining automatisch durchgeführt.

8 Der Durchsatz in Multiratensystemen

8.1 Die Iterationsperiode und deren Schranke

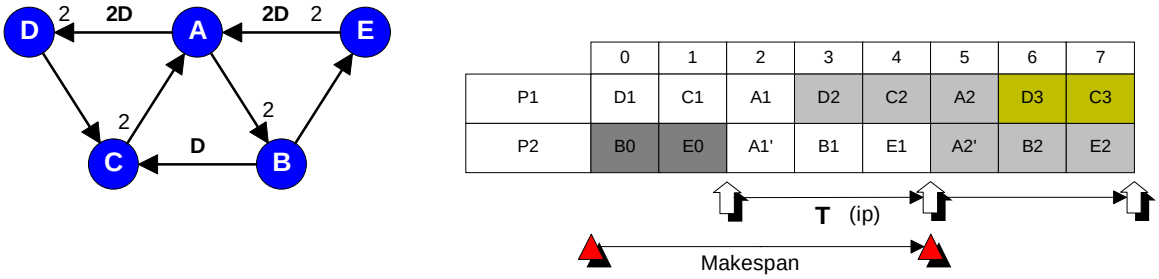
Definition: Die Iterationsperiode ist die durchschnittliche Zeit im Schedul zwischen dem Start einer vollständigen Aktivierungssequenz (Iteration), innerhalb der jeder Knoten genau q_i mal aufgerufen wird, und dem Start der nächsten.

Dabei ist $\vec{q}$ ein Vektor aus natürlichen Zahlen aus dem Nullraum von Γ. Mehrere Iterationen können zu einen Block zusammengefaßt werden {blocked schedule} und bilden eine Periode. Das Schedulmuster {pattern} wiederholt sich identisch nach jeder Periode, während innerhalb einer Periode keine periodischen Muster zu erkennen sein müssen [18],[21].

Die Iterationsperiode ist der Kehrwert des Durchsatzes. Die Iterationsperiode hat bei zyklischen Graphen eine untere Grenze (bei Annahme unerschöpflicher Ressourcen, d.h. beliebig vieler Prozessoren), die sogenannte *iteration period bound*, kurz *iteration bound*. Ziel eines Scheduling ist es, diese Grenze zu erreichen, um den Durchsatz zu maximieren. Die 'iteration bound' ist für einen Graphen mit gegebener Struktur und Delaysumme (im Einheitsratenfall) konstant, kann aber prinzipiell erreicht werden (sie ist daher 'tight' [32]). Sie kann aber bei fester Delayverteilung nicht immer durch ein einfaches Schedul erreicht werden.

Durch Retiming kann eine andere Delayverteilung erzeugt werden, die dem optimalen Schedul näher kommt; Retiming kann jedoch kein optimales Schedul garantieren [21]. Durch Unfolding kann im Einheitsratenfall ein raten-optimales statisches Schedul gefunden werden; dabei bilden *J* Iterationen eine Periode [23].

Im abgebildeten Beispiel ist die Iterationsperiode 3 Zeiteinheiten lang und optimal. Wenn *D* ein Eingangs- und *E* ein Ausgangsknoten ist, beträgt die Latenz 5 Zeiteinheiten.



Bekannt ist bisher nicht, wie die optimale Iterationsperiode bei fest vorgegebener Prozessoranzahl bestimmt werden kann und wie man sie erreicht.

Im Multiratenfall sind bisher wenige Eigenschaften bekannt. Für die Grenze der Iterationsperiode existieren nur sehr grobe Abschätzungen, z.B. wenn sich alle Delays auf einer Kante befinden und die *SWT* ganzzahlig in jeder Schleife ist (dies entspricht ganzzahligen Vielfachen der komfortablen Markierung *CM*) [42]. Eine weitere Abschätzung [41] benutzt statistische Verweilzeiten von Tokens in Places. Es ist jedoch keine enge Grenze bekannt, die unabhängig von der Summe der Delays *WST*, erreicht werden kann. Weiterhin ist unbekannt, ob und wie man diese Grenze immer erreichen kann. Diese offenen Probleme waren die Motivation für die folgenden Untersuchungen.

8.2 Die Iteration Bound für Einheitsraten-Graphen

In Einheitsratengraphen wird jeder Knoten nur einmal pro Periode aktiviert. Außerdem ist hier immer *HDM*=0 und *LLM*=*CM*=1. Die Lebendigkeit ist also gesichert, wenn in jeder Schleife mindestens ein Delay vorhanden ist.

Für Einheitsratengraphen existiert ein geschlossener Ausdruck für die Iterationsperiodengrenze {iteration period bound} [19],[24],[37]:

$$T_{ib} = \mathop{\mathbf{Max}}_{\substack{alleSchleifen \\ l \in L}} \left\{ \frac{T_l}{D_l} \right\} \tag{8.1}$$

Dabei ist T_l die Summe der Ausführungszeiten aller Knoten und D_l die Summe der Delays auf der Schleife l. Wie schon früher erwähnt, ist für diese Grenze erst einmal eine unbegrenzte Anzahl von Prozessoren anzunehmen. Für jede Schleife l des Graphen gibt es eine Schranke [19], die zur obigen Gleichung führt: {loop bound inequality}

$$T_l \leq D_l \cdot T_{ib} \tag{8.2}$$

Die Schleife, für die T_l/D_l maximal ist, erfüllt die Ungleichung mit Gleichheit und wird kritische Schleife {critical loop} genannt. Die Differenz zwischen der *iteration bound* und der *loop bound* wird *slack time* genannt [24]. Sie ist für die kritische Schleife Null.

Trotz des bekannten Ausdrucks für die Schranke ist in der Praxis die Bestimmung derselben nicht trivial, da die Anzahl der zu testenden verschiedenen Schleifen stärker als polynomial in n=|V| und |E| wachsen kann [31]. Nur die Anzahl der *fundamentalen* Schleifen ist mit |E|-|V|+1 polynomial. Ein Algorithmus, der alle c elementaren Schleifen findet, ist mit einem Zeitaufwand von O((|V|+|E|)(c+1)) pseudopolynomial durchführbar [64].

Praktische Ansätze verwenden eine ‘Shortest Path’-Methode [32],[31] mit einer Gesamtkomplexität O(n^3log$n$) oder zur Schätzung für stochastische Netze eine Zerlegung in Untersysteme [47].

8.3 Die Iterationsperiode für einen Prozessor

Für nur einen Prozessor (dessen Auslastung dann 100% beträgt) ergibt sich für den Single- und Multiratenfall eine Iterationsperiode von

$$T = \vec{q} \cdot \vec{t} \tag{8.3}$$

Alle Tasks werden hier sequentiell ausgeführt. Der t-Vektor enthält in seiner i.ten Zeile die Ausführungszeit von Knoten v_i.

8.4 Die ‘Processor-Bound’

Die Mindestanzahl benötigter Prozessoren ist im Einheitsratenfall gegeben durch [22]:

$$P_0 = \left\lceil \frac{\sum_{v \in V} t_v}{T_{ip}} \right\rceil \tag{8.4}$$

Diese als ‘processor bound’ bekannte Grenze hängt von der Dauer der erwünschten Iterationsperiode T_{ip} und den Knoten-Ausführungszeiten t_v ab. Eine Erweiterung für multiple Raten muß die notwendigen Aktivierungen q_v jedes Knotens $v \in V$ berücksichtigen:

$$P_0 = \left\lceil \frac{\sum_{v \in V} q_v t_v}{T_{ip}} \right\rceil \tag{8.5}$$

Bei gegebener Prozessoranzahl ist umgekehrt auch die Iterationsperiode beschränkt durch:

$$T_{ip} \geq \frac{\sum_{v \in V} q_v t_v}{P} \tag{8.6}$$

8.5 Auslastung

Ein periodisches Schedul, das in einer Periode *J* Iterationen bearbeitet und das auf *P* Prozessoren eine Zeit *T* pro Periode benötigt, führt zu einer Auslastung der Prozessoren von

$$Auslastung = \frac{J \cdot \vec{q} \cdot \vec{t}}{P \cdot T} \cdot 100\% \tag{8.7}$$

8.6 Schwierigkeiten im Multiratenfall

Die Größen, die für Einheitsraten-Graphen nur aus Nullen und Einsen bestehen (u.a. *q*, *y*, *C*, *LLM*, *HDM*, *CM*), haben im allgemeinen von Eins verschiedene Werte. Der *q*-Vektor gibt dann für einen Knoten beispielsweise eine Aktivierungshäufigkeit von 128 an, für einen anderen 48 usw.; eine Iteration des Scheduls muß nun genau so viele Aktivierungen enthalten. Man weiß nun aber nicht, wieviel Aktivierungen gleichzeitig durchgeführt werden können, da man die Delayverteilung nicht zu jedem Zeitpunkt kennen kann, außer empirisch durch versuchsweises Scheduling (mit entsprechend hoher Komplexität; wobei nicht gesichert ist, daß man das optimale Schedul findet). Weiterhin weiß man nicht, wieviel Iterationen zu einer Periode zusammengefaßt werden müssen, um ein optimales Schedul zu erreichen. Außerdem hängt der maximale Durchsatz nicht, wie im Einheitsratenfall, linear von der Delaysumme in der kritischen Schleife ab. Hinzu kommt noch, daß die Delayverteilung und nicht nur die gewichtete Summe einen Einfluß auf die Performance hat. Bei mehreren gekoppelten Schleifen ist die Grenze möglicherweise nicht mehr gleich der der kritischen Schleife allein, da die Kopplungen komplexer sind.

Im Folgenden wird immer nur eine Schleife betrachtet. Systeme mit mehreren Schleifen können maximal den Durchsatz einer einzigen Schleife haben, daher ist eine Grenze für eine Schleife auf jeden Fall auch eine Grenze für den gesamten Graph.

Vorsicht ist dann angebracht, wenn die Elemente des *q*-Vektors für die Knoten einer Schleife im gekoppelten System sich von den Elementen der herausgetrennten Schleife unterscheiden. Man sollte die Elemente des ursprünglichen *q*-Vektors für das Schedul benutzen, damit die benötigte Zeit für eine Iteration vergleichbar bleibt.

8.7 Maximale Anzahl von Prozessoren

Eine Abschätzung für die maximal mögliche Anzahl von Aktivierungen eines Knotens ist möglich, wenn man alle Delays der Schleife (*WST*) auf der Eingangskante *e* dieses Knotens *v=Zielknoten*(*e*) gesammelt annimmt (mehr gibt's nicht). Dividiert man diese (*D*(*e*)) durch die

Eingangsrate $I(v)$, dann kann der Knoten v maximal so oft gleichzeitig (parallel) ausgeführt werden:

$$\mathbf{max}\ Aktivierungen(v) = \left\lfloor \frac{D(e)}{I(v)} \right\rfloor \tag{8.8}$$

$D(e)$ berechnet sich aus der *WST* unter Berücksichtigung des P-invarianten Vektors $\vec{y}$ an dieser Kante ($y(e)$), daher ist

$$\mathbf{max}\ Aktivierungen(v) = \left\lfloor \frac{\lfloor WST / y(e) \rfloor}{I(v)} \right\rfloor \tag{8.9}$$

Für Normschleifen ($y(e)$=1) vereinfacht sich der Ausdruck wegen D_{sum}=*WST*=ΣD_e zu

$$\mathbf{max}\ Aktivierungen(v) = \left\lfloor \frac{\sum D_e}{I(v)} \right\rfloor = \left\lfloor \frac{D_{sum}}{I(v)} \right\rfloor \tag{8.10}$$

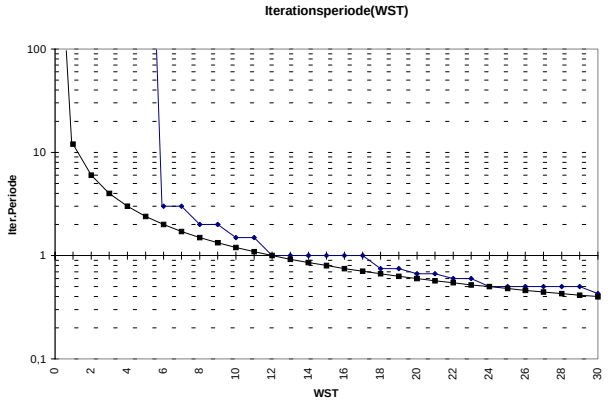
Daraus kann man eine Obergrenze für die Anzahl benötigter Prozessoren angeben, indem das Maximum über alle Knoten gebildet wird:

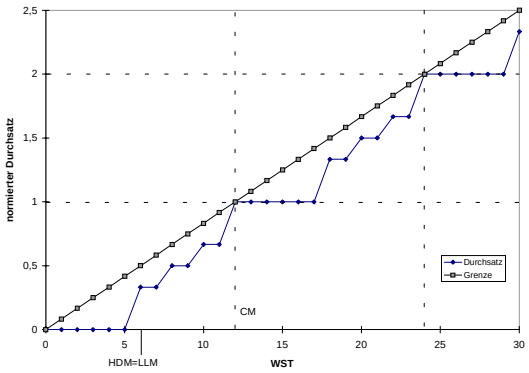
$$\begin{aligned} &\mathbf{max}\ AnzahlProzessoren = \max_{v \in V} \{\mathbf{max}\ Aktivierungen(v)\} \\ &= \max_{v \in V} \left\{ \left\lfloor \frac{\lfloor WST / y(Eingangskante(v)) \rfloor}{I(v)} \right\rfloor \right\} \end{aligned} \tag{8.11}$$

Diese Grenze gilt natürlich auch, wenn nicht alle Delays auf einer Kante konzentriert sind, da dann die Anzahl parallel aktivierbarer Knoten kleiner oder gleich der Anzahl der Aktivierungen sein muß, die bei Anhäufung der Delays vor dem Knoten mit minimalem $y(e)*I(v)$ möglich ist.

8.8 Schranken für den Durchsatz

Die Iterationsperiode eines Scheduls für einen zyklischen Graphen hat eine untere Grenze, die als *iteration period bound* für den Einheitsratenfall bekannt ist (Bild links). Da sich der Durchsatz als Kehrwert der Iterationsperiode für die Untersuchungen als praktischer erwiesen hat, wird dieser im Folgenden betrachtet (Bild rechts). Er ist dann nach oben durch eine Durchsatzgrenze beschränkt.





8.8.1 Stand der Forschung

Die Iterationsperiodengrenze {system cycle time} kann nach Chao [38] unter folgenden Voraussetzungen berechnet werden (und ist dann erreichbar {tight}):

- In jeder Schleife eines 'nonordinary marked graph' ist nur ein Place markiert, d.h. im Datenflußkonzept sind alle Delays einer Schleife auf einer Kante versammelt.
- Die Summe der gewichteten Tokens, *SWT*, ist in jeder Schleife *k* eine natürliche Zahl. Ausgedrückt mit hier verwendeten Begriffen ist die gewichtete Delaysumme *WST* ein ganzzahliges Vielfaches N_K der komfortablen Markierung *CM*.

Zusammenfassend ergibt sich dann für die Iterationsperiodengrenze mit $T_K=\Sigma t_i$ (Summe der Ausführungszeiten der Knoten in der Schleife *k*):

$$T_{ib} = \max_K \left\{ \frac{T_K}{N_K} \right\} = \max_K \left\{ \frac{T_K}{\lfloor WST/CM \rfloor} \right\} \tag{8.12}$$

Der rechte Ausdruck ist eine mit den vertrauteren Begriffen ausgedrückte Erweiterung, die davon ausgeht, daß für $(n\text{-}1)CM<WST\leq nCM$ die Zeit nicht besser sein kann als für $WST=nCM$. Die notwendigen Bedingungen sind in der Praxis sehr selten anzutreffen, daher ist diese Darstellung nicht sehr brauchbar.

Eine andere Grenze für die Iterationsperiode, jedoch keine 'tight bound', gibt Murata für 'Timed Petri-Nets' [37]. Dabei ist jeder Transition (Knoten) oder jedem Place (Kante) eine Verarbeitungszeit zugeordnet. Hier werden nur Knoten mit Ausführungszeit betrachtet.
Die Knoten-Ausführungszeiten t_i werden zu einer Diagonalmatrix **T** zusammengefaßt (Im Folgenden werden übliche Formelzeichen benutzt) und damit ein Ressource-Time-Product (RTP) als Vektor $\vec{R}$ gebildet, der für jede Kante angibt, wie lange reservierte[1] Delays dort während einer Iteration verweilen:

$$\mathbf{T} = \begin{pmatrix} t_1 & 0 \cdots & 0 \\ 0 & & \vdots \\ \vdots & \ddots & \\ 0 & \cdots 0 & t_n \end{pmatrix}, \qquad \vec{R} = \Gamma^- \cdot \mathbf{T} \cdot \vec{q} \tag{8.13}$$

Durchschnittlich befinden sich auf den Kanten während einer Iteration (T_{ip}) mehr Delays ($\vec{\bar{d}}$) (reservierte und nicht-reservierte). Dies führt zu einem RTP von

$$\vec{\bar{d}} \cdot T_{ip} \geq \vec{R} = \Gamma^- \cdot \mathbf{T} \cdot \vec{q} \tag{8.14}$$

Wegen der Invarianz $\vec{y}^T \cdot \vec{\bar{d}} = \vec{y}^T \cdot \vec{d}_0$ ergibt die Multiplikation mit $\vec{y}^T$ (Der Index *k* steht für eine der Schleifen des Graphen):

$$\begin{aligned} &\vec{y}_k^T \cdot \vec{d}_0 \cdot T_{ip} \geq \vec{y}_k^T \cdot \Gamma^- \cdot \mathbf{T} \cdot \vec{q} \\ &\Rightarrow \quad T_{ip} \geq \vec{y}_k^T \cdot \Gamma^- \cdot \mathbf{T} \cdot \vec{q} \,/\, \vec{y}_k^T \cdot \vec{d}_0 \end{aligned} \tag{8.15}$$

[1] Reservierte Delays sind die Werte, die der nachfolgende Knoten schon eingelesen hat. Für das hier erwähnte Konzept bleiben diese jedoch während der Bearbeitung des Knotens auf der Eingangskante.

Die letzte Ungleichung stellt eine Grenze für die Iterationsperiode dar, wenn das Maximum über alle Schleifen *k* gebildet wird.
Dieser Ausdruck läßt sich aber einfacher mit unseren Abkürzungen darstellen. Betrachtet man dazu nun jeweils einzelne Schleifen und erkennt, daß

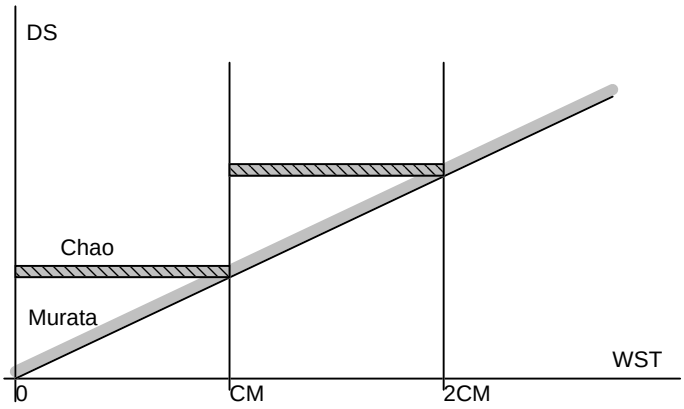
$$\begin{aligned} \vec{y}^T \cdot \vec{d}_0 &= WST \\ \vec{y}^T \cdot \Gamma^- \cdot \vec{q} &= CM \cdot \vec{1} \end{aligned} \tag{8.16}$$

dann ergibt sich für die Iterationsperiode T_{ip} oder den Durchsatz *DS*:

$$T_{ip} \geq \frac{CM \cdot \sum t_i}{WST} \quad , \quad DS \leq \frac{WST}{CM \cdot \sum t_i} \tag{8.17}$$

Diese Grenze wächst linear mit *WST* und wird durch eine alternative Betrachtung in den folgenden Abschnitten bestätigt.

Die beiden bekannten Grenzen werden in der Grafik visualisiert:



8.8.2 Erweiterungen

Die von Chao gegebene Grenze erfordert, daß sich alle Delays einer Schleife auf einer Kante befinden. Durch Retiming kann genau so eine Verteilung erreicht werden, sofern keine NMD's vorhanden waren und die Schleifen nicht zu stark gekoppelt sind. Die Grenze ist dann auch für eine größere Klasse von Graphen gültig.

Bei Konzentration der Delays auf eine Kante können Vielfache von *CM* auch der *WST* abgespalten werden. Jedes vollständige CM hat einen gleichmäßigen Zuwachs des Durchsatzes um $1/\Sigma t_i$ zur Folge. Die restlichen Delays können den Durchsatz nur so verbessern, als seien sie alleine auf der Kante, d.h. für $WST_{\text{Rest}} = WST \,\mathbf{mod}\, CM < LLM$ ist keine weitere Verbesserung zu erzielen.

