Zur Realisierbarkeit des PRAM Modelles

Dissertation
zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften
der Technischen Fakultät
der Universität des Saarlandes
vorgelegt
von

Jörg Keller

6600 Saarbrücken
1992

Tag des Kolloquiums:	27. Mai 1992
Dekan:	Prof. Dr. R. Maurer
Berichterstatter:	Prof. Dr. W.J. Paul Prof. Dr. J. Buchmann

Vorwort

Der Ursprung dieser Arbeit liegt in einer Aufgabe, die mir Herr Prof. Paul im Spätherbst 1989 zuteilte: der Entwurf eines Prozessors, der mehrere Instruktionsströme in einer Pipeline verarbeitet. Ihm danke ich für seine Betreuung und Unterstützung und für sein unerschütterliches Vertrauen.

Das hier vorgestellte Ergebnis sieht anders aus als damals erwartet, es verdankt seine Gestalt einer kontinuierlichen Entwicklung und Veränderung. Diese Entwicklung fand nicht im stillen Kämmerlein statt. Sie profitierte vom Aufeinanderprallen verschiedener Meinungen, vielfachem Überdenken und Überarbeiten von bereits als endgültig angesehenen Fakten und vielen Diskussionen innerhalb und außerhalb des Fachbereiches.

Ich danke Prof. Friedhelm Meyer auf der Heide, Herrn Dr. Martin Dietzfelbinger, Herrn Dr. Torben Hagerup, Herrn Dr. Helmut Seidl, Herrn Georg Sander und allen Mitarbeitern des Instituts für Rechnerarchitektur für ihr reges Interesse, ihre Kritik und ihre Anregungen.

Eine weitere Arbeit aus dem großen Gebiet der PRAM entstand im gleichen Zeitraum. Herrn Ferri Abolhassan, meinem Partner und Zimmergenossen, danke ich für eine gelungene Zusammenarbeit ohne Konkurrenzdenken.

Herrn Curd Engelmann, der im Rahmen seiner Diplomarbeit wesentliche Teile der Simulationssoftware implementierte, danke ich für seine Geduld und seine Mühe, alle Änderungen des Modells in die Simulation zu übertragen.

Frau Stefanie Urig danke ich für ihre Nachsicht gegenüber unberechenbaren Arbeitszeiten und Fällen von geistiger Versunkenheit in irgendwelchen PRAM Problemen.

Meinen Eltern danke ich für die finanzielle Unterstützung in den Jahren meiner Ausbildung, ohne die die vorliegende Arbeit nicht möglich gewesen wäre.

Sulzbach, im Februar 1992

Jörg Keller

Inhaltsverzeichnis

1 Einleitung — Vom Abacus zur PRAM **1**

2 Parallelrechner in der Theorie **5**

2.1 Das PRAM Modell 5

2.1.1 Definitionen und Notationen 5

2.1.2 Programmierung von PRAMs 7

2.1.3 Realisierungsansätze 9

2.2 Theoretische Vorarbeiten 10

2.2.1 Verteilen des globalen Speichers 10

2.2.2 Verbindungsstrukturen 13

2.2.3 Einige Netzwerke 15

2.2.4 Routingalgorithmen 16

2.3 Die Fluent Machine 17

2.3.1 Verteilung des globalen Speichers 17

2.3.2 Routing 18

2.3.3 Konkurrierende Zugriffe 19

2.3.4 Ghost Pakete 20

2.3.5 Rückweg 21

2.3.6 Aufbau 22

3 Bewertung von Rechnern **23**

3.1 Formales Modell 23
3.1.1 Hochsprache und Compiler 25
3.1.2 Hardware 25
3.2 Änderungen des Modells 32
3.3 Wichtige Hardware Bausteine 33
3.3.1 Multiplexer 33
3.3.2 Addierer 34
3.3.3 Subtrahierer 35
3.3.4 Inkrementierer 35
3.3.5 Vergleicher 35
3.3.6 Multiplizierer 36
3.3.7 Rotierer und Shifter 39
3.3.8 FIFO–Schlange 41
3.4 Bewertung der Fluent Machine 42
3.4.1 Maschinensprache und Prozessor 43
3.4.2 Netzwerk 48
3.4.3 Analyse 51

4 Veränderungen **53**

4.1 Pipelining 53
4.1.1 Virtuelle Prozessoren 55
4.1.2 Sortierfelder 57
4.1.3 Änderungen des Routingalgorithmus 59
4.2 Delayed LOAD 60
4.3 Simulationen 61
4.3.1 Einfluß von Hashing 61
4.3.2 Bestimmung der Konstanten c und z 62

4.4 Analyse der Änderungen 63
4.4.1 Änderung der Hardwarekosten 63
4.4.2 Änderung des Zeitverhaltens 64
4.4.3 Vergleich der Maschinen 65

5 Design Varianten **67**
5.1 Unterstützung konkurrierender Zugriffe 68
5.1.1 Konstruktion einer EREW PRAM 68
5.1.2 Simulation konkurrierender Zugriffe 70
5.1.3 Analyse 72
5.1.4 Einfluß auf CREW 72
5.2 Unterstützung paralleler Präfixberechnung 74
5.2.1 Definition und Eigenschaften 74
5.2.2 Hardware für Multipräfix 74
5.2.3 Hardware–MP versus Software–MP 76
5.2.4 Einschränkungen 77
5.2.5 Anwendungen von Multipräfix und Sync 78
5.3 Simulation großer Prozessorzahlen 82
5.3.1 Kontextwechsel 83
5.3.2 Unterstützende Hardware für schnellen Kontextwechsel 84
5.3.3 Unterstützende Hardware für mehrere Kontexte 85
5.3.4 Kombination beider Ansätze 85
5.3.5 Analyse 86
5.4 Bewertung am Beispiel 90
5.4.1 Berechnung von Zusammenhangskomponenten 90
5.4.2 Analyse und Messungen 91
5.4.3 Abschätzung der Anzahl von Kontextwechseln 93

6 Entwurf eines Prototyps 95

6.1 Der Prozessor 96

6.2 Sortier- und Netzknoten 98

6.3 Physikalischer Aufbau — 2 Alternativen 105

6.3.1 Größe von PCBs 107

6.3.2 On–Board und Off–Board Links 107

6.3.3 Anordnung von Boards 109

6.3.4 Eine Alternative 115

6.4 Anbindung an die Außenwelt 116

7 Zusammenfassung 117

Literaturverzeichnis 119

Anhang 125

A Code für Bitonisches Sortieren 125

B Kontrolle der Fluent Machine 127

B.1 Kontrollsignale des Prozessors 127

B.2 Kontrollsignale des Netzknotens 130

Abbildungsverzeichnis

2.1 6 Phasen Routing der Fluent Machine 19

2.2 Funktion von Ghost Paketen 21

3.1 Grobstruktur der Hardware 26

3.2 Multiplexer 33

3.3 Volladdierer, 3–2–Addierer, 4–2–Addierer 38

3.4 Rotierer und Shifter für 4–bit Zahlen 41

3.5 Aufbau einer FIFO Schlange 42

3.6 Datenpfade des Prozessors 45

3.7 Datenpfade des Netzknotens 49

4.1 Datenpfade des veränderten Prozessors 56

4.2 Datenpfade des Sortierknotens der Phase 1 58

4.3 Quotienten der Kosten, Zeit– und Gütenfunktionen 66

5.1 Bitonischer Sortier Algorithmus 71

5.2 Minimal notwendiger Anteil von konkurrierendem Zugriff 73

5.3 Minimal notwendiger Anteil von Multipräfix und Sync 78

5.4 Synchronisierung von Prozessoren 80

5.5 Datenpfade zum Wechseln von Kontexten 86

5.6 Quotienten der Gütenfunktionen von D7 und D6 in Abhängigkeit von x . . 88

5.7 Quotienten der Gütenfunktionen 89

5.8 Programm für Zusammenhangskomponenten 92

6.1 Auslastung der Busse vom und zum Prozessorchip 97

6.2 Schnitt durch Netzwerkknoten 102

6.3 Aufteilung von Netzwerkknoten für $x = 1$ 103

6.4 Anordnung von Boards nach Wise 111

6.5 Anordnung der Boards in 3 Teilen 114

6.6 Anordnung von Steckverbindern auf Boards 115

B.1 Berechnung eines Kontrollsignales 128

B.2 Instruktionsformate 128

Tabellenverzeichnis

3.1 Delay der Grundbausteine 30

3.2 Basiskosten der Grundbausteine 31

3.3 Packungsfaktoren 31

3.4 Befehlssatz 44

3.5 Aufteilung der Kosten des Prozessors 47

6.1 Benötigte Anzahlen von Netzwerkchips 101

6.2 Anzahl von Slices bei Halbierung der Links 104

6.3 Benötigte Anzahlen von Netzwerkchips bei halbierten Links 105

6.4 Größe von Subnetzwerken auf PCBs 107

6.5 Notwendige Kabel bei verschiedenen Partitionierungen 108

6.6 Notwendige Kabel in Off–Board Links 109

6.7 Knoten in Subnetzwerk j 112

6.8 Zuordnung Subnetzwerk zu Knoten y 112

Kapitel 1

Einleitung — Vom Abacus zur PRAM

Schon seit vielen Jahrhunderten versucht der Mensch, Berechnungen aller Arten zu beschleunigen. Bereits 500 Jahre vor Beginn unserer Zeitrechnung beginnt dieses Streben mit der Erfindung des Abacus und zieht sich über die Addiermaschinen des 17. Jahrhunderts bis zu den Computern heutiger Tage. Mit der Leistungssteigerung der Rechenmaschinen ging aber stets ein Anwachsen der zu verarbeitenden Datenmenge einher. Waren es zuerst nur einzelne Rechnungen zur Steuereintreibung oder zur Erstellung von Logarithmentafeln, die man zur Standortbestimmung von Schiffen benötigte, so berechnet man in unserem Jahrhundert Statistiken aus Milliarden von Daten und simuliert komplizierte physikalische Vorgänge.

Diesem Wunsch nach Leistungssteigerung versuchten Rechnerarchitekten stets Rechnung zu tragen. Ging die Entwicklung von Computern zuerst dahin, Rechner mit immer schnellerer Zentraleinheit und immer mächtigerem Befehlssatz zu bauen, so suchte man bereits in den siebziger Jahren nach anderen Wegen zur Leistungssteigerung: man begann mit der Entwicklung von Vektor– und Parallelrechnern. Erstere beschleunigen vorwiegend numerische Rechnungen durch Überlappung von komplizierten (und zeitintensiven) Operationen mit Fließkommazahlen; Beispiele sind Cray XMP [Cra84] und IBM SPARK [FOP+92]. Letztere versuchen Berechnungen verschiedenster Art durch den gleichzeitigen Einsatz mehrerer Prozessoren zu beschleunigen. Auch Kombinationen, bei denen jeder Prozessor eines Parallelrechners ein Vektorcomputer ist, sind denkbar und existieren, so zum Beispiel der SUPRENUM Rechner [Tro88].

Jeder Prozessor eines Parallelrechners ist mit einigen wenigen anderen Prozessoren fest verbunden und kann über diese Verbindungen Daten mit ihnen austauschen. Nicht jede Verbindungsstruktur ist allerdings für alle Probleme geeignet. Zum Beispiel ist das Shuffle–Exchange–Netz zum Sortieren, der Torus zum Multiplizieren von Matrizen sehr gut geeignet, jedoch nicht notwendigerweise umgekehrt. In diesem Sinne sind die existierenden Parallelrechner Spezialrechner. Die Übertragung paralleler Algorithmen zwischen Parallelrechnern

mit unterschiedlicher Verbindungsstruktur ist oft nicht möglich oder zumindest aufwendig. Teilweise ist die Verbindungsstruktur auch nicht zur Skalierung auf größere Prozessorzahlen geeignet. Beispiele hierfür sind Verbindungsstrukturen wie Busse oder vollständig verbundene Graphen. Deshalb suchte man nach einem parallelen Äquivalent zum von–Neumann–Rechner, einem „general purpose parallel computer". Erste Veröffentlichungen hierzu stammen von SCHWARTZ [Sch80] und GALIL und PAUL [GP83].

Die theoretische Informatik liefert ein solches Modell, die sogenannte PRAM (parallel random access machine). Hierbei arbeiten N Prozessoren synchron auf einem gemeinsamen globalen Speicher. Man erhält so ein sehr einfaches und komfortables Programmiermodell, das die Benutzung beliebiger Verbindungsstrukturen erlaubt. Die heute verfügbare Technologie erlaubt allerdings nicht, Speicherelemente zu bauen, die beliebig viele gleichzeitige Zugriffe in konstanter Zeit bearbeiten können. Existierende Parallelrechner mit globalem Speicher wie der ALLIANT FX/8 [PM86] haben nur wenige Prozessoren und können auch nicht weiter skaliert werden. Deshalb suchte man nach Möglichkeiten, eine PRAM auf einem Parallelrechner mit fester Verbindungsstruktur effizient zu simulieren. Hierzu gibt es mittlerweise einige Ansätze. Das Ziel dieser Arbeit ist es, diese Ansätze auf ihre Realisierbarkeit hin zu prüfen und so die „beste" PRAM Realisierung zu finden.

Um allerdings von einer „besten" Maschine reden zu können, benötigt man ein genaues und faires Vergleichsmaß. Bei der Bewertung eines Rechners sind in diesem Zusammenhang die wichtigsten Kriterien Preis und Leistung, ein einfaches Maß ist das Preis–Leistungs–Verhältnis. Schwierig ist die Festlegung der Einheiten für Preis und Leistung. Ein Preis in Dollar oder Deutscher Mark ist unangebracht, da er neben Materialkosten auch Löhne, Verwaltungs– und Entwicklungskosten berücksichtigen muß und zusätzlich noch vielen äußeren Faktoren wie Schwankungen der Umrechungskurse oder der Verfügbarkeit von Teilen abhängt. Ebenso sind für die Leistung Einheiten wie Instruktionen pro Sekunde oder Fließkommaoperationen pro Sekunde nicht geeignet. Sie hängen zu sehr vom speziellen Programm und vom Befehlssatz der Maschine ab. Außerdem sollte das gewählte Vergleichsmaß schnell „von Hand" zu berechnen sein, also ohne die geplante Maschine zu bauen oder aufwendige Simulationen durchführen zu müssen. MÜLLER und PAUL entwickelten 1990 ein solches Maß, das geeignet scheint.

Die vorliegende Arbeit ist wie folgt gegliedert. Kapitel 2 beschreibt das PRAM Modell als theoretisches paralleles Programmiermodell und gibt einen kurzen Überblick über Ansätze zu seiner Realisierung. Theoretische Vorarbeiten der Bereiche Hashing und Routing werden skizziert und verglichen. Der favorisierte Ansatz wird detailliert beschrieben. Kapitel 3 stellt das oben erwähnte formale Modell zum Bewerten und Vergleichen von Rechnern vor, beschreibt notwendige Änderungen beim Übergang zu Parallelrechnern und stellt die in dieser Arbeit verwendeten Hardware Komponenten vor. Als Beispiel wird eine Hardware für obigen Ansatz im Detail entwickelt und bewertet. In Kapitel 4 wird obiger Ansatz „ingenieursmäßig" überarbeitet, um die konstanten Faktoren in der Bewertung zu verbessern. Eine Analyse zeigt die Auswirkungen der Veränderungen und den Grad der Verbesserung. In Kapitel 5 werden verschiedene Designvarianten vorgestellt und diskutiert: konkurrierender

Zugriff, parallele Präfixberechnung und Kontextwechsel. Bei allen diesen Varianten stellt sich die Frage, ob eine Unterstützung in Hardware notwendig ist oder ob eine Simulation in Software genügt. Die Antworten auf diese Fragen geben auch Aufschluß darüber, welche dieser Eigenschaften des original Ansatzes essentiell sind und welche lediglich schmückendes Beiwerk sind. Kapitel 6 beschreibt einen möglichen Prototyp nach heutigem Stand der Technologie. Der Schwerpunkt liegt auf dem Übergang vom theoretischen Modell zur kommerziell verfügbaren und finanzierbaren Technik.

Kapitel 2

Parallelrechner in der Theorie

Mit dem Aufkommen der ersten Überlegungen zum Bau von Parallelrechnern Ende der siebziger Jahre wurde auch die theoretische Informatik auf diese Rechner aufmerksam. Man benötigte ein ideales Modell eines Parallelrechners — ähnlich der Random Access Machine im sequentiellen Fall —, das es erlaubte, die Komplexität von Algorithmen zu bestimmen, ohne von einer speziellen existierenden Maschine direkt beeinflußt zu werden. Dieses Modell muß deshalb eine sehr einfache Verbindungsstruktur aufweisen. FORTUNE und WYLLIE definierten in [FW78] ein solches Modell und nannten es *parallel random access machine* oder abgekürzt PRAM. Auch andere Forscher definierten ähnliche Modelle, zum Beispiel LEV, PIPPENGER und VALIANT in [LPV81], in [FW78] wurde das Modell jedoch meines Wissens erstmalig definiert und zur Komplexitätsbestimmung benutzt.

2.1 Das PRAM Modell

2.1.1 Definitionen und Notationen

Definition 2.1 *Eine* Parallel Random Access Machine (PRAM) *besteht aus einer Menge von Prozessoren* $P_0, \ldots, P_{N-1}$*, und einem globalen Speicher der Größe* M*. Ein Prozessor ähnelt weitestgehend einer random access machine wie sie in [AHU74] definiert wird. Er führt in einem Schritt entweder eine interne Operation aus, liest ein Wort aus dem globalen Speicher oder schreibt ein Wort in den globalen Speicher. Ein Schritt hat feste konstante Länge.*

In einigen Arbeiten, z.B. bei KARP und RAMACHANDRAN [KR90], wird zusätzlich jedem Prozessor ein lokaler Speicher zugeordnet, auf den nur er allein zugreifen kann. Der Zugriff auf lokalen Speicher gehört zu den internen Operationen.

Greifen im gleichen Schritt mehrere Prozessoren auf eine Zelle des globalen Speichers zu, so kommt es zu Konflikten. Zur Regelung dieser Konflikte müssen Konventionen getroffen wer-

den. Verschiedene Konventionen definieren verschiedene PRAM Modelle. Eine gemeinsame Konvention aller Modelle ist, daß in einem Schritt auf eine Zelle nur lesend oder nur schreibend zugegriffen wird. Einige Arbeiten umgehen dieses Problem, indem sie jeden Schritt als zwei Halbschritte auffassen, wobei im ersten Halbschritt alle Leseoperationen und im zweiten Halbschritt alle Schreiboperationen abgewickelt werden. Die wichtigsten Modelle sollen hier kurz vorgestellt werden. Die Nomenklatur der Modelle wurde aus [MV84] und [DR86] übernommen.

- Im *exclusive read exclusive write (EREW)* Modell darf eine Zelle in einem Schritt von höchstens einem Prozessor gelesen oder geschrieben werden. Dieses Modell wurde in [LPV81] eingeführt.
- Im *concurrent read exclusive write (CREW)* Modell darf eine Zelle in einem Schritt von mehreren Prozessoren gelesen oder von einem geschrieben werden. Dies ist das ursprüngliche Modell aus [FW78].
- Im *concurrent read concurrent write (CRCW)* Modell darf eine Zelle in einem Schritt von mehreren Prozessoren gelesen oder von mehreren geschrieben werden.
- Im *concurrent read owner write (CROW)* Modell darf eine Zelle in einem Schritt von mehreren Prozessoren gelesen oder von einem bestimmten Prozessor, ihrem „owner“ geschrieben werden. Dieses Modell wurde in [DR86] eingeführt.

Weitere Kombinationen von Lese– und Schreibrechten sind denkbar. Einige werden verwandt, das Modell der ERCW (exclusive read concurrent write) PRAM hingegen wurde meines Wissens bisher nirgends benutzt.

In den Modellen, die concurrent write benutzen, muß zusätzlich festgelegt werden, welchen Wert eine Zelle erhält, die in einem Schritt von mehreren Prozessoren geschrieben wird. Drei Hauptkonventionen werden verwandt.

- Im *arbitrary* Modell gewinnt einer der schreibenden Prozessoren, aber es kann nicht vorhergesagt werden, welcher.
- Im *common* Modell müssen alle Prozessoren, die in einem Schritt eine Zelle schreiben, den gleichen Wert schreiben.
- Im *priority* Modell gibt es eine injektive Abbildung von Prozessornummern in die natürlichen Zahlen. Schreiben mehrere Prozessoren im gleichen Schritt die gleiche Zelle, so übernimmt die Zelle den Wert des Prozessors, der bezüglich obiger Funktion den größten Wert hat.

Die Mächtigkeit der obigen Modelle war und ist Gegenstand von komplexitätstheoretischen Untersuchungen. Als Maß der Komplexität eines Problems der Größe n nimmt man die

Anzahl der benötigten Prozessoren $p(n)$ und die benötigte Zeit $t(n)$. Man bezeichnet parallele Algorithmen dann als *optimal*, wenn sie $p(n) \cdot t(n) = O(T_s(n))$ erfüllen, wobei $T_s(n)$ die asymptotische Laufzeit des schnellsten bekannten sequentiellen Algorithmus für das gleiche Problem mit Größe n ist.

2.1.2 Programmierung von PRAMs

Zum Beschreiben von Algorithmen für PRAMs benutzt man eine sequentielle von–Neumann Programmiersprache wie PASCAL. Die Benutzung von Parallelität erfolgt durch die zusätzliche Anweisung

for $x \in X$ **pardo** $A(x)$ **od**

X ist eine Menge und die Abarbeitung der Anweisung beinhaltet die Schritte

1. Ordne jedem x aus X einen Prozessor zu
2. Führe auf allen diesen Prozessoren gleichzeitig die Anweisungsfolge $A(x)$ aus.

Die Anweisungen werden synchron ausgeführt. Das parallele Konstrukt ist also erst beendet, wenn alle Prozessoren $A(x)$ abgearbeitet haben. Auf eine genaue Beschreibung der Semantik soll hier verzichtet werden. Zwei Beispiele dienen zur Verdeutlichung. Eine Programmiersprache für PRAMs namens FORK wurde 1990 an der Universität des Saarlandes entwickelt [HSS91] und wird gerade für eine abstrakte parallele Maschine, die weitestgehend einem Simulator des in Kapitel 6 beschriebenen Prototyps entspricht, implementiert.

Beispiel 1: Sei $a[1..n]$ ein Feld mit n verschiedenen Zahlen. Das folgende Programm berechnet das Maximum und legt es in der Variable *erg* ab. Das Feld b aus n Zahlen dient zur Aufnahme von Zwischenergebnissen. Es gilt $t(n) = O(1)$ und $p(n) = n^2$. Der Algorithmus stammt aus [GR88].

(1) **for** $i \in \{1..n\}$ **pardo** $b[i] := 0$ **od**;
(2) **for** $(i_1, i_2) \in \{1..n\} \times \{1..n\}$ **pardo**
(3)   **if** $a[i_1] < a[i_2]$ **then** $b[i_1] := 1$ **fi**;
(4) **od**;
(5) **for** $i \in \{1..n\}$ **pardo**
(6)   **if** $b[i] = 0$ **then** $erg := a[i]$ **fi**;
(7) **od**.

Beispiel 2 (Parallel Prefix): Sei $a[1..n]$ ein Feld mit n Zahlen. Das folgende Programm berechnet alle Präfixe der Form $s[i] = \sum_{j=1}^{i} a[j], j = 1, \ldots, n$. Es gilt $t(n) = O(\log n)$ und $p(n) = O(n)$. Der Algorithmus wurde in [LF80] erstmalig vorgestellt. Er gilt

nicht nur für die Addition, sondern für alle assoziativen Operationen. Die Felder $b[1\ldots 2n-1]$ und $c[1\ldots 2n-1]$ realisieren Beschriftungen vollständiger binärer Bäume, wobei Element 1 die Wurzel darstellt und ein Element i die Söhne $2i$ und $2i+1$ hat.

(1) **for** $x \in \{1..n\}$ **pardo** $b[n-1+x] := a[x]$ **od**;
(2) **for** $i := \log n - 1$ **downto** 0 **do**
(3) **for** $x \in \{2^i..2^{i+1}-1\}$ **pardo** $b[x] := b[2x] + b[2x+1]$ **od**;
(4) **od**;
(5) $c[1] := 0$;
(6) **for** $i := 1$ **to** $\log n - 1$ **do**
(7) **for** $x \in \{2^i..2^{i+1}-1\}$ **pardo**
(8) **if even**(x) **then** $c[x] := c[\frac{x}{2}]$
(9) **else** $c[x] := c[\lfloor\frac{x}{2}\rfloor] + b[x-1]$
(10) **fi**
(11) **od**
(12) **od**;
(13) **for** $x \in \{1..n\}$ **pardo** $s[x] := c[n-1+x] + a[x]$ **od**;

Beispiel 1 ist nur auf einer CRCW PRAM auszuführen, da in Zeile 3 jeweils n Prozessoren ein Feldelement aus a gleichzeitig lesen und eventuell mehrere Prozessoren ein Element aus b gleichzeitig schreiben. Man kann jedes der Modelle common, arbitrary oder priority write wählen, da gleichzeitig schreibende Prozessoren den gleichen Wert schreiben. Laufzeit und Prozessorzahl sind direkt einzusehen. Die Korrektheit zeigt man wie folgt. Am Ende der Ausführung von Zeile 3 gibt es noch genau ein Element in Feld b mit Wert 0. Gäbe es zwei solcher Elemente mit Indices i_1 und i_2, so hätte Prozessor (i_1, i_2) in Zeile 3 eines der beiden Feldelemente mit 1 besetzt. Gäbe es kein solches Element mehr, so könnte man einen Zyklus von Bedingungen $a[i_j] > a[i_{j+1}]$ konstruieren. Beides führt zum Widerspruch. Das eindeutige Element $a[i]$ mit $b[i] = 0$ nach Ausführung von Zeile 3 ist das Maximum, da es nach Zeile 3 größer als alle anderen sein muß.

Beispiel 2 kann auf einer EREW PRAM ausgeführt werden. Auch hier können Laufzeit und Prozessorzahl direkt abgelesen werden. Die Korrektheit kann aus den folgenden Behauptungen hergeleitet werden:

1. Die Beschriftungen b der Blätter des Baumes enthalten nach Ausführung von Zeile 1 die n Zahlen.

2. Nach Ausführung der Zeile 4 enthalten die Beschriftungen $b[i]$ der Knoten i des Baumes die Summe der Beschriftungen b an den Blättern des Unterbaumes, von dem i die Wurzel ist.

3. Nach Ausführung der Zeile 12 enthalten die Beschriftungen $c[i]$ der Knoten i des Baumes die Summe der Beschriftungen b der Blätter, die im Baum links von i liegen.

Der Algorithmus kann so weit verbessert werden, daß nur noch $p(n) = n/\log n$ Prozessoren benötigt werden. Jeder Prozessor $i \in \{1 \ldots n/\log n\}$ berechnet zuerst sequentiell die Summe $a'[i]$ der Zahlen $a[i(\log n - 1) + 1]$ bis $a[i \log n]$. Dazu wird Zeit $O(\log n)$ benötigt. Danach führen die Prozessoren obigen Algorithmus für die $n' = n/\log n$ Zahlen im Feld $a'[1 \ldots n/\log n]$ aus. Lediglich die letzte Zeile wird zu $s'[x] := c[n-1+x]$ verändert. Dazu benötigen sie Zeit $t(n') = O(\log n') = O(\log n)$. Man erhält ein Feld $s'[1 \ldots n/\log n]$ der Präfixe der Zahlen aus a'. Jeder Prozessor i berechnet jetzt sequentiell die Präfixe der Zahlen $a[i(\log n - 1) + 1]$ bis $a[i \log n]$ und addiert zu jedem Präfix $s'[i]$. Auch hierzu wird Zeit $O(\log n)$ benötigt. Die Gesamtzeit ist $t(n) = O(\log n)$, die Prozessorzahl $p(n) = O(n/\log n)$. Der Algorithmus ist optimal. Diese Verbesserung mag den Anschein von „Erbsenzählerei“ haben, da sich die Laufzeit $t(n)$ größenordnungsmäßig nicht ändert und sich sogar ungefähr um den Faktor 3 erhöht. Sie erlangt ihre Bedeutung erst im Zusammenhang mit der unten angesprochenen Tatsache, daß die Prozessorzahl P realer Rechner sehr viel kleiner ist als n.

Die beiden Beispiele demonstrieren, wie einfach sich parallele Algorithmen für eine PRAM formulieren und analysieren lassen. Die PRAM bietet also ein sehr komfortables Programmiermodell. Eine Vielzahl von PRAM Algorithmen findet man in [GR88, KR90]. Aus diesem Grunde gab und gibt es viele Versuche, eine PRAM zu realisieren.

2.1.3 Realisierungsansätze

Bei einer Realisierung von PRAMs treten zwei Hauptschwierigkeiten auf:

1. Eine reale PRAM hat nur konstant viele Prozessoren, die ideale hat Prozessorzahlen in Abhängigkeit der Eingabegröße.
2. Die heutige Technologie kennt keine Speicherbausteine, die mehr als 2 oder 3 Ports in konstanter Zeit bedienen. Man kann den globalen Speicher daher nicht direkt realisieren.

Das erste Problem tritt nicht nur bei der PRAM, sondern bei allen Parallelrechnern auf. Man löst es, indem man auf jedem der N vorhandenen Prozessoren $p(n)/N$ logische Prozessoren simuliert. Jeder der logischen Prozessoren wird durch den Zustand aller von ihm benutzten Register des realen Prozessors einschließlich Stackzeiger, Programmzähler und Statusregister repräsentiert. Man nennt dies den *Kontext* [PS85]. Zusätzlicher Aufwand entsteht nur, wenn die Simulation eines logischen Prozessors angehalten und die eines anderen fortgesetzt wird. Man spricht hierbei von *Kontextwechsel* [PS85]. Asymptotisch hat der Kontextwechsel keinen Einfluß auf die Laufzeit eines Programms, da der Kontext jedes logischen Prozessors nur Größe $O(1)$ hat und deshalb auch nur Zeit $O(1)$ zum Laden oder Speichern eines Kontextes benötigt wird. Der tatsächliche Einfluß und eventuelle Unterstützung durch Hardware wird in Kapitel 5.3 diskutiert.

Das zweite Problem löst man, indem man den globalen Speicher in mehrere Module, z.B. N Stück, zerteilt und zwischen Prozessoren und Speichermodule ein Netzwerk schaltet. Da die Knoten bekannter Netzwerke konstanten Grad haben (im Fall des $\log N$–dimensionalen Würfels auch $\log N$), hat ein Netzwerk mit N Eingängen und Ausgängen mindestens Tiefe $\log N$. Der Zugriff auf den globalen Speicher kann also nicht mehr in konstanter Zeit erfolgen, wie es die Definition der PRAM fordert.

Alle mir bekannten Ansätze zur Realisierung von PRAMs (also von Simulationen auf realistischen Maschinen) folgen diesem Weg und beschreiben, wie der globale Speicher auf die verschiedenen Module verteilt wird, welches Netzwerk und welcher Routing Algorithmus gewählt wird. Danach folgt der Beweis der Korrektheit und der Effizienz der Simulation. Im folgenden Abschnitt werden die bekanntesten Ansätze kurz geschildert und ihre Vor– und Nachteile vorgestellt. VALIANT gibt in [Val90b] eine Übersicht der bekannten theoretischen Ergebnisse.

Es soll hier bereits angemerkt werden, daß die Hardware für eine PRAM Realisierung nach dem oben gesagten nicht sehr verschieden von der Hardware für derzeit kommerziell erhältliche Parallelrechner ist. Die Bedeutung dieser Beobachtung ist Gegenstand der Dissertation von ABOLHASSAN [Abo92]. Sie soll hier nicht weiter verfolgt werden.

2.2 Theoretische Vorarbeiten

Bevor nachfolgend einzelne Ansätze kurz geschildert werden, sollen einige grundlegende Entscheidungen begründet werden, die den meisten Ansätzen zugrunde liegen. Die erste Beobachtung betrifft die Art der Verteilung des globalen Speichers auf Module. Der zweite wichtige Punkt ist die Auswahl der Verbindungsstruktur.

2.2.1 Verteilen des globalen Speichers

Die Zeit zur Simulation eines Schrittes einer PRAM ist nach unten beschränkt durch die Tiefe des verwendeten Netzwerkes und durch die maximale Anzahl von Speicherzugriffen, die ein Speichermodul zu bearbeiten hat (Modulbelastung). Ist die Verteilung des globalen Speichers fest und eindeutig in dem Sinne, daß von keiner Zelle des globalen Speichers mehrere Kopien auf verschiedenen Modulen angelegt werden, so kann ein Zugriffsmuster der Prozessoren konstruiert werden, daß die Zugriffe aller N Prozessoren in einem Modul stattfinden. Die Zeit zur Simulation wäre damit linear. Eine lineare Simulationszeit kann aber auch sequentielle Simulation aller Prozessoren auf einem sequentiellen Rechner erreicht werden. Deshalb muß dieser Fall möglichst vermieden werden.

Deterministische Verfahren

Deterministische Strategien zur Verteilung des globalen Speichers arbeiten mit mehreren Kopien jeder Speicherzelle. Die Kopien befinden sich auf verschiedenen Modulen. Formal ist die Verteilung durch einen bipartiten Graphen gegeben. Die Teilmengen bestehen aus den zu verteilenden Zellen des globalen Speichers und den Modulen. Eine Kante zwischen einer Zelle und einem Modul bedeutet, daß eine Kopie dieser Zelle auf diesem Modul gehalten wird. Bei einem Zugriff wird stets nur ein Teil (aber mehr als die Hälfte) der Kopien aktualisiert bzw. gelesen. Dies garantiert, daß bei einem Lesezugriff stets mindestens eine aktuelle Kopie gelesen wird. Die Auswahl der Kopien wird abhängig vom Zugriffsmuster so gewählt, daß die Modulbelastung möglichst gering ist.

Unabhängig voneinander zeigten ALT, HAGERUP, MEHLHORN und PREPARATA [AHMP87] und KARLIN und UPFAL [KU88], daß bei dieser Methode die Simulationszeit eines Schritts mindestens $\Omega(\log^2 N/\log\log N)$ beträgt. Die beste bekannte Strategie ist die von BILARDI und HERLEY [BH88], sie benötigt Zeit $O(\log N \log M/\log\log N)$. Die Konstante in der asymptotischen Notation ist sehr groß, da eine Sortierung nach AJTAI, KOMLÓS und SZEMERÉDI [AKS83] verwendet wird. Eine verbesserte Version dieses Verfahrens durch UPFAL [Upf89] hat nach [Mas90] eine Konstante von 25000. Die deterministischen Verfahren sind deshalb bisher noch nicht von praktischem Nutzen und werden nicht weiter betrachtet.

Anmerkung: Das Verfahren von BILARDI und HERLEY benötigt einen Verteilungsgraphen mit Expandereigenschaft. Die einzige derzeit bekannte Möglichkeit zur effizienten Erzeugung großer Expandergraphen ist das Würfeln. In diesem Sinne ist somit auch dieses Verfahren randomisiert.

Randomisierte Verfahren — Hashing

Bei randomisierten Verfahren versucht man durch Verwendung von Zufallszahlen in der Funktion, die den Speicher auf die Module verteilt, die Wahrscheinlichkeit zu begrenzen, daß ein Simulationsschritt lineare Zeit erfordert. Man benutzt die Strategie des *universellen Hashings*. Universelle Hash–Funktionen wurden von CARTER und WEGMAN eingeführt [CW79]. Bei universellem Hashing benutzt man eine Klasse von Hash–Funktionen, die die Eigenschaft besitzt, daß es für jedes Muster aus disjunkten Adressen und jedes Muster aus Speichermodulen nur sehr wenige Funktionen dieser Klasse gibt, die diese Adressen auf diese Module abbilden. Wählt man vor dem Beginn der Simulation aus einer solchen Klasse eine Hash–Funktion zufällig aus, so ist die Wahrscheinlichkeit sehr klein, daß die Funktion den Speicher so auf die Speichermodule aufteilt, daß die Modulbelastung hoch wird. MEHLHORN und VISHKIN benutzen in [MV84] die einfach zu konstruierende Klasse $\mathcal{H}_s$:

$$\mathcal{H}_s = \{pol(x) \bmod P \bmod N : pol(x) \text{ ist Polynom vom Grad } s\}\,.$$

Hierbei ist N die Anzahl der Speichermodule, auf die verteilt werden soll. P ist eine

Primzahl mit $P \geq N$. Auch KARLIN und UPFAL führen diese Klasse ein [KU88]. Bei vorgegebenem Muster und zufälliger Auswahl eines Polynoms dieser Klasse (durch zufälliges Auswählen der Koeffizienten) ist die Wahrscheinlichkeit, daß mehr als s Werte auf ein bestimmtes Modul abgebildet werden, durch N^{1-s} beschränkt. Man erreicht also mit sehr hoher Wahrscheinlichkeit eine Modulbelastung von höchstens s. Da wegen der Netzwerktiefe von mindestens $\log N$ die Simulation eines Schrittes mindestens Zeit $O(\log N)$ benötigt, ist eine Modulbelastung von $s = \log N$ ausreichend. Man benötigt also Polynome des Grades $\log N$. Neuere Forschungen von DIETZFELBINGER [Die91b] zeigen, daß sogar Polynome konstanten Grades größer 1 ausreichen, daß jedoch lineare Funktionen den Anforderungen im allgemeinen nicht genügen [Die91a].

Lokales Verteilen

Bisher wurde nur die Verteilung des globalen Speichers auf die Module betrachtet. Die Zellen des globalen Speichers jedoch, die auf ein Modul abgebildet werden, müssen noch lokalen Zellen dieses Moduls zugeordnet werden. Hierzu berechnet man erst $g(x) = pol(x) \bmod P \bmod M$ und bildet dann die Modulnummer $h(x) = g(x) \bmod N$ und die lokale Adresse $l(x) = g(x) \operatorname{div} N$. Allerdings ist für Polynome eines Grades größer 1 die Funktion $g(x)$ nicht bijektiv, d.h. daß mehrere Zellen des globalen Speichers auf die gleiche Zelle eines Moduls abgebildet werden können. Hier wäre die Benutzung von linearen Funktionen hilfreich. Zusätzlich könnte die Berechnung von $g(x)$ vereinfacht werden, da bei linearen Hash–Funktionen statt einer Primzahl auch eine Zweierpotenz eingesetzt werden kann, bei Polynomen höheren Grades jedoch nicht [Die90]. Die Berechnung „modulo Zweierpotenz“ ist im Binärsystem, das von Computern im allgemeinen verwendet wird, in einem Schritt möglich, die Berechnung „modulo Primzahl“ jedoch nicht.

Eine einfache Methode zur Bewältigung des Problems wäre eine Vergrößerung der Module um den Faktor Grad des Polynoms, da maximal soviele Zellen des globalen Speichers in eine Zelle eines Moduls abgebildet werden können. Abgesehen davon, daß dies teuer wird, wird auch der Zugriff schwieriger, da bei jeder Zelle eines Moduls gespeichert werden muß, welche Zelle des globalen Speichers auf sie abgebildet wurde.

RANADE gibt ein Verfahren an [Ran91], das lokales Adressieren auch bei Hash–Funktionen realisiert, die nicht bijektiv sind, ohne die Module asymptotisch zu vergrößern, wobei sich auch die Zugriffszeit größenordnungsmäßig nicht ändert. Die Konstanten in zusätzlich benötigtem Speicher und zusätzlicher Zugriffszeit können jedoch nicht vernachlässigt werden.

Da Simulationen [Eng92, RBJ88] darauf hindeuten, daß in der Praxis der Grad der Hash–Funktion kaum Einfluß auf die Simulationszeit hat, werden später lineare Hash–Funktionen verwendet.