Die Zuwächse des Durchsatzes sind periodisch über *WST* mit der Periode *CM*. Wenn die Durchsatzgrenze *DG=f*(*WST*) für 0<*WST*<*CM* bekannt ist, dann ist sie damit auch für größere *WST* bestimmt:

$$DG(WST) = \left\lfloor \frac{WST}{CM} \right\rfloor \cdot \frac{1}{\sum t_i} + DG(WST \,\mathbf{mod}\, CM) \tag{8.18}$$

8.8.3 Normierung

Die Ausdrücke für die Iterationsperiode und den Durchsatz enthalten Σt_i als linearen Faktor. Am besten normiert man den Durchsatz, um einen allgemeingültigen Ausdruck zu bekommen:

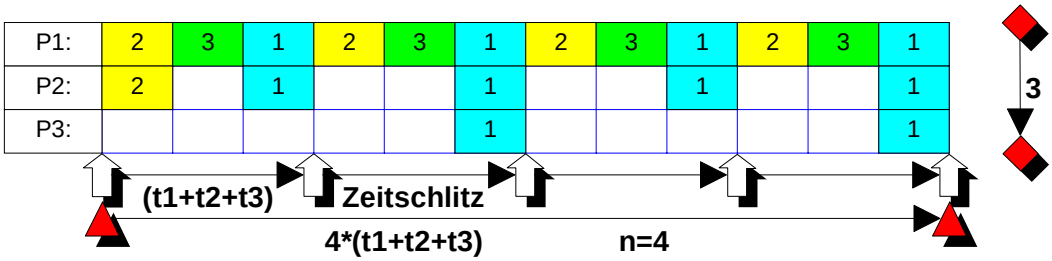
$$normierter\ Durchsatz = Durchsatz \cdot \sum_{v \in L} T_v \tag{8.19}$$

8.8.4 Neue Ansätze

Wenn alle Delays auf einer Kante konzentriert[2] sind, kann immer nur der nachfolgende Knoten aktiviert werden, dann der nächste usw., aber keine zwei verschiedenen Knoten gleichzeitig parallel. Daher muß ein periodischer Abschnitt des Scheduls ein ganzzahliges Vielfaches der Summe der Ausführungszeiten (Zeitschlitz) dauern, $n*\Sigma T_v$. In dieser Zeit kann eine ganze Zahl m von Iterationen (jeweils q_v Aktivierungen von Knoten v) abgearbeitet werden.

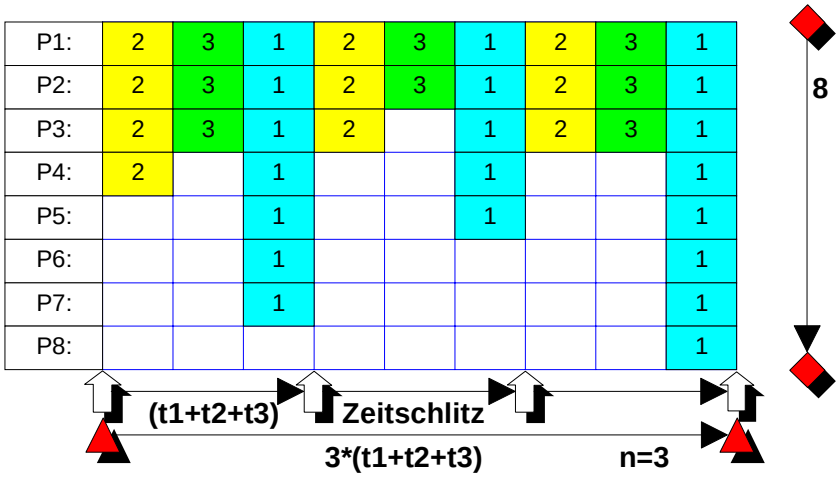
Man kann sich m als Packungsfaktor vorstellen, ähnlich dem Unfolding- oder Blocking-Faktor, denn genauso wie bei diesen Verfahren im Einheitsratenfall werden mehrere Iterationen zu einer Periode zusammengefaßt, die sich dann regelmäßig im Schedul wiederholt. Innerhalb der Periode muß kein periodisches Muster erkennbar sein. Durch die Zusammenfassung mehrerer (J=m) Iterationen wird ein dichtes nicht-überlappendes Schedul erreicht.

Im Bild ist ein Schedul für eine Schleife mit $\vec{q}^T$=(10,5,4) zu sehen. Es werden drei Prozessoren benötigt; bei einem Packungsfaktor m=1 dauert eine Periode (=Iteration) n=4 Zeitschlitze á (t_1+t_2+t_3) Sekunden.



Im nächsten Bild ist wieder ein Schedul für die Schleife mit $\vec{q}^T$=(10,5,4) zu sehen. Es werden acht Prozessoren benötigt; bei einem Packungsfaktor m=2 dauert eine Periode (=2 Iterationen) n=3 Zeitschlitze á (t_1+t_2+t_3) Sekunden.

[2] Die Konzentration aller Delays auf eine Kante führt zu schnelleren Scheduls im Vergleich zu anderen Delayverteilungen, siehe Kapitel „Retiming“



Die Iterationsperiode und deren Grenze muß also darstellbar sein als rationale Gewichtung der Summe der Ausführungszeiten:

$$iteration\,bound = T_{ib} = \frac{n \cdot \sum_{v \in L} T_v}{m}, \text{ normiert: } \frac{n}{m} \tag{8.20}$$

Die Durchsatzgrenze muß dann als Kehrwert so darstellbar sein:

$$Durchsatzgrenze = DG = \frac{m}{n \cdot \sum_{v \in L} T_v}, \text{ normiert: } NDG = \frac{m}{n} \tag{8.21}$$

Wenn während eines m Perioden dauernden Abschnitts $m*q_v$ Aktivierungen des Knotens v geschehen müssen, dann benötigt man dafür mindestens n_v Zeitschlitze á ΣT_v Sekunden:

$$n_v = \left\lceil \frac{m \cdot q_v}{\mathbf{max}\ Akt(v)} \right\rceil = \left\lceil \frac{m \cdot q_v}{\left\lfloor \frac{\lfloor WST/y(e) \rfloor}{I(v)} \right\rfloor} \right\rceil, \tag{8.22}$$

wobei $e=E(v)$ die Eingangskante von v ist. Das Maximum über alle Knoten gibt die obere Grenze für die tatsächlich benötigten Zeitschlitze an:

$$n(m) = \mathbf{max}_{v \in L}\{n_v\} = \mathbf{max}_{v \in L}\left\{\left\lceil \frac{m \cdot q_v}{\mathbf{max}\ Akt(v)} \right\rceil\right\} = \mathbf{max}_{v \in L}\left\{\left\lceil \frac{m \cdot q_v}{\left\lfloor \frac{\lfloor WST/y(E(v)) \rfloor}{I(v)} \right\rfloor} \right\rceil\right\} \tag{8.23}$$

Dadurch ist die normierte Durchsatzgrenze *NDG* gegeben durch

$$NDG(m) = \frac{m}{n} = \frac{m}{\max\limits_{v \in L}\left\{\left\lceil \frac{m \cdot q_v}{\left\lfloor \frac{\lfloor WST/y(E(v)) \rfloor}{I(v)} \right\rfloor} \right\rceil\right\}} \tag{8.24}$$

Bei Kenntnis von m läßt sich der Durchsatz dadurch sehr genau abschätzen. In den meisten Fällen ist diese Grenze ‘tight’. In der Regel wird das Schedul mit wachsendem m komplexer und beansprucht einen zu m und Σq proportionalen Speicherplatz im Speicher des DSP. Daher wird man häufig ein m oder maximales m_{max} vorgeben. Die Grenze läßt sich dann ziemlich genau mit obiger Formel oder dem Maximum über alle m bis m_{max} berechnen:

$$NDG \le \max_{m \in [1; m_{max}]} NDG(m) \tag{8.25}$$

Bei Unkenntnis müßte man die Abschätzung gröber vornehmen; dazu liefert der Grenzwert für unendliches m:

$$NDG \le \lim_{m\to\infty} \frac{m}{n(m)} = \lim_{m\to\infty} \frac{m}{m \cdot \max\limits_{v \in L}\left\{\left\lfloor \frac{q_v}{\left\lfloor \frac{WST/y(E(v))}{I(v)} \right\rfloor} \right\rfloor\right\}} = \frac{1}{\max\limits_{v \in L}\left\{\left\lfloor \frac{q_v}{\left\lfloor \frac{WST/y(E(v))}{I(v)} \right\rfloor} \right\rfloor\right\}}$$

$$= \min_{v \in L}\left\{\frac{\left\lfloor \frac{WST/y(E(v))}{I(v)} \right\rfloor}{q_v}\right\} \le \min_{v \in L}\left\{\frac{WST/y(E(v))}{I(v) \cdot q_v}\right\} = \min_{v \in L}\left\{\frac{WST}{CM}\right\} = \frac{WST}{CM} \tag{8.25}$$

Diese Grenze wächst linear mit *WST* und ist in den folgenden Graphen mit „Grenze“ betitelt eingezeichnet. Sie entspricht genau der bei Murata [37] erwähnten Grenze.

Eine Interpretation dieser (linearen) Grenze ist möglich: Wenn man rationale (nicht ganze) Aktivierungen von Knoten zuläßt, so daß z.B. ein Knoten mit Eingangsrate 4 durch 2,5-fache Aktivierung 10 Delays von seiner Eingangskante einlesen könnte, dann werden immer sämtliche Delays verbraucht und das Schedul würde diese lineare Grenze als Durchsatz aufweisen. Die Differenz zwischen linearer und realer Grenze kommt also durch unbenutzte Delays zustande.

Problematisch ist zur Bestimmung der engeren Grenze die Unkenntnis des optimalen m. Dieses müßte sich, wie J_{opt}, aus dem äquivalenten Einheitsraten-Graph bestimmen lassen; diese Vorgehensweise ist aber wegen der Komplexität ausgeschlossen. In der Praxis wird man aber den Packungsfaktor vorgeben, den man verwenden möchte, und erhält dann einen geeigneten Richtwert für den Durchsatz.

Die angegebene Grenze dürfte die engste Beschränkung für den Durchsatz in allgemeiner Form (analytisch) darstellen. Seltene Abweichungen von der ‘tight’-Grenze ergeben sich

durch unbenutzte Delays, die sich nur bei genauer Analyse des speziellen Graphen feststellen lassen.

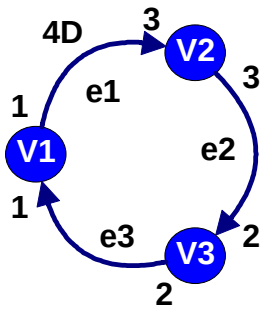
Im Bereich *WST*<*LLM* kann durch die Formel nicht immer der exakte Wert Null berechnet werden. Dort ist der Graph nicht lebendig. Die Bestimmung von *LLM* muß daher durch andere Verfahren geschehen.

Im Bereich *WST*∈[0; *CM*[steigt das optimale *m* monoton von 0 (dead) über 1 (bei *LLM*) bis zu einem Maximalwert an. Es ist nur dieser Bereich zu betrachten, da sich ab *WST*=*CM* der Zuwachs des Durchsatzes periodisch wiederholt. Der Wert *LLM* kann nicht mit obiger Formel ermittelt werden. Für *WST* zwischen *LLM* und dem Mittelwert zwischen *LLM* und *CM* ist *m*=1, denn es sind zuwenig Delays vorhanden, um viele parallele Aktivierungen durchzuführen; ein Vorteil durch Zusammenfassung mehrerer Iterationen ist nicht erzielbar.

Für *WST*≈(*CM*+*LLM*-1)/2 (Mittelwert) ist der normierte Durchsatz immer ½ (*n*=2). Experimente ergaben außerdem, daß der Verlauf des Durchsatzes *DS*=*f*(*WST*) punktsymmetrisch um diesen Mittelwert zu sein scheint.

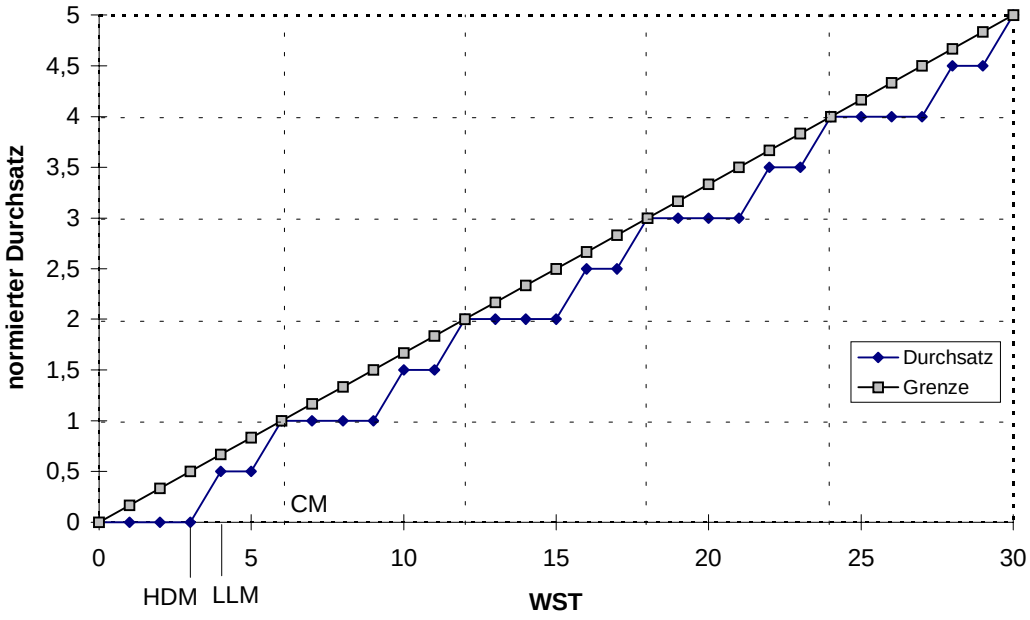
Anhand einiger Beispielgraphen sollen das Verhalten des Durchsatzes vermittelt werden.

Für den folgenden Graphen ist $\vec{q} = \begin{pmatrix} 6 \\ 3 \\ 2 \end{pmatrix}$, *LLM*=4, HDM=3, *CM*=6.

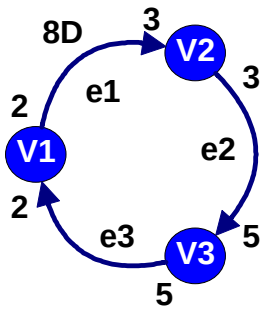


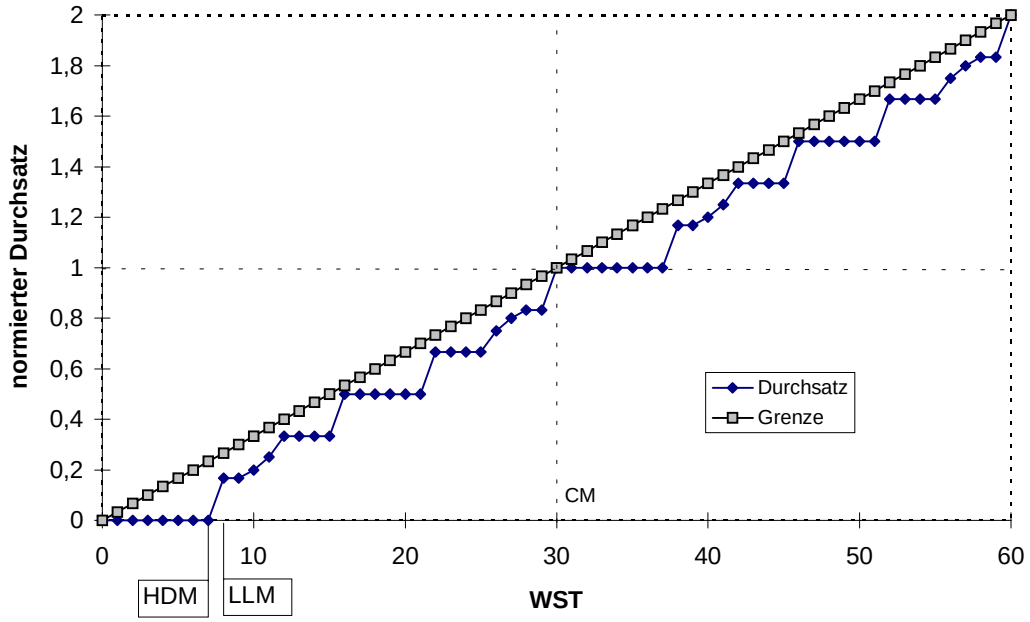
Es wurde der optimale normierte Durchsatz *nDS* in Abhängigkeit von *WST* empirisch durch Scheduling ermittelt. Zum Vergleich wurde die normierte Durchsatzgrenze *NDG*(*m*) nach der genauen und der linearen Grenze berechnet. *NDG*(*m*) stimmt hier exakt. Im Graph sind *nDS*(*WST*)=*nDG*(*WST*) und die lineare Grenze eingezeichnet. Die folgende Tabelle gibt die berechneten und durch Scheduling ermittelten Werte in Abhängigkeit der *WST* an:

WST	0	1	2	3	4	5	6	7	8	9	10	11	12
nDS	0	0	0	0	1/2	1/2	1	1	1	1	3/2	3/2	2
m	0	0	0	0	1	1	1	1	1	1	3	3	2
n(m)	-	-	-	-	2	2	1	1	1	1	2	2	1
NDG(m)	0	0	0	0	1/2	1/2	1	1	1	1	3/2	3/2	2
linear	0	1/6	2/6	3/6	4/6	5/6	1	7/6	8/6	9/6	10/6	11/6	2



Für den nächsten Graph mit $\vec{q}^{\,T}$=(15,6,10), *HDM*=7, *LLM*=8, *CM*=30 treten neun Stufen unterhalb *CM* auf. Bis *WST*=21 ist *m*=1, danach in jeder Stufe um eins höher. Die enge Grenze *NDG*(*m*) liefert genau diese Werte außer für *WST*=10; dort ist *DS*=1/5, aber NDG=1/4. Die Symmetrie um *DS*=1/2 @ *WST*=18,5 ist deutlich erkennbar.



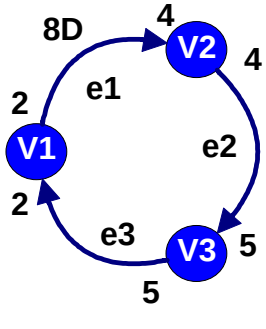


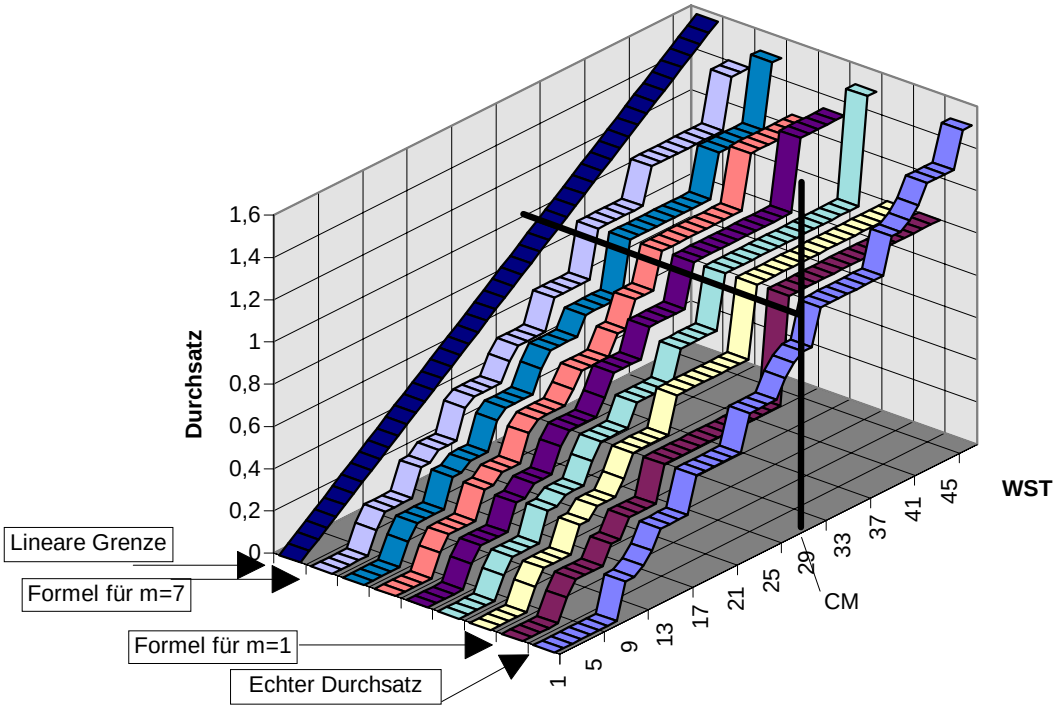
Der folgende Graph zu dem oben vorgestellten Beispiel zeigt die echte Durchsatzgrenze für optimales *m* (erster Graph), danach die Grenzen nach der *m*/*n*-Formel für *m* zwischen 1 und 7, sowie die lineare Grenze *WST*/*CM*.

Man erkennt die Monotonie von $NDG_m(WST)$. Weiterhin wird verständlich, daß *NDG* nicht monoton über *m* ist, d.h. es existiert ein optimales *m* in Abhängigkeit von *WST*. Für unendliches *m* nähert sich *NDG*(*m*) dem Maximum.

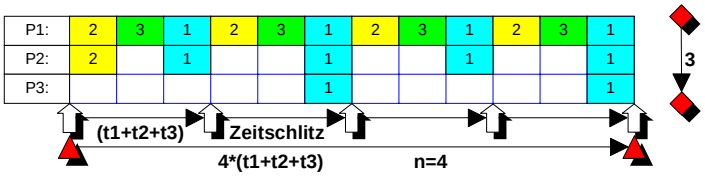
Im Bereich *WST*<*LLM* sind die Differenzen zwischen realem Durchsatz, der hier Null ist, weil der Graph nicht lebendig ist, und den Ergebnissen der Formeln erkennbar. Die Größe *LLM* muß daher immer anders ermittelt werden.