2.2.2 Verbindungsstrukturen

Als Verbindungsstrukturen benutzt man *Netzwerke*. Diese kann man als Graphen $G = (V, E)$ betrachten. Einige der Knoten des Graphen, die sogenannten Sender, können Nachrichten ins Netzwerk einspeisen, einige Knoten, die sogenannten Empfänger, nehmen für sie bestimmte Nachrichten aus dem Netzwerk. Alle Knoten des Netzwerkes, auch die, die weder Sender noch Empfänger sind, übernehmen die Weiterleitung von Nachrichten. Nachrichten werden über die Kanten des Graphen von einem Knoten zu einem anderen weitergeleitet. Sind die Kanten ungerichtet, können sie Nachrichten in beide Richtungen transportieren, sind sie gerichtet, nur in eine Richtung. Jede Kante kann in einer Zeiteinheit nur eine bestimmte Menge an Information transportieren. Der *Durchmesser* eines Netzwerkes ist der Durchmesser des zugrunde liegenden Graphen. Entscheidende Kriterien eines Verbindungsnetzwerkes sind nach [HB84] die Art der Vermittlung von Nachrichten, die Kontrollstrategie, die Betriebsart und die Topologie des zugrunde liegenden Graphen.

Die beiden wichtigsten Vermittlungsarten von Nachrichten sind Leitungsvermittlung *(circuit switching)* und Paketvermittlung *(packet switching)*. Bei leitungsvermittelnden Netzwerken wird zwischen den Kommunikationspartnern ein physikalischer Pfad geschaltet, über den die Nachrichten verschickt werden. Die Zeit zum Erstellen des Pfades ist im Allgemeinen größer als die Zeit zum Transport einer Nachrichteneinheit (nach [Mül89] etwa Faktor 100). Bei paketvermittelnden Netzwerken werden die zu versendenden Nachrichten in sogenannte Pakete verpackt, die außer der Nachricht selbst noch Informationen über Sender und Empfänger der Nachricht enthalten. Diese Pakete werden durch das Netzwerk geschickt, ohne daß dazu ein physikalischer Pfad zwischen Sender und Empfänger als ganzes geschaltet werden müßte. Im einfachsten Fall wird das Paket als Ganzes von einem Knoten zum nächsten transportiert (Store–and–Forward Prinzip, [Tan89]). Innerhalb der Knoten werden die Pakete in Queues zwischengespeichert. Es gibt auch Netzwerke, in denen das Paket in mehrere Teile, die sogenannten „Flits“, aufgeteilt werden kann (Cut–Through Prinzip, [KK79]). Dort wird zumindest teilweise ein physikalischer Pfad etabliert, dessen Länge mit der Anzahl der Flits variiert. Bei paketvermittelnden Netzwerken wird keine Zeit zum Aufbau eines Pfades benötigt. Die Zeit zum Transport eines Paketes von einem Knoten des Netzwerkes zu einem anderen ist größer als bei einer Leitungsvermittlung, da die Kante, über die das Paket weitertransportiert werden soll, erst aus der Nummer des Empfängers berechnet werden muß und da eventuell mehrere Pakete die gleiche Kante benutzen wollen und nacheinander geschickt werden müssen. Letzeren Fall nennt man einen Konflikt.

Man kann aus oben gesagtem ersehen, daß sich leitungsvermittelnde Netzwerke eher für Anwendungen eignen, bei denen vorwiegend Nachrichten großen Umfangs verschickt werden, paketvermittelnde Netzwerke hingegen eher für Anwendungen, bei denen sehr viele kurze Nachrichten verschickt werden. Eine Nachricht soll hierbei kurz sein, wenn die Größe der Nachricht selbst ungefähr der Größe der Information über Sender und Empfänger entspricht. Bei einer PRAM–Simulation bestehen die Nachrichten aus Anforderungen für Zugriffe auf den globalen Speicher. Sie enthalten nur Informationen über die Art des Zugriffs, die Adresse der betroffenen Speicherzelle und eventuell das zu schreibende Datum, sind also

kurz. Außerdem sind sie häufig. Im weiteren Verlauf der Arbeit werden deswegen nur paketvermittelnde Netze nach dem Store–and–Forward Prinzip betrachtet.

Bekannte Kontrollstrategien sind zentrale und verteilte Kontrolle. Bei einer zentralen Kontrolle berechnet eine Kontrolleinheit die Einstellung aller Knoten abhängig von den Sender/Empfänger–Kombinationen aller auftretenden Nachrichten. Bei einer verteilten Kontrolle berechnet jeder Knoten selbst seine Einstellung abhängig von den Sender/Empfänger–Kombinationen der gerade bei ihm anliegenden Nachrichten, unabhängig von allen anderen Nachrichten. Zentrale Kontrolle findet man in der Regel bei leitungsvermittelnden Netzwerken, oft findet sie off–line statt, d.h. die Berechnung findet nicht zur Laufzeit, sondern schon vorher, zum Beispiel zur Compilezeit, statt. Verteilte Kontrolle findet man bei allen mir bekannten paketvermittelnden Netzwerken. Der Grund hierfür liegt darin, daß die Kante, über die ein Paket weitergeschickt werden muß, sich in der Regel sehr einfach aus der Nummer des Empfängers berechnen läßt. Die Entscheidung, ob dieses Paket weitertransportiert werden kann, hängt nur davon ab, ob ein Konflikt auftritt. Hierzu genügt die Kenntnis der anderen Pakete, die gerade in diesem Knoten vorliegen. Eine verteilte Kontrolle ist also möglich und wegen ihrer Einfachheit auch vorzuziehen. Es ist auch ein leitungsvermittelndes Netzwerk mit verteilter Kontrolle bekannt. Innerhalb des ALEWIFE Projektes am MIT wird in [Kni89] ein solches Netzwerk beschrieben, das nach dem Prinzip des Wormhole Routing [DS87] funktioniert. Da in dieser Arbeit aber keine leitungsvermittelnden Netze betrachtet werden, wird nicht weiter darauf eingegangen. Im Weiteren werden alle betrachteten Netzwerke mit einer verteilten Kontrolle versehen sein.

Als Betriebsarten unterscheidet man synchronen und asynchronen Betrieb. Bei synchronem Betrieb dürfen nur in bestimmten Zeitabständen neue Nachrichten ins Netzwerk eingespeist werden, bei asynchronem Betrieb darf dies zu beliebigen Zeitpunkten geschehen. Synchroner Betrieb kann als Spezialfall des asynchronen Betriebs betrachtet werden, bei dem die Möglichkeit des Einspeisens zu beliebigem Zeitpunkt nicht ausgenutzt wird.

Bei Topologien unterscheidet man regelmäßige und unregelmäßige Netzwerke, innerhalb der regelmäßigen *statische* und *dynamische* Netzwerke. Bei statischen Netzwerken ist jeder Knoten ein Sender und Empfänger, bei dynamischen unterscheidet man zwischen Sendern/Empfängern, die in der Regel Prozessoren darstellen, und anderen Knoten, die nur als „Schalter“ zur Weiterleitung von Nachrichten (routing switch) dienen. Zu den statischen Netzwerken zählen das eindimensionale Feld *(array)*, der Ring, der Stern *(star)*, der Baum *(tree)*, das k–dimensionale Gitter *(k–dimensional mesh)*, der k–dimensionale Würfel *(k–dimensional cube, Hypercube)* und der *Cube–Connected–Cycle*. Zu den dynamischen Netzwerken zählen das *Shuffle–Exchange Netzwerk* und der *Crossbar* als *Single–stage Netzwerke* und das *Butterfly Netzwerk*, das *Baseline Netzwerk* und die *Clos–Netzwerke* [Clo53] als *Multistage Netzwerke*. Die Netzwerke, die im Verlauf der Arbeit noch benutzt werden, werden im folgenden definiert. Für alle anderen sei auf das bereits erwähnte Buch von Hwang und Briggs [HB84] verwiesen.

Die beiden letzten Kriterien wurden der Vollständigkeit halber aufgenommen. Da im weiteren Verlauf der Arbeit die Unterscheidung zwischen statischen und dynamischen bzw.

synchronen und asynchronen Netzwerken nicht von Bedeutung ist, werden wir die Begriffe nicht weiter verwenden.

2.2.3 Einige Netzwerke

In den folgenden beiden Definitionen werden die den benutzten statischen und dynamischen Netzwerken zugrunde liegenden Graphen definiert.

Definition 2.2 *Das* k–dimensionale Gitter *mit Seitenlänge* s *ist ein ungerichteter Graph* $\mathcal{G}_{s,k} = (V_G, E_G)$ *mit* $V_G = \{0, \ldots, s-1\}^k$ *und* $\{(a_0, \ldots, a_{k-1}), (b_0, \ldots, b_{k-1})\} \in E_G$ *dann und nur dann wenn es genau ein* $i \in \{0, \ldots, k-1\}$ *gibt mit* $|a_i - b_i| = 1$ *und* $a_j = b_j$ *für* $j \neq i$. *Man nennt dies die* i–te Kante *von* $(a_0, \ldots, a_{k-1})$ *beziehungsweise* $(b_0, \ldots, b_{k-1})$. *Der* k–dimensionale Würfel (Hypercube) $\mathcal{H}_k$ *ist ein* k*–dimensionales Gitter der Seitenlänge* $s = 2$.

Definition 2.3 *Das* s–Butterfly *mit Grad 2 ist ein gerichteter Graph* $\mathcal{B}_s = (V_B, E_B)$ *mit* $V_B = \{0, \ldots, s\} \times \{0, \ldots, 2^s - 1\}$ *und für* $a_0 \in \{0, \ldots, s-1\}$ *ist* $((a_0, a_1), (a_0 + 1, b_1)) \in E_B$ *genau dann wenn* $b_1 = a_1$ *oder* $b_1 = a_1 \oplus 2^{a_0}$.

Die Schreibweise $x \oplus y, x, y \in \{0, \ldots, 2^s - 1\}$ ist eine Abkürzung für die Zahl, deren s–stellige Binärdarstellung das bitweise exklusive Oder der s–stelligen Binärdarstellungen der Zahlen x und y ist.

Ich werde die folgenden Sprechweisen benutzen. Bei einem Knoten $(a_0, a_1) \in V_B$ nennt man a_0 die *Spaltennummer* und a_1 die *Zeilennummer*. Alle Knoten mit gleicher Spaltennummer bilden eine *Spalte*, alle Knoten mit gleicher Zeilennummer bilden eine *Zeile*. Der *Row Leader* einer Zeile a_1 ist $(0, a_1)$. Kanten, die der ersten Bedingung der Definition genügen, heißen *gerade Kanten*, Kanten, die der zweiten Bedingung der Definition genügen, heißen *kreuzende Kanten*.

In einigen Artikeln werden die Spalten 0 und s identifiziert, d.h. $V_B = \{0, \ldots, s-1\} \times \{0, \ldots, 2^s - 1\}$ und bei der Definition der Kanten gilt $((a_0, a_1), ((a_0 + 1) \bmod s, b_1)) \in E_B$ falls eine der beiden Bedingungen erfüllt ist. Anschaulich entspricht diese Methode einem „Rundkleben“ des Graphen. Es werden auch „umgekehrte Butterfly Netzwerke“ betrachtet. Bei diesen ist die zweite Bedingung für eine Kante geändert zu $b_1 = a_1 \oplus 2^{s-a_0}$. Anschaulich entspricht diese Methode dem Umdrehen der Kanten und einer umgekehrten Numerierung der Spalten.

Netzwerke wie Baseline, Delta und einige mehr werden nicht definiert. Wu und Feng zeigen in [WF80], daß diese Netzwerke alle äquivalent zum Butterfly Netzwerk sind. Dies bedeutet, daß man das eine Netzwerk durch Permutieren der Ein– und Ausgänge aus dem anderen erhält.

2.2.4 Routingalgorithmen

Dieser Abschnitt soll die Entwicklung von Algorithmen zum schnellen Versenden von Paketen in verschiedenen Netzwerken darstellen. Kriterien zur Beurteilung von solchen Routingalgorithmen sind in diesem Zusammenhang die Laufzeit zum Transport der Pakete L im Vergleich zum Durchmesser des Netzwerkes D, die Anzahl der Pakete S, die in dieser Zeit von ihrem Sender zu ihrem Empfänger transportiert werden können, die maximale Länge Q der in den Knoten benötigten Queues, und bei probabilistischen Algorithmen die Wahrscheinlichkeit, mit der die Ergebnisse erreicht werden. Die Zeit wird in *Schritten* gemessen, ein Schritt ist die Zeit zum Auswählen eine Paketes in einem Knoten und zum Transport über eine Kante zum nächsten Knoten, falls dies möglich ist.

Ein Routingalgorithmus ist *optimal*, wenn $L = O(D)$, $Q = O(1)$ und $S = |V|$. Ein weiteres Kriterium ist die Menge der möglichen Sender/Empfänger Kombinationen. Hier sind nur h–Relationen erlaubt. Dies bedeutet, daß jeder Sender höchstens h Pakete ins Netz sendet und daß jeder Empfänger höchstens h Pakete erhält. Es gilt dann $h < L$. Den Spezialfall $h = 1$ nennt man *Permutation*. In unserem Fall ist jedoch der Fall $h = \log|V|$ von größerer Bedeutung, da auch bei sehr gutem Hashing die Modulbelastung $\log N$ betragen kann und die Speichermodule hier gerade die Empfänger sind. Desweiteren folgt, daß die Zahl der Sender und die der Empfänger durch $|V|/h$ nach oben beschränkt ist.

Ein erster Durchbruch für das Routen von Permutationen war VALIANT's probabilistischer Algorithmus für den k–dimensionalen Würfel [Val82]. Der Algorithmus arbeitet in 2 Phasen. In der ersten Phase wird für jedes Paket ein Zwischen–Empfänger zufällig gewürfelt und das Paket dorthin geschickt. In der zweiten Phase werden die Pakete von den Zwischen–Empfängern zu den Empfängern geschickt. Der Pfad von einem Sender A zu einem Empfänger B wird dabei folgendermaßen gewählt. Man erzeuge einen Bitstring $a_0, \ldots, a_{k-1}$ der Länge k als bitweises exklusives Oder von A und B. Man suche $j = \min\{i | a_i = 1\}$, setze $a_j = 0$ und schicke das Paket über die j–te Kante von A nach A'. Dort verfährt man ebenso. Das Paket hat B erreicht, wenn $a_0, \ldots, a_{k-1}$ nur noch Nullen enthält. Bei Konflikten wird eines der Pakete in einer Queue gespeichert, der Algorithmus funktioniert für alle Queue Disziplinen wie First–In–First–Out, Last–In–First–Out usw.

VALIANT bewies, daß mit Wahrscheinlichkeit mindestens $1 - 2^{-O(k)}$ nach $L = O(k)$ Schritten alle $S = 2^k$ Pakete bei ihrem Empfänger angekommen sind. Die Queues der Knoten können eine Länge von $Q = O(k)$ erreichen. Da der Durchmesser eines Hypercubes $D = k$ ist, sind L und S optimal, nicht jedoch Q.

In der Folge wurden eine Reihe von Routing Algorithmen für Hypercube und ähnliche Netzwerke wie Butterfly oder Shuffle–Exchange Netzwerk entwickelt [Ale82, Upf82, Pip84], die jedoch alle nicht optimal waren, da sie nur für Permutationen gedacht waren. Der erste optimale Algorithmus stammt von RANADE [Ran91, RBJ88], er wurde von LEIGHTON, MAGGS und RAO verallgemeinert [LMRR90, LMR88].

Auch für Gitter wurden eine Reihe von Routing Algorithmen entwickelt [KRT88, Kun88,

KT89, LMT89]. Diese sind aber alle nur für Permutationen geeignet und deshalb nicht optimal in unserem Sinne.

Die beste Lösung bietet der Algorithmus von RANADE, da er auch die Behandlung von konkurrierenden Zugriffen mit einschließt. Seine Lösung umfaßt außerdem sowohl Hashing als auch Routing, deshalb soll diese Lösung im nächsten Abschnitt weiter entwickelt werden.

2.3 Die Fluent Machine

RANADE veröffentlichte 1987 einen probabilistischen Algorithmus [Ran87], der auf einem n–Butterfly Netzwerk arbeitet und $N = n2^n$ Pakete mit Wahrscheinlichkeit mindestens $1 - 2^{-O(n)}$ in Zeit $O(n)$ routet und nur Knoten mit Queues der Länge $O(1)$ benötigt. Der Algorithmus ist damit optimal. Sender und Empfänger sind alle Knoten des Butterfly Netzwerkes. Jeder Sender sendet ein Paket, die Empfänger der Pakete können beliebig gewählt werden. RANADE benutzte diesen Algorithmus zum Entwurf der sogenannten „Fluent Machine“ [RBJ88], die eine PRAM schrittweise simuliert. An jedem Knoten des Butterfly Netzwerkes befinden sich ein Prozessor und ein Speichermodul. Ein Prozessor sendet am Anfang eines Schrittes ein Paket ins Netzwerk, wenn er in diesem Schritt einen Zugriff auf den globalen Speicher simuliert. Der Algorithmus unterstützt konkurrierendes Lesen und Schreiben von Zellen. In den folgenden Abschnitten wollen wir die Verteilung des globalen Speichers und den Algorithmus beschreiben.

2.3.1 Verteilung des globalen Speichers

Die Verteilung erfolgt mittels Hashfunktionen in folgender Art und Weise. Die Speicherzelle $x \in \{0, \ldots, m-1\}$ des globalen Speichers wird im Speichermodul $h(x) = g(x) \bmod N$ an Adresse $l(x) = g(x) \operatorname{div} N$ abgelegt. $g(x)$ wird zufällig aus der Menge G gewählt:

$$G = \left\{ g | g(x) = \left(\sum_{0 \le i < \zeta} a_i x^i \right) \bmod P \right\}$$

Hierbei ist $P \ge m$ eine Primzahl und die Koeffizienten $a_i \in \{0, \ldots, P-1\}$. Die Variable ζ wird zu $\zeta = 8n$ gewählt. Die Funktion g ist also im allgemeinen nicht bijektiv. Bis zu ζ verschiedene Adressen können mittels g auf eine abgebildet sein. Eine einfache Methode, um dieses Problem zu lösen ist die folgende. Jede der m/N Zellen pro Modul wird durch 2ζ viele Zellen ersetzt. Die ersten ζ dieser Zellen erhalten die Originaladressen, die auf diese Zelle abgebildet wurden, die anderen die Inhalte dieser Zellen. Dadurch wird allerdings der Gesamtspeicher um den Faktor $2\zeta = 16n$ vergrößert. RANADE skizziert in [Ran87] eine Methode, die diesen Faktor zur Konstante reduziert. Da im weiteren Verlauf an Stelle dieser Hashfunktion eine lineare (und damit bijektive) Funktion verwendet wird, wird auf die Schilderung dieser Lösung verzichtet. Siehe hierzu auch Absatz 2.2.1.

Die Modulnummer $h(x)$ einer Zelle des globalen Speichers ist ein Wert aus $\{0, \ldots, N-1\}$, die Knoten des Butterfly Netzwerkes, die Module tragen, sind allerdings Werte aus $\{0, \ldots, n-1\} \times \{0, \ldots, 2^n-1\}$. Zur Umsetzung wird die Bijektion $h'(x) = (x \text{ div } 2^n, x \bmod 2^n)$ benutzt. Im Rest dieses Kapitels wird aus Gründen der Einfachheit in der Regel $h(x)$ dem exakten $h'(h(x))$ vorgezogen.

2.3.2 Routing

Will ein Prozessor einen Zugriff auf eine Zelle x des globalen Speichers machen, so schickt er ein Paket ins Netzwerk. Ein solches Paket besteht aus der Adresse x, der gehashten Adresse $h(x)$, einem Typ, der die Art des Zugriffs (lesend oder schreibend) angibt und bei einem schreibenden Zugriff aus dem zu schreibenden Datum. Wir werden im folgenden nur Lesezugriffe betrachten. Schreibzugriffe werden genauso behandelt mit der Vereinfachung, daß hier kein Ergebnis zum Prozessor zurückgeschickt werden muß.