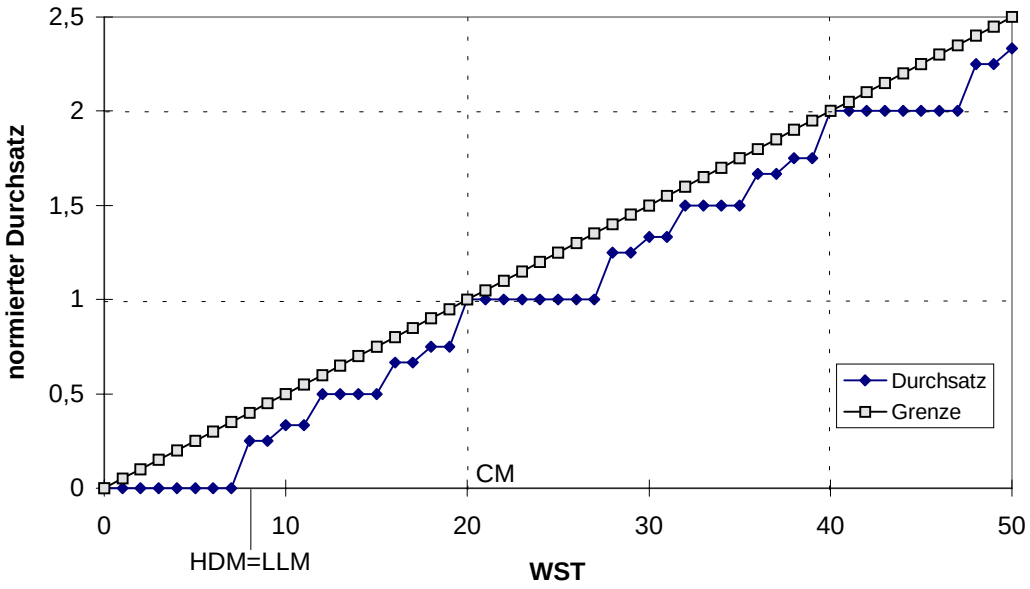
Je nach gewünschtem Packungsfaktor *m* ist eine der Grenzen *NDG*(*m*) für die Praxis brauchbar. Sind mehrere Packungsfaktoren denkbar, dann könnte man durch Betrachtung der zugehörigen Grenzen das optimale *m* bestimmen.





Der nebenstehende Graph mit $\vec{q}^{\,T}$=(10,5,4), *HDM*=*LLM*=8, *CM*=20 stellt auch kein Problem dar. Beispielhaft ist hier das Schedul für *WST*=8 angegeben.





8.9 Anwendungen

Die neu bestimmten Formeln für die Grenze des Durchsatzes stellen die engste bisher bekannte Grenze für den Durchsatz von Multiratensystemen dar. Sie ist in den meisten Fällen oberhalb der minimalen lebendigen Delaysumme (*LLM*) ‘tight’. Die Ausnahmefälle sind durch die Anomalien im Multiratenfall gegeben (unbenutzte Delays, d.h. Tokens, die nicht zur Aktivierung des nachfolgenden Knotens reichen).

Die Annahme, daß sich die Delays auf einer Kante der Schleife befinden, führt zu der oberen Grenze des Durchsatzes. Durch Retiming kann meistens eine solche Verteilung erreicht werden (siehe Kapitel 7.4). Wenn dies in Einzelfällen nicht möglich sein sollte, so stellt die vorgestellte Formel dennoch eine obere Grenze dar; lediglich der Abstand zum realen Durchsatz erhöht sich etwas.

Für den praktischen Gebrauch reicht diese Grenze als Abschätzung aus. Oft ist sowieso die genaue Ausführungszeit eines Blockes nicht bekannt, so daß die geringe Unsicherheit keine rolle spielt.

Die Kenntnis des Packungsfaktors, der angibt, wieviel Iterationen in einer Periode zusammengefaßt werden, ist für die praktische Implementation gegeben. Meistens ist der Speicherplatz zur Speicherung des Schedulmusters begrenzt, daher wird man diesen Faktor so klein wie möglich halten; er geht linear in den Speicherbedarf für das Schedul ein.

Die Grenze ergab sich aus der Betrachtung einer einzelnen Schleife. Für einen Graphen mit mehreren Schleifen ist der Durchsatz der langsamsten Schleife entscheidend, d.h. der minimale Durchsatz aller Schleifen bestimmt den Durchsatz des Gesamtsystems.

9 Implementationen

9.1 Objektorientierte Programmierung in C++

In den letzten Jahren hat sich in der Softwaretechnologie eine Wandlung vollzogen, die bessere Ansätze zur Wiederverwendbarkeit und Flexibilität von Code verwirklicht, als die klassische Programmierstrategie mit Prozeduren und Funktionen und Benutzer-‘Abfertigung’ in Eingabeschleifen. Das Konzept der objektorientierten Programmierung [54] verbessert zudem die Kapselung von Daten und Code (Information hiding). Das Objekt als programmtechnische Datenstruktur besteht aus *Methoden* und *Datenfeldern*; sie enthält zwei Arten dieser Informationen: *private* und *öffentliche*. Erstere sind für den Zugriff von außen gesperrt, letztere sind frei manipulierbar. Der optimale Stil wäre die Programmierung nur privater Datenfelder und teilweise öffentlicher/privater Methoden, damit ist der Zugriff auf die Daten nur über zugeordnete Methoden möglich. Eine mißbräuchliche Manipulation ist dadurch weitestgehend ausgeschlossen. Die Instanziierung von Objekten geschieht über *Konstruktoren*, die Beseitigung dynamischer oder lokaler Objekte wird durch *Destruktoren* vorgenommen.

Die *Vererbung* spielt eine große Rolle für die Erweiterung bereits vorhandenen Codes. Objekte können einfach durch neue Methoden und Daten ergänzt werden, ohne daß das Objekt seine früheren Eigenschaften verliert. *Virtuelle* Methoden sind als solche deklarierte Prozeduren oder Funktionen, die in abgeleiteten Objekten die früheren virtuellen Methoden überschreiben, jedoch werden bei Typumwandlung (z.B. bei Parameterübergabe als Funktionsargument) nicht die alten Methoden aufgerufen, sondern die mit dem neuen Objekt verknüpften Methoden. Dies wird durch dynamische Bindung und Referenzierung über eine virtuelle Methodentabelle erreicht. Durch die flexible Anbindung der Methoden ist *Polymorphie* möglich, die Erscheinung eines Objekts in verschiedener Weise, je nach Verwendung. Ein Objekt kann so immer als ein Objekt der übergeordneten Klasse betrachtet werden, bei Bedarf aber seine besonderen Fähigkeiten ausspielen.

Die Sprache C++ [52],[53],[54] ist eine auf die objektorientierte Programmierung zugeschnittene Erweiterung der Sprache C [51]. „Erweiterung“ bedeutet insbesondere, daß im alten Stil geschriebene C-Programme weiterhin mit C++ compiliert werden können. Das wäre allerdings ganz und gar nicht im Sinne des Erfinders: Ein Programm sollte von vorne bis hinten objektorientiert aufgebaut sein. Dazu gehört im Idealfall, daß das gesamte Programm ein (Applikations-)objekt darstellt, dessen Methoden und Event-Handler die Struktur des Programms aufbauen.

Neben C++ gibt es auch objektorientierte Weiterentwicklungen von Pascal und die Sprache Eiffel, mit denen objektorientierte Programmierung möglich ist.

9.2 Klassenbibliotheken

Die Vererbung vereinfacht die Wiederverwendung von Code erheblich („Code-Recycling“). Herkömmliche Bibliotheken für C, die Funktionen und Datentypen bereitstellen, lassen keine Änderungen an den Datentypen mehr zu. Die Funktionen sind starr für einen speziellen Zweck programmiert. Wollte man z.B. statt einem linearen Speicherbereich eine Verkettete Liste zur Speicherung einer Datenmenge nutzen, dann müßten sämtliche Routinen für diesen Datentyp neu geschrieben werden.

Für C++ sind sogenannte Klassenbibliotheken erhältlich oder leicht selbst zu implementieren. Ziel ist es, zu einem Problem passende Objektklassen anzubieten, die die wichtigsten Datentypen und Methoden enthalten, die dem Problem angepaßt sind. Beispielsweise sind Klassen *komplexe_Zahl, Matrix, Bildschirmfenster, Graph, String* nützliche Erweiterungen. Mit diesen läßt sich dann so leicht umgehen, wie mit fundamentalen Datentypen. Bei Bedarf kann der anwendende Programmierer die Klassen erweitern und für ihn wichtige Methoden oder Felder ergänzen, z.B. *Matrix::Gauss_Jordan*. Einzigartig ist in C++ die Möglichkeit *Templates* (Schablonen) zu verwenden. Diese erlauben es, Datentypen erst bei der Compilation festzulegen. Die Klasse mit ihren Methoden wird im Quelltext nur einmal geschrieben, kann aber in der Deklaration des Benutzers mit irgendeinem sinnvollen Typ gebildet werden, z.B. *Vektor<int>, Vektor<float>, Vektor<Farbe>* u.s.w. .

9.3 Die LEDA-Klassenbibliothek

Eine im Rahmen der Diplomarbeit eingesetzte Klassenbibliothek ist die am Max-Planck-Institut für Informatik in Saarbrücken entwickelte Software LEDA[1] [55] in der Version 3.0. Es sind zahlreiche Datentypen schon implementiert, die zum Standard der Softwareentwicklung gehören, beispielsweise Strings, Listen, Stacks, Sequenzen, Queues (FIFO), aber auch Graphen (gerichtet oder ungerichtet und parametriert) und graphische Objekte (Geraden, Linien, Kreise, Punkte). Zahlreiche angebotenen Algorithmen operieren auf diesen Datentypen oder sind Teil der Objekte (Methoden). Viele Standardaufgaben, wie z.B. Such- und Sortieralgorithmen, Einfügen und Löschen von Daten, Ein- und Ausgabedienste sind in benutzerfreundlicher Weise (aus Sicht des Programmierers) implementiert. Es sind aber auch sehr spezielle Algorithmen, z.B. zur Berechnung von Voronoi- oder Delaunay-Diagrammen enthalten.

Ein großer Vorteil dieser C++-Implementation ist die häufige Zulassung von Template-Datentypen als Parameter für Objekte und Funktionen. Dadurch kann man bei der Deklaration von Objekten noch den verwendeten Datentyp angeben. Als Beispiel ist Liste<float> genauso möglich wie Liste<Stack>. Besonders ist der Graph-Datentyp zu gebrauchen (er besteht aus Listen von Typ Node und Edge), da seinen Knoten und Kanten Template-Parameter zugewiesen werden können, die sich zur Modellierung erweiterter Graphen eignen (davon wird in den entwickelten Programmen ausgiebig Gebrauch gemacht).

Als sehr brauchbar erweisen sich Iterator-Makros, die in einer Programmzeile einen Durchlauf durch eine Datenstruktur ermöglichen, z.B. läuft `forall_nodes(v,G) {...}` durch alle Knoten eines Graphen.

Ein recht komfortables Interface zur X-Window-Oberfläche unter UNIX ermöglicht es, Ein- und Ausgaben in einem graphischen Fenster vorzunehmen. Dazu stehen Routinen zur Feststellung von Mausaktionen und zur Ausgabe von Text- und Grafikobjekten (Linien, Kreise etc.) zur Verfügung. Vorhandene Prozeduren zur Erstellung von Dialogfenstern können die Fähigkeiten einer Applikation zur interaktiven Bedienung stark verbessern.

Die Dokumentation [56] ist ordentlich und übersichtlich, die Namen von Klassen und Methoden wurden zweckmäßig (englischsprachig) gewählt.

Trotz aller Vorzüge ist die Bibliothek längst nicht vollständig, gerade mathematische Algorithmen, beispielsweise zur linearen Algebra oder linearen Programmierung, wären in einer ähnlichen Form wünschenswert gewesen.

[1] LEDA=Library of Efficient Data Types and Algorithms

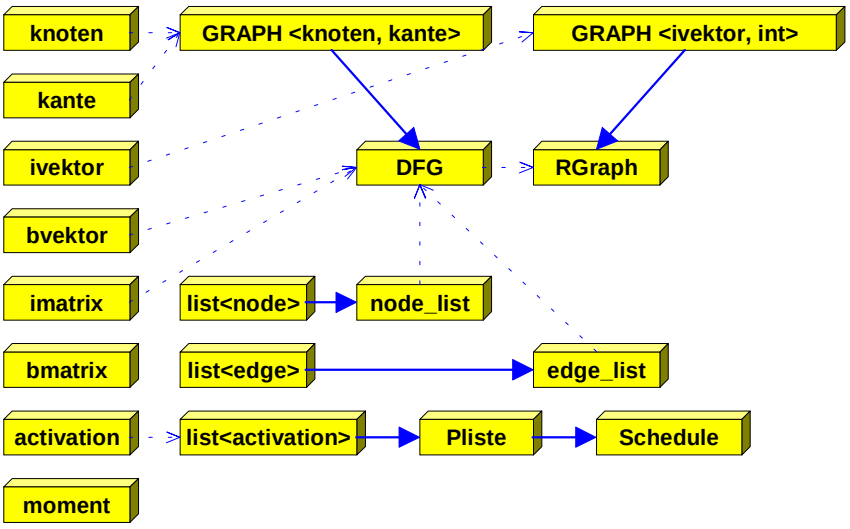
9.4 Eigene Klassen

Folgende Klassen wurden entworfen und implementiert:

- Bruch Arithmetik mit rationalen Zahlen
- ivektor Arithmetik für Vektoren variabler Länge aus ganzen Zahlen
- bvektor Arithmetik für Vektoren variabler Länge aus rationalen Zahlen
- imatrix Arithmetik für Matrizen variabler Länge aus ganzen Zahlen; mit Matrix-Vektor und Matrix-Matrix Multiplikation, Transponierung usw.
- bmatrix Arithmetik für Matrizen variabler Länge aus rationalen Zahlen; wie 'imatrix', zusätzlich Gauß-Algorithmus zur Lösung des inhomogenen Gleichungssystems mit Bestimmung aller homogenen Lösungen (bei singulären[2] Matrizen)
- node_list Einfache Liste von Knoten; für Kreissuche, Schedul und zusammenhängende Komponenten benötigt
- edge_list Einfache Liste von Kanten; für Schleifensuche, Pfadsuche und Methoden im Zusammenhang mit WST u. HDM benötigt
- knoten Parameter für LEDA-Knoten; mit Information über Laufzeit, Nummer, Name, Position (X-Window)
- kante Parameter für LEDA-Kanten; mit Quell- und Zielrate, Delayanzahl, Nummer, Multiplizität (f. parallele Kanten), Markierung (X-Window)
- DFG Datenflußgraph aus Knoten und Kanten; Zentrales Objekt jedes Programms
- activation Klasse für Scheduler; enthält u.a. Prozessor, gestarteten Knoten, Start- und Endzeit
- Pliste Liste von 'activation'; enthält das Schedul für einen Prozessor
- Schedule Array von 'Pliste'; enthält Multiprozessor-Schedul
- Rgraph Erreichbarkeitsgraph; enthält Knoten aus (Zustands-)Vektoren und Kanten aus DFG-Knotennummern, sowie eine Liste von Anfangs- und Endzuständen
- moment Zustand zu einen bestimmten Zeitpunkt während des Scheduls; vermerkt ist der Zustands- und Aktivierungsvektor, sowie der Zeitpunkt

Die Vererbungshierarchie ist im folgenden Bild dargestellt[3]:

[2] Singuläre Matrizen haben kleineren Rang als Zeilen- oder Spaltenanzahl, det(A)=0, nicht regulär
[3] Durchgezogene Pfeile: echte Vererbung; Gestrichelt: Einbindung/Verwendung



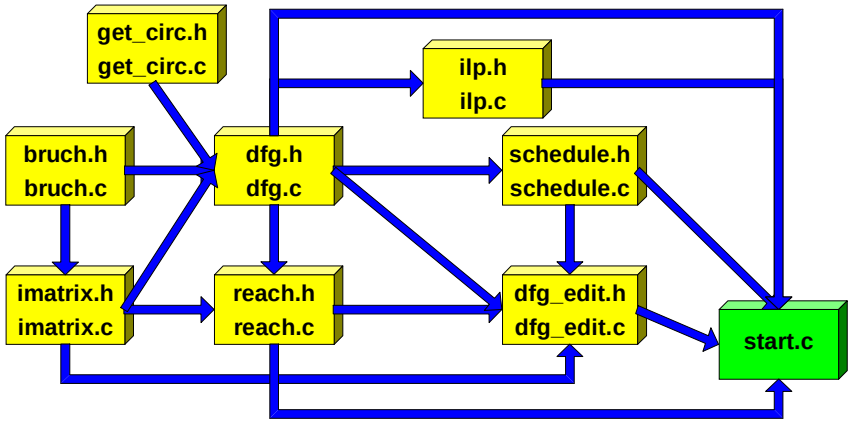
9.5 Methoden der Klasse DFG

Die von ‘Graph’ geerbten Methoden mußten wesentlich erweitert werden, um den Graphen zum Multiraten-Datenflußgraphen zu machen. Neben elementaren Aktionen wie Einfügen und Löschen von Knoten und Kanten, die nicht mehr die geerbten Methoden verwenden können, sind die Graphentransformationen auch als Methoden formuliert. Eine Kollektion von Prozeduren, die adjazente Knoten und Kanten ermitteln, schließen sich an. Die Berechnungen der Invarianten (Γ, q, y, C, w, n) sind als Methoden implementiert. Weiterhin können Knoten (zufällige oder bestimmte Wahl) vorwärts oder rückwärts aktiviert werden und das wahlweise mit Auswirkungen im Graphen oder nur im Delay-Vektor. Schließlich sind Methoden rund um die ‘Weighted Sum of Tokens’ vorhanden. Hinsichtlich der graphischen Darstellung wurden Maßnahmen zur Erleichterung der Bearbeitung ergriffen.

Im Anhang sind die Header-Dateien der Module abgedruckt, die Methoden können daraus entnommen werden. Der gesamte Sourcecode wird in einem separaten Skript nachzulesen sein.

9.6 Weitere implementierte Module

Es wurden zur Durchführung der Untersuchungen diverse C++-Module implementiert, jeweils Programmrumpf (*.C) und -kopf (Header, *.H) für eingebundene Routinen und Code-Dateien für die Hauptprogramme (Main, *.C). Anstelle von ‘start.c’ kann jedes andere Hauptprogramm (automatische Tests) stehen. Die Grafik soll die Zusammenhänge der Module untereinander veranschaulichen; dies ist besonders dann wichtig, wenn die Dateien neu compiliert werden müssen: Die Reihenfolge der Compilation ist genau die, die für ein Schedul des zugrundeliegenden azyklischen Graphen verwendet werden müßte.



(get_circ: Kreise bestimmen; dfg: Klasse DFG; reach: Erreichbarkeit; ilp: Lineare Optimierung; dfg_edit: Interaktive Bedieneroberfläche; start: Hauptprogramm mit Menüstruktur)

9.7 Interaktive Bedienoberfläche

Im Laufe der Arbeit an den Methoden der Klasse DFG bestand die Notwendigkeit, die generierten Graphen zu betrachten und zu verändern. Eine einfache Textausgabe in der Form:

" {V1}2-->4{V2}4-->2{V1} " für eine Schleife

erwies sich als wenig praktikabel für die Auswertung von Graphen mit mehr als zwanzig Schleifen. Eine Eingabe in dieser Form würde jeden Benutzer nach kurzer Zeit ermüden und führt zu häufigen Fehlern. Die angenehmste vorstellbare Lösung wäre (nach heutigen Erkenntnissen) eine Implementation mit einer CUA[4]-Oberfläche (GUI[5]) nach dem SAA[6]-Standard, mit einer Menüleiste und Popup-Fenstern, kontextsensitiver Hilfe und Pushbuttons. Dieser von OS/2 und Microsoft Windows® gewohnte Komfort ist aber nur bei entsprechender Unterstützung durch das Betriebssystem (mit vertretbarem Aufwand) realisierbar. In Systemen, die das Betriebssystem UNIX nutzen, sind solche Applikationen noch nicht so üblich, obwohl mit X-Window eine adäquate Plattform existiert.
Für das Projekt der Auswertung von Datenflußgraphen wurde dennoch mit Hilfe von Maus- und Grafikroutinen eine graphische Oberfläche und ein statisches Menü implementiert. Teilweise sind auch interaktive Dialogfenster verfügbar, zum Beispiel für die Menüpunkte Datei und Optionen. Das Modul „dfg_edit“ enthält alle dazu notwendigen Funktionen. Der Bildschirm sieht beispielsweise so aus:

[4] CUA=Common User Access: Standard, der das Aussehen einer Applikation sowie gewisse Tastenbelegungen normiert: Menüleiste, Statuszeile, Desktop und Mausunterstützung. Teilmenge des SAA-Standards.
[5] GUI=Graphical User Interface (Graphische Benutzeroberfläche)
[6] SAA=System Application Architecture: von IBM 1987 definierter Standard zur Integration verschiedener Software in eine gemeinsame Oberfläche.