Der Algorithmus arbeitet in 6 Phasen. Zur einfacheren Beschreibung wollen wir vorerst annehmen, daß das benutzte Netzwerk aus 6 aneinandergehängten Butterfly Netzwerken besteht, von denen die mit gerader Nummer umgekehrt sind. Unter Aneinanderhängen von Netzwerken verstehen wir hier, daß die Spalte n des Netzwerkes i mit Spalte 0 des Netzwerkes $i+1$ identifiziert wird.

Jeder Netzwerkknoten hat 2 eingehende und zwei ausgehende Kanten. Pakete, die über eine eingehende Kante den Knoten erreichen, werden in Puffern gespeichert, die maximal b Pakete beinhalten können. Die Puffer arbeiten nach dem First–in–First–out Prinzip. Pakete dürfen über eine ausgehende Kante nur verschickt werden, wenn der betreffende Eingangspuffer des Nachfolgeknotens nicht voll ist.

Ein Paket, das von einem Prozessor (a_0, a_1) ausgeschickt wird und zum Speichermodul $h(x) = (b_0, b_1)$ will, benutzt den folgenden Pfad:

1. In Phase 1 folgt es von (a_0, a_1) dem Pfad der geraden Kanten nach (n, a_1), der gleichzeitig $(0, a_1)$ der Phase 2 ist.
2. In Phase 2 folgt das Paket dem eindeutigen Pfad von $(0, a_1)$ nach (n, b_1) der Phase 2, der gleichzeitig $(0, b_1)$ der Phase 3 ist.
3. In Phase 3 folgt das Paket von $(0, b_1)$ dem Pfad der geraden Kanten nach (b_0, b_1). Dort findet der Speicherzugriff statt.
4. – 6. In diesen Phasen läuft das Antwortpaket genau den gleichen Pfad wie auf dem Hinweg in umgekehrter Reihenfolge.

In Abbildung 2.1 ist der Weg eines Paketes über alle 6 Phasen dargestellt. Der Knoten (a_0, a_1) ist dort mit Source, der Knoten (b_0, b_1) ist mit Destination bezeichnet. Die Abbildung zeigt auch, daß das Paket in den Phasen 1, 3, 4 und 6 nur über gerade Kanten

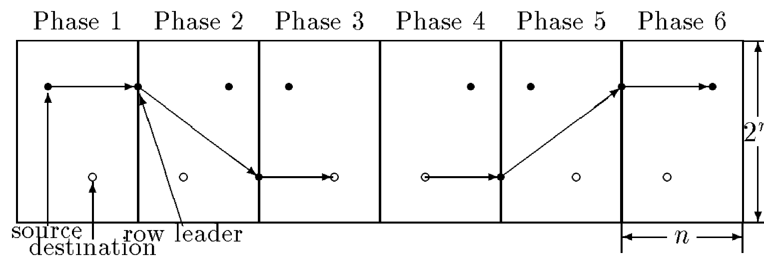

Abbildung 2.1: 6 Phasen Routing der Fluent Machine

weitertransportiert wird. Nur in den Phasen 2 und 5 werden also die kreuzenden Kanten zum Transport von Paketen im Netzwerk benutzt. In Phase 1 benutzen die Prozessoren die kreuzenden Kanten, um die Pakete ins Netzwerk einzuspeisen. Die kreuzenden Kanten enden in Phase 1 also zwar in einem Netzwerkknoten, ihr Anfang ist aber mit einem Prozessor verbunden. In Phase 3 hängt am Ende von kreuzenden Kanten ein Speichermodul, über diese Kanten laufen die Pakete, wenn sie ihren Zielknoten erreicht haben, um den Speicherzugriff auszuführen. In Phase 4 speisen die Speichermodule über die kreuzenden Kanten die Antworten von LOAD Paketen wieder ins Netzwerk ein. In Phase 6 hängt an ausgehenden kreuzenden Kanten ein Prozessor. Hier werden die Pakete, die ihren Ausgangsprozessor wieder erreicht haben, über die kreuzenden Kanten aus dem Netzwerk ausgespeist.

2.3.3 Konkurrierende Zugriffe

Erfolgt in einem Schritt der zu simulierenden PRAM ein konkurrierender Zugriff auf eine Adresse x, so schicken mehrere Prozessoren bei der Simulation dieses Schrittes Pakete mit gleichen Originaladresen x und damit gleichen gehashten Adressen $h(x)$ ins Netz. Die Pfade dieser Pakete bis zum Speichermodul $h(x)$ bilden einen Baum im Netzwerk. Allerdings ist es nicht notwendig, mehr als ein solches Paket über jede Kante des Baumes zu senden. Um dies zu erreichen, muß man dafür sorgen, daß sich nicht nur die Pfade dieser Pakete, sondern die Pakete selbst treffen. Treffen sich zwei solche Pakete in einem Knoten, so können sie zu einem *kombiniert* werden. Bei lesendem Zugriff kann ein Paket gelöscht werden, allerdings muß gewährleistet sein, daß das übriggebliebene Paket auf dem Rückweg an entsprechender Stelle wieder dupliziert wird. Bei schreibendem Zugriff wird das Paket gelöscht, das von einem Prozessor kleinerer Nummer kommt, wodurch eine Priority Regelung realisiert wird. Die Idee der Kombinierung von Paketen wird bereits beim Design des NYU Ultracomputers beschrieben [GGK+83].

Um zu gewährleisten, daß sich zu kombinierende Pakete auch wirklich treffen, verlangt der Algorithmus, daß die Pakete, die einen Knoten verlassen, nach ihren gehashten Adressen aufsteigend sortiert sind. Deshalb erfolgt die Auswahl, welches der beiden führenden Pakete der beiden Eingangspuffer eines Knotens weitergeschickt werden soll, über die gehashten Adressen der beiden Pakete. Das Paket mit der kleineren Adresse darf zuerst weiter, das

andere muß warten. Ist ein Eingangspuffer leer, so muß das führende Paket des anderen Eingangspuffers warten. Unter der Annahme, daß die Pakete in den Eingangspuffern eines Knotens sortiert eintreffen, verlassen sie den Knoten auch sortiert. Durch Induktion über die Spalten des Netzwerks läßt sich nun beweisen, daß die Auswahl den Anforderungen des Algorithmus genügt.

Treffen sich also die Pfade zweier zu kombinierender Pakete, so befinden sich beide Pakete zu gleicher Zeit an den Spitzen der beiden Eingangspuffer und können kombiniert werden. Die Gegenannahme führt direkt zum Widerspruch zur Voraussetzung, daß die von diesem Knoten ausgehende Folge von Paketen nach Adressen aufsteigend sortiert ist. In diesem Fall würden die beiden Pakete nämlich nacheinander den Knoten verlassen und dazwischen mindestens ein anderes Paket mit gehashter Adresse $h(y) \neq h(x)$. Die Folge $h(x), h(y), h(x)$ ist aber nicht aufsteigend sortiert.

Erfolgen in einer Runde sowohl lesende als auch schreibende Zugriffe auf eine Adresse x, so werden nur Pakete gleichen Typs kombiniert. Treffen Pakete für Adresse x und verschiedenen Zugriffsarten aufeinander, so hat das Paket mit lesendem Zugriff Vorrang. So lesen alle Prozessoren, die lesenden Zugriff auf Adresse x haben, noch den alten Wert, der neue Wert wird erst später, aber noch in der Simulation des gleichen PRAM Schrittes geschrieben.

2.3.4 Ghost Pakete

Die Idee, die Pakete nach Adressen sortiert zu halten, hat einen großen Nachteil. Betrachtet man Abbildung 2.2, die aus [Ran87] entnommen ist, so stellt man fest, daß das Paket mit Adresse 25 nicht versendet werden kann, weil der andere Eingangspuffer leer ist, obwohl über die Kante von Knoten A nach B in diesem Schritt wegen der Sortierung nie mehr ein Paket mit einer Adresse kleiner als 35 kommen kann. Um solche unnützen Wartezeiten zu vermeiden, sendet ein Knoten, der ein Paket über eine ausgehende Kante verschickt, über die andere ausgehende Kante ein Paket mit gleicher Adresse und Typ Ghost. Dieses Ghost Paket informiert den nachfolgenden Knoten, in unserem Beispiel B, daß kein Paket mit kleinerer Adresse mehr kommen kann. So könnte im Fall der Abbildung 2.2 das Paket in B versendet werden.

Ein Ghost Paket würde auch von A nach B geschickt, wenn das Paket mit Adresse 35 nicht über die ausgehende Kante versendet werden könnte, weil der daran hängende Eingangspuffer des nachfolgenden Knotens voll wäre. Damit sendet jeder Knoten, nachdem er zum ersten Mal ein Paket ausgesandt hat, in jedem Schritt über jede Kante ein Paket, falls nicht der betreffende Eingangspuffer im nachfolgenden Knoten voll ist. Ein Paket muß also in einem Knoten nur noch warten, wenn wirklich die Möglichkeit besteht, daß noch ein anderes Paket vor ihm diesen Knoten verläßt.

Ein Ghost erfüllt seinen Zweck allerdings nur, wenn er im nächsten Schritt auch wirklich versandt werden kann. Steht in dem Eingangspuffer, in den er gerade aufgenommen wurde, noch mindestens ein Paket vor ihm, so kann er im nächsten Schritt nicht verschickt werden.

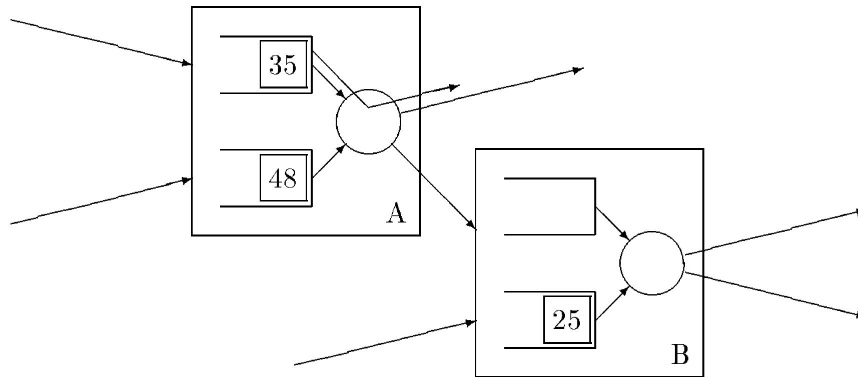

Abbildung 2.2: Funktion von Ghost Paketen

In diesem Schritt trifft aber wieder ein Paket mit einer nicht kleineren Adresse ein. Gleiches gilt für ein Ghost Paket an der Spitze eines Eingangspuffers, das nicht versandt wird, weil der nachfolgende Eingangspuffer voll ist, oder weil an der Spitze des anderen Eingangspuffers ein Paket mit kleinerer Adresse steht. Solche Ghost Pakete werden gelöscht.

Ghost Pakete halten das Netzwerk „fluent", daher der Name der Maschine. Eine ähnliche Funktion haben „End Of Stream" Pakete (EOS). EOS Pakete haben den Typ Ghost, aber als Adresse m. Jeder Prozessor schickt ab dem Zeitpunkt, zu dem er ein Paket ins Netzwerk geschickt hat, ständig EOS Pakete ins Netz. EOS Pakete verlassen aufgrund ihrer Adresse einen Knoten nur, wenn er kein Paket und keinen Ghost mehr enthält und damit auch nicht mehr erhält. Dies garantiert, daß das letzte Paket, das einen Knoten verläßt, nicht aufgrund eines leeren Eingangspuffers hängenbleibt.

2.3.5 Rückweg

Erreicht ein Paket sein Speichermodul und hat einen Lesezugriff ausgeführt, so muß es wieder zurück zu dem Prozessor, von dem es ausgeschickt wurde. Dies könnte prinzipiell genauso geschehen wie auf dem Hinweg, allerdings gibt es eine einfachere Methode. Jeder Netzwerkknoten in den Phasen 1 bis 3 merkt sich in einem zusätzlichen FIFO Puffer, der sog. „Direction Queue", für jedes Paket, das einen Lesezugriff ausführen will und ihn verläßt, aus welchem Eingangspuffer es kam und über welche ausgehende Kante es den Knoten verließ. Wurde eine Kombinierung zweier Pakete mit Lesezugriffen ausgeführt, so merkt sich der Knoten die Kombinierung und über welche ausgehende Kante das übriggebliebene Paket den Knoten verließ. Ein Knoten der Phasen 4 bis 6 liest jetzt den führenden Eintrag der Direction Queue, wartet bis eine Antwort über die entsprechende Kante zurückkommt, dupliziert diese falls notwendig und schickt die Antwort bzw. die Antworten weiter, wie es der Eintrag vorschreibt.

Da die Direction Queue nur 6 Fälle erfassen muß (Eingang i nach Ausgang j, $i, j \in \{0, 1\}$

und Kombinierung nach Ausgang j, $j \in \{0,1\}$), muß sie bei kompakter Kodierung nur 3 Bit breit sein. Da die Laufzeit zur Simulation mit sehr großer Wahrscheinlichkeit höchstens $O(n)$ Schritte dauert, benötigt die komplette Direction Queue $O(n)$ Bits. Da eine Adresse eines Paketes ein Wert aus $\{0,\ldots,m-1\}$ ist, benötigt ein Paket mindestens $\Omega(\log m) = \Omega(n)$ Bits zu seiner Kodierung, da $m \geq N = n2^n$. Da ein Eingangspuffer konstant viele Pakete enthält, benötigt er $\Omega(n)$ Bits. Die Direction Queue benötigt also nicht mehr Platz als ein Eingangspuffer, obwohl sie eine größere Länge hat.

In den seltenen Fällen, in denen die Simulation länger als $O(n)$ Schritte dauert, könnte die Direction Queue mehr Platz benötigen. Hier muß dafür gesorgt werden, daß die Queue nicht überläuft. Ist die Direction Queue eines Netzwerkknotens voll, so wird das nächste LOAD Paket, das diesen Knoten auf dem Hinweg durchläuft und deshalb etwas in diese Queue schreiben müßte, so lange aufgehalten, bis die Direction Queue nicht mehr voll ist. Dies beeinträchtigt die Korrektheit des Verfahrens nicht. Die zusätzlich auftretende Verzögerung wird hingenommen, da die Simulation ohnehin Zeit $O(2^n)$ benötigen kann.

2.3.6 Aufbau

Zur Realisierung der Fluent Machine hängt man ein Butterfly Netzwerk und ein umgekehrtes Butterfly Netzwerk an beiden Enden aneinander. Das erste Netzwerk simuliert die Phasen 1,3,5, das zweite die Phasen 2,4,6. Jeder Netzwerkknoten realisiert drei verschiedene Netzwerkknoten und benötigt deshalb 3 Direction Queues. Da aber zu jedem Zeitpunkt nur eine der drei Phasen in einem Knoten abläuft, müssen die Eingangspuffer nicht verdreifacht werden. Allerdings kann ein Eingangspuffer oder ein Ausgang eines Knoten in verschiedenen Phasen mit verschiedenen Dingen (anderem Knoten, Prozessor, Speichermodul) verbunden sein.

Der Grund, weshalb man als zweites Netzwerk ein umgekehrtes Butterfly Netzwerk benutzt, liegt bei den Direction Queues. Ein naheliegender Aufbau der beiden Netzwerke ist, Knoten (a_0, a_1) des ersten Netzwerkes und Knoten $(n - a_0, a_1)$ des zweiten Netzwerkes räumlich direkt nebeneinander anzuordnen, um eine einfachere Verdrahtung zu erhalten. Dies sind allerdings die Knoten, zwischen denen Direction Queues bestehen. Diese können dann sehr einfach realisiert werden.

Kapitel 3

Bewertung von Rechnern

In diesem Kapitel wird ein formales Modell zur Beschreibung und Bewertung von Rechnern vorgestellt. Das Modell wurde von MÜLLER und PAUL entwickelt [MP90] und in [Mül91] ausgearbeitet. Es war vorrangig zur Bewertung und zum Vergleich von Prozessoren gedacht, so zum Beispiel zur Beantwortung von Fragen der Art „Sollten in einem RISC Prozessor Ausdrücke besser auf dem Stack oder nach Möglichkeit in den Registern ausgewertet werden?“ oder „Sollte man in einen Motorola MC68000 besser einen Carry–Lookahead Addierer oder einen Carry Chain Addierer einbauen?“. Die Beantwortung solcher Fragen (und damit der Entwurf neuer Rechner) wird mit diesem Modell von einer Kunst zu einer mathematischen Optimierungsaufgabe. Der erste Abschnitt erläutert das Modell in der hier benötigten Tiefe. Im zweiten Abschnitt werden Änderungen in Zielsetzung und Modell beschrieben, die zum Vergleich von skalierbaren Parallelrechnern notwendig sind. Im dritten Abschnitt werden Standardkomponenten von Rechnern, z.B. Addierer, als Beispiele aufgeführt, im vierten Abschnitt wird die Fluent Machine bewertet.

3.1 Formales Modell

In dem verwendeten formalen Modell werden *Architekturen* beschrieben. Zu einer Architektur zählt hierbei aber nicht nur die Hardware einer bestimmten Maschine. Auch die Software spielt eine erhebliche Rolle. Da in der Anwendungsprogrammierung heute generell Hochsprachen wie PASCAL oder C benutzt werden, sollte also zumindest der (oder die) benutzte(n) Compiler ebenfalls dazuzählen. Auf die Integration des Betriebssystems und der Peripheriehardware in die Architektur wird verzichtet.

Definition 3.1 *Eine* Architektur *A wird dargestellt durch ein Tupel $\mathcal{A} = (IS, H, L, Co)$. Hierbei ist IS eine Maschinensprache, H eine Hardware, auf der IS ausführbar ist, L eine Menge von Hochsprachen und Co eine Menge von Compilern, die Programme, die in Hochprachen aus L geschrieben sind, in Programme in IS übersetzen.*

Anmerkung: Wir werden uns im weiteren auf Architekturen mit nur einer Hochsprache beschränken und die einelementigen Mengen L und Co mit ihrem Element identifizieren.

Zu gegebenen IS und H kann für jede Instruktion $I \in IS$ eine Ausführungszeit $T_H(I)$ angegeben werden, ferner können die Kosten $C(H)$ der Hardware bestimmt werden. Vorgehensweisen zum schnellen Berechnen von C und T per Hand werden in 3.1.2 formal beschrieben.

Zur Bewertung einer Architektur benötigt man Angaben über die *Workload*, also ein Modell des Benutzerverhaltens in dem Sinne, wie der Anwender die Ressourcen der Maschine auslastet. Die Workload wird durch ein kurzes Programm B in der Sprache L einschließlich einer Eingabe beschrieben. B heißt *Benchmark*. Das Problem, geeignete Benchmarks zu finden, ist in der Literatur vielfach behandelt worden. Für einige Fälle konnte man sich auf „Standardbenchmarks" einigen, so zum Beispiel auf die *Livermore Loops* [McM88] als Benchmarks für Vektorrechner.

Compiliert man B mit Co in ein Programm in der Maschinensprache IS und analysiert man dieses, so erhält man für jede Instruktion $I \in IS$ ihre absolute Häufigkeit $H_B(I)$. Die Vorgehensweise bei Übersetzung und Analyse (ohne den Compiler zu implementieren) ist in 3.1.1 angegeben. Die Gesamtzeit zur Ausführung des Programmes ist dann

$$T_H(B) = \sum_{I \in IS} H_B(I) \cdot T_H(I)$$

.

Der Reziprokwert von $T_H(B)$ ist ein Maß für die Leistung der Maschine.

Für eine gegebene Architektur A und gegebenes Benchmark B können nun verschiedene Gütemaße $G(\mathcal{A}, B)$ definiert werden. Sind Architektur und Benchmark bekannt, können die Indices entfallen. Das Aussehen des Gütemaßes hängt von der Zielsetzung des Architekten ab. Die 3 vorherrschenden Zielsetzungen sind:

Leistung um jeden Preis. Diese Zielsetzung kommt in der Regel nur bei Höchstleistungsrechnern vor. Hier wählt man $G = 1/T$.

Gutes Preis/Leistungsverhältnis. Diese Zielsetzung ist die gängigste bei „normalen" Computern. Hier wählt man $G = 1/(T \cdot C)$.

Leistung wichtiger als Preis. Diese Zielsetzung ist ein Kompromiß zwischen den beiden ersten. Hier wählt man $G = 1/(T^i \cdot C)$ mit $i > 1$. Je größer i, desto wichtiger wird die Leistung. In [Mül91] wird hier $i = 2$ benutzt.

Wir werden im folgenden immer das zweite Gütemaß benutzen.

Die Fragen, die zu Beginn des Kapitels gestellt wurden, kann man mit Hilfe dieses Modells lösen. Man vergleicht dazu zwei Architekturen, die sich im ersten Fall nur im verwendeten

Compiler unterscheiden und im zweiten Fall leicht verschiedene Hardware haben. Viele Design–Entscheidungen in der Rechnerarchitektur können jetzt als Optimierungsproblem gelöst werden. Als Beispiel diene die Frage „Welches ist die beste Registerzahl für eine ansonsten fest vorgegebene Architektur A und gegebenes Benchmark B?“. Man erhält abhängig von der Zahl x der Register verschiedene Kosten und Laufzeiten. Die Güte G ist damit eine Funktion mit der Variablen x. Gesucht ist der absolute (oder zumindest ein guter lokaler) Maximumpunkt für $G(x)$.

3.1.1 Hochsprache und Compiler

Die meisten heutigen Hochsprachen wie PASCAL oder C unterstützen sehr viele verschiedene Anweisungen. Compiler von Hochsprachen nach Maschinensprachen hingegen verfolgen meist nur wenige Strategien und erzeugen nur wenige verschiedene Code–Sequenzen [Mül91]. Ein Beispiel für eine Strategie ist die Ausdrucksauswertung. Hier gibt es die Möglichkeit, Ausdrücke möglichst in den Registern auszuwerten, auf dem Stack oder in einem festen Speicherbereich, dem Scratch.

Ein Compiler für eine weitere Hochsprache oder für eine veränderte Maschine kann also sehr einfach durch Veränderung der Strategien (falls notwendig) und der Code–Sequenzen beschrieben werden. Die Übersetzung eines Hochsprachenprogrammes besteht nun aus der Zerlegung in Code–Sequenzen. Eine Analyse des so entstandenen Maschinenprogramms ist einfach, da die Eingabe des Programms bekannt ist.

3.1.2 Hardware

Eine bereits seit Jahrzehnten bekannte Darstellungsform für Hardware ist die Darstellung durch *Schaltkreise*. Die Schaltelemente oder Komponenten bilden hierbei die Knoten eines gerichteten Graphen, ihre Verbindungen die Kanten. Gewisse Verschaltungsregeln müssen beachtet werden, so zum Beispiel maximaler Ingrad der verschiedenen Typen von Komponenten. Ausgezeichnete Knoten sind Eingänge (Ingrad 0) und Ausgänge (Outgrad 0). Jeder Typ von Komponente (in der Regel die verschiedenen Arten von Gattern) hat gewisse Kosten und eine gewisse Durchlaufzeit *(Delay)*. Die Kosten eines solchen Schaltkreises sind die Summe der Komponentenkosten. Die Durchlaufzeit eines Pfades von einem Eingang zu einem Ausgang ist die Summe der Durchlaufzeiten der Knoten auf dem Pfad. Die Tiefe des Schaltkreises ist die maximale Durchlaufzeit eines Pfades von einem Eingang zu einem Ausgang.

Computer lassen sich mit dem Modell des Schaltkreises allerdings nicht zufriedenstellend beschreiben, da Computer auch Komponenten wie Register, Speicher und Treiber beinhalten. Register können zu Zyklen im Graphen führen, Treiber zu Uneindeutigkeiten (mehrere Treiber führen zu einem Bus), außerdem benötigen beide Kontrollsignale.

Das Modell des Schaltkreises wird deshalb wie folgt zum Begriff der Hardware erweitert.

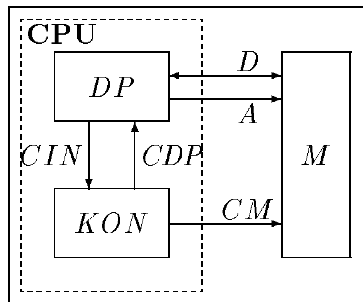

Abbildung 3.1: Grobstruktur der Hardware

Die Beschreibung der Hardware ist eine verkürzte Form der Beschreibung in [Mül91].

Definition 3.2 *Die Hardware H einer Architektur besteht aus den Datenpfaden DP, der Kontrollogik KON, dem Speichersystem M und den sie verbindenden Bussen. Datenpfade und Kontrollogik werden zur CPU zusammengefaßt. Zur Erläuterung dient Abbildung 3.1.*

Um die Kontrollsignale zu berechnen, empfängt KON über CIN einige Daten aus DP. DP und M werden über CDP und CM gesteuert. Der Datenaustausch zwischen DP und M wird über den bidirektionalen Datenbus D und den Adressbus A abgewickelt.

Grundbausteine

Wir beschreiben zuerst die Grundbausteine der Hardware und die Regeln ihrer Verschaltung. Signale können hierbei Werte aus der Menge $E = \{0, 1, ?\}$ sein, wobei „?" einen undefinierten Wert bedeutet. Ein Signal heißt *aktiv* bei Wert 1, *inaktiv* bei Wert 0.

Als Grundbausteine stehen AND, OR, NOR, NAND, EXOR, EQUAL, NOT, Treiber, Register, RAM, ROM und Tristate–Busse zur Verfügung. In der Schaltkreistheorie sind die von den Bausteinen AND bis NOT berechneten Schaltfunktionen nur auf den Signalwerten 0 und 1 definiert. Zur Erweiterung auf die Menge E definiert man den Wert der Schaltfunktion von EXOR, EQUAL und NOT als undefiniert „?", sobald mindestens ein Eingang undefiniert ist. Bei OR (NOR) hat die Schaltfunktion den Wert 1 (0), falls ein Eingang undefiniert ist und der andere den Wert 1 hat, sonst ist der Wert undefiniert. Bei AND (NAND) hat die Schaltfunktion den Wert 0 (1), falls ein Eingang undefiniert ist und der andere den Wert 0 hat, sonst ist der Wert undefiniert.

Ein n–bit Treiber hat n Dateneingänge, n Datenausgänge und einen Kontrolleingang *output enable (OE).* Ist das Kontrollsignal aktiv, berechnet der Treiber die Identität. Ist OE inaktiv oder undefiniert, so sind die Ausgänge undefiniert.

Ein n–bit Tristate–Bus verbindet die n–bit breiten Ausgänge von k Komponenten K_i^A mit

den n–bit breiten Eingängen von j Komponenten K_j^E. Hat der Ausgang genau einer Komponente K_i^A einen Wert ungleich $?^n$, so liegt dieser Wert an den Eingängen aller Komponenten K_j^E an, ansonsten $?^n$,was physikalisch einem Kurzschluß entspricht.

Die Hardware von Computern besteht nicht nur aus Schaltkreisen, sondern aus Schaltwerken, die in *Zyklen* arbeiten. Deshalb benötigt man Bausteine zum Speichern von Werten. Solche Bausteine sind Register, RAMs (Random Access Memory) und ROMs (Read only Memory). Gespeicherte Werte können sich nur am Ende eines Zyklus ändern, bei ROMs gar nicht. Die Kontrollsignale der Speicherbausteine sind während eines Zyklus fest.

Ein n–bit Register verfügt über je n Dateneingänge und –ausgänge und über einen Kontrolleingang, der mit *clock (ck)* bezeichnet wird. Ein Register speichert einen n–bit Wert, der an den Ausgängen anliegt. Am Ende eines Zyklus wird bei aktiver Clock der Wert der Eingänge gespeichert, ansonsten bleibt der bisher gespeicherte Wert.

Für natürliche Zahlen a, d und $A = 2^a$ besitzt ein $A \times d$–RAM a Adreßeingänge Ad und je d Dateneingänge und –ausgänge Din und $Dout$, zusätzlich einen Kontrolleingang rw (read–write). Ein solches RAM speichert A Werte, die je d bit breit sind. Hat das Signal rw den Wert 1, so liegt an $Dout$ der gespeicherte Wert an, der durch Ad spezifiziert wird. Hat rw Wert 0, so wird der Wert, der an Din anliegt, am Ende des Zyklus an Adresse Ad gespeichert. $Dout$ hat dann den Wert $?^n$, ebenso für $rw =?$.

Ein $A \times d$–Dual Port RAM besitzt jeweils doppelt soviele Adress– und Datenein– und –ausgänge und Kontrolleingänge wie ein RAM. Hier können während eines Zyklus zwei Zugriffe auf das RAM erfolgen, allerdings müssen dabei an den beiden Adresseingängen verschiedene Werte anliegen.

Ein ROM verhält sich wie ein RAM, auf das nur lesend zugegriffen werden darf. Die gespeicherten Werte sind beliebig, aber fest.

Zusammenschalten von Komponenten

Die Regeln zum Zusammenschalten von Komponenten umfassen für die Komponenten AND bis NOT die Regeln der klassischen Schaltkreistheorie (siehe z.B. [Weg87]):

- Ausgänge einer Komponente dürfen nur mit Eingängen anderer Komponenten verbunden sein.
- Eingänge von Komponenten dürfen mit Ausgängen von Komponenten oder Eingängen des Schaltkreises verbunden sein. Eingänge von Komponenten dürfen miteinander verbunden werden, wenn genau einer davon mit dem Ausgang einer Komponente oder einem Eingang des Schaltkreises verbunden ist.
- Der Graph des Schaltkreises muß azyklisch sein.

Beim Zusammenschalten mit Treibern, Registern usw. gelten weiter die folgenden Regeln:

- Die Eingänge eines Tristate–Bus dürfen nur mit Ausgängen von Treibern verbunden sein. Die Ausgänge eines Tristate–Bus dürfen mit Eingängen aller Komponenten außer Tristate–Bussen verbunden sein.
- Der Ausgang eines Treibers darf nur mit einem Tristate–Bus verbunden sein.
- Entweder ist der Graph des Schaltwerkes azyklisch oder jeder Zyklus beinhaltet mindestens ein Register oder 2 Treiber.

Für das Zusammenschalten von Schaltwerken zu einer CPU gelten weitere Regeln Hierbei ist zu beachten, daß CDP und CIN keine Tristate–Busse sind.

- Kontrolleingänge von Bausteinen in DP dürfen nur über den CDP Bus mit Ausgängen von Komponenten von KON verbunden sein. CDP besteht damit aus den ck aller Register, den oe aller Treiber und den rw aller RAMs in DP. Zusätzlich können weitere Signale vorkommen, z.B. Steuersignale für Multiplexer. Verfolgt man die Pfade der Signale von CDP aus rückwärts, so muß man stets ein Register vor einem Treiber finden.
- Signale in CIN haben ihren Ursprung in Ausgängen von Komponenten in DP. Beim Zurückverfolgen von diesen Ausgängen aus muß man stets ein Register vor einem Treiber finden.
- Verfolgt man die Pfade der Kontrollsignale von Registern und Treibern in KON rückwärts, so dürfen keine Treiber zu finden sein.

Es gibt zwei verschiedene Strategien zum Aufbau einer Kontrollogik: programmiert und festverdrahtet *(hardwired)*. Wir werden im folgenden nur hardwired Logik verwenden. Hier werden Kontrollsignale aus dem Mikro–Status, dies ist der Inhalt aller Register in KON, und den Werten des CIN–Busses durch boole'sche Polynome berechnet und, falls notwendig, in Registern der Kontrollogik abgelegt. Die Schaltkreise zur Realisierung der boole'schen Polynome benutzen nur AND, OR und NOT Gatter, sie berechnen zuerst Literale, dann Monome und zuletzt Polynome. Zur Tiefenreduktion verwendet man balancierte binäre Bäume.

Zu Beginn eines Zyklus werden aus Werten in den Registern in KON und aus Werten in CIN alle Kontrollsignale berechnet. Sie ändern sich nicht vor dem Ende des Zyklus, da die Register auf ihren Pfaden, die nach Definition vorhanden sind, ihre Ausgänge nicht vor Ende des Zyklus ändern. Durch die Kontrollsignale wird eine *modifizierte Hardware* definiert:

- Treiber, deren oe nicht aktiv ist, werden entfernt, die mit aktivem oe berechnen die Identität.
- RAMs, aus denen gelesen wird, und ROMs werden ersetzt durch ein Register mit nachfolgender Logik zur Adressverarbeitung.

- RAMs, in die geschrieben wird, werden ersetzt durch einen Schaltkreis zur Adressverarbeitung und ein nachfolgendes Register.

- Nach diesen Schritten verfolgt man von Registern mit aktiver *ck* aus die Pfade in der Hardware rückwärts und trifft dabei eine Anzahl von Komponenten. Alle Schaltkreise und Treiber, die man nicht trifft, werden entfernt.

Der letzte Schritt gewährleistet, daß alle Komponenten, die in einem Zyklus nichts tun, aus der modifizierten Hardware entfernt werden.

Nicht alle Kombinationen von Kontrollsignalen sind jedoch sinnvoll bzw. zulässig. Haben zum Beispiel zwei Treiber, deren Ausgang mit dem gleichen Bus verbunden ist, beide aktive *oe* Signale, so tritt ein Kurzschluß auf. Zulässige Kombinationen von Kontrollsignalen müssen deshalb die folgenden Bedingungen erfüllen.

- Der Graph der modifizierten Hardware ist azyklisch oder in jedem Zyklus liegt mindestens ein Register.

- Jeder Tristate–Bus der modifizierten Hardware ist mit dem Ausgang genau einer Komponente verbunden.

- Jedes Register, dessen *ck* aktiv ist, verfügt über keine offenen Eingänge, d.h., alle Eingänge sind mit Ausgängen von Komponenten der modifizierten Hardware verbunden. An den Eingängen darf kein undefinierter Wert anliegen.

Zeitverhalten

Um einen Zyklus beenden zu können, müssen alle Signale der Hardware *stabil* sein. Dies bedeutet, daß jedes Signal den Wert angenommen hat, den es bei gegebenen Registerinhalten und Kontrollsignalen als Schaltfunktion berechnen soll. Dies benötigt Zeit, da jeder Grundbaustein eine gewisse Zeit d, die sogenannte Verzögerung *(delay)*, benötigt, bis Änderungen an den Eingängen sich als Änderungen der Ausgänge fortgesetzt haben. Der Delay der Grundbausteine ist in Tabelle 3.1 zu sehen. Er hängt in der Realität von der Technologie ab, in der die Schaltung realisiert ist. Zum Beispiel ist die Verzögerungszeit eines AND Gatters in ECL Technologie geringer als bei CMOS. Da Architekturen hier aber möglichst unabhängig von bestimmten Technologien untersucht werden sollen, normieren wir unser Maß bezüglich der Durchlaufzeit von AND Gattern. Zur Berechnung der relativen Verzögerungen anderer Gatter legen wir eine feste Technologie zugrunde, hier wurde die CMOS Technologie der Firma Motorola gewählt [Mot89a].

Gesucht ist die minimale Dauer eines Zyklus bei gegebenen zulässigen Kontrollsignalen und beliebigen Daten. Hierzu muß man für alle Komponenten berechnen, wann ihre Ausgänge stabil werden, und das Maximum dieser Zeiten bilden. Es gilt folgendes:

Baustein	d(Baustein)
`NOT`	1
`AND`, `OR`, `NAND`, `NOR`	1
`EXOR`, `EQU`	2
n–bit Treiber	1
n–bit Register	4
$A \times d$ RAM	$1.5 \log_2 A$
$A \times d$ ROM	$1.5 \log_2 A$

Tabelle 3.1: Delay der Grundbausteine

- Die Ausgänge der Register sind zu Beginn des Zyklus stabil.
- Der Ausgang einer Komponente, die kein Register ist, ist genau dann zu Beginn des Zyklus stabil, wenn alle Pfade zu dieser Komponente in Registern entspringen, die zu Ende des vorhergehenden Zyklus kein aktives ck haben.
- Sei K ein Treiber, ein RAM, oder ein ROM und t der früheste Zeitpunkt nach Beginn des Zyklus, zu dem alle Eingänge von K stabil sind. Dann sind nach $t + \mathrm{d}(K)$ die Ausgänge von K stabil.
- Sei K eine Komponente, die nicht unter die vorherigen Punkte fällt. Sei t der früheste Zeitpunkt nach Beginn des Zyklus, zu dem alle Eingänge von K stabil sind oder ein Eingang stabil ist und einen Wert hat, so daß der Wert des Ausgangs unabhängig vom Wert der anderen Eingänge berechnet wird. Dann sind die Ausgänge von K zum Zeitpunkt $t + \mathrm{d}(K)$ stabil.

Sei S die Menge der Register, deren ck am Ende des Zyklus aktiv ist. Der früheste Zeitpunkt nach Beginn des Zyklus, zu dem die Eingänge aller Register aus S stabil sind (bei festen Kontrollsignalen und beliebigen RAM– und Registerinhalten) sei $\tilde{t}_z$. Die Länge des Zyklus ist dann $t_z = \tilde{t}_z + \delta$. Der Parameter δ modelliert Setup– und Hold–Zeiten beim Clocken der Register am Ende des Zyklus. Er ist notwendig, um unerwünschte Effekte bei Pipelines zu verhindern. In [Mül91] wird $\delta = 1 + \mathrm{d}(\text{Register})$ gewählt, in dieser Arbeit wird ebenfalls dieser Wert benutzt.

Kostenmaß

Die Kosten der Grundbausteine lassen sich aus Datenbüchern von VLSI Herstellern ableiten. Im weiteren wird für diese sogenannten Basiskosten die Funktion c verwendet. Die Basiskosten sind bezüglich der Kosten eines NOT Gatters normiert. Die relativen Kosten entstammen der Technologie von Motorola [Mot89a]. Eine Auflistung ist in Tabelle 3.2 zu sehen.

Baustein	c(Baustein)
NOT	1
AND, OR, NAND, NOR	2
EXOR, EQU	6
n–bit Treiber	$6n$
n–bit Register	$12n$
$A \times d$ RAM	$12dA$
$A \times d$ Dual Port RAM	$24dA$
$A \times d$ ROM	$6dA$

Tabelle 3.2: Basiskosten der Grundbausteine

Struktur	Parameter	Wert
Einzel	ρ	1
Arithmetik	ρ_A	0.75
RAM klein	ρ_K	0.45
RAM groß	ρ_G	0.31

Tabelle 3.3: Packungsfaktoren

Die Basiskosten für RAMs und ROMs wurden so berechnet, als seien sie aus Registern bzw. Treibern aufgebaut. VLSI Hersteller bieten in ihren Design–Katalogen aber zusätzlich zu den Grundbausteinen sehr viele Makrozellen an, darunter auch RAM– und ROM–Blöcke. Deren Kosten sind niedriger als hier berechnet, da man diese regelmäßigen Zellen sehr platzsparend aufbauen kann. Um das Kostenmaß realitätsnah zu halten, führt man Packungsfaktoren für RAMs abhängig von der Größe ein. Das gleiche Argument gilt auch für andere verfügbare Makrozellen. Dies sind im wesentlichen Zellen zur Unterstützung arithmetischer Operationen, für die man ebenfalls Packungsfaktoren einführt. Die in dieser Arbeit verwendeten Packungsfaktoren wurden aus [Mot89a] abgeleitet und sind in Tabelle 3.3 aufgeführt. Die Angabe von zwei Packungsfaktoren für RAMs abhängig von der Größe („klein, groß“) beruht auf der Tatsache, daß in der Regel nur zu wenige RAM Größen zur Verfügung stehen, um daraus eine Funktion zu extrahieren. Mit „klein“ sind hier RAMs mit $Ad \leq 288$, mit „groß“ RAMs mit $Ad \approx 4600$ gemeint.