(Bildschirmhardcopy)

Die Menüpunkte haben folgende Bedeutung:

redraw	Auffrischung des Bildschirms; Neuzeichnen
exit	Programm beenden
clear	Graph löschen
file	Graph laden, speichern, Verzeichnis wechseln
options	Einstellungen zum Bearbeiten, Statistik und Scheduling
help	Übersicht über Verwendung der Maustasten
Swap G	Umschalten zwischen zwei gespeicherten Graphen
Info	Ausnutzung der Ressourcen
new	Generation eines neuen Graphen: Schleife, Normschleife etc.
matrix	Berechnet die Inzidenzmatrix
write	Gibt den graphen in interner Darstellung aus: Knoten+Kantenliste
loops	Bestimmt die im Graphen enthaltenen Schleifen
calc_q	Berechnet den *q*-Vektor
calc_C	Berechnet die ‘fundamental circuit matrix’
StrCmps	Bestimmt die stark zusammenhängenden Komponenten
StrCon	Führt ‘Strongly Connecting’ durch
1-Gain	Führt die Einheitsverstärkungs-Transformation durch

1-Rate	Bestimmt den äquivalenten Einheitsraten-Graph
DelParEdg	Entfernt parallele Kanten
->APG	Bestimmt den azyklischen Präzedenzgraphen
calc_RG	Berechnet den Erreichbarkeitsgraphen
show_RG	Zeigt den berechneten Erreichbarkeitsgraphen an
Del_NMD	Entfernt arc-NMD's auf Kanten
show_NMD	Bestimmt NMD's und deren Gewicht auf Pfaden
Gauss	Ermittelt eine Lösung für $\Gamma^* r = d_r - d_0$; Unterschied zweier Graphen
ILP	Benutzt ganzzahlige lineare Programmierung für das Problem $\Gamma^* r = d_r - d_0$
Opt_Del	Optimiert die Delayverteilung durch lineare Programmierung
Opt_DelSR	Optimierung der Delayverteilung im äquivalenten SR-Graph
calc_r	Berechnet den Retimingvektor r zu einer Neuverteilung
calc_HDM	Berechnet HDM für jede Schleife
HDM+1	Macht den Graph lebendig durch Verteilung von HDM+1 auf Schleifen
RedDel	Neuverteilung der Delays mit gleicher WST in jeder Schleife
Schedule	Erstellt ein Schedul solange, bis sich ein vormaliger Zustand wiederholt
Tmin(WST)	Führt Scheduling f. 1 Schleife durch WST=1..x; Bestimmt Iterationsperiode

9.8 Automatische Tests

Zum Test der entwickelten Verfahren und schneller empirischer Verifikation von Annahmen wurden automatische Tests implementiert, die zufällige Graphen mit bestimmten Eigenschaften in ausreichender Anzahl generieren und darauf Testverfahren anwenden.

Meistens wurden je 2000 bis 20000 Graphen mit Größen zwischen drei und zehn Knoten generiert, deren Kantenanzahl bis zu 20 festgelegt wurde, evtl. durch Festlegung des Verhältnisses Kanten/Knoten, das beschreibt, wie dicht der Graph ist {sparse-dense}. Wahlweise wurden nur einzelne Schleifen oder allgemeine Multiraten-Graphen erzeugt. Letztere wurden durch „Strongly Connecting" zyklisch gemacht, da nur zyklische Graphen hier von Interesse waren. Wahlweise konnte dann eine Einheitsverstärkungs-Transformation durchgeführt werden, um eine normierte Darstellung zu gewinnen, bei der die Raten jedes Knotens gleich sind. Dabei tritt die Schwierigkeit auf, daß bei zufälliger Neuverteilung der Delays mit gleicher WST die Wahrscheinlichkeit für ein NMD sehr viel größer wird (aus einer Kante mit 1-Raten wird eine mit Raten der Größe w; w-1 von w Verteilungen enthalten dort ein NMD). Also wurde die Normierung erst möglichst spät durchgeführt und die Algorithmen so entwickelt, daß sie auch auf unnormierten Graphen funktionieren.

Eine Einheitsratentransformation war der nächste mögliche Schritt, mit dem die Ergebnisse verifiziert werden konnten, denn für (äquivalente) Einheitsraten-Graphen sind vorhandene Algorithmen immer gültig, dafür aber von so hoher Komplexität, daß dadurch die Größe der getesteten Graphen begrenzt wurde.

Für einige Tests konnte nun eine neue Delayverteilung erzeugt werden, indem unter Einhaltung der Bedingung $C \cdot \vec{d} = \overline{WST} = const$ entweder

- ein Monte-Carlo-Verfahren für zufällige Verteilung sorgt,
- durch lineare Programmierung eine Zielfunktion über d_i maximiert wurde,
- mit ganzzahliger linearer Programmierung ein legales Retiming durchgeführt wurde.

Die lineare Optimierung wurde auch unter Hinzufügung der zusätzlichen Schnittebenen durchgeführt. Zur Optimierung kam das Programm „lp_solve" zum Einsatz (als Subprocess), das Ein- und Ausgabe der Daten über Textdateien durchführt. Der System-Verwaltungsaufwand und der Zeitbedarf für den integrierten Parser verhielt sich im Vergleich zum Simplex-Algorithmus relativ gering.

Graphen, die entgegen der Annahme unerwartetes Verhalten aufwiesen, wurden in durchnumerierten Dateien (mit den LP-Dateien) abgespeichert und der späteren Auswertung zugänglich gemacht. Das Programm konnte so ohne Benutzerdialog auch zu Zeiten mit geringer Systemauslastung laufen.

10 Zusammenfassung und Ausblick

Diese Arbeit hatte die Behandlung von Modellen für Multiraten-Datenflußprogramme zum Ziel. Betrachtet wurden alternative Beschreibungsformen und konkurrierende Modelle, die zum Erkenntnisstand beitragen. Mit dem Modell der ‘non-ordinary’-Petri-Netze und dem regulären Datenfluß sind geeignete Methoden zur Beschreibung konkurrierender und paralleler Software mit multiplen Raten vorhanden, die auch begrenzt für Hardware-Implementationen eingesetzt werden können.

Wichtige und markante Kenngrößen der Modelle in ihrer mathematischen Beschreibung wurden zitiert, erweitert und neu definiert. Die Größen HDM, LLM, CM, die hier genannt wurden, stellten sich im späteren Kapitel „Durchsatz“ als wichtige Parameter heraus. Die Berechnung der Invarianten und anderer struktureller Eigenschaften wurden im Detail ausgearbeitet und vorgestellt. Die Bedeutung der Erreichbarkeit, die anhand graphischer Beispiele verdeutlicht wurde, steht im direkten Zusammenhang mit einem Schwerpunkt der Arbeit, der Multiraten-Retiming-Transformation.

Es wurden alle bekannten Transformationen der Modelle in einem eigenen Kapitel vorgestellt, teilweise wegen ihrer Bedeutung für die folgenden Kapitel „Scheduling“ und „Retiming“, aber auch wegen einer vollständigen Übersicht der in der Literatur anzutreffenden Methoden, die hier teilweise erweitert und ergänzt wurden.

Das Scheduling von Multiraten-Programmen ist als Titel der Arbeit in ein Kapitel eingeflossen und wurde mehr als Übersicht über Theorie und Verfahren angelegt, um dem Schwerpunkt „Retiming“ den aus dieser Richtung nötigen Hintergrund zu verschaffen.

Die Retiming-Transformation, die wegen ihrer Komplexität im Multiratenfall bisher große Schwierigkeiten bereitete, wurde ausführlich untersucht. Dabei stellten sich Bedingungen heraus, die zusätzlich zu den im Einheitsratenfall nötigen Bedingungen für legales Retiming erfüllt sein müssen. Die Konsequenzen der Bedingungen nahmen einen großen Teil der Arbeit in Anspruch und führten zu Ansätzen und Algorithmen, die die Komplexität der Operationen erheblich reduzieren können. Die lineare Optimierung kann dadurch auch für Multiraten-Graphen eine gültige Neuverteilung der Delays in polynomialer Zeit durchführen, wobei lediglich eine Umwandlung in einen minimalen äquivalenten gewöhnlichen Graphen durchzuführen ist. Die Feststellung oder Distanzierung von der NP-Vollständigkeit des allgemeinen Multiraten-Retimings ist ein Punkt zukünftiger Forschung. Leider sind viele andere für Einheitsraten brauchbare Algorithmen aus der Graphentheorie nicht unmittelbar einsetzbar.

Die Analyse des asymptotischen Laufzeitverhaltens von Datenflußprogrammen bei optimalem Schedul war ein weiterer Schwerpunkt der Arbeit. Hier wurden die bisher bekannten Grenzen vorgestellt und durch einen neuen Ansatz in der Genauigkeit wesentlich verbessert. Das angestrebte Ziel, eine Formel für die Iteration-bound im Multiratenfall zu finden, die immer erreicht werden kann (‘tight bound’), konnte nicht ganz verwirklicht werden, wenn auch die Ergebnisse der engsten Grenze recht nahe kommen und meistens mit dieser übereinstimmen. Wegen der Anomalien der Multiraten-Graphen bezüglich ihrer Delayverteilung ist aber anzunehmen, daß die engste Grenze niemals durch eine geschlossene Formel, wie im Einheitsratenfall, beschrieben werden kann.

Neben der Vorstellung der erarbeiteten Konzepte und Algorithmen, die auch in der Praxis für Multiprozessor-Implementationen brauchbar sind, sollten mit diesem Schriftstück auch viele der bisher bekannten, aber in der Literatur weit zerstreuten, Informationen gesammelt dargestellt werden. Daher ist an einigen Stellen umfangreicher Text zur Aufklärung vorhanden.

Die im Rahmen der Diplomarbeit entstandene Software „MuDaG“ steht nun als einfaches Auswertungswerkzeug von Datenflußgraphen zur Verfügung.