Die Gesamtkosten einer Hardware lassen sich dann wie folgt berechnen. Man unterteilt die Menge B der benutzten Grundbausteine in die disjunkten Mengen A, K, G, L der Grundbausteine in Arithmetik, kleinen RAMs, großen RAMs und dem Rest. B umfaßt nur die Komponenten der CPU, der Speicher M fließt nicht in das Kostenmaß ein. Die Gesamtkosten ergeben sich aus der Gleichung

$$c(H) = \rho_A \cdot \sum_{k \in A} c(k) + \rho_K \cdot \sum_{k \in K} c(k) + \rho_G \cdot \sum_{k \in G} c(k) + \rho \cdot \sum_{k \in L} c(k)$$

3.2 Änderungen des Modells

In [Mül91] können Zyklen zur Ausführung verschiedener Instruktionen verschiedene Länge haben. Sind zur Ausführung einer Instruktion mehrere Zyklen erforderlich, so können auch diese verschiedene Längen haben. Beim Entwurf realer Systeme versucht man Berechnungen so in Zyklen zu unterteilen, daß alle Zyklen gleiche Länge haben. Die Ausführungszeit verschiedener Instruktionen unterscheidet sich dann nur noch in der Anzahl der benötigten Zyklen. Ein gutes Beispiel hierfür ist in [Mot89b, Section 10] zu finden.

In dieser Arbeit werden Parallelrechnerkonzepte verglichen. Diese Konzepte unterscheiden sich im verwendeten Netzwerk, im Routingalgorithmus usw. Da die meisten Originalarbeiten, insbesondere die von RANADE, keine Angaben über den verwendeten Prozessor enthalten, wird im Abschnitt 3.4.1 ein Prozessor entwickelt und dieser mit minimalen Abwandlungen für alle hier verglichenen Architekturen verwendet. Der Prozessor benötigt für alle Instruktionen die gleiche Zeit. Eine Ausnahme bildet nur der LOAD Befehl. Das Ergebnis eines LOAD Befehls steht im veränderten Prozessor erst im übernächsten Befehl zur Verfügung. Man spricht von *delayed LOAD*. Wird nach dem LOAD ein Befehl ausgeführt, der das Ergebnis des LOAD Befehls nicht benötigt, so hat das delayed LOAD keinen Einfluß auf die Laufzeit. Im anderen Fall muß eine NOP Instruktion eingefügt werden. Nach HENNESSY und PATTERSON kann aber in den allermeisten Fällen ein vom LOAD unabhängiger Befehl ausgeführt werden [HP90]. Wir werden deshalb den Einfluß von Delayed LOAD Befehlen auf die Laufzeit vernachlässigen und allen Befehlen gleiche Laufzeit zuordnen.

Da alle betrachteten Architekturen den gleichen Instruktionssatz haben und da in dieser Arbeit keine Compilerstrategien untersucht werden, beschränken wir uns auf die Hochsprache PASCAL mit der PARDO Erweiterung und benutzen einen festen Compiler Co.

Um die Ausführungszeit eines Benchmarks zu bestimmen, muß man nur die Anzahl der ausgeführten Maschineninstruktionen berechnen, da die Ausführungszeit aller Befehle gleich ist. Deshalb ist für die Berechnung der Laufzeit die spezielle Gestalt des Benchmarks nicht von Bedeutung. Dies enthebt der Mühe, ein repräsentatives Benchmark für eine Architektur zu suchen, die bisher theoretisch ist und deshalb auch keinen „realen“ Anwenderkreis hat. Deshalb wird auch der Compiler nicht näher spezifiziert.

Um allerdings die Kosteneffektivität einiger Design Varianten zu bestimmen, sind die relativen Häufigkeiten bestimmter Befehle von Bedeutung. Diese relativen Häufigkeiten werden parametrisiert und die Kosteneffektivität in Abhängigkeit dieser Parameter bewertet. Um einen Hinweis auf reale Werte der Parameter zu erhalten, wurde die Berechnung der Zusammenhangskomponenten eines ungerichteten Graphen [SV82] als Benchmark gewählt,

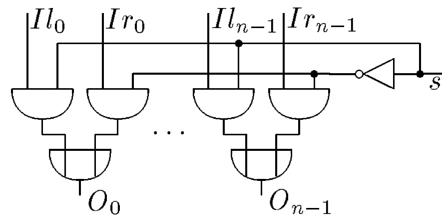

Abbildung 3.2: Multiplexer

das Benchmark mit einem Compiler der PARDO Sprache [F+92] übersetzt und mittels eines Simulators der Architektur gestartet. Der Simulator liefert die relativen Häufigkeiten aller Befehle [San90].

Die in [Mül91] betrachteten Architekturen waren sequentielle Rechner, die aus CPU und Speicher bestanden. Die hier untersuchten Architekturen sind Parallelrechner mit vielen CPU's, vielen Netzwerkknoten und vielen Speichern. Wir bewerten die Netzwerkknoten wie CPU's, die Speicher haben weiterhin keine Kosten.

Die mit dem puren Modell gemachten Aussagen sind beweisbar, sie hängen nicht von Implementierungen oder Simulationen ab. Wir werden von dieser Vorgehensweise bei der Beweisbarkeit der Laufzeit von Routingalgorithmen abweichen und Grad der Hashfunktion, Puffergröße in Netzwerkknoten und Anzahl der Routingschritte aus Simulationen gewinnen.

3.3 Wichtige Hardware Bausteine

Um Analysen von Maschinen zu erleichtern, werden hier einige wichtige und oft benutzte Bausteine vorgestellt. Effiziente Realisierungen dieser Bausteine findet man in der Literatur, z.B. in [Spa76, Weg87]. Wir werden uns hier auf eine kurze Darstellung der von uns verwendeten Lösung beschränken und auf Korrektheitsbeweise verzichten.

3.3.1 Multiplexer

Definition 3.3 *Ein n–bit* Multiplexer ($\mathtt{MUX}_n$) *verfügt über zwei n–bit breite Eingänge Il und Ir, einen n–bit breiten Ausgang O und über einen Kontrolleingang s* (select). *Der Wert des Ausgangs ist gleich dem Wert des linken Eingangs bei select=0, gleich dem Wert des rechten Eingangs bei select=1, undefiniert sonst.*

Ein Multiplexer wird realisiert durch den in Abbildung 3.2 gezeigten Schaltkreis.

Kosten und Delay des Multiplexers lassen sich nun sehr einfach berechnen.

$$\begin{aligned} c(\texttt{MUX}_n) &= 2nc(\texttt{AND}) + nc(\texttt{OR}) + c(\texttt{NOT}) = 6n+1 && (3.1)\\ d(\texttt{MUX}_n) &= d(\texttt{AND}) + d(\texttt{OR}) = 2 && (3.2) \end{aligned}$$

Anmerkung: Delay bezeichnet hier die Verzögerung Eingang nach Ausgang, zur Berechnung der Verzögerung select nach Ausgang muß der Delay des NOT Gatters addiert werden. Da in späteren Berechnungen normalerweise Verzögerungen auf Datenpfaden berechnet werden (die Kontrollsignale liegen bereits kurz nach Beginn des Zyklus an), soll dies auch bei den folgenden Komponenten gelten.

Ein Multiplexer wird bei der Bewertung zu den arithmethischen Komponenten gerechnet.

3.3.2 Addierer

Definition 3.4 *Ein n–bit* Addierer *ist ein Schaltkreis mit zwei n–bit breiten Eingängen und einem $n+1$–bit breitem Ausgang, der folgende Schaltfunktion berechnet: liegen an den Eingängen die n–stelligen Binärdarstellungen der Zahlen $x, y \in \{0, \ldots, 2^n - 1\}$ an, so erscheint am Ausgang die $n+1$–stellige Binärdarstellung der Zahl $x+y$.*

Anmerkung: Das höchstwertige Bit des Ausgangs wird *Carry* oder *Carry out* genannt. Addierer werden manchmal auch so definiert, daß sie einen dritten, 1–bit breiten Eingang, den sogenannten *Carry in (c_{in})* besitzen. Am Ausgang liegt dann die Zahl $x + y + c_{in}$ an.

Ein n–bit Addierer hat mindestens Kosten $\Omega(n)$, da er n Ein– und Ausgänge hat, und Delay $\Omega(\log n)$. Ein Addierer, der dies bei kleinen konstanten Faktoren erreicht, ist der Carry–Lookahead Addierer. Dieser berechnet die Summe $(s_0, \ldots, s_n)$ aus zwei Zahlen $(a_0, \ldots, a_{n-1})$ und $(b_0, \ldots, b_{n-1})$ mit $s_i = a_i \oplus b_i \oplus c_{i-1}$ für $0 \le i < n$ und $s_n = c_{n-1}$. c_{-1} ist der Carry in, c_i für $0 \le i < n$ ist der Übertrag, der von Stelle i erzeugt wird. Diese Überträge werden durch eine Schaltung logarithmischer Tiefe erzeugt. Diese Schaltung besteht aus zwei hintereinandergeschalteten binären Bäumen mit einer Gesamttiefe von $2\log n - 1$ inneren Knoten plus Blättern. Die Anzahl der inneren Knoten ist $2n - \log n - 2$, die Anzahl der Blätter ist n. Die inneren Knoten dieser Bäume bestehen aus zwei `AND` und einem `OR` Gatter, die Tiefe der Knoten ist $d(\texttt{AND}) + d(\texttt{OR})$. Die Blätter des ersten Baumes bestehen aus einem `AND` und einem `OR` Gatter, die Tiefe ist $\max(d(\texttt{AND}), d(\texttt{OR}))$.

Die Basiskosten und Verzögerung des Addierers betragen damit

$$\begin{aligned} c(\texttt{ADD}_n) &= 2nc(\texttt{EXOR}) + (5n - 2\log n - 4)c(\texttt{AND}) + (3n - \log n - 2)c(\texttt{OR}) \\ &= 28n - 6\log n - 12 && (3.3)\\ d(\texttt{ADD}_n) &= 2\log n d(\texttt{AND}) + (2\log n - 1)d(\texttt{OR}) + d(\texttt{EXOR}) = 4\log n + 1 && (3.4) \end{aligned}$$

3.3.3 Subtrahierer

Ein Subtrahierer besteht aus einem Addierer, bei dem das Vorzeichen des zweiten Arguments umgekehrt wird. Bei Zahlendarstellung im Zweier Komplement erfolgt dies durch bitweises Invertieren und Addition einer 1, die als Carry in übergeben wird.

Ein Schaltkreis sowohl zum Addieren als auch zum Subtrahieren besteht dann aus einem Addierer mit einem zusätzlichen Eingang s, der angibt, ob addiert werden soll ($s = 0$) oder subtrahiert ($s = 1$). Der Carry in des Addierers ergibt sich dann aus dem exklusiven Oder des Carry in des Schaltkreises und s, der zweite Eingang des Addierers ergibt sich aus dem bitweisen exklusiven Oder des zweiten Eingangs des Schaltkreises und s.

Ein Addierer/Subtrahierer für n–bit Zahlen hat dann folgende Kosten und Verzögerung:

$$\mathrm{c}(\texttt{ADDSUB}_n) = \mathrm{c}(\texttt{ADD}_n) + (n+1)\mathrm{c}(\texttt{EXOR}) = 34n - 6\log n - 6 \tag{3.5}$$

$$\mathrm{d}(\texttt{ADDSUB}_n) = \mathrm{d}(\texttt{ADD}_n) + \mathrm{d}(\texttt{EXOR}) = 4\log n + 3 \tag{3.6}$$

3.3.4 Inkrementierer

Definition 3.5 *Ein n–bit* Inkrementierer *ist ein Schaltkreis mit n–bit breitem Eingang und n–bit breitem Ausgang, der folgende Schaltfunktion berechnet: liegt am Eingang die n–stellige Binärdarstellung einer Zahl $x \in \{0, \ldots, 2^n - 1\}$ an, so erscheint am Ausgang die n–stellige Binärdarstellung der Zahl $x + 1 \bmod 2^n$.*

Der Inkrementierer ist also ein Spezialfall eines Addierers mit $y = 0$ und $c_{in} = 1$. Hieraus resultieren einige Vereinfachungen, da ein Teil der Gatter nicht benötigt wird. Die Berechnung der Summe vereinfacht sich zu $s_i = a_i \oplus c_{i-1}$. Die Blätter des ersten Baumes entfallen.

Die Basiskosten und die Verzögerung eines Inkrementierers betragen damit

$$\begin{aligned}\mathrm{c}(\texttt{INC}_n) &= n\mathrm{c}(\texttt{EXOR}) + (4n - 2\log n - 4)\mathrm{c}(\texttt{AND}) + (2n - \log n - 2)\mathrm{c}(\texttt{OR}) \\ &= 18n - 6\log n - 12 \end{aligned} \tag{3.7}$$

$$\mathrm{d}(\texttt{INC}_n) = (2\log n - 1)\mathrm{d}(\texttt{AND}) + (2\log n - 1)\mathrm{d}(\texttt{OR}) + \mathrm{d}(\texttt{EXOR}) = 4\log n \tag{3.8}$$

3.3.5 Vergleicher

Bei Vergleichern unterscheidet man Untersuchung auf Identität und Untersuchung auf Vorzeichen.

Definition 3.6 *Ein n–*bit Vergleicher auf Identität *ist ein Schaltkreis mit zwei n–bit breiten Eingängen und einem Ausgang, der folgende Schaltfunktion berechnet: Liegen am Eingang die n–stelligen Binärdarstellungen zweier Zahlen $x, y \in \{0, \ldots, 2^n - 1\}$ an, so hat der Ausgang v den Wert* 1 *genau dann wenn $x = y$, sonst 0.*

Ein Vergleicher auf Identität läßt sich durch folgende Schaltfunktion realisieren:

$$v = \overline{(a_0 \oplus b_0) \vee \ldots \vee (a_{n-1} \oplus b_{n-1})}$$

Realisiert man die ODER Verknüpfung als balancierten binären Baum, so erhält man folgende Basiskosten und Verzögerung:

$$\mathrm{c}(\mathtt{IDENT}_n) = n\mathrm{c}(\mathtt{EXOR}) + (n-2)\mathrm{c}(\mathtt{OR}) + \mathrm{c}(\mathtt{NOR}) = 8n - 2 \quad (3.9)$$

$$\mathrm{d}(\mathtt{IDENT}_n) = \mathrm{d}(\mathtt{NOR}) + (\log n - 1)\mathrm{d}(\mathtt{OR}) + \mathrm{d}(\mathtt{EXOR}) = \log n + 2 \quad (3.10)$$

Definition 3.7 *Ein n–*bit Vergleicher auf Vorzeichen *ist ein Schaltkreis mit zwei n–bit breiten Eingängen und einem Ausgang, der folgende Schaltfunktion berechnet: Liegen am Eingang die n–stelligen Binärdarstellungen zweier Zahlen $x, y \in \{0, \ldots, 2^n - 1\}$ an, so hat der Ausgang v den Wert* 1 *genau dann wenn $x < y$, sonst 0.*

$x < y$ ist äquivalent zu $x - y < 0$. Ein Vergleicher auf Vorzeichen besteht also prinzipiell aus einem Subtrahierer, bei dem vom Resultat allerdings nur der Ausgangsübertrag benutzt wird. Ein Subtrahierer für Zahlen in Zweier Komplement Darstellung besteht aus einem Addierer, bei dem die zweite Eingabe bitweise negiert wird, der Eingangsübertrag hat den Wert 1. Als Addierer benutzt man den in Abschnitt 3.3.2 eingeführten Carry Lookahead Addierer. Von ihm wird allerdings nur der erste Baum samt Blättern benötigt.

Basiskosten und Verzögerung eines Vergleichers auf Vorzeichen ergeben sich zu

$$\mathrm{c}(\mathtt{VGL}_n) = n\mathrm{c}(\mathtt{NOT}) + (3n-2)\mathrm{c}(\mathtt{AND}) + (2n-1)\mathrm{c}(\mathtt{OR}) = 11n - 6 \quad (3.11)$$

$$\mathrm{d}(\mathtt{VGL}_n) = \mathrm{d}(\mathtt{NOT}) + \mathrm{d}(\mathtt{AND}) + \log n \cdot (\mathrm{d}(\mathtt{OR}) + \mathrm{d}(\mathtt{AND})) = 2\log n + 2 \quad (3.12)$$

3.3.6 Multiplizierer

Definition 3.8 *Ein (i, j)* Multiplizierer *ist ein Schaltkreis mit zwei i–bit breiten Eingängen und einem $(j+1)$–bit breiten Ausgang. Liegen an den Eingängen die i–stelligen Binärdarstellungen zweier Zahlen $a, b \in \{0, \ldots, 2^i - 1\}$ an, so liegt am Ausgang die $(j+1)$–stellige Binärdarstellung der Zahl $a \cdot b$ an, falls $a \cdot b < 2^j$, $(1, *, \ldots, *)$ sonst.*

Anmerkung: Das Symbol $*$ gibt an, daß das betreffende Bit entweder den Wert 0 oder den Wert 1 haben kann.

Ein Multiplizierer geringer Tiefe ist der Wallace–Tree. Dieser berechnet für Zahlen a, b mit Binärdarstellungen $a_{i-1}, \ldots, a_0$ und $b_{i-1}, \ldots, b_0$ zuerst $2i$–stellige Darstellungen $c^{(l)}$, $l \in \{0, \ldots, i-1\}$, die folgendermaßen definiert sind:

$$c^{(l)} = \begin{cases} (0, \ldots, 0) & : \quad b_l = 0 \\ (0, \ldots, 0, a_{i-1}, \ldots, a_0, \underbrace{0, \ldots, 0}_{l}) & : \quad b_l = 1 \end{cases}$$

Die Zahlen, die durch die $c^{(l)}$ dargestellt werden, entsprechen den Multiplikation der Zahl a mit je einer (Binär)Stelle der Zahl b. Um das Produkt zu erhalten, müssen die Teilergebnisse addiert werden. Hierzu benutzt man die folgenden Addierer:

Definition 3.9 *Ein* Volladdierer *(engl. Fulladder, FA) ist ein Schaltkreis mit Eingängen x, y, c_{in} und Ausgängen s, c_{out}, der folgende Schaltfunktion berechnet:*

$$\begin{aligned} s &= x \oplus y \oplus c_{in} \\ c_{out} &= xy \vee (x \vee y)c_{in} \end{aligned}$$

Ein 3–2–Addierer *für n–bit Zahlen ($\texttt{ADD}_{n,3,2}$) hat drei n–bit breite Eingänge und zwei n–bit breite Ausgänge. Liegen an den Eingängen die n–stelligen Binärdarstellungen der Zahlen $a, b, c \in \{0, \ldots, 2^n - 1\}$ an, so liegen an den Ausgängen die Binärdarstellungen von zwei Zahlen $d, e \in \{0, \ldots, 2^n - 1\}$ an, so daß gilt: $a + b + c = d + 2e$.*

Ein 4–2–Addierer *für n–bit Zahlen ($\texttt{ADD}_{n,4,2}$) hat vier n–bit breite Eingänge und zwei $(n+1)$–bit breite Ausgänge. Liegen an den Eingängen die n–stelligen Binärdarstellungen der Zahlen $a, b, c, d \in \{0, \ldots, 2^n - 1\}$ an, so liegen an den Ausgängen die $(n+1)$–stelligen Binärdarstellungen von zwei Zahlen $e, f \in \{0, \ldots, 2^{n+1} - 1\}$ an, so daß gilt: $a + b + c + d = e + f$.*

Die Schaltkreise für Volladdierer, 3–2–Addierer und 4–2–Addierer sind in Abbildung 3.3 zu sehen.