11 Anhang

11.1 Programme

Auf den folgenden Seiten sind die Header-Dateien zu den Modulen abgedruckt, die im Rahmen der Diplomarbeit entwickelt wurden. Datentypen, Objektklassen mit zugehörigen Methoden und Definitionen lassen sich daraus ablesen. Die Implementationen in den Hauptdateien (*.C) werden aus Platzgründen nicht oder nur auszugsweise veröffentlicht.

```
/*****************************************************
*****
 *            Bruch.h
 *     Arithmetik fuer rationale Zahlen
*****************************************************
*****/

#ifndef _Bruch_h
#define _Bruch_h

// Standard-Header-Dateien einbinden
#include <stdlib.h>
#include <iostream.h>
#include <LEDA/basic.h>

inline long l_abs (long x) { return ((x < 0) ? (-x):(x)); };
long next_prim(long z1);
long ggt(long z1,long z2);
int  ggt(int  z1, int z2);
long kgv(long z1,long z2);
long div_ceil(long x, long y); // int_up[x/y]

typedef long ityp; // Basistyp fuer Zaehler und Nenner

class Bruch {
  private:
    ityp zaehler;
    ityp nenner;
    bool auto_kuerz; // default: on
  public:
    // Default-Konstruktor, Konstruktor aus Zaehler und
    // Konstruktor aus Zaehler und Nenner
    Bruch (ityp = 0, ityp = 1);

    void kuerze();
    bool set_autokuerz(bool on)
      { bool last=auto_kuerz; auto_kuerz=on; return last; }

    //Bruch operator =(Bruch&);// nicht noetig, default kopiert
    /* Multiplikation
     *  - globale Friend-Funktion, damit automatische
     *    Typumwandlung des ersten Operanden moeglich ist
     */
    friend Bruch operator * (const Bruch&, const Bruch&);
    friend Bruch operator / (const Bruch&, const Bruch&);
    friend Bruch operator + (const Bruch&, const Bruch&);
    friend Bruch operator - (const Bruch&, const Bruch&);
    Bruch operator - () const {return Bruch(-zaehler,nenner);}

    // multiplikative Zuweisung etc.
    const Bruch& operator *= (const Bruch&);
    const Bruch& operator /= (const Bruch&);
    const Bruch& operator += (const Bruch&);
    const Bruch& operator -= (const Bruch&);
    const Bruch operator ! () // Kehrwert
      { if (zaehler ==0) { cerr << "Kehrwert-Fehler" << endl;
exit(1);}
        return Bruch(nenner, zaehler);}
    Bruch kehrwert() {
      if (zaehler ==0) { cerr << "Kehrwert-Fehler" << endl;
exit(1);}
      ityp tm=zaehler; zaehler=nenner; nenner=tm; return (*this);
}
    ityp Zaehler() const { return zaehler;}
    ityp Nenner() const { return nenner;}
    bool zero() const { return (zaehler == 0); }

    // Vergleich
    //  - globale Friend-Funktion, damit automatische
    //    Typumwandlung des ersten Operanden moeglich ist
    friend bool operator < (const Bruch&, const Bruch&);
    friend bool operator > (const Bruch&, const Bruch&);
    friend bool operator >= (const Bruch&, const Bruch&);
    friend bool operator <= (const Bruch&, const Bruch&);
    friend bool operator == (const Bruch&, const Bruch&);
    friend bool operator != (const Bruch&, const Bruch&);

    // Ausgabe mit Streams
    friend void Print (const Bruch& b, ostream& out = cout);
    // Eingabe mit Streams
    friend void Read (Bruch& b, istream& in = cin);
    friend bool compare(const Bruch& b1, const Bruch& b2);
//return (b1 > b2); }
    friend bool compare_abs(const Bruch& b1, const Bruch&
b2); // return (|b1| > |b2|); }
    // Konvertierung nach double
    double asDouble () const;
    Bruch abs() const;
};

/* Standard-Ausgabeoperator
 *  - global ueberladen und inline definiert
 */
inline ostream& operator << (ostream& strm, const Bruch& b)
{   Print (b,strm);       // Elementfunktion zur Ausgabe aufrufen
    return strm;          // Stream zur Verkettung zurueckliefern
}

/* Standard-Eingabeoperator
 *  - global ueberladen und inline definiert
 */
inline istream& operator >> (istream& strm, Bruch& b)
{   Read (b,strm);        // Elementfunktion zur Eingabe aufrufen
    return strm;          // Stream zur Verkettung zurueckliefern
}

/* asDouble
 *  - Bruch in double umwandeln
 */
inline double Bruch::asDouble () const
{   // Quotient aus Zaehler und Nenner zurueckliefern
    return (double)zaehler / (double)nenner;
}
```

```
inline Bruch Bruch::abs() const
{ return Bruch(l_abs(zaehler),l_abs(nenner)); }

LEDA_TYPE_PARAMETER(Bruch);

#endif  /* _Bruch_h */
```

```
// Definition der Klassen fuer ganzzahlige und rationale
Vektorarithmetik
// Datei : imatrix.h
// Klassen: ivektor, imatrix - Matrizen und Vektoren aus
Integerwerten
// bmatrix, bvektor - Matrizen und Vektoren aus Bruechen

#ifndef IMATRIXHEADER
#define IMATRIXHEADER

#include <stdlib.h>
#include <fstream.h>
#include <LEDA/basic.h> // f. string, LEDA_TYPE_PARAM
#include <LEDA/vector.h> // f. vector(float)
#include "bruch.h"

// ivektor: erlaubte Indizes: 0..(zeile-1) -> root[0..(zeile-1)]
class ivektor {
 private:
  int zeile;
  long * root;
 public:
  bool valid;
  friend class imatrix;
  friend class bvektor;
  friend class bmatrix;
  ivektor (int dim=1);                  // Konstruktor
  ivektor (int dim, long initwert); // Konstruktor + init
  /* ivektor();                                    //
Standardkonstruktor */
  virtual ~ivektor () ;                 // Destruktor
  ivektor (const ivektor& robj); // Kopier-Konstruktor
  void init (int zeile);  // Neue Dimension
  void init (int zeile, long initwert);  // Neue Dimension
  int dim () const { return zeile; }
  const ivektor& operator= (const ivektor& op);
  ivektor operator+ (const ivektor& op) const;
  const ivektor& operator+= (const ivektor& op); // Vektoradd.
v+=w
  ivektor operator- () const; // Vorzeichenwechsel
  ivektor operator- (const ivektor& op) const;
  const ivektor& operator-= (const ivektor & op); // Vektorsub.
v-=w
  ivektor operator* (long r) const;
  const ivektor& ivektor::operator*= (long r); // Streckung v*=s
  long ivektor::operator* (const ivektor & op) const; //
Skalarmultiplikation
  ivektor ivektor::operator% (const ivektor & op) const; //
Vektormodulo u=v % w (komp.weise)
  const ivektor& ivektor::operator%= (const ivektor & op); //
Vektormodulo v%=w
  ivektor operator * (const imatrix& m); // Vektor*Matrix
  long& operator[] (int i); // ACCESS
  long element(int i) const; // ACCESS
  int operator == (const ivektor& op2) const;
  int operator == (const long op2) const;
  int operator != (const ivektor& op2) const;
  int operator != (const long op2) const;
  // im folgenden muss operator fuer jede Komponente gelten -
>true
  int operator < (const ivektor& op2) const;
  int operator < (const long op2) const;
  int operator <= (const ivektor& op2) const;
  int operator <= (const long op2) const;
  int operator > (const ivektor& op2) const;
  int operator > (const long op2) const;
  int operator >= (const ivektor& op2) const;
  int operator >= (const long op2) const;
  vector to_vector() const; // integer -> double
  long ggT();
  long kgV();
  long max();
  long min();
  int min_int(); // minimaler Integer-Vektor
  friend ivektor min_int(const ivektor& s) { ivektor e=s;
e.min_int(); return e; }
  int kehrwert(); // HN/Komponente
  friend ivektor kehrwert(const ivektor& s) { ivektor e=s;
e.kehrwert(); return e; }

  friend void Read(ivektor& v, istream& in);
  friend void Print(const ivektor& v, ostream& out);
  friend int compare(const ivektor& v1, const ivektor& v2) {
return (v1.zeile - v2.zeile);};
  friend ostream & operator << (ostream & out, const ivektor&
v);
  friend istream & operator >> (istream & in, ivektor& v);
 };

//------------------------ iMatrix: -------------------------

class imatrix {
 private:
  int zeile,spalte;
  long * * wurzel;
 public:
  bool valid;
  friend class ivektor;
  friend class bvektor;
  friend class bmatrix;
  imatrix (int li=1,int co=1);                  // Konstruktor
  imatrix (int li,int co,long init);  // Konstruktor mit init
  virtual ~imatrix();                   // Destruktor
  imatrix (const imatrix& obj);                 // Kopier-
Konstruktor
  const imatrix& init (const ivektor& vek);   // Konstruktor
(Spalte)
  const imatrix& init(int li, int co);                  // Neue
Dimension !
  const imatrix& init(int li, int co, long init);// Neue Dimension +
initwert
  const imatrix& Einheit(int dim);
  int dimz () const { return zeile; }
  int dims () const { return spalte; }
  long& operator() (int i,int j); // ACCESS on elements
  long element(int i,int j) const;   // ACCESS on elements
  const imatrix& operator = (const imatrix& op);
  imatrix operator + (const imatrix& op) const;
  const imatrix& operator += (const imatrix& op);
  imatrix operator - (const imatrix& op) const;
  const imatrix& operator -= (const imatrix& op);
  imatrix operator ! () const; // Transponieren

  ivektor operator* (const ivektor& x) const;           // Matrix
* Vektor
  imatrix operator* (const imatrix& m) const;           // Matrix
* Matrix
  int rang() const;

  ivektor Zeile(int znr) const;
  ivektor Spalte(int snr) const;

  friend imatrix transpone(const imatrix& m);
  friend imatrix transpone(const ivektor& v); // Spalte->Zeile

  friend void Read(imatrix& m, istream& in);
  friend void Print(const imatrix& m, ostream& out);
```

```
  friend int compare(const imatrix& m1, const imatrix& m2) {
return (m1.zeile - m2.zeile);};
  friend ostream & operator << (ostream & out, const imatrix&
m);
  friend istream & operator >> (istream & in, imatrix& m);
};

//------------------------- bvektor ---------------------------------

// bvektor: erlaubte Indizes: 0..(zeile-1) -> root[0..(zeile-1)]
class bvektor {
 private:
  int zeile;
  Bruch * root;
 public:
  bool valid;
  friend class ivektor;
  friend class imatrix;
  friend class bmatrix;
  bvektor (int dim=1);                // Konstruktor
  bvektor (int dim, const Bruch& initwert);  // Konstruktor + init
  /* ivektor();                                //
Standardkonstruktor */
  virtual ~bvektor () ;                  // Destruktor
  bvektor (const bvektor& robj); // Kopier-Konstruktor
  bvektor (const ivektor& robj); // Typwandlungs-Konstruktor
  const bvektor& init (int zeile);  // Neue Dimension
  const bvektor& init (int zeile, const Bruch& initwert);  // Neue
Dimension
  int dim () const { return zeile; }
  const bvektor& operator= (const bvektor& op);
  bvektor operator+ (const bvektor& op) const;
  const bvektor& operator+= (const bvektor& op); // Vektoradd.
v+=w
  bvektor operator- () const; // Vorzeichenwechsel
  bvektor operator- (const bvektor& op) const;
  const bvektor& operator-= (const bvektor & op); // Vektorsub.
v-=w
  bvektor operator* (const Bruch& r) const;
  const bvektor& bvektor::operator*= (const Bruch& r); //
Streckung v*=s
  Bruch bvektor::operator* (const bvektor & op) const; //
Skalarmultiplikation
  bvektor operator * (const bmatrix& m); // Vektor*Matrix
  Bruch& operator[] (int i); // ACCESS
  Bruch element(int i) const; // ACCESS
  int operator == (const bvektor& op2) const;
  int operator == (const long op2) const;
  int operator != (const bvektor& op2) const;
  int operator != (const long op2) const;
  // im folgenden muss operator fuer jede Komponente gelten -
>true
  int operator < (const bvektor& op2) const;
  int operator < (const long op2) const;
  int operator <= (const bvektor& op2) const;
  int operator <= (const long op2) const;
  int operator > (const bvektor& op2) const;
  int operator > (const long op2) const;
  int operator >= (const bvektor& op2) const;
  int operator >= (const long op2) const;
  vector to_vector() const; // Bruch -> double
  Bruch max();
  Bruch min();
  long min_int(); // minimaler Integer-Vektor
  friend ivektor min_int(const bvektor& s);

  friend void Read(bvektor& v, istream& in);
  friend void Print(const bvektor& v, ostream& out);
  friend int compare(const bvektor& v1, const bvektor& v2) {
return (v1.zeile - v2.zeile);};
  friend ostream & operator << (ostream & out, bvektor& v);
  friend istream & operator >> (istream & in, bvektor& v);
 };

//------------------------- bmatrix: ---------------------------

class bmatrix {
 private:
  int zeile,spalte;
  Bruch * * wurzel;
 public:
  bool valid;
  friend class ivektor;
  friend class bvektor;
  friend class imatrix;
  bmatrix (int li=1,int co=1);                  // Konstruktor
  bmatrix (int li,int co, const Bruch& init);  // Konstruktor mit
init
  virtual ~bmatrix();                 // Destruktor
  bmatrix (const bmatrix& obj);                 // Kopier-
Konstruktor
  bmatrix (const imatrix& obj);                 // Typwandlungs-
Konstruktor
  const bmatrix& init(const bvektor& vek);              //
Konstruktor (Spalte)
  const bmatrix& init(int li, int co);                  // Neue
Dimension !
  const bmatrix& init(int li, int co, const Bruch& initwert);//
Neue Dimension + initwert
  const bmatrix& Einheit(int dim);
  int dimz () const { return zeile; }
  int dims () const { return spalte; }
  Bruch& operator() (int i,int j); // ACCESS on elements
  Bruch element(int i,int j) const; // ACCESS on elements
  const bmatrix& operator= (const bmatrix& op);
  bmatrix operator + (const bmatrix& op) const;
  const bmatrix& operator += (const bmatrix& op);
  bmatrix operator - (const bmatrix& op) const;
  const bmatrix& operator -= (const bmatrix& op);
  bmatrix operator ! () const; // Transponieren

  bvektor operator* (const bvektor& x) const;           // Matrix
* Vektor
  bmatrix operator* (const bmatrix& m) const;           // Matrix
* Matrix
  int rang() const;
  int gauss(bmatrix b, bmatrix& x);   // loest Ax=b

  bvektor Zeile(int znr) const;
  bvektor Spalte(int snr) const;

  friend bmatrix transpone(const bmatrix& m);
  friend bmatrix transpone(const bvektor& v); // Spalte->Zeile

  friend void Read(bmatrix& m, istream& in);
  friend void Print(const bmatrix& m, ostream& out);
  friend int compare(const bmatrix& m1, const bmatrix& m2) {
return (m1.zeile - m2.zeile);};
  friend ostream & operator << (ostream & out, bmatrix& m);
  friend istream & operator >> (istream & in, bmatrix& m);
};

LEDA_TYPE_PARAMETER(ivektor);
LEDA_TYPE_PARAMETER(imatrix);
LEDA_TYPE_PARAMETER(bvektor);
LEDA_TYPE_PARAMETER(bmatrix);

//------------------------- Misc: --------------------------------

inline string long_string(long l) { return string("%4d",l); };

#endif
```

```
//
// get_circ.h
```

```
//
// Routinen zur Ermittlung von gerichteten Schleifen
// Es wird der Algorithmus von Johnson benutzt
//
#ifndef GETCIRCUITS
#define GETCIRCUITS


#include <LEDA/graph.h>
#include <LEDA/list.h>
#include <LEDA/graph_alg.h>
#include <LEDA/array.h>
//#include "dfg.h"

// "node_list" is just a simple list of nodes.
// "node_list" is used, because we need a list of a list of nodes !
// Every elementary cycle is such a list of nodes
// and we have usually more than one cycle in each graph.

// jetzt auch in dfg.h:
class node_list:public list<node>
{
 public:
 node_list() { this->clear(); }
 node_list(const node_list& robj) { (*this)=robj; }
 friend void Read (node_list& k, istream& is) { k.read(is); }
 friend void Print (node_list& k, ostream& os) { k.print(os); }
};
LEDA_TYPE_PARAMETER(node_list)
//LEDA_MEMORY(node_list);

class edge_list:public list<edge>
{
 public:
 edge_list() { this->clear(); }
 edge_list(const edge_list& robj) { (*this)=robj; }
 friend void Read (edge_list& k, istream& is) { k.read(is); }
 friend void Print (edge_list& k, ostream& os) { k.print(os); }
};
LEDA_TYPE_PARAMETER(edge_list)

void get_selfloops(const graph &G, list<node_list> &
allselfloops);
// nur Selbstschleifen
void get_selfloops(const graph &G, list<edge> & allselfloops);

void get_circuits (const graph &G, list<node_list> &
allcircuits);
// nur Schleifen mit >=2 Knoten
// (Algorithmus von Donald B. Johnson)

int get_loops(const graph &G,list<node_list> & all_loops);
// alle Schleifen incl. Selbstschleifen. Return=Anzahl

int get_loops(const graph &G,list<edge_list> & all_loops);
// alle Schleifen incl. Selbstschleifen. Return=Anzahl

#endif
```

```
/* Diese Datei "dfg.h" liefert die Klassen fuer den Knotentyp
und den
   Kantentyp, der fuer die Multiraten-Graphen benoetigt wird.

  LEDA: Class types have to provide the following operations:

  a constructor taking no arguments   T::T()
  a copy constructor                  T::T(const T&)
  a Read function                     void Read(T&, istream&)
  a Print function                    void Print(const T&, ostream&)

  A compare function  "int compare(const T&, const T&)" (cf.
section 1.4 of the user manual) has to be defined if required by
the data type.

*/

#ifndef dfgHEADER
#define dfgHEADER

#include <fstream.h>
#include <LEDA/basic.h>
#include <LEDA/list.h>
#include <LEDA/graph.h>
#include <LEDA/plane.h> // fuer point
#include "imatrix.h"
#include "bruch.h"
#include "get_circ.h"
//typedef list<node> node_list;
//typedef list<edge> edge_list;

// for random numbers :
//#define granularity 100

class knoten   // DFG-Knoten mit Informationen ueber Zeit +
Name + Koordinatenposition
{
 public:
  double time;                        // Ausfuehrungszeit des Knoten
  int nr;                  // Nummer des Knoten (f. Inzidenzmatrix)
  string name;                        // Name des Knotens
  point position;        // Position im X-Window
 knoten() { time=0; nr=0; point p(0,0); position=p; }  //
Konstruktor
 ~knoten() {;};
 knoten(const point& p) { time=0; nr=0; position=p; }
 knoten(const point& p, const knoten& ohnpkt) {
time=ohnpkt.time; nr=ohnpkt.nr; position=p; }
 knoten(const knoten& robj) { time=robj.time; nr=robj.nr;
position=robj.position; name=robj.name; }
 friend void Read (knoten& k, istream& is) { is >> k.nr >>
k.time >> k.position >>k.name; }
 friend void Print (knoten& k, ostream& os) { os << " " << k.nr
<< " " << k.time <<" "<< k.position <<" "<< k.name; }
 friend int compare (const knoten& k1,const knoten& k2) {
return (k1.nr > k2.nr); }
};

LEDA_TYPE_PARAMETER(knoten);
//LEDA_MEMORY(knoten);

class kante  // Kante von Knoten u nach v
{
 public:
  int source;   // erzeugte Samples beim Quellenknoten (u)
  int target;   // verbrauchte Samples beim Targetknoten  (v)
  int delay;    // Anzahl Delays auf der Kante
  int nr;      // Nummer fuer Inzidenzmatrix // nie vom user zu
veraendern!
  int multiplicity; // zur Zusammenfassung paralleler Kanten
                                        // wird nicht mit abgespeichert
!
  int marke;                // Normal Null
  kante() { source=1; target=1; delay=0; nr=0; multiplicity=1;
marke=0; };               // Konstruktor
  kante(const kante& robj) { source=robj.source;
target=robj.target; delay=robj.delay; nr=robj.nr;
marke=robj.marke; multiplicity=robj.multiplicity; };
  int get_delay() {return delay;};
  int set_delay(int d) {delay=d; return delay;};
 friend void Read(kante& k, istream& is) { is >> k.nr >>
k.source >> k.target >> k.delay; };
 friend void Print(kante& k, ostream& os) { os << k.nr << " " <<
k.source << " " << k.target << " " << k.delay; };
```

```
 friend int compare (const kante& k1,const kante& k2) { return
(k1.nr > k2.nr); }
};

LEDA_TYPE_PARAMETER(kante);
//LEDA_MEMORY(kante);

//------------------------------------------------------------------
// DatenFlussGraph mit o.g. Knoten und Kanten :
//    DFG

class DFG:public GRAPH<knoten, kante> {
 private:
  int          max_node_indx;
  int          max_edge_indx;
  bool         node_array_OK;
  node_array<int> nodeindx;      //
nodeindx[node]=DFG::inf(node).nr
  bool         edge_array_OK;
  edge_array<int> edgeindx;      //
edgeindx[edge]=DFG::inf(edge).nr
  void        make_arrays_unvalid() {
node_array_OK=edge_array_OK=false;
    max_node_indx=max_edge_indx=0; nodeindx.clear();
edgeindx.clear(); };
  int          randomdelay(); // Zufallsdelaywert mit spez. Statistik

  // Rekursive Prozeduren:
  void  visit_neighbors(node n, const Bruch& qn, bvektor&
qtemp, edge_array<bool> & echeck, node_array<bool> &
ncheck);//DFS f. q-Vektor
  bool  visit_n(node n, long rn, ivektor& rtemp,
edge_array<bool> & echeck, node_array<bool> & ncheck, const
ivektor& dd);//DFS f. r-Vektor
  void DFG::visit_edge(node_array<bool> visited,
list<edge_list> &allpaths, edge_list path, edge testedge); // Fuer
get_paths

 public:
  double mintime;      // Minimale Ausfuehrungszeit
  double maxtime;      // Maximale Ausfuehrungszeit
  int          granularity;
  int    maxrate;      // Maximale In/Out-Rate
  int    maxdelays;    // Maximale Delayanzahl je Kante bei
Zufallsgr.
  int    ohne_zu_mit;  // Verhaeltnis Anzahl arcs ohne/mit delays
  bool         changed;  // Wird bei Aenderungen (der Struktur)
auf
                        // true gesetzt, sonst nix !
Eigenverantwortung !

  DFG() ;//: GRAPH<knoten,kante> () ;       // Std.Konstruktor
  DFG(const DFG& origin);                   // Copy-Konstruktor
  const DFG& operator = (const DFG& origin);          //
Zuweisung
  virtual ~DFG() { clear();};
              //Destruktor
  DFG(int nodes, int arcs);                 // Konstruktor fuer
Zufallsgraph
  virtual void clear() {GRAPH<knoten, kante>::clear();
make_arrays_unvalid(); max_node_indx=0; max_edge_indx=0;
};
  virtual void set_stat_params(double _mintime,double
_maxtime, int _granularity, int _maxrate, int _maxdelays, int
o2m); // Statistik beeinflussen

// Erzeugung eines DFG:

  virtual void load_interactive();
  virtual void makeuser_graph();
  virtual char create_graph(char prechoice=' '); // Abfrage:
Random/Datei/Komplett/Planar/User/Maus
  virtual void makerandom_graph(int anz_knoten, int
anzahl_kanten); // Erzeugt Zufallsgraph
  virtual void makerandom_loop(int anz_knoten, int& dsum,
bool mitHDM);
  // Normschleifen:
  virtual void makerand_n_loop(int anz_knoten, int& dsum, bool
mitHDM=false, bool coprime=false);
  virtual void makerand_n_loop(const DFG& origin, int
anz_knoten, int dsum);
  virtual void show_loops(ostream& os, ivektor& y);

// Node / Edge Information

  virtual int nodenumbers(node_array<int>& narr); // rechnet
Zuordnung
  virtual int edgenumbers(edge_array<int>& earr); // node,edge -
> (matrix-)index
  virtual int nodenumber(node v) const;
  virtual int edgenumber(edge e) const;
  virtual node nodenumber(int nn) const;
  virtual edge edgenumber(int en) const;
  virtual void makeNodeNames();
  // Adjazente Knoten und Kanten:
  virtual list<edge>    pred_edges(node v);
  virtual edge                  first_pred_edge(node v);
  virtual list<edge>    succ_edges(node v);
  virtual edge                  first_succ_edge(node v);
  virtual list<node>    pred_nodes(node v);
  virtual node                  first_pred_node(node v);
  virtual list<node>    succ_nodes(node v);
  virtual node                  first_succ_node(node v);
  virtual list<edge>    edges_between(node u, node v);
  virtual edge                  edge_between(node u, node v);
  virtual node                  getothernode(node n, edge e,
Bruch & ratio, bool& src); // ratio: n:other
  virtual node getothern(node n, edge e, int& r1, int& r2, bool&
src);
  // Knoten und Kanten erzeugen und loeschen
  virtual edge          new_edge(node u, node v); // overwrites
inherited fkt.
  virtual edge          new_edge(node u, node v, kante& k); //
with rate-info etc.
  virtual node          new_node(); // overwrites inherited
  virtual node          new_node(knoten& k); // mit info
  virtual int           del_edge(edge e); // overwrites inherited
  virtual int           del_node(node u); // overwrites inherited
  virtual void  del_parallel_edges(int marke=0); // mit min.
Delays.

  double randomtime()
   { double time=(double) random(1,granularity) / (double)
granularity;
    time*= (double) (maxtime-mintime); // Gleichverteilt in
[mintime...maxtime]
    time+= mintime;
    return time; };

  // Invarianten berechnen:
  void incidence_matrix(imatrix& m, ivektor& b0, int& nodes,
int& arcs);
  bool calc_q(ivektor& q); // true <- Graph war konsistent
  bool calc_r(ivektor& r, const ivektor& dd, const ivektor& q); //
true <- legal Retiming
  long calc_n(ivektor& n, ivektor& q);
  void calc_w(ivektor& w, ivektor& n, ivektor& q);
  void calc_C(list<edge_list>& all_loops, imatrix& C, ivektor&
w, ivektor& n, ivektor& q);
  // Circuit-Matrix (not fundamental)

// Transformationen:

  bool strongly_connect(const ivektor& q); // true, wenn vorher
schon str. conn.
```

```
  void unit_gain_trafo(ivektor& w, ivektor& n, ivektor& q); //
Trafo auf Einheitsraten
  bool delNMD_arc();
  void singlerate(DFG& SRG, int marke=0);
  void make_APG(DFG& APG, int J);

// Ausfuehren von Knoten:

  bool runnable(node v); // testing of delays on arcs!
 friend bool runnable_m(int nodenr,const imatrix& m, const
ivektor& d);
  bool run_node(node v); // execution +redistribution of delays
on arcs!
 friend bool run_node_m(int nodenr,const imatrix& m, ivektor&
d); // chg. vector
 friend bool run_node_m(int nodenr,const imatrix& m, ivektor&
d, ivektor& qsum); // chg. vector
  bool choose_runnable_node(node& v); // false if none found
// Rueckwaerts-Ausfuehren von Knoten
  bool R_runnable(node v); // testing of delays on arcs!
 friend bool R_runnable_m(int nodenr,const imatrix& m, const
ivektor& d);
  bool R_run_node(node v); // execution +redistribution of
delays on arcs!
 friend bool R_run_node_m(int nodenr,const imatrix& m,
ivektor& d); // chg. vector
 friend bool R_run_node_m(int nodenr,const imatrix& m,
ivektor& d, ivektor& qsum); // chg. vector
  bool choose_R_runnable_node(node& v); // false if none found

bool activate_node(node v);
bool release_node(node v);

// Delays & Co

  ivektor delay_vector() const; // returns Delay distribution
vector
  void delay_vector(const ivektor& dv); // sets delays on arcs
  long calc_WST(edge_list& loopedges, ivektor& y, bool
mit_y=true);
  void calc_WST(ivektor& WST, list<edge_list>& all_loops,
imatrix& C);
   // weighted sum of tokens
  void get_paths(list<edge_list> &allpaths);
  void showNMDpaths(ivektor& w);
  bool hasNMDs(ivektor& w);
  bool getNMDpaths(ivektor& w, list<edge_list> &NMDpaths,
list<edge_list> &allpaths);
  bool getPotNMDpaths(ivektor& w, list<edge_list>
&NMDpaths, list<edge_list> &allpaths);
  bool getPotNMD(ivektor& w, list<edge_list> &allpaths,
ivektor& NMD);
  void calc_HDM(ivektor& HDM, list<edge_list>& all_loops,
imatrix& C); // fuer jede Schleife ein Wert
  bool redistribute_delays(DFG& NeuGraph, list<edge_list>&
all_loops, imatrix& C, ivektor& WST1, long max_it=10000);
};

//LEDA_MEMORY(DFG);

// Fuer Scheduler benoetigte Klasse:
class activation
{ public:
  int proc;  // Prozessor, auf dem Knoten laeuft
  int nr;     // Knotennr.
  node v;    // Knoten selbst
  double time_start;    // Startzeit
  double time_end;     // Endzeit
  double time;                          // Laufzeit = (End-Start)
  activation() { nr=proc=0; time_start=time_end=0.0; v=nil; }
  const activation& init(DFG& G,const knoten& kn,const double
start)
    { nr=kn.nr; v=G.nodenumber(nr); time=kn.time;
time_start=start;
      time_end=start+kn.time; return (*this);}
  activation(DFG& G,const knoten& kn,const double start)
    { init(G,kn,start); }
  const activation& init(const DFG& G,const node& u,const
double start)
    { knoten kn=G.inf(u); nr=kn.nr; v=u; time=kn.time;
time_start=start;
      time_end=start+kn.time; return (*this);}
  activation(const DFG& G,const node& u,const double start)
    { init(G,u,start); }
 friend void Read (activation& k, istream& is) { is>>k.nr; }
 friend void Print (activation& k, ostream& os)
   { os<<"Knoten"<<k.nr<<" auf Proz "<<k.proc<<" von
t="<<k.time_start<<" bis "<<k.time_end<<endl; }
 friend int compare (const activation& k1,const activation& k2)
   { double t=(k1.time_end - k2.time_end);
     if (t==0.0) return 0;
              else if (t>0.0) return 1; else return -1; }

};

LEDA_TYPE_PARAMETER(activation);

// Ein-Prozessor-Schedul
class Pliste: public list<activation>
{private:
  int nr;
 public:
 double freetime; // Zeit, ab der Prozessor wieder frei ist.
 Pliste() { list<activation>::clear(); nr=0; freetime=0.0; }
 virtual void clear() { list<activation>::clear(); freetime=0.0; };
 Pliste(const Pliste& robj) { (*this)=robj; }
 friend void Read (Pliste& k, istream& is) { k.read(is); }
 friend void Print (Pliste& k, ostream& os) { k.print(os); }
 friend int compare (const Pliste& k1,const Pliste& k2) { return
(k1.nr-k2.nr); }
};

LEDA_TYPE_PARAMETER(Pliste);

inline string int_string (int n) { return string("%d",n); }

#endif
```

```
// reach.h
// All about reachability of States in a DFG
//
// Definition of Class RGraph

#ifndef reachHEADER
#define reachHEADER

#include <LEDA/basic.h> // f. bool
#include "imatrix.h"
#include "dfg.h"

// Reachability Graph for DFG
class RGraph:public GRAPH<ivektor, int> {
 private:
  bool calculated;
  DFG *Basis;
  int knotenzahl;
  int kantenzahl;
  imatrix IM; // Inzidenzmatrix
  ivektor qmax; // Komponenten=Ausfuehrungshaeufigkeit d.
Knoten
  bool limited_search;
  ivektor nullvek; // Nullvektor der Dimension = Kantenzahl
  bool check_DFGnode(int knotennr, const ivektor& d1, node
d1_zustand, const ivektor& qsum); //_rekursiv
```

```
void check_Statenode(const ivektor& d1, node d1_zustand,
const ivektor& qsum);  //_rekursiv
  bool R_check_DFGnode(int knotennr, const ivektor& d1, node
d1_zustand, const ivektor& qsum); //_rekursiv
void R_check_Statenode(const ivektor& d1, node d1_zustand,
const ivektor& qsum);  //_rekursiv

 public:
  list<node> start_states;
  list<node> end_states;
  bool live; // DFG ist live gdw. RG. strongly connected ist.

  RGraph(); // : GRAPH<ivektor, int> (); // Std.Konstruktor
  RGraph(const RGraph& origin); // Copy-Konstruktor
  RGraph(DFG& origin); // Konstruktion aus DFG
  RGraph(const imatrix& m, const ivektor& q, const ivektor&
d0, bool limited=true);
  void init(DFG& origin); // Konstruktion aus DFG
  void init(const imatrix& m, const ivektor& q, const ivektor&
d0, bool limited=true);

  list<int> run_sequence(const ivektor& d1, const ivektor& d2)
const;
  node state_node(const ivektor& d) const; // nil if none exists
  bool is_reachable(const ivektor& d) const;  // d erreichbar ?
  bool is_reachable(const ivektor& d, const ivektor& start) const;
// d erreichbar von start
  bool shortest_path(const ivektor& d, const ivektor& start,
list<int> exe_seq) const;  // kuerzeste Aktivierungssequenz von
start->d
  void calc_dmin(ivektor& dmin) const;
  void write_end_states(ostream& out) const;
  void write_start_states(ostream& out) const;
};

//--------------------------------------------------------------------------

// jeweils true, wenn NMD vorhanden:
bool DelNMD_lokal(DFG& G, const RGraph& RG);
bool DelNMD_path(DFG& G, const RGraph& RG, ivektor&
w);

#endif
```

```
// ilp.h
//
// Schnittstelle zum Programm "LPSOLVE"
//
// (ganzzahlige) lineare Programmierung

#ifndef ILPHEADER
#define ILPHEADER

#include "dfg.h"
//#include "imatrix.h"

// ILP-Loeser
bool solve_ILP(const imatrix& A, ivektor& x, const ivektor&
b);
// true, wenn Integer-Loesung fuer Ax=b existiert -> x

// LP-Loeser: max wgt*d s.t. C*d=WST
bool opt_delays(const ivektor& wgt, ivektor& d, const imatrix&
C, const ivektor& WST);

// LP-Loeser mit zusaetzlichen Ungleichungen (NMD-Cut-
Planes)
bool opt_delays_N(DFG& Gr, const ivektor& wgt, ivektor& d,
const ivektor& w, const imatrix& C, const ivektor& WST, const
list<edge_list>& allpaths, ivektor& NMD);

//bool opt_delays_N(DFG& Gr, const ivektor& wgt, ivektor& d,
ivektor& w, const imatrix& C, const ivektor& WST, const
list<edge_list>& NMDpaths);

// LP-Loeser: max wgt*d s.t. d-G*r=d0
bool opt_delays2(const ivektor& wgt, const ivektor& d0,
ivektor& d, ivektor& r, const imatrix& G);

#endif
```

```
/* schedule.h

   Tools zum manuellen und automatischen Scheduln
   `Geradeausmethode`:
   Aktivierung aller startbaren Knoten parallel;
   Ermittlung des naechsten Fertigstellungszeitpunkts;
   dort wieder Start aller startbaren Knoten;
   Abbruch, wenn sich Delayzustand wiederholt hat,
   d.h. frueher schon einmal dieser Zustand auftrat.
   -> Iterationsperiode*J (Wenn alle Delays auf einer Kante
waren)
*/


#ifndef SCHEDULEHEADER
#define SCHEDULEHEADER

#include <math.h>
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>
#include <LEDA/point.h>
#include <LEDA/window.h>
#include <LEDA/array.h>

#include "dfg.h"
//#include "imatrix.h"

typedef array<Pliste> ScheduleTyp;

class moment
{ public:
  double time;
  ivektor vsum;
  ivektor d;
  moment() { time=0.0; vsum.init(1); d.init(1); };
 friend void Read (moment& k, istream& is) { is>>k.time; }
 friend void Print (moment& k, ostream& os)
   { os<<k.time<<": "<<k.d<<"  "<<k.vsum; }
 friend int compare (const moment& k1,const moment& k2)
   { double t=(k1.time - k2.time);
     if (t==0.0) return 0;
              else if (t>0.0) return 1; else return -1; }
};

LEDA_TYPE_PARAMETER(moment);

typedef list<moment> MomenteTyp;

//void clear_schedule(array<Pliste> &pl);

double plot_schedule(window &W, ScheduleTyp & pl,int procs,
int ycoord=10,double sk=-1);

void init_schedule(DFG& G, ScheduleTyp &sched);

int activate(DFG& G, node v, ScheduleTyp& schedule, int
procs);
// -1=No Proc. free; 0=Node not runnable; 1=OK

bool nexttime(DFG& G);
// 0=all nodes released; 1=OK; 2=OK & period
```

```
bool auto_schedule(DFG& G, ScheduleTyp& schedule, int
maxprocessors, int& processors, double &period, long& k);

#endif
```

```
/*****************************************************
****
+  dfg_edit.h
+  Interaktive graphische Ein- und Ausgabe
*****************************************************
****/

#ifndef DFG_ED_HEADER
#define DFG_ED_HEADER

#include <math.h>  // sin,cos...
#include <LEDA/window.h>
#include <LEDA/plane_graph.h>
#include "dfg.h"
#include "imatrix.h" // int2string
#include "reach.h"
#include "schedule.h"

// Initialisierung + Window-Zuweisung
void window_init(window&, DFG&);
// Hauptroutine, Rueckgabewert Menuepunkt:
int graph_edit(window&, DFG&, bool redraw=true);
// Erreichbarkeitsgraph:
int RG_edit(window&, RGraph&);
// Menue erweitern:
int new_menu_entry(const string& s);
// Schedul verankern:
void set_schedule(ScheduleTyp& Sched, int procs);

#endif
```

12 Glossar

12.1 Fachbegriffe aus der Literatur

Wegen der fast ausschließlich englischsprachigen Fachliteratur wurde das „Glossar der Fachbegriffe“ in Englisch ausgeführt. Referenzen mit dem Symbol „§“ sind in dieser Arbeit zu finden. Für Begriffe der Parallelverarbeitung ist zusätzlich das Glossar [c1] zu empfehlen.

:Begriff	:Erläuterung	:Referenz
APG=acyclic precedence graph	visulization of dependencies for limited J periods	[17]
admissible schedule	no deadlock, only runnable nodes executed	[1]-1241
arc	signal path; transports and buffers data (samples)	[17]
asynchronous graph	data rates cannot be specified statically	[18]-33
atomic SDF graph	very simple nodes, performing elementary indivisible operations	[1]-1236
backward-concurrent-free GPN (bc)	every transition has outdegree=1	[40]-260
backward-conflict-free GPN (bf)	every place has indegree=1	[40]-260
balanced schedule	schedule based on consistent DFG's	[30]
Bellman Ford algorithm	graph algoritm for finding the shortest path	[58],[63],[33]
basic variables (LP)	variables that contain LP solution, other variables are zero	[61],[62],[68]
BFS	breadth first search in a graph visiting all nodes	[41]-72, [58]-490
Big-M method (LP)	method for getting initial feasible point, if not apparent at start	[61]-161
blending problem (LP)	optimization of mixing ingredients for maximum cash gain	[61]-80
blocking; blocked schedules	collection of J (minimal-)periods for an APG	[1]-1241, [20]-65
bounded (k) (petri-nets)	number of tokens does never exceed k in each place	[37]-547
bounded fairness (petri-nets)	t1,t2 are B-fair if max. nbr. of t1-firings while t2 sleeps is bounded	[37]-550
bounded SDF/petri-net	no data overflow on arcs (buffer); consistent sample rates	[28]
branch and bound	algorithm for solving ILP’s by partitioning solution space recursively	[61],[62]
buffering	storing data samples after generation before consumption	[18]-26
canonical multirate graph		[27]
capital budgeting problem (LP)	determining optimal financial decisions for investments	[61]-73
chord	single arc (belonging to only one fund. loop) connecting a tree to a cyclic graph	[28]
circular buffer	fast (but bounded) implementation of FIFO queue	[17]-1242
class S algorithm	a~ that schedules runnable nodes and terminates if deadlocked	[17]-1240
clustering	hierarchical nodes may contain a 'subnet'	[25]
CM=comfortable marking	delay-WST, such that any following node can be scheduled q times at once	§
coarse grain data flow	opposite of fine grain data flow	[23]
complementary place transformation	trafo finite capacity net -> infinite capacity net	[37]-543
component of a graph	all nodes of a component are connected	[41]-72
computable graph	live+bounded=>Delay distribution equal after period	[28]
computation graph	basically equivalent to SDF graph; special case of petri nets	[17]-1238
concurrency	property of nodes, that can be scheduled parallel	[17]-1240
conservative petri-net	special case of structural boundedness	[37]-567
consistent petri-net	every trans. occurs at least once in fir-seq s from M0->M0	[37]-567
consolidation of a subgraph		[20]-71
constraint (LP)	row (in-)equality in LP like: 2x+3y-5z>3 defining hyperplane for feasible region	[61]
contention	conflict about ressources being used by multiple processors	[17]-1238
contiguous schedule	nodes are scheduled without gap or idle time	[17]-190
controllability (petri-nets)	any marking is reachable from any other marking	[37]-566,
convex set (LP)	LP’s feasible region is convex; any point betw. feasible points is also feasible	[61]

coverability graph (petri-n)	derived from coverabiltity tree	[37]-551
coverability tree (petri-nets)	nodes represent markings in R(M0), arcs=transition firings	[37]-550
coverable marking M (petri n.)	c~ if ex. marking M' in R(M0) with M'(p)>=M(p) for each p in net	[37]-548
CPM=critical path method	method for scheduling projects with precedences, determining total time needd.	[61]
critical loop	loop of directed arcs with maximum sum execution time	[19]-182, [24]-863, [20]-83
critical path	path with longest sum execution time in precedence graph	[19]-181
cumulative delay count	delay count @ end of path, all prev. delays transf. to end	[19]
cutset transformation	Shift of delays from incoming arcs to outgoing arcs	[19]-184; [21]-1883
cutting planes	algorithm for solving ILP's by reduction to partial integer polyhedron	[61],[68]
cycle time (petri-nets)	time to complete firing sequence leading back to initial marking	[37]-570
cycling vector (f. FSFG)	defines relative permutation of uP's in schedule matrix	[24]-861
cyclo-static schedule	tasks run alternately on all processors	[19]-182, [20]-67
DAG=directed acyclic graph	directed graph without any cycles	[58]-543
data stationary coding	special assembly programming method	[2]
data-driven	execution of nodes depends on amount of input data	[17]-1235; [22]-1279
dead transition/node	can never be fired in any sequence in L(M0)	[37]-548
deadlock-freeness (p-n)	at least one transition is enabled@every reachable marking	[39]-351
delay	sample offset between in/out on arc; buffer content (No. of samples)	[17]-1236
delay conservation law	C*dr=const.	[37],[29]
delay optimal realization	satisfies the periodic delay bound (PDB)	[24]-864
demand ratio	total number of memory cycles per instruction cycle	[1]-14
determinate (computation graph)	same computations are performed by any proper execution	[18]-25
Dijkstra's algorithm	graph algorithm for finding shortest path, edge weights must be positive	[58]
DFG=data flow graph	generic; not necessarily known sample rates	[18]
DFS=depth first search	visiting all nodes in a graph trying adjacent nodes first	[41]-71, [58]-482
dual (petri) net	net achieved by commuting places and transitions	[29]-351
dual LP problem	LP-problem with transponed constraint matrix and exchanged rhs+cost	[61],[63],[68]
edge	connecting arc between nodes	[58]
entering variable (LP)	nonbasic variable to be taken into the set of basic variables during pivot	[61]
ero=elementary row operation (LP)	allowed for matrix equations: scaling, addition of rows, exchanging rows	[61]
excess variables (LP)	variables for converting inequations to eqns.: ax>b <=> ax-e=b, e>=0	[61]
exhaustive search	algorithm testing all possible combinations of topics. Often O(e^N)	[58]-702
expansion rules (LSMG's)	live save marked graphs synthesis rules adding arcs/nodes	[37]-562
extended petri-nets	petri-nets with inhibitor arcs -> Turing machines	[37]-546
extended transitive arc	arc A->B, if other path A->B exists with lower delay count	[19]-181
feasible region (LP)	LP solution space: polyhedron defined by hyperplanes (constraints)	[61]
feasible solution (LP)	LP solution which satisfies all LP constraints. Located in polyhedron	[61]
feed forward cutset	cut through non-recursive arcs, s.t. 2 seperate graphs are generated	[17]-1244, [20]-68
feedback arc set (FAS)	graph without FAS is acyclic	[37]-556
finite capacity petri-net	each place p has an associated capacity K(p)	[37]-543
finite state machine	subclass of petri-nets; each trans. has exactly 1 in/out arc	[37]-544
fire (node)	node performs computation if input data available	[17]-1235
firing vector u	vector containing only 0's but one 1 in the position of firing tr.	[37]-552
Fixed point arithmetic	DSP having integer valued registers + ALU	[1]
Floating point arithmetic	numerical calculations with mantissa and exponent	[1]
Floyd-Warshall algorithm	graph algorithm for finding the shortest path	[58],[33],[63]
Ford-Fulkerson algoritms	graph algorithm for solving maximum-flow problems	[61]-421
forward-concurrent-free GPN (fc)	every transition has indegree=1	[40]-260
forward-conflict-free GPN (ff)	every place has outdegree=1	[40]-260

FSFG=fully specified f.graph	node operations are atomic operations of DSP	[24]-859
fully dynamic scheduling	node assign+order+timing done at runtime	[22]-1281
fully static scheduling	node assign+order+timing done at compile time	[22]-1280
fully-static schedule	all iterations of a task are scheduled in same DSP	[19]-182
fundamental circuit matrix	matrix consisting of S-invariants; C*G=0	[37]-552; [39]-352; [28]
gain of a weighted circuit	product over all (output/input)-rates	[39]-353
Gauss-Jordan method	algorithm for solving linear equations (matrix Ax=b)	[61]
generalized petri net (GPN)	corresponds to weigted T-system, multirate system	[40]-251
GFG=generic flow graph	directed flowgraph of processing nodes, data edges, delays	[24]-859
graph coloring	putting additional information on nodes/arcs	[20]-105
greatest dead marking	maximal weight of marking for which deadlocks are possible	[39]-363
HDM=heaviest dead marking	all markings with WST>HDM are live	[39],§
high-level petri-net	individual tokens allowed	[37]-572
home state M' (petri-nets)	for each M in R(M0), M' is reachable from M	[37]-548
homogenious SDF	all nodes produce same amount of samples per invocation	[17]-1236
Hu-level scheduling	critical path method using levels for nodes	[18]-32
Hu-level scheduling algorithm	approx. to opt. schedule (level=time on path to the terminating node)	[18]-32
ILP=integer linear programming	optimization of x, subject to Ax>b, x with integer components	[61], [62], [63], [68]
immediate predecessor	A is ~ to B, if there is a directed arc from A->B	[19]-180
immediate successor node	B is ~ to A, if there is a directed arc from A->B	[19]-180
incidence matrix	matrix of integers containing arc weights	[37]-551
inconsistent sample rates	invocation of nodes leads to buffer overflow (not bounded)	[18]
indegree of a node	number of arcs entering the node	[40]-260
index mapping transformation	trafo. of algorithms (eqns) to achieve max. concurrency	[21]-1893
infinite capacity petri-net	each place can accomodate an unlimited number of tokens	[37]-543
inherent processor inefficiency	1- true efficiency for all uP's	[24]-864
inhibitor arc (petri-nets)	disables transition, if input place has a token	[37]-546
initial marking M0	contents of places in petri-nets = delays in SDF-graphs	[37]-542
initial node	has no predecessor (Source)	[19]-181
inter-iteration precedence	constraint among iterations (arc with delay)	[19]-180; [21]-1880
interlocking	automatic pipeline-protection by cycle-time stretching	[2]
intra-iteration precedence	constraint for node A to be executed after B within an iteration	[19]-180; [21]-1880
iteration period	length (runtime) of one cycle of a blocked schedule /J	[17]-1236
iteration period bound	upper bound for it.per. (affected by loop computation time & delays)	[17]-1236; [19]-1883
iteration tile	area in gantt-chart (schedule matrix) for one iteration	[24]-861
Karmarkar's method (LP)	polynomial time algorithm for solving LP problems	[61]-604
L(M0)	Set of all possible firing sequences from initial marking M0	[37]-547
L1-live transition t	t can be fired at least once in some sequence in L(M0)	[37]-548
L2-live transition t	t can be fired at least k times (k>1) ...	[37]-548
L3-live transition t	t appears infinitely often in some firing sequence in L(M0)	[37]-548
L4-live transition t	t is L1-live for every marking M in R(M0) {reachable}	[37]-548
large grain data flow (LGDF)	large granularity, complex operations in nodes	[17]-1237
linear optimization	finding vector x with f(x)=Max. and A*x<b	[58]-687
linaer programming	solving linear optimization problems (usually simplex algorithm)	[58]-687
linear vectorization	vectorization with constant vect.-factor for all nodes	[28]
live petri-net/SDF-graph	at least one node can fire at any time in the firing sequence	[37]-548
live, alive, liveness	deadlockfree; enough delays on arcs (no negative buffer length)	[37]
LLM=lightest live marking	smallest WST, such that graph is live	[39],§
look ahead transformation	transformation of recursion by back-substitution of equations	[21]-1885..1893

loop bound	minimum runtime for one loop	[19]-182; [21]-1883
loop bound inequity	Tl<=Dl*Tinfinity	[19]-182
loop gain GL	GL=Prod{all nodes of a loop}(bi/ai) should be =1	[38]-260, [43]-1391
LP=linear programming	optimization (max/min) of x, subject to Ax>b, solved by Simplex algorithm	[61], [62], [63], [68]
makespan	maximum completion time for all nodes	[22]-1281
marked (+directed) graph	=event graph; Petri net with exactly one in/out arc on place	[18]-26
maximal loop	loop doesn't contain a transitive arc	[21]-1883
maximum flow problem	network with arcs having capacities; optimizing flow from source to sink node	[61]-412
MCNFP=minimum cost network flow	problem; LP-problem efficiently solved by network-simplex	[61]-448
mean flow time	average of firing times of all nodes	[22]-1281
MIMD	multiple instruction stream-multiple data stream	[24]-862
MIMO	multiple-input multiple-output (nodes in DFG)	[21]-1890
minimal support invariant	vector with smallest possible int. values as S-/T-invariant	[37]-568
minimum cycle time (petri-n)	lower bound for cycle time for M0->M0 via fire-sequence s	[37]-570
multiperiod decision problem (LP)	decisions are made during current period to influence future periods	[61]-95
multiperiod work scheduling (LP)	optimizing work-force ressources for future periods	[61]-104
multi-rate SDF graph	number of samples in/out not equal for each node	[17]-1238
multiple-weighted marked graphs	equivalent to multiple rate data flow graphs	[38]-259
multirate retiming	transformation on delays: dr(e)=d(e)+o(u,e)*r(u)-i(v,e)*r(v)	[29]
network simplex method (LP)	algorithm for solving MCNFP efficiently (only additions and subtractions)	[61]-459
neutral WMG	marked graph with every loop gain =1 <=> rank(O-I)=s-1	[38]-260
NMD=non movable delay	delay on arc/path, on which never exist less than dNMD delays	§
node	computational unit; performs operation on input data delivering output data	[17]
node assignment	decision what node-function is executed on what DSP	[26]
non-preemptive scheduling	task switch only allowed after completion of all computations	[22]-1281
non-terminating	schedule can run forever without deadlock	[18]-26
nonoverlapped schedule	execution of iteration (n+1) begins after ex. of all tasks of nth it.	[19]-181
normalized delay	^D:=Di/ai*qu=Di/bi*qv for arc i from u->v in multirate DFG	[21]-1885
NP-complete problem	in "set of all problems, solvable (only) by nondeterministic algorithms"	[58]-717;[70]
objective function (LP)	weighted sum of free variables in LP problem, to be optimized (max,min)	[61]
optimal solution (LP)	point in feasible region with largest objective function value	[61]
optimum unfolding factor Jopt	Jopt=kgV(Sum Dl); DFG can be scheduled rate-optimally	[19]-192
optimum vectorization	vectorization with v~ factor equals linear vectorization bound	[28]
ordinary petri-net	all arcs are weighted by 1's	[37]-543
outdegree of a node	number of arcs leaving the node	[40]-260
overlapped schedule	any task of it. (n+1) is started before all task of nth it. completed	[19]-181
PAPS	periodic admissible parallel schedule	[18]-26
partial runtime	time from start of node to arrival of data at special output	[20]-128?
PASS	periodic admissible sequential schedule	[18]-26
path	path A->B ex., if B can be reached over directed arcs + nodes from A	[19]-181
perfect-rate PRG/DFG	one delay on each loop; rate-optimal scheduling possible	[19]-178
period transformation matrix	defines row permutations of schedule matrix	[24]-861
periodic delay bound (PDB)	max. periodic throughput delay at fastest possible rate	[24]-864
periodic throughput delay	max. avg. time delay over all dir. paths betw. in & out	[24]-864
persistent petri-net	for any two enabled t1,t2 , firing of t1 will not disable t2	[37]-548
PERT=program eval. & review techn.	estimation of duration of a project, node duration not known witth certainty	[61]-438
petri-net transformations	rules modifying graph preserving live-, safe-, boundedness	[37]-553
petri-nets	tool for describing systems similar to flow charts	[37]-541
pipelining	dividing computation path into stages, separated by delays	[20]-73
place (Petri nets)	corresponds to arcs in DFG; Symbol: circle with points=delays	[38]-259
planar graph	graph can be displayed without crossing edges	[41]-76
potentially reachable marking	M is ~, if ex. s, t.h. M=M0+A*s holds, s rational. Equals C*(M-M0)=0	[39]-352