Damit ergeben sich die folgenden Kosten und Verzögerungen:

$$\begin{aligned} \mathrm{c}(\texttt{FA}) &= 2\mathrm{c}(\texttt{EXOR}) + 2\mathrm{c}(\texttt{AND}) + 2\mathrm{c}(\texttt{OR}) = 20 && (3.13) \\ \mathrm{d}(\texttt{FA}) &= \max(2\mathrm{d}(\texttt{EXOR}), 2\mathrm{d}(\texttt{OR}) + \mathrm{d}(\texttt{AND})) = 4 && (3.14) \\ \mathrm{c}(\texttt{ADD}_{n,3,2}) &= n\mathrm{c}(\texttt{FA}) = 20n && (3.15) \end{aligned}$$

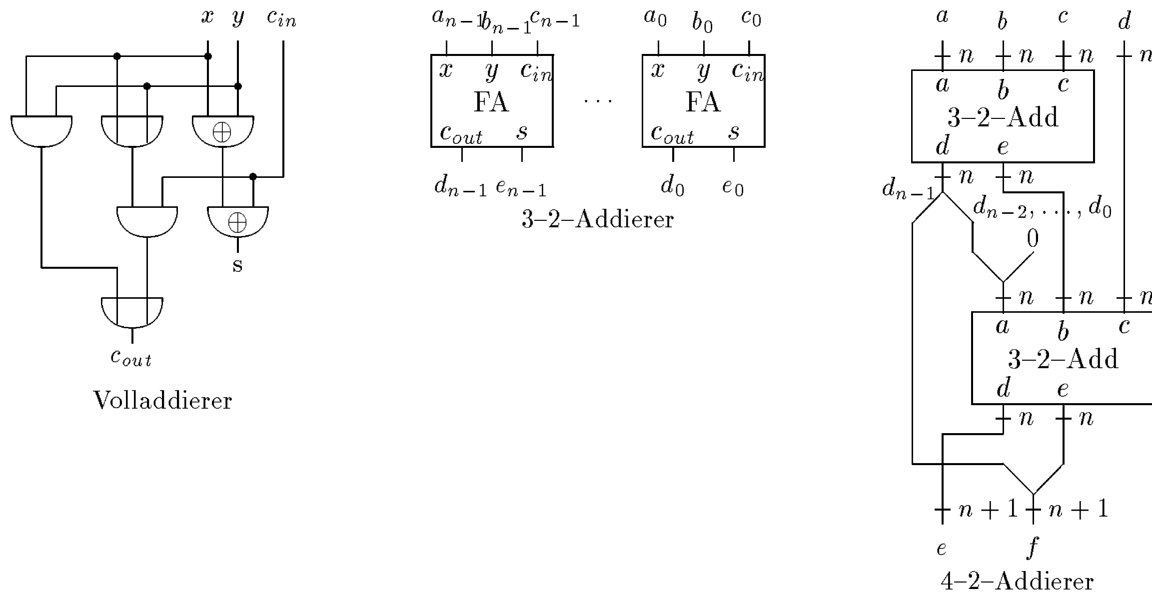

Abbildung 3.3: Volladdierer, 3–2–Addierer, 4–2–Addierer

$$\mathrm{d}(\mathtt{ADD}_{n,3,2}) = \mathrm{d}(\mathtt{FA}) = 4 \tag{3.16}$$

$$\mathrm{c}(\mathtt{ADD}_{n,4,2}) = 2\mathrm{c}(\mathtt{ADD}_{n,3,2}) = 40n \tag{3.17}$$

$$\mathrm{d}(\mathtt{ADD}_{n,4,2}) = 2\mathrm{d}(\mathtt{ADD}_{n,3,2}) = 8 \tag{3.18}$$

Die Teilergebnisse können nun in einem binären Baum von 4–2–Addierern bis auf zwei Ergebnisse zusammengefaßt werden, die abschließend in einem Carry–Lookahead Addierer zum Endergebnis addiert werden. Dieser Baum hat Tiefe $\lceil \log_2 i - 1 \rceil$. Die Breite der 4–2–Addierer beträgt stets $2i$, das oberste Bit der beiden Ergebnisse kann nie gesetzt sein.

Für $j < 2i$ muß nun überprüft werden, ob eines der Bits $j, \ldots, 2i-1$ den Wert 1 hat. Wenn ja, dann muß das Ergebniss auf $(1, 0, \ldots, 0)$ gesetzt werden. Die Überprüfung erfolgt durch einen binären Baum aus `OR` Gattern der Tiefe $\lceil \log(2i - j) \rceil$. Um Kosten zu sparen, werden allerdings auf allen Bits, die nur zu diesem Baum führen, keine Berechnungen mehr durchgeführt. Weiterhin werden alle Addierer eliminiert, bei denen alle Eingänge stets 0 sind.

Für den Fall $j = i$, den wir im weiteren annehmen, ergibt sich dann folgender Aufbau: Von $c^{(l)}, \ldots, c^{(l+3)}$ werden die Bits l bis $i-1$ addiert, $l \in \{0, 4, \ldots, i-4\}$ (wir nehmen an, daß i durch 4 teilbar ist). Die Bits $i, \ldots, l+i-1$ der $c^{(l)}$ und die $(i+1)$–ten Bits der Ausgänge der 4–2–Addierer werden direkt zum ODER Baum geführt. In den folgenden Stufen des Addiererbaumes wird analog verfahren. Am Ende werden die beiden letzten Ergebnisse durch einen Carry–Lookahead Addierer der Breite i addiert.

Man benötigt damit in der r–ten Stufe des Addiererbaumes, $0 \leq r < \log i - 1$, 4–2–Addierer der Breiten $i, i - 2^{r+2}, \ldots, 2^{r+2}$. Damit hat man in der r–ten Stufe 4–2–Addierer,

deren Breiten zusammen $W_r = \sum_{j=1}^{i/2^{r+2}} 2^{r+2} \cdot j$ ergeben. Die Breiten aller 4–2–Addierer im Addiererbaum summieren sich dann zu $\sum_{r=0}^{\log i-2} W_r = 0.25 \cdot i^2 - i + 0.5 \cdot i \log i$.

Der ODER Baum hat $\sum_{t=1}^{i-1} t$ Eingänge von den $c^{(l)}$, einen Eingang vom Carry–Lookahead Addierer und von jedem der 4–2–Addierer zwei Stück. Die Anzahl der 4–2–Addierer in Stufe r beträgt $i/2^{r+2}$. Insgesamt hat der Baum dann $0.5 \cdot i^2 + 0.5 \cdot i - 1$ Eingänge.

Damit ergeben sich folgende Kosten und Verzögerungen:

$$\begin{aligned} c(\texttt{MULT}_{i,i}) &= i^2 \cdot c(\texttt{AND}) + c(\texttt{ADD}_i) + (\frac{i^2}{2} + \frac{i}{2} \cdot c(\texttt{OR}) + (\frac{i^2}{4} - i + \frac{i}{2}\log i) \cdot 2 \cdot c(\texttt{FA}) \\ &= 13i^2 - 11i + 20i\log i - 16 \qquad (3.19) \\ d(\texttt{MULT}_{i,i}) &= d(\texttt{AND}) + (\log i - 1)d(\texttt{ADD}_{n,4,2}) + d(\texttt{ADD}_i) \\ &\quad + \log(i^2 - (3i)/2 + 1)d(\texttt{OR}) \\ &< 14\log i - 6 \qquad (3.20) \end{aligned}$$

Anmerkung: In [Weg87] wird die Konstruktion des Wallace Tree Multiplizierers mit 3–2–Addierern beschrieben, was zu einer unregelmäßigen Struktur führt. Die Konstruktion mit 4–2–Addierern entstammt [Vui83].

3.3.7 Rotierer und Shifter

Definition 3.10 *Ein* Rotierer *für i–bit Zahlen ist ein Schaltkreis mit einem i–bit, einem* $\log i$*–bit und einem* 1*–bit Eingang und einem i–bit breiten Ausgang. Liegt am ersten Eingang* $(a_{i-1}, \ldots, a_0)$ *an, am zweiten Eingang die Binärdarstellung* $(b_{\log i-1}, \ldots, b_0)$ *einer Zahl* $b \in \{0, \ldots, i-1\}$ *und am dritten Eingang s, so liegt im Fall* $s = 0$ *am Ausgang* $(a_{i-b-1}, \ldots, a_0, ai - 1, \ldots, a_{i-b})$ *an, (Linksrotation),* $(a_{b+1}, \ldots, a_0, a_{i-1}, \ldots, a_b)$*, falls* $s = 1$ *(Rechtsrotation).*

Ein logischer Shifter *für i–bit Zahlen unterscheidet sich vom Rotierer im Ausgangsverhalten nur dahingehend, daß am Ausgang* $(a_{i-b-1}, \ldots, a_0, \underbrace{0, \ldots, 0}_{b})$ *anliegt, falls* $s = 0$ *(Linksshift), falls* $s = 1$*, liegt* $(\underbrace{0, \ldots, 0}_{b}, a_{i-1}, \ldots, a_b)$ *am Ausgang an (logischer Rechtsshift).*

Ein arithmethischer Shifter *für i–bit Zahlen unterscheidet sich vom logischen Shifter im Ausgangsverhalten nur dahingehend, daß am Ausgang* $(\underbrace{a_{i-1}, \ldots, a_{i-1}}_{b}, a_{i-1}, \ldots, a_b)$ *statt* $(\underbrace{0, \ldots, 0}_{b}, a_{i-1}, \ldots, a_b)$*, falls* $s = 1$ *(arithmethischer Rechtsshift). Für* $s = 0$ *unterscheiden sie sich nicht.*

Ein Schaltkreis zur Realisierung eines Rotierers benutzt Trimuxe. Dies sind Multiplexer mit 3 Eingängen und einem Ausgang. Man gewinnt sie durch eine Schaltung ähnlich der

Hintereinanderschaltung von 2 Multiplexern. Im ersten „Mux“ wird mittels eines ersten Select Signals s_1 zwischen erstem ($s_1 = 1$) und drittem Eingang gewählt, im zweiten „Mux“ wird mittels eines zweiten Select Signals s_2 zwischen dem ersten Ergebnis ($s_2 = 1$) und dem zweiten Eingang gewählt. Für Eingänge i_1, i_2, i_3 berechnet sich Ausgang o nun folgendermaßen:

$$o = \bar{s_1} \cdot \bar{s_2} \cdot i_1 \vee s_1 \cdot \bar{s_2} \cdot i_2 \vee s_2 \cdot i_3$$

Berechnet man $\bar{s_2}$ nur einmal, so erhält man folgende Kosten und, bei vorgegebenen Select Signalen, folgende Verzögerung:

$$\mathrm{c}(\texttt{TRIMUX}) = 5\mathrm{c}(\texttt{AND}) + 2\mathrm{c}(\texttt{OR}) + 2\mathrm{c}(\texttt{INV}) = 16 \quad (3.21)$$

$$\mathrm{d}(\texttt{TRIMUX}) = \mathrm{d}(\texttt{AND}) + 2\mathrm{d}(\texttt{OR}) = 3 \quad (3.22)$$

Ein Rotierer besteht aus $\log i$ Stufen zu je i Trimuxen. Trimux $j \in \{0, \ldots, i-1\}$ in Stufe $k \in \{1, \ldots, \log i\}$ wird mit $\langle k, j \rangle$ bezeichnet. Eingang a_j ist mit dem zweiten Eingang von $\langle 1, j \rangle$, mit dem dritten Eingang von $\langle 1, (j+1) \bmod i \rangle$ und mit dem ersten Eingang von $\langle 1, (j-1) \bmod i \rangle$ verbunden. Der Ausgang von $\langle k, j \rangle$, $k < \log i$, ist mit dem zweiten Eingang von $\langle k+1, j \rangle$, mit dem dritten Eingang von $\left\langle k+1, (j+2^k) \bmod i \right\rangle$ und mit dem ersten Eingang von $\left\langle k+1, (j-2^k) \bmod i \right\rangle$ verbunden. Die Ausgänge der letzten Stufe sind die Ausgänge des Schaltkreises. Select Signal s_2 der Trimuxe in Stufe k ist mit Eingang b_{k-1} verbunden. Select Signal s_1 aller Trimuxe ist mit Eingang s verbunden. Daher braucht auch der entsprechende Inverter für alle Trimuxe nur einmal vorhanden zu sein und die Kosten der Trimuxe reduzieren sich zu 15.

Ein Rotierer kann zusätzlich die Funktion eines logischen Shifters übernehmen, wenn er einen Eingang t erhält, der angibt, ob rotiert ($t = 1$) werden soll oder geshiftet ($t = 0$). Man baut dazu in alle Verbindungen, bei denen $j + 2^k \geq i$ oder $j - 2^k < 0$ ist, ein AND Gatter ein. Der freie Eingang dieser Gatter wird mit t verbunden. Abbildung 3.4 zeigt einen Rotierer/Shifter für 4–bit Zahlen.

Ein solcher Rotierer/Shifter kann zusätzlich die Funktion eines arithmethischen Rechtsshifters übernehmen, wenn er einen zusätzlichen Eingang u erhält, der angibt, ob bei ein Rechtshift logisch ($u = 0$) oder arithmetisch ($u = 1$) sein soll. Hierzu baut man bei allen Verbindungen mit $j - 2^k < 0$ anstatt des AND Gatters folgende Schaltung ein: Wird vom Ausgang des Trimuxes am Anfang der Verbindung ein Signal l geliefert, so wird das Signal $t \cdot l \vee \bar{t} \cdot u \cdot a_{i-1}$ geliefert.

Damit ergeben sich folgende Kosten und Verzögerung:

$$\mathrm{c}(\texttt{SHIFT}_i) = i \log i \cdot (\mathrm{c}(\texttt{TRIMUX}) - 1) + (i-1) \cdot \mathrm{c}(\texttt{AND}) + (i-1)(3\mathrm{c}(\texttt{AND}) + \mathrm{c}(\texttt{OR}))$$

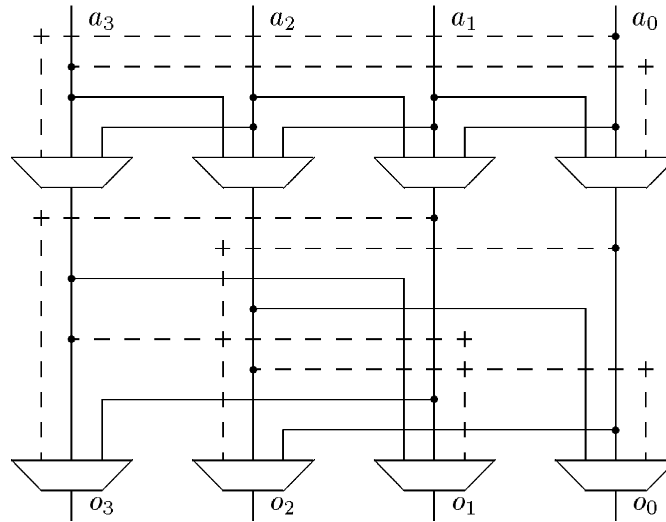

Abbildung 3.4: Rotierer und logischer Shifter für 4–bit Zahlen. Die gestrichelten Linien sind die Verbindungen, an denen AND Gatter eingefügt werden müssen.

$$
\begin{aligned}
&= 15i\log i + 10i - 10 && (3.23)\\
\mathrm{d}(\mathtt{SHIFT}_i) &= \log i\cdot \mathrm{d}(\mathtt{TRIMUX}) + \log i\cdot(\mathrm{d}(\mathtt{AND}) + \mathrm{d}(\mathtt{OR})) &&\\
&= 5\log i && (3.24)
\end{aligned}
$$

3.3.8 FIFO–Schlange

Definition 3.11 *Eine* FIFO–Schlange der Länge k und Breite n *ist ein Schaltwerk mit je einem n–bit breiten Ein– und Ausgang und einer Eingangs– und Ausgangsclock. Das Schaltwerk kann maximal k Worte, die n–bit breit sind, speichern. Enthält das Schaltwerk weniger als k Worte und ist die Eingangsclock während eines Zyklus aktiv, so wird die Belegung des Eingangs als neues Wort aufgenommen. Am Ausgang liegt stets das Wort an, das sich am längsten im Schaltwerk befindet. Ist die Ausgangsclock in einem Zyklus aktiv und enthält das Schaltwerk mindestens ein Wort, so wird das Wort entfernt, das sich am längsten im Schaltwerk befindet.*

Man realisiert eine FIFO–Schlange durch die in Abbildung 3.5 dargestellte Hardware. Es ergeben sich folgende Basiskosten

$$
\begin{aligned}
\mathrm{c}(n, k-\mathtt{FIFO}) &= 3\mathrm{c}(\mathtt{INV}) + 4\mathrm{c}(\mathtt{AND}) + \mathrm{c}(\mathtt{NAND}) + \mathrm{c}(\mathrm{REG}_2) + 2\mathrm{c}(\mathrm{REG}_{\log k}) &&\\
&\quad +2\mathrm{c}(\mathtt{INC}_{\log k}) + \mathrm{c}(\mathtt{IDENT}_{\log k}) + \mathrm{c}(k\times n\text{–Dual Port RAM}) &&\\
&= 24kn + 68\log k - 12\log\log k + 11 && (3.25)
\end{aligned}
$$

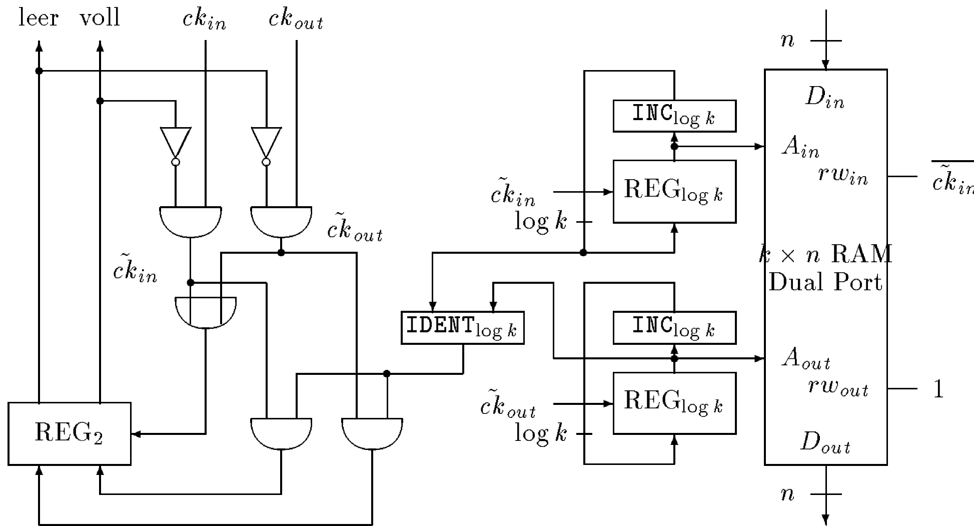

Abbildung 3.5: Aufbau einer FIFO Schlange

Im Gegensatz zu allen bisher vorgestellten Bausteinen besteht die FIFO–Schlange nicht nur aus einem arithmetischen Teil, sondern aus einem Logikteil (in der Zeichnung links), der die Gatter und das 2–Register umfaßt, einem Arithmetikteil (in der Mitte der Zeichnung), der die Adressregister, Inkrementierer und den Vergleicher umfaßt, und einem Speicherteil, der das Dual Port RAM umfaßt. Die Basiskosten werden gemäß dieser Aufteilung mit verschiedenen Packungsfaktoren gewichtet, die Gesamtkosten ergeben sich zu

$$37 + 0.75 \cdot (68 \log k - 12 \log \log k - 26) + 0.45 \cdot 24kn = 10.8kn + 51 \log k - 9 \log \log k + 17.5$$

Die Verzögerung ck_{out} nach D_{out} ergibt sich aus den Verzögerungen des `AND` Gatters, des Adressregisters und der Zugriffszeit des RAMs. Die minimale Zeit bis zum nächsten Zyklus ergibt sich aus der Verzögerung des Adressregisters und des Inkrementers.

$$d(ck_{out} \rightarrow D_{out}) \quad = \quad d(\mathrm{REG}_{\log k}) + d(k \times n\text{–RAM}) = 4 + 1.5 \log k \tag{3.26}$$

3.4 Bewertung der Fluent Machine

Um die in Abschnitt 2.3 eingeführte Maschine zu bewerten, müssen zunächst für Prozessoren und Netzwerkknoten Schaltkreise und Schaltwerke angegeben werden. Für den Netzwerkknoten ergibt sich dies sehr genau aus der Beschreibung seiner Funktionsweise. Über den Prozessor wird in [RBJ88] nichts Näheres ausgesagt. Es wird lediglich die Verwendung eines RISC Prozessors vorgeschlagen. Deshalb wird im nächsten Abschnitt ein einfacher RISC

Befehlssatz vorgestellt, der aus dem Berkeley RISC Prozessor [PS82] entwickelt wurde. Für diesen Befehlssatz wird eine einfache Hardware angegeben. Im übernächsten Abschnitt wird eine Hardware für den Netzwerkknoten angegeben, im letzten Abschnitt wird die Fluent Machine bewertet.