preemptive scheduling	interruption of node computations possible	[22]-1281
primal GPN (petri-net)	conservative + strongly connected + no str. conn. subset ex.	[40]-259
principal lattice vector	defines displacement in time&processor space (Cycl.Sched)	[24]-868
processor assignment vector	C(k); maps rows of schedule matrix onto uP's	[24]-862
processor bound (PB)	minimum number of uP's need'd f. reaching iteration period	[24]-863
processor displacement	shift of executing uP for one task in cyclo-static schedule	[21]-1883
processor efficiency (n)	percentage of utilization of all uP's	[24]-864
processor modulo constraint	requires schedule to be infinitely extendible	[24]-868
processor optimal scheduling	using minimum number of uP's	[24]-864
production process models (LP)	optimization of running processes stage by stage	[61]-90
PSSIMD	parallel single instruction stream - multiple data stream	[24]-867
pure petri-net	net has no self-loops	[37]-543
rate gain of a node	g(v,p)=o(v,p)/i(v,p) on path p	[29]
rate-optimal (schedule)	actual iteration period = iteration period bound	[19]-178
reachability of retimed states	delay vector space for retimed states equals reachability spc.	[29],
reachability tree (petri-nets)	contains all reachable markings (=coverability f. bounded nets)	[37]-550
reachable marking Mn	reachable, if exists sequence s of firings transforming M0->Mn	[37]-547
reachablitity graph (petri-n)	derived from reachability tree	[37]-551
recursive doubling	technique for effective computation of back-substituted eqns.	[21]-1886
recursive nodes	nodes located in a loop	[19]
reduced dependence graph	periodic acyclic precedence graph (one perion displayed)	[17]-1238
register constrained retiming	force a node to have one/no delay on each in(out) arcs	[19]-189
reduced cost (LP)	information, how LP’s optimal solution changes by changing obj.funct.params,	[61]-216
regular stream flow graph	"=SDFG"	[30]
repetitive petri-net	net is ~ if ex M0 & firing-seq. s, s.t. every trans. occurs inf. often	[37]-567
reservation table	Zeilen: Ressourcen(RAMi,ALU,...); Spalten: Instr. Cycle	[2]-5
resource-time product (RTP)	(number of tokens)*(time the tokens stay in a place)	[37]-570
retiming	redistribution of delays, creating new precedence relations	[19]-184
reverse (petri) net	net achieved by reversing arcs	[39]-351
reverse dual (petri) net	net with transponed incidence matrix vs. original net	[39]-351
reversible petri-net	M0 is reachable from each M in R(M0) {get back to initial state}	[37]-548
reversible system (petri-n)	initial marking M0 can be reached from any reachable marking	[39]-351
rhs=right hand side	right (constant) side of a matrix (in-)equation or LP problem	[61]
runable node	enough data on arc for execution	[17]
S-/P-invariant, S-/P-flow	vector y (dim. m=place count) is ~ if A*y=0	[37]-568
safe petri-net	= 1-bounded net (buffer contains max. 1 token)	[37]-548
sample threshold	node can only be fired if s.th. tokens (samples) exceeded	[18]-25
schedule matrix	represents one period of schedule; row=uP-nbr., col=time slot	[24]-860
scheduling	determination, when nodes are executed and where	[18]
self-loop (Petri nets)	transition t has place p both as input & output place	[37]-543
self-timed scheduling	node assign+order @compile time; timing @runtime	[22]-1280
sensivity analysis (LP)	determining, how LP’s paramters affect optimal solution	[61]-207
separator	element in synchronous circuits marking border to new iteration	[35]
shared buffer memory	single memory is used by different uP's and processes	[20]-105
shimming delay	slack runtime on an arc	[20]-82
shortest path in a graph	path between two nodes with minimum cost (cost on edges)	[58]-541, [41]-73, [61]-406
short-term financial planning (LP)	production optimization for maximum gain	[61]-77
side effects	node A has influence on node B without arc A->B	[17]-1235
SIFG	shift invariant flow graph; structure doeasn't change with time	[24]-859
signal flow graph	SDF graph describing linear single-rate systems	[17]-1238
SIMD	single instruction stream - multiple data stream	[24]-866
simplex algorithm	method for solving LP problems by row manipulations, generally not polynomial	[61],[62],[68]

sink transition (Petri nets)	t~ without any output place	[37]-543
siphon and trap (petri-nets)	subset of places with special properties	[37]-556
SISO	single-input, single-output (nodes in DFG)	[21]-1890
slack time	difference between iteration bound - loop bound on one DSP	[19]-182, [24]-863
slack variables (LP)	variables to convert inequations to eqns.: ax<b <=> ax + s = b , b>=0	[61]-124
source transition (Petri nets)	t~ without any input place	[37]-543
spanning tree of a graph	tree consisting of the arcs of the graph only	[58]-475
SSIMD	skewed single instruction stream - multiple data stream	[24]-867
standard form of LP	all constraints are equations, all variables nonnegative	[61]-120
state equation f. markings	eqn. describing changes of M due to firing sequence s	[37]-552
static allocation scheduling	node assign @compile time; order+timing @runtime	[22]-1281
static buffering	specific memory location used instead of FIFO buffer	[17]-1242
stochastic petri-net	each transition has exp-distributed random time (delay)	[37]-570
strict transition rule (petri-nets)	nb. of tokens in output places must not exceed capacity	[37]-543
strongly connected components	subgraphs in which each node is reachable from another	[58]-545, [41]-72
strongly connected graph	each node can be reached from any other by a path	[58]
structural B-fairness (petri-n)	B-fair for any initial marking	[37]-569
structural boundedness	system is bounded for every initial marking	[39]-351
structural boundedness (p-n)	bounded for any finite initial marking M0	[37]-567
subclasses of petri-nets	nets with limited abilities	[37]-553
sum of weighted tokens	sum {all loop places}(mi/bi*R(i+1))=sum(mi/ai*Ri)	[38]-261
support of an (S-/T-)invariant	set of places/trans. corresp. to nonzero entries in inv-vector	[37]-568
sychronous system	all sample rates are rational multiples of all other sample rates	[17]-1237
synchronic distance (petri net)	metric between t1,t2. d12=max|/s(t1)-/s(t2)|	[37]-549
synchronic distance matrix	matrix containing synchronic distances betw. nodes i & j	[37]-565
synchronous data flow (SDF)	data flow, where number of in/out samples is known initially	[17]
systolic array	field of processors connected in a special manner	[17]-1238, [50]-654
systolic scheduling	using modular regular processing elements	[24]-866
T-invariant, T-flow	vector x (dim. n=transition count) is ~ if A^T *x=0	[37]-568
terminal node	has no successor (Sink)	[19]-181
terminating transition	number of invocations is limited	[40]-254
throughput (Durchsatz)	reciprocal of iteration period	[17]-1236
throughput delay	time delay betw. input and corresp. DFG-output	[24]-864
tiling constraint	copies of uP-schedule shifted must not overlap	[24]-869
time displacement	period between same tasks in cyclo-static schedule	[21]-1883
time stationary coding	special assembly programming method	[2]
time-driven	execution of nodes is determined by time	[22]
token	data stored in places (arcs), used by trans. (nodes)	[37]-545
token bound (petri-net)	upper bound Mm(p) on number of tokens a place p can have	[37]-569
token distance matrix (MG's)	min. token content among all posbl. paths from node i->j	[37]-564
topological sorting of a graph	ordering of nodes of an acyclic graph, s.th. all arcs point right	[58]-543, [41]-71
topology matrix	matrix containing input/output rates; Input:nodes, output:arcs	[17]-1239
transition (Petri nets)	corresponds to nodes in DFG; Symbol: thick line	[38]-259
transitive closure (transitive Hülle)	graph obtained by adding all edges (u,v) if path u->v exists	[58]-539
transitivity	property of arc A->B, if (other) path A->B exists (same delays)	[19]-181
transportation problem (LP)	optimizing costs for node-to-node transport	[61]-349
transportation simplex (LP)	efficient algorithm like simplex but only additions and sub's needed	[61]-372
tree (special graph)	graph without any cycle (loop) connecting all nodes	[58]-475
trellis	diagram with states + time steps + connections	[20]-104
TSP=travelling salesman problem	problem of visiting all cities in minimum time - NP-complete	[61]-535
two-phase simplex method	method for getting initial feasible point, if not apparent at start	[61]-167

unconditional fairness (global)	firing sequence s is finite or every trans. appears inf. often	[61]-550
unfolding	graph transformation preserving inter-iteration precedence constraints	[19]-186
unfolding factor J	number of basic iterations joined to form a new block itertion	[19]-185
unit-gain transformation	graph transformation to set node gain to one (same rates on node)	§
unit-rate SDF graph	number of samples in/out always unity (1)	[17]-1238
unit-rate transformation	graph transformation multirate->unitrate	[20]
unrestricted in sign variables (LP)	real variables to solve for, not suitable for simplex demanding vars>=0	[61]-172
Vectorisation of a graph	increase in/out-rates by multiples of previous value	[27], [28]
vectorization bound (linear v~)	maximum vectorization factor leading to computable graphs	[28]
vectorization factor	number by which input/output-rates of a node are multiplied	[28]
vertex, vertices	= node in a graph	[58]
weak transition rule	transition may fire if enough input tokens available	[37]-543
weight of a marking (P-n)	W(m):=Y^T*M for a circuit C (Y*C=0)	[39]-362
weighted circuit (petri-nets)	directed loop with no additional arcs, places, transitions	[39]-353
weighted sum of tokens	M*I [(marking vector)*(S-invariant)] =const for all M in R(M0)	[37]-563
weighted T-system	petri net with bulk input/output counts; multirate case	[39]-348
weighted token	=mi/bi*Rv=mi/ai*Ru=normalized delay on arc i: u->v	[38]-261
well-behaved GPN	left and right annullors of inc.-matrix G have pos. integer solution	[40]-258
well-behaved system	live and bounded	[39]-352
work-scheduling problem (LP)	determining minimum cost for satisfying work-force requirements	[61]-69
WST=weigted sum of tokens	M*I [(marking vector)*(S-invariant)] =const for all M in R(M0)	[37], [39]

12.2 Index

—A—

Adjazenzmatrix 15
Aktivierungssequenz 20
  Algorithmus 55
andere Äquivalenztransformation 70
Anfangswerte 74
Anfangszustand 20
  Anwendungen 75
Anwendungen von Retiming 75
Anzahl ganzzahliger Zustände 41
atomare Knoten 10
Ausführungszeit 14
Ausgangsknoten 12
Auslastung 79
automatische Tests 95
Azyklischer Präzedenzgraph 32

—B—

Bellman-Ford-Shortest-Path 73
Berechenbarkeit 20
Berechnung des Retiming-Vektors 54
Beschränktheit 20
Bestimmung aller Pfade 58
Beweis durch Einschränkung 46
Bipartiter Graph 21
Blockdiagramm 10
Blocking 36
Blocking-Faktor 32; 37
Branch and Bound 44

—C—

C++ 90
Chinesisches Restwert-Theorem 58
Circuit-Matrix 33
Clusterbildung 37
CM 26
Codegenerator 4
Compilation 93
Computation Graphs 11
coprime Normschleife 40
COSSAP 4
CPM 50
Cutting Planes 44;67
cyclo-statisches Schedul 49

—D—

d-Vektor 24
Datenfluß-Konzept 12
Datenflußgraph 11
datengesteuert 9
Deadline 46
deadlock 20
Delay 12
Determinismus 22
digitale Signalverarbeitung 10
distributed memory 7
DMA 7
DSP’s 5
Dualität 43
Durchsatz 14;77
Durchsatz-Grenzwert 85
Durchsatz-Verbesserung 75
Durchsatzgrenze 80
dynamisches Scheduling 48

—E—

Ebenengleichungen 20
Eingangsknoten 12
Einheitsraten-DFG 13
Einheitsraten-Transformation 30
Einheitsverstärkungs-Transformation 39
Endliche Automaten 22
Entfernung von NMD's 61
Entscheidungsproblem 45
Entscheidungsvariablen 42
Erreichbarkeit 27
Erreichbarkeitsgraph 27
Exponentielle Komplexität 44

—F—

Free-Choice Net 22
fundamental circuit matrix 18
fundamentale Schleife 33

—G—

Gantt-Chart 15
gewichtete Delaysumme 24
graphische Oberfläche 94
Gültigkeit des Multiraten-Retiming 59

—H—

Harvard-Architektur 4
HDM 25
Heuristik 50
Hierarchisierung 33
HLF 50
Hocker 61
homogenes diophantisches Gleichungssystem 54

—I—

ILP-Problem 43
Initial-Programm 74
Inter-Prozessor-Kommunikation IPC 52
Interlocking 6
Invarianz der WST 40
Inzidenz-Matrix 15
iteration bound 77
Iterationsperiode 14;77
Iterationsperiodengrenze 78

Iterator-Makros 91

—K—

Kanten 12
Kanten-NMD’s 60
Kausalität 15; 20
Klasse *co-NP* 46
Klasse DFG 93
Klasse NP 45
Klasse S Algorithmus 47
Klassenbibliotheken 90
Klassifikation nach Flynn 7
Knoten 12
Kommunikations-Knoten 52
Kommunikationsleistung 7
Komplexität 44
Konstruktoren 90
konvexe Hülle 68

—L—

Latenz 15
Lebendigkeit 20
LEDA 91
Lee’s Synchroner Datenfluß 9
legales Einheitsraten-Retiming 52
lineare Grenze 82
lineare Programmierung 42
  lineare Programmierung 65
lineare Vektorisierung 38
List-Scheduler 50
LLM 25
LP- Problem 42
LP-Standardform 43

—M—

Makespan 50
Marked Graphs 22
Massiv parallele Rechner 7
maximale Aktivierung von Knoten 79
Menüpunkte 95
Methoden 90
MIMD 7
Mindestanzahl benötigter Prozessoren 78
Minimaler retimingäquivalenter Einheitsraten-Graph 69
Minimierung der Speicherelemente 76
MISD 7
Modelle 11
Modulo-Bedingung 56
Monte-Carlo-Verfahren 96
MuDaG-Bildschirm 94
Multiplizierer-ALU-Einheit 5
Multiprozessor-Schedul 48
Multiraten-DFG 13
Multiraten-Retiming 53
  neue Bedingung 56

—N—

n-Vektor 18
NMD's 60
Non-ordinary Petri-Net 21
Normgraph 40
normierte Durchsatzgrenze 84
normierter Durchsatz 82
Normschleife 40
notwendige Bedingung 53
NP-hard 46
NP-Vollständigkeit 45

—O—

objektorientierte Programmierung 90
Operations Research 10
Optimierungsrichtung 66
Ordnung 44

—P—

P-Flow 18
Packungsfaktor 83
PAPS 48
Parallelität 23
Parallelrechner 7
Parikh-Vektor 27
Partitionierung 42
PASS 47
PERT 51
Petri-Netze 21
Pfad 13
Pfad-NMD's 61
Pipeline 5
Pipelinestufen 8
Pipelining. 8
Place 21
Polyhedron 43; 44; 65
Polynomiale Algorithmen 44
potentiell erreichbare Zustände 27
präemptives Schedul 48
Präzedenz 23; 36
Precedence Constrained Scheduling 47
Prozesse 15
pseudo-polynomialer Aufwand 46

—Q—

q-Vektor 16

—R—

Raten 11
Raten-optimale Graphen 35
Ratenbedingung 61
Ratenkonsistenz 21
Räumliche Parallelität 23
Redundanz 71
regulärer Datenfluß 11
Retiming 37; 52
Retiming azyklischer Graphen 64
Retiming von Schleifen 63
Retiming zyklischer Graphen 64
Retiming-Koeffizient r 53
Retiming-Vektor-Berechnung 55
RTP 81
Rückumwandlung SR-MR 30

—S—

S-Invariant 18
sammelbare Delays 61
Schedul 15
Scheduling 42
Scheduling-Problem 46
Schedulmuster 77
Schleife 13
Schnittebenen 68
Sehne 33; 38
Selbstschleifen 16
Separator 8
Separatoren zwischen Clustern 76
Shared Memory 6
shimming delay 8
Shortest Path 73
Signallaufzeit 8
SIMD 7
Simplex 43
Singleraten-DFG 13
SISD 7
Spannbaum 18; 33
statisches Scheduling 48
Steuerbarkeit 27
Strongly Connecting 34
Strukturelle Eigenschaften 11
SWT 25
synchroner Datenfluß 11

—T—

T-Flow 16
TDM 6
Tiefensuche 17
Token 12
Topologie-Matrix 15
topologisch äquivalenter Einheitsratengraph 59
Transition 21
transitive Kanten 13
Transputer 4

—Ü—

überlappende Pfade 71
überlappendes Schedul 49

—U—

Umwandlung einer Kante 70
Umwandlung w. T-System -> PN 31
Unbewegliche Delays 60
Unfolding-Transformation 35

—V—

Vektor n 39
Vektor w 40
Vektorisierung 37
Vektorrechner 7
Vererbung 90
Vererbungshierarchie 92
Verklemmung 20
VLSI 8
Voraussetzungen Datenfluß 9

—W—

w-Vektor 18
Weighted P/T-System 21
Weighted T-Systems 22
WST 24

—X—

X-Window 94

—Y—

y-Vektor 17

—Z—

Zeitliche Parallelität 23
Zielhardware 3
Zusammenhängende Komponente 14; 34
Zusatzbedingungen f. Retiming 68
Zustandsgleichung 19
Zustandsvektor 20
Zuwächse des Durchsatzes 82

13 Literaturverzeichnis

13.1 Verwendete Literatur

[1] E.A.Lee, "Programmable DSP Architectures: Part I", IEEE ASSP Magazine, pp. 4..19, October 1988.

[2] E.A.Lee, "Programmable DSP Architectures: Part II ", IEEE ASSP Magazine, pp. 4..14, January 1989.

[3] Markt & Technik, "Marktübersicht: Digitale Signalprozessoren", Markt & Technik Wochenzeitung, S. 36..39, Heft Nr.40 v. 30. Sep.1994

[4] Texas Instruments, "TMS320C5x User's Guide", TI, 1993

[5] Texas Instruments, "The DSP Starter Kit User's Guide", TI, 1994

[6] Texas Instruments, "Parallel Processing with the TMS320C4x", TI, 1994

[7] Elektor Verlag, "Mikroprozessor-Datenbuch 1", Bibl.: 1²Bm7816

[8] B. Barrera and E.A.Lee, "Multirate Signal Processing in Comdisco's SPW", IEEE, pp. 1113..1116, 1991

[9] J.Buck, S.Ha, E.A.Lee and D.G.Messerschmitt, "Multirate Signal Processing in PTOLEMY", IEEE, pp. 1245..1248, 1991

[10] Synopsys,"COSSAP User Guide"

[11] Synopsys,"DESCARTES"

[12] Hendrik Koerner, "Codegenerierung Datenflußorientierter Programme für Multi-DSP-Architekturen", Diplomarbeit am ISS, Juni 1994

[13] Landesinitiative SOFTECH NRW, "Situation und Perspektiven des Einsatzes der parallelen Datenverarbeitung", Ministerium für Wirtschaft, Mittelstand und Technologie des Landes NRW

[14] J.A.Sharp, "Verteilte und parallele Computernetze - Architektur und Programmierung", VCH Verlag, 1989, ISBN 3-527-27814-1

[15] G.V.Wilson, „A Glossary of Parallel Computing Terminology", IEEE Parallel & Distributed Technology, 1993

[16] R.Duncan, "A Survey of Parallel Computer Architectures", IEEE: Computers, Feb.1990

[17] E.A.Lee and D.G.Messerschmitt, "Synchronous Data Flow", Proceedings of the IEEE, pp. 1235..1245, vol.75 No.9 Sep.1987

[18] E.A.Lee and D.G:Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", IEEE Transactions on Computers, pp. 24..35, Vol.C-36 No.1, Jan.1987

[19] K.K.Parhi and D.G.Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding", IEEE Transactions on Computers, pp. 178..195, vol.40 No.2, Feb.1991

[20] E.A.Lee, "A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors", PhD thesis, University of California, Berkeley, 1986

[21] K.K.Parhi, "Algorithm Transformation Techniques for Concurrent Processors", Proceedings of the IEEE, pp. 1879..1895, vol.77 No.12, Dec.1989

[22] E.A.Lee and S.Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP", IEEE: GLOBECOM, pp. 1279..1283, Dallas, Texas, Nov. 1989

[23] P.D.Hoang and J.M.Rabaey, "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput", IEEE Transactions on Signal Processing, pp. 2225..2235, vol.41 No.6, June 1993

[24] P.R.Gelabert and T.P.Barnwell, "Optimal Automatic Periodic Multiprocessor Scheduler for Fully Specified Flow Graphs", IEEE Transactions on Signal Processing, vol.41 No.2, Feb. 1993

[25] Vojin Zivojnovic, "High Performance DSP Software Using Data-Flow Graph Transformations", ISS-Paper, ASILOMAR 1994

[26] Vojin Zivojnovic, Hendrik Koerner und H.Meyr, "Multiprocessor Scheduling with a-priori Node Assignment", ISS-Paper, 1994

[27] Vojin Zivojnovic, "Multiprocessor Iteration Period Bound For Multirate Data-Flow Graphs", ISS-Paper, 1994

[28] Vojin Zivojnovic, S.Ritz, H.Meyr, "Retiming of DSP Programs for Optimum Vectorization", ICASSP 1994

[29] Vojin Zivojnovic, S.Ritz, H.Meyr, "Optimizing DSP Programs Using the Multirate Retiming Transformation", EUSIPCO 1994

[30] R.Govindarajan + G.R.Gao, "A Novel Framework for Multi-Rate Scheduling in DSP Applications", ASAP'93

[31] S.H.Gerez, S.M.deGroot, O.E.Herrmann, "A Polynomial-Time Algoritm for the Computation of the Iteration-Period Bound in Recursive Data-Flow Graphs", IEEE Trans. on Circuits & Systems, Vol.39, No.1, Jan.1992

[32] D.Y.Chao, D.T.Wang, "Iteration Bound of Single-Rate Data Flow Graphs for Concurrent Processing", IEEE Trans. on Circuits & Systems, Vol.40, No.9, Sep.1993

[33] C.E.Leiserson + F.M.Rose + J.B.Saxe, "Optimizing Synchronous Circuitry by Retiming", Proceedings of Caltech Conf. on VLSI, 1983

[34] X.Hu, S.C.Bass, R.G.Harber, "Minimizing the Number of Delay Buffers in the Synchronization of Pipelined Systems", IEEE Trans. on VLSI, 1994

[35] H.V.Jagadish, T.Kailath, "Obtaining Schedules for Digital Systems", IEEE Trans. on Signal Processing, Vol.39, No.10, Oct.1991

[36] P.Zepter, T.Grötker, H.Meyr, "Digital Receiver Design using VHDL Generation from Data Flow Graphs", ISS Paper, 1995

[37] T.Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, pp. 541..579, vol.77 No.4, April 1989

[38] D.T.Chao, M.Zhou, D.T.Wang, "Multiple-Weighted Marked Graphs", Proceedings of IFAC, vol.1, pp. 259..262, Sydney, 1993

[39] E.Teruel, P.Chrzastowski-Wachtel, J.M.Colom, M.Silva, "On Weighted T-Systems", Application and Theory of Petri Nets, pp. 348..367, 1992

[40] Y.E.Lien, "Termination Properties of Generalized Petri Nets", SIAM Journal of Computing, vol.5 No.2, June 1976

[41] M.Silva, J.M.Colom, "On the Computation of Structural Synchronic Invariants in P/T Nets", APN, LNCS, Vol.340, pp.386..417, Springer Verlag, 1988

[42] T. Murata, "State Equation, Controllability, and Maximal Matchings of Petri Nets", IEEE Trans. on Automat. Control, Juni 1977

[43] R.M.Karp und R.E.Miller, "Properties of a Model for Parallel Computations: Determinancy, Termination, Queuing", SIAM J., vol.14, pp. 541-580, April 1989

[44] T.Murata, "Relevance of Network Theory to Models of Distributed/Parallel Processing", Journal of the J.Franklin Institute, Vol.310, No.1, pp.41..50, 1980

[45] K.H.Pascoletti, "Diophantische Systeme und Lösungsmethoden zur Bestimmung aller Invarianten in Petri-Netzen", Dissertation, R.Oldenbourg Verlag, ISBN 3-486-20205-7, Bibl.:Za661

[46] J.L.Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall Verlag, ISBN 0-13-661983-5, 1981

[47] J.Campos, J.M.Colom, H.Jungnitz, M.Silva, "Approximate Throughput Computation of Stochastic Marked Graphs", IEEE Trans. on Software Engineering, Vol.20, No.7, July 1994

[48] E.Teruel, M.Silva, "Liveness and Home States in Equal Conflict Systems", Application & Theory of Petri-Nets, 1992, LNCS 691, Springer Verlag

[49] J.Desel, J.Esparza, "Shortest Path in Reachability Graphs", Application & Theory of Petri-Nets, 1993, LNCS 691, Springer Verlag

[50] H RRZN, "UNIX, Eine Einführung", Uni Hanover, 7.Auflage, Juli 1993

[51] H.Erlenkötter, "Programmiersprache C", rororo-Computerfachbuch, 1990

[52] RRZN, "Die Programmiersprache C++ für C-Programmierer", Uni Hannover, 5.Auflage, April 1994

[53] S.Lippmann, "C++ Primer", Addison-Wesley Publishing, 1990

[54] N. Josuttis, "Objektorientiertes Programmieren in C++", Addison-Wesley, ISBN 3-89319-637-4

[55] K.Mehlhorn und S.Näher, "LEDA - A Library of Efficient Data Types and Algorithms", Max-Planck-Institut f. Informatik, Saarbrücken

[56] S. Näher, "LEDA User Manual. Version 3.0", Max-Planck-Institut f. Informatik, Saarbrücken

[57] P. Zepter, "An Object Oriented Generic Class System for Hierarchical Data Flow Graphs", ISS Internal Report, RWTH Aachen, April 1992

[58] R.Sedgewick, "Algorithmen", Addison-Wesley Publishing, ISBN 3-89319-402-9, 1991

[59] W.H.Press, B.P.Flannery et al., "Numerical Recipes in Pascal - The Art of Scientific Computing", Cambridge University Press, 1989

[60] I.N.Bronstein und K.A:Semendjajew, "Taschenbuch der Mathematik", Teubner Verlagsgesellschaft, Leipzig, ISBN 3-322-00259-4, 1989

[61] W.L.Winston, "Introduction to Mathematical Programming", Duxbury Press, 1995, ISBN 0-534-23046-6

[62] C.H.Papadimitriou + K.Steiglitz, "Combinatorial Optimization - Algorithms and Complexity", Prentice-Hall, 1982, ISBN 0-13-152462-3

[63] E.L.Lawler, "Combinatorial Optimization - Networks and Matroids", Holt,Rinehart and Winston, 1976, ISBN 0-03-084866-0

[64] D.B.Johnson, "Finding all the Elementary Circuits of a Directed Graph", SIAM J. on Computers, Vol.4, No.1, Mar.1975

[65] R.Blahut, "Theory and Practice of Error Control Codes", Addison-Wesley, ISBN 0-201-10102-5

[66] W.M.Schmidt, "Analytische Methoden für Diophantische Gleichungen", Birkhäuser Verlag, ISBN 3-7643-1661-6, Bibl.:Bb1498

[67] S.Lang, "Encyclopaedia of Mathematical Sciences, Number Theory III", Springer Verlag, ISBN 3-540-53004-5, Bibl.:Bb1690

[68] A.Schrijver, "Theory of Linear and Integer Programming", J.Wiley&Sons, ISBN 0-471-90854-1

[69] D.E.Knuth, "The Art of Computer Programming - Seminumerical Algorithms", Addison-Wesley, ISBN 0-201-03822-6

[70] M.R.Garey, D.S.Johnson, "Computers and Intractability - Theory of NP-Completeness", W.H.Freeman and Company, ISBN 0-7167-1045-5

[71] V.Zivojnovic, R.Schoenen, H.Meyr,“Liveness and Reachability of Nonordinary Marked Graphs“, Vorschlag f. Konferenz VLSI’95

[1] E.A.Lee, "Programmable DSP Architectures: Part I", IEEE ASSP Magazine, pp. 4..19, October 1988.
[2] E.A.Lee, "Programmable DSP Architectures: Part II ", IEEE ASSP Magazine, pp. 4..14, January 1989.
[3] Markt & Technik, "Marktübersicht: Digitale Signalprozessoren", Markt & Technik Wochenzeitung, S. 36..39, Sep.1994
[4] Texas Instruments, "TMS320C5x User's Guide", TI, 1993
[5] Texas Instruments, "The DSP Starter Kit User's Guide", TI, 1994
[6] Texas Instruments, "Parallel Processing with the TMS320C4x", TI, 1994
[7] Elektor Verlag, "Mikroprozessor-Datenbuch 1", Bibl.: 1²Bm7816

[8] B. Barrera and E.A.Lee, "Multirate Signal Processing in Comdisco's SPW", IEEE, pp. 1113..1116, 1991
[9] J.Buck, S.Ha, E.A.Lee and D.G.Messerschmitt, "Multirate Signal Processing in PTOLEMY", IEEE, pp. 1245..1248, 1991
[10] Synopsys,"COSSAP User Guide"
[11] Synopsys,"DESCARTES"
[12] Hendrik Koerner, "Codegenerierung Datenflußorientierter Programme für Multi-DSP-Architekturen", Diplomarbeit am ISS, Juni 1994
[13] Landesinitiative SOFTECH NRW, "Situation und Perspektiven des Einsatzes der parallelen Datenverarbeitung"
[14] J.A.Sharp, "Verteilte und parallele Computernetze - Architektur und Programmierung", VCH Verlag, 1989, ISBN 3-527-27814-1
[15] G.V.Wilson, „A Glossary of Parallel Computing Terminology", IEEE Parallel & Distributed Technology, 1993
[16] R.Duncan, "A Survey of Parallel Computer Architectures", IEEE: Computers, Feb.1990

[17] E.A.Lee and D.G.Messerschmitt, "Synchronous Data Flow", Proceedings of the IEEE, pp. 1235..1245, vol.75 No.9 Sep.1987
[18] E.A.Lee and D.G:Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", IEEE
[19] K.K.Parhi and D.G.Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding", IEEE
[20] E.A.Lee, "A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors", PhD thesis, UCB, 1986
[21] K.K.Parhi, "Algorithm Transformation Techniques for Concurrent Processors", Proceedings of the IEEE, pp. 1879..1895, Dec.1989
[22] E.A.Lee and S.Ha, "Scheduling Strategies for Multiprocessor Real-Time DSP", IEEE: GLOBECOM, pp. 1279..1283, Nov. 1989
[23] P.D.Hoang and J.M.Rabaey, "Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput", IEEE, June 1993
[24] P.R.Gelabert and T.P.Barnwell, "Optimal Automatic Periodic Multiprocessor Scheduler for Fully Specified Flow Graphs", IEEE, 1993
[25] Vojin Zivojnovic, "High Performance DSP Software Using Data-Flow Graph Transformations", ISS-Paper, ASILOMAR 1994
[26] Vojin Zivojnovic, Hendrik Koerner und H.Meyr, "Multiprocessor Scheduling with a-priori Node Assignment", ISS-Paper, 1994
[27] Vojin Zivojnovic, "Multiprocessor Iteration Period Bound For Multirate Data-Flow Graphs", ISS-Paper, 1994
[28] Vojin Zivojnovic, S.Ritz, H.Meyr, "Retiming of DSP Programs for Optimum Vectorization", ICASSP 1994
[29] Vojin Zivojnovic, S.Ritz, H.Meyr, "Optimizing DSP Programs Using the Multirate Retiming Transformation", EUSIPCO 1994
[30] R.Govindarajan + G.R.Gao, "A Novel Framework for Multi-Rate Scheduling in DSP Applications", ASAP'93
[31] S.H.Gerez, S.M.deGroot, O.E.Herrmann, "A Polynomial-Time Algoritm for the Computation of the Iteration-Period Bound in Recursive Data-Flow Graphs", IEEE Trans. on Circuits & Systems, Vol.39, No.1, Jan.1992
[32] D.Y.Chao, D.T.Wang, "Iteration Bound of Single-Rate Data Flow Graphs for Concurrent Processing", IEEE Sep.1993
[33] C.E.Leiserson + F.M.Rose + J.B.Saxe, "Optimizing Synchronous Circuitry by Retiming", Proceedings of Caltech Conf. on VLSI, 1983
[34] X.Hu, S.C.Bass, R.G.Harber, "Minimizing the Number of Delay Buffers in the Synchronization of Pipelined Systems", IEEE 1994
[35] H.V.Jagadish, T.Kailath, "Obtaining Schedules for Digital Systems", IEEE Trans. on Signal Processing, Vol.39, No.10, Oct.1991
[36] P.Zepter, T.Grötker, H.Meyr, "Digital Receiver Design using VHDL Generation from Data Flow Graphs", ISS Paper, 1995

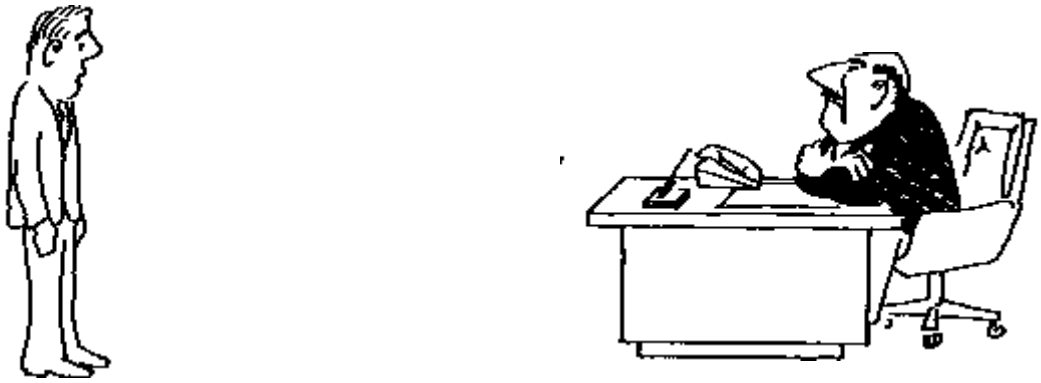
[37] T.Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, pp. 541..579, vol.77 No.4, April 1989
[38] D.T.Chao, M.Zhou, D.T.Wang, "Multiple-Weighted Marked Graphs", Proceedings of IFAC, vol.1, pp. 259..262, Sydney, 1993
[39] E.Teruel, P.Chrzastowski-Wachtel, J.M.Colom, M.Silva, "On Weighted T-Systems", Application and Theory of Petri Nets, 1992
[40] Y.E.Lien, "Termination Properties of Generalized Petri Nets", SIAM Journal of Computing, vol.5 No.2, June 1976
[41] M.Silva, J.M.Colom, "On the Computation of Structural Synchronic Invariants in P/T Nets", APN, LNCS, Vol.340, pp.386..417, 1988
[42] T. Murata, "State Equation, Controllability, and Maximal Matchings of Petri Nets", IEEE Trans. on Automat. Control, Juni 1977
[43] R.M.Karp und R.E.Miller, "Properties of a Model for Parallel Computations: Determinancy, Termination, Queuing", SIAM J., 1989
[44] T.Murata, "Relevance of Network Theory to Models of Distributed/Parallel Processing", Journal of the J.Franklin Institute, Vol.310
[45] K.H.Pascoletti, "Diophantische Systeme und Lösungsmethoden zur Bestimmung aller Invarianten in Petri-Netzen", Dissertation
[46] J.L.Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall Verlag, ISBN 0-13-661983-5, 1981
[47] J.Campos, J.M.Colom, H.Jungnitz, M.Silva, "Approximate Throughput Computation of Stochastic Marked Graphs", IEEE, July 1994
[48] E.Teruel, M.Silva, "Liveness and Home States in Equal Conflict Systems", Application & Theory of Petri-Nets, 1992, LNCS 691
[49] J.Desel, J.Esparza, "Shortest Path in Reachability Graphs", Application & Theory of Petri-Nets, 1993, LNCS 691, Springer Verlag

[50] H RRZN, "UNIX, Eine Einführung", Uni Hanover, 7.Auflage, Juli 1993
[51] H.Erlenkötter, "Programmiersprache C", rororo-Computerfachbuch, 1990
[52] RRZN, "Die Programmiersprache C++ für C-Programmierer", Uni Hannover, 5.Auflage, April 1994
[53] S.Lippmann, "C++ Primer", Addison-Wesley Publishing, 1990
[54] N. Josuttis, "Objektorientiertes Programmieren in C++", Addison-Wesley, ISBN 3-89319-637-4
[55] K.Mehlhorn und S.Näher, "LEDA - A Library of Efficient Data Types and Algorithms", Max-Planck-Institut f. Informatik, Saarbrück.
[56] S. Näher, "LEDA User Manual. Version 3.0", Max-Planck-Institut f. Informatik, Saarbrücken
[57] P. Zepter, "An Object Oriented Generic Class System for Hierarchical Data Flow Graphs", ISS Internal Report, April 1992

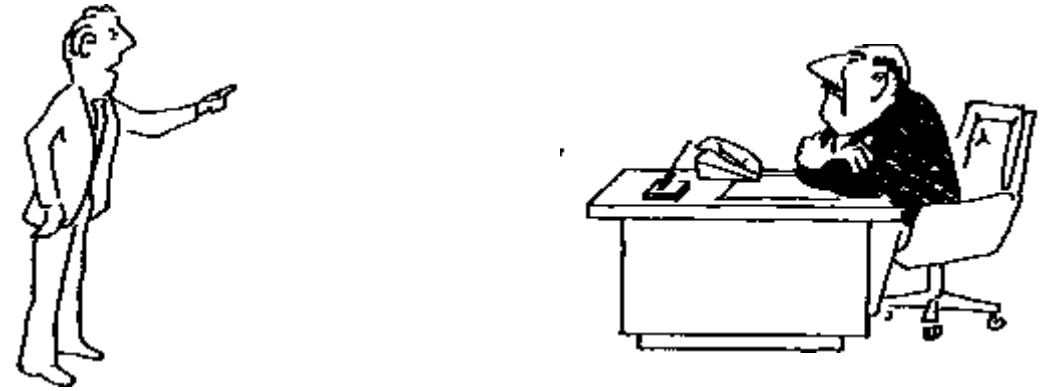
[58] R.Sedgewick, "Algorithmen", Addison-Wesley Publishing, ISBN 3-89319-402-9, 1991
[59] W.H.Press, B.P.Flannery et al., "Numerical Recipes in Pascal - The Art of Scientific Computing", Cambridge University Press, 1989

[60] I.N.Bronstein und K.A:Semendjajew, "Taschenbuch der Mathematik", Teubner Verlagsgesellschaft, 1989
[61] W.L.Winston, "Introduction to Mathematical Programming", Duxbury Press, 1995, ISBN 0-534-23046-6
[62] C.H.Papadimitriou + K.Steiglitz, "Combinatorial Optimization - Algorithms and Complexity", Prentice-Hall, 1982
[63] E.L.Lawler, "Combinatorial Optimization - Networks and Matroids", Holt,Rinehart and Winston, 1976, ISBN 0-03-084866-0
[64] D.B.Johnson, "Finding all the Elementary Circuits of a Directed Graph", SIAM J. on Computers, Vol.4, No.1, Mar.1975
[65] R.Blahut, "Theory and Practice of Error Control Codes", Addison-Wesley, ISBN 0-201-10102-5
[66] W.M.Schmidt, "Analytische Methoden für Diophantische Gleichungen", Birkhäuser Verlag, ISBN 3-7643-1661-6, Bibl.:Bb1498
[67] S.Lang, "Encyclopaedia of Mathematical Sciences, Number Theory III", Springer Verlag, ISBN 3-540-53004-5, Bibl.:Bb1690
[68] A.Schrijver, "Theory of Linear and Integer Programming", J.Wiley&Sons, ISBN 0-471-90854-1
[69] D.E.Knuth, "The Art of Computer Programming - Seminumerical Algorithms", Addison-Wesley, ISBN 0-201-03822-6
[70] M.R.Garey, D.S.Johnson, "Computers and Intractability - Theory of NP-Completeness", W.H.Freeman and Company
[71] V.Zivojnovic, R.Schoenen, H.Meyr,“Liveness and Reachability of Nonordinary Marked Graphs“, Vorschlag f. Konferenz VLSI’95

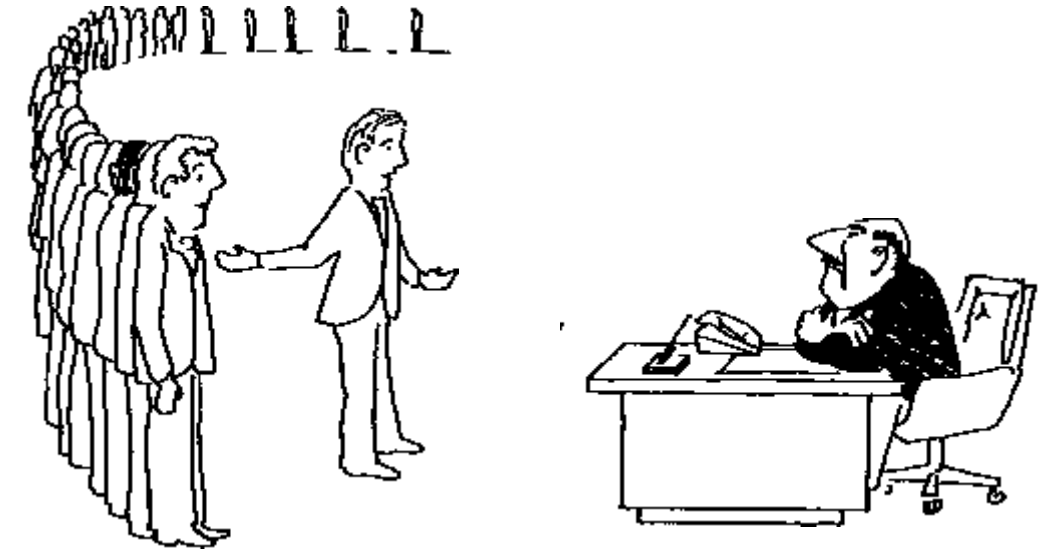
„I can’t find an efficient algorithm, I guess I’m just too dumb“



„I can’t find an efficient algorithm, because no such algorithm is possible“



„I can’t find an efficient algorithm, but neither can all these famous people“



aus: [70] (The Theory of NP-Completeness)