Es wird sich zeigen, daß Kosten und Verzögerung der Maschine abhängig sind von $\log m$ und n. Wir werden annehmen, daß die Größe des Speichers höchstens polynomiell in der Anzahl der Prozessoren ist, das heißt $m = N^a$ mit $a > 1$. Damit ist $\log m = O(n)$. Um die Maschine realistisch zu machen, wählen wir $m = 2^{32}$. Damit benutzen wir im Prozessor 32 bit breite Register und Datenpfade und wir beschränken $\log m = 32$ und $n < 32$.

3.4.1 Maschinensprache und Prozessor

Der verwendete Prozessor benutzt 32 Register der Breite $\log m$ Bit. Die Register 0 bis 3 haben spezielle Bedeutungen. Register 0 hat stets den Wert 0, ein ihm zugewiesener Wert geht verloren. Register 1 enthält den Programmzähler (PC), Register 2 und 3 enthalten Stackzeiger (SP_0 und SP_1).

Im folgenden steht R_x für Register x, S_2 bezeichnet entweder ein Register oder eine ($\log m-18$)–bit Konstante in 2er Komplement Darstellung. Es findet Vorzeichenerweiterung statt. Y bezeichnet eine 19–bit Konstante. Bei einer Zuweisung werden damit die Bits $\log m - 18$ bis $\log m - 1$ des Zielregisters besetzt, die Bits 0 bis $\log m - 19$ des Zielregisters erhalten den Wert 0. Zusätzlich enthält der Prozessor ein Statusregister (ST), das Flags für Null (Z), Negativ (N), Überlauf (O), Übertrag (C). $S(Ausdruck)$ bezeichnet den Inhalt der Speicherzelle mit Adresse $Ausdruck$.

Alle Instruktionen sind in $\log m$–bit Worten kodiert. Das Format ist einheitlich für alle Befehle außer LDHI. Das Format ist ähnlich dem in [PS82]. Als einziger Datentyp wird ein $\log m$–bit Wort (Long) verwendet. Eine Aufstellung des Befehlssatzes ist in Tabelle 3.4 zu sehen.

Gültige Bedingungen im JMP Befehl sind $flag$ und $nichtflag$, wobei $flag \in \{C, N, Z, O\}$.

Die Datenpfade einer Hardware zur Ausführung dieses Befehlssatzes sind in Abbildung 3.6 zu sehen. Sie arbeitet wie im Folgenden beschrieben. Wir werden lediglich, um eine einheitliche Taktlänge zu erhalten, am Ende die einzelnen Takte in mehrere zerteilen, was allerdings den Ablauf nicht beeinflußt.

- Im 1. Takt einer Befehlsausführung wird der neue Befehl geladen. Da der Ausgang des PC stets auf dem Adressbus zum Programmspeicher liegt, muß lediglich am Ende des Taktes der neue Befehl ins Befehlsregister *(Instruction Register, IR)* geclockt werden. Die Länge des Taktes wird durch die Zugriffszeit des Programmspeichers bestimmt.

- Im 2. Takt erfolgen abhängig vom gerade geladenen Befehl verschiedene Dinge:

LOAD Befehle		
LD	R_x,S_2,R_d	R_d:= S($R_x + S_2$)
LDHI	R_d, Y	$R_d\langle 13:31\rangle := Y$, $R_d\langle 0:12\rangle := 0$
GETPSW	R_d	$R_d := ST$
LDI	R_d	$R_d :=$ Prozessor Nummer
STORE Befehle		
ST	R_x, S_2, R_m	$S(R_x + S_2) := R_m$
PUTPSW	R_x	$ST := R_x$
COMPUTE Befehle		
ADD	R_x, S_2, R_d	$R_d := R_x + S_2$
ADC	R_x, S_2, R_d	$R_d := R_x + S_2 + carry$
SUB	R_x, S_2, R_d	$R_d := R_x - S_2$
SBC	R_x, S_2, R_d	$R_d := R_x - S_2 - carry$
AND	R_x, S_2, R_d	$R_d := R_x \wedge S_2$
OR	R_x, S_2, R_d	$R_d := R_x \vee S_2$
NAND	R_x, S_2, R_d	$R_d := R_x \overline{\wedge} S_2$
XOR	R_x, S_2, R_d	$R_d := R_x \oplus S_2$
ASL	R_x, S_2, R_d	$R_d := R_x$ shift links arithm. um S_2 mod $\log m$ Bits
ASR	R_x, S_2, R_d	$R_d := R_x$ shift rechts arithm. um S_2 mod $\log m$ Bits
LSL	R_x, S_2, R_d	$R_d := R_x$ shift links logisch um ...
LSR	R_x, S_2, R_d	$R_d := R_x$ shift rechts logisch um ...
WRL	R_x, S_2, R_d	$R_d := R_x$ rotiert links um S_2 mod $\log m$ Bits
WRR	R_x, S_2, R_d	$R_d := R_x$ rotiert rechts um ...
MUL	R_x, S_2, R_d	$R_d := R_x \cdot S_2$
SPRUNG und STACK Befehle ($i \in \{0,1\}$)		
JSRi	R_x, S_2	$S(SP_i++) := PC$, $PC := R_x + S_2$
PSHi	R_x	$S(SP_i++) := R_x$
POPi	R_d	$R_d := S(--SP_i)$
JMP	Bed.,R_x, S_2	Falls (Bedingung ist erfüllt) $PC := R_x + S_2$

Tabelle 3.4: Befehlssatz

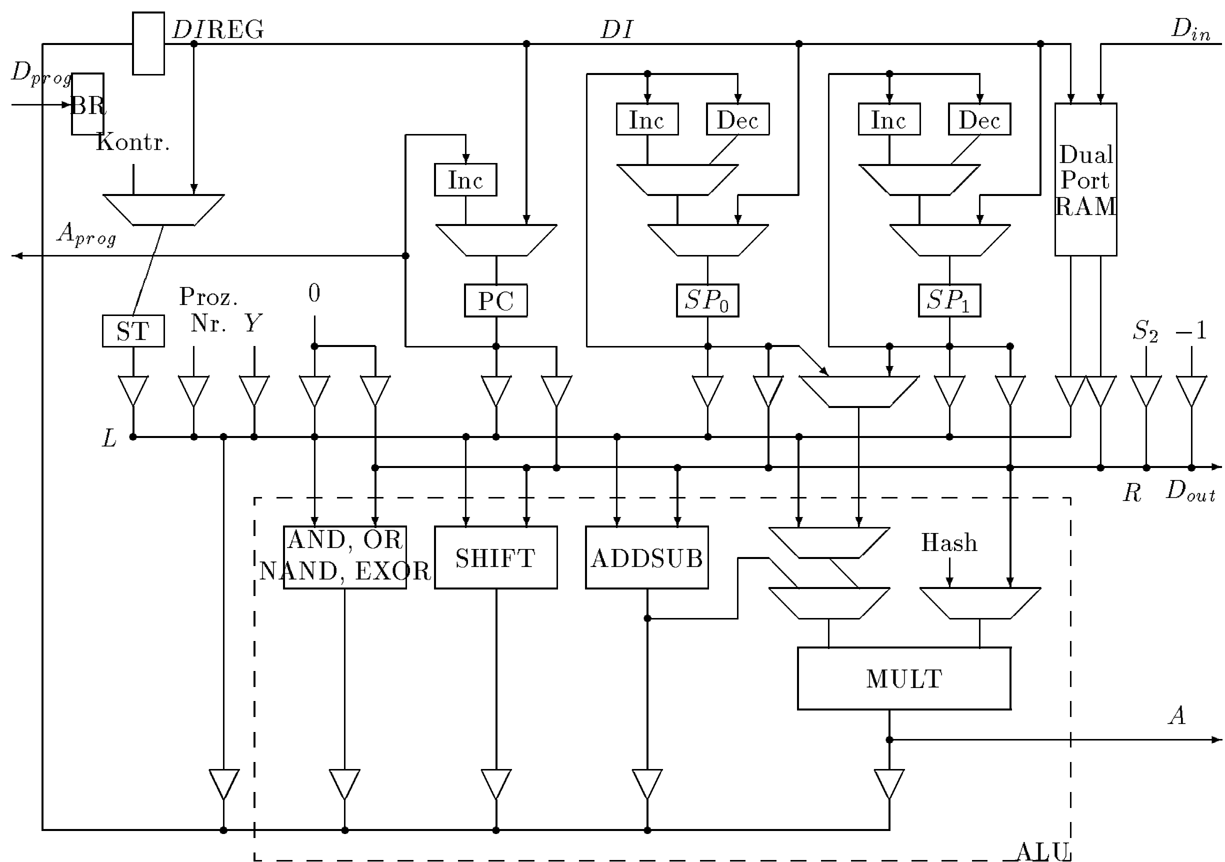

Abbildung 3.6: Datenpfade des Prozessors

- Bei Befehlen LD und ST werden die beiden Argumente R_x und S_2 selektiert (aus dem Dual Port RAM, den 3 anderen Registern, oder direkt als Konstante) und auf L und R Bus weitergeleitet. Die Ansteuerung der Adressen des Dual Port RAM erfolgt durch die Kontrollogik. Bei PSHi (POPi) wird SP_i auf den L Bus und 0 (-1) auf den R Bus geleitet. Der Addierer summiert die Adresse und leitet sie zum Multiplizierer weiter. Am anderen Eingang des Multiplizierers wird der Faktor a der linearen Hash Funktion selektiert, das Multiplikationsergebnis wird zum Adressbus A gegeben. Der Faktor a befindet sich in einem Register der Kontrollogik. Dort wird er einmal beim Start der Maschine eingespeist.
- Beim Befehl JSRi wird der Inhalt von SP_i direkt am Multiplizierer zusammen mit dem Faktor a selektiert und am Ende des Taktes das Ergebnis der Multiplikation in das Register zum Adressbus geclockt. Zusätzlich werden R_x und S_2 selektiert, auf L und R Bus geleitet, addiert, auf den DI Bus geleitet und am Ende des Taktes ins DI Register geclockt.
- Bei allen Compute Befehlen und bei JMP werden R_x und S_2 selektiert, auf L und R Bus geleitet, in der ALU verarbeitet, das Ergebnis auf den DI Bus gelegt und am Ende des Taktes in das DI Register geclockt.
- Beim Befehl LDHI (LDI) wird Y (die Prozessornummer) auf den DI Bus geleitet und am Ende des Taktes ins DI Register geclockt.
- Beim Befehl GETPSW (PUTPSW) wird ST (R_x) selektiert, auf den L Bus geleitet und am Ende des Taktes ins DI Register geclockt.

Die Länge des Taktes wird von den Verzögerungen von Addierer und Multiplizierer bestimmt.

- Im 3. Takt wird bei ST R_m, bei PSH R_x, bei JSR der PC selektiert und über den R Bus auf den D_{out} Bus gelegt. Am Ende des Taktes werden dann Datum und Adresse von D_{out} und A in den Eingangspuffer des Netzwerkes geclockt. Bei Compute, LDHI, LDI, GETPSW, PUTPSW, JSR (und bei auszuführenden JMP) Befehlen wird der Wert des DI Registers am Ende des Taktes in das Zielregister geclockt. Bei Compute Operationen wird zusätzlich ST am Ende des Taktes geclockt. Bei allen Operationen, die nicht den PC als Zielregister haben, wird der inkrementierte PC geclockt. Bei PSHi und JSRi (POPi) wird zusätzlich der inkrementierte (dekrementierte) SP_i geclockt.

- Nach $x \cdot n$ weiteren Takten kehrt bei LD und POPi Befehlen die Antwort einer LOAD Anfrage aus dem Netzwerk zurück und wird über den Bus D_{in} am Ende des Taktes im Zielregister abgespeichert. Die Verzögerung von $x \cdot n$ Takten rührt vom Netzwerk her. In [RBJ88] wird x mit 11 angegeben.

Der Anschluß des Prozessors ans Netzwerk besteht darin, daß die Werte von Adressbus A, Datenbus D_{out} und einigen Signalen der Kontrollogik zum Steuern des Modus (LOAD, STORE) ein Paket bilden. Diese Leitungen werden mit einem Netzwerkknoten verbunden wie in Abschnitt 3.4.2 beschrieben. Der Anschluß des Rücknetzwerkes an den Prozessor

Arithmethik	18528.5
RAM	11059.2
Reg., Treiber, Logik	7808.0
Gesamt	37395.7

Tabelle 3.5: Aufteilung der Kosten des Prozessors

erfolgt über den Bus D_{in}. Dieser Bus wird mit einem Netzwerkknoten verbunden wie in Abschnitt 3.4.2 beschrieben.

Die Hardware hat folgende Besonderheiten: Die Busse für Programm und Daten sind getrennt, es liegt also eine sogenannte *Harvard Architektur* [HP90] vor. Der Prozessor hat eine sogenannte *LOAD–STORE–Architektur*, das heißt, daß nur bei Befehlen LD, ST, PSH, POP und JSR auf den Speicher zugegriffen wird, bei Compute Befehlen aber nicht. Deshalb können Addierer und Multiplizierer sowohl für ADD und MUL als auch zur Adressberechnung und zum Auswerten der linearen Hash Funktion benutzt werden.

Um das Verhalten der Hardware zu spezifizieren, muß man alle Kontrollsignale für alle Treiber, Multiplexer, Register und RAM beschreiben. Dies erfolgt in Anhang B.1. Wir setzen die Kosten für die Kontrolle pauschal mit 2000 an, die Verzögerung mit 10. Mit den Kenntnissen über die verwendeten Bausteine kann man nun Kosten und Geschwindigkeit des Prozessors bestimmen. Die Gesamtkosten ergeben sich zu:

$$\begin{aligned}
\mathrm{c}(\texttt{PROZ}) &= \rho_A \cdot (9\mathrm{c}(\texttt{MUX}_{\log m}) + 5\mathrm{c}(\texttt{INC}_{\log m}) + \mathrm{c}(\texttt{MULT}_{\log m, \log m}) \\
&\quad + \mathrm{c}(\texttt{ADDSUB}_{\log m}) + \mathrm{c}(\texttt{SHIFT}_{\log m}) + \mathrm{c}(\texttt{MUX}_4) \\
&\quad + \log m \cdot (\mathrm{c}(\texttt{AND}) + \mathrm{c}(\texttt{NAND}) + \mathrm{c}(\texttt{OR}) + \mathrm{c}(\texttt{EXOR}))) \\
&\quad + \rho_K \cdot \mathrm{c}(32 \times \log m\text{–Dual Port RAM}) \\
&\quad + \rho \cdot (5\mathrm{c}(\mathrm{REG}_{\log m}) + \mathrm{c}(\mathrm{REG}_4) + \mathrm{c}(\mathrm{LOGIK}) + 20\mathrm{c}(\mathrm{Treiber}_{\log m})) \\
&= 667.35 \log m + 9.75 (\log m)^2 - 27.0 \log\log m + 26.25 \log m \log\log m \\
&\quad - 8.5 + \mathrm{c}(\mathrm{LOGIK}) && (3.27) \\
&= 37395.7 && (3.28)
\end{aligned}$$

Eine Aufteilung in Kosten für Arithmetik, RAM und Register/Treiber/Logik in Tabelle 3.5 zeigt deutlich das Übergewicht der Arithmetik. Dies ist nicht gut, da die Arithmetik nur sehr selten benutzt wird. Diese Beobachtung wird im nächsten Kapitel auch zu einer Änderung des Prozessors führen.

Die Länge des ersten Taktes ergibt sich zu $1.5 \log m + \delta = 53$, wenn der Programmspeicher so groß wie der Datenspeicher ist. Die Länge des zweiten Taktes ergibt sich aus

$$\begin{aligned} & \mathrm{d(LOGIK)} + \mathrm{d}(32 \times \log m\text{–RAM}) + \mathrm{d(Treiber)} \\ & + \mathrm{d}(\mathtt{ADDSUB}_{\log m}) + \mathrm{d}(\mathtt{MUX}_{\log m}) + \mathrm{d}(\mathtt{MULT}) + \mathrm{d(Treiber)} + \delta \\ = \;& 18 \log\log m + 13.5 + \mathrm{d(LOGIK)} = 113.5 \end{aligned} \tag{3.29}$$

Die Länge des 3. Taktes ergibt sich aus

$$\mathrm{d}(32 \times \log m\text{–RAM}) + \delta = 1.5 \log 32 + 5 = 12.5$$

Wir wählen die Länge eines Taktes in Anlehnung an die Taktlänge des Netzwerkknotens im nächsten Abschnitt zu 30 Gate Delays. Damit wird Takt 1 in zwei Takte unterteilt, Takt 2 in 4 Takte (einen zum Addierern, drei zum Multiplizieren), Takt 3 bleibt. Damit werden zur Ausführung eines Befehles bzw. bis zum Einspeisen in Netzwerk 7 Takte benötigt. Die Länge aller weiteren Takte beträgt ebenfalls 30 Gate Delays. Ihre Anzahl beträgt $11n$.

3.4.2 Netzwerk

Die Datenpfade der Hardware des Netzwerkknotens sind in Abbildung 3.7 zu sehen. Sie ergibt sich direkt aus der Funktionsweise des Knotens. Die in der Zeichnung mit „Vergl.“ bezeichneten Vergleicher enthalten sowohl einen Vergleicher auf Vorzeichen als auch einen Vergleicher auf Identität zum Erkennen von Kombinierung. Ein Knoten des Netzwerkes enthält je einen Knoten des ersten und des umgekehrten Butterfly Netzwerkes. Jeder dieser Knoten realisiert 3 Phasen von RANADE's Algorithmus und verhält sich in den Phasen unterschiedlich. Allen Phasen ist jedoch folgendes gemeinsam: ankommende Pakete werden in FIFO–Schlangen gepuffert. Da Pakete aus einer Adresse, einem Datum und einem der vier Modi (READ, WRITE, GHOST, EOS) bestehen, sind Pakete $2 \log m + 2$ Bit breit. Ein Paket kann nur verschickt werden, wenn der Eingangspuffer des Knotens am Ende des Ausgangslinks, über den es verschickt wird, nicht voll ist. Wird das Paket verschickt, wird es am Ende des Taktes aus dem Eingangspuffer entfernt.

Die Knoten des ersten Butterfly Netzwerkes arbeiten folgendermaßen:

Phase1: Der Eingang des kreuzenden Links ist mit dem Prozessor verbunden. Pakete kommen an beiden Eingängen an. Die Auswahl des zu versendenden Paketes erfolgt durch einen Vergleich der Adressen auf Vorzeichen. Das selektierte Paket wird stets über den geraden Ausgang gesendet, falls ein Senden möglich ist. Ist eine FIFO–Schlange am Eingang leer oder ist der Eingangspuffer des nachfolgenden Knotens voll, so kann kein Paket verschickt werden. Hat das Paket den Modus READ, so muß die Richtungsqueue 1 am Ende des Taktes geclockt werden. Konnte das selektierte Paket verschickt werden, so wird es am Ende des Taktes aus dem Eingangspuffer entfernt, bei Kombinierung werden beide Eingangspuffer geclockt.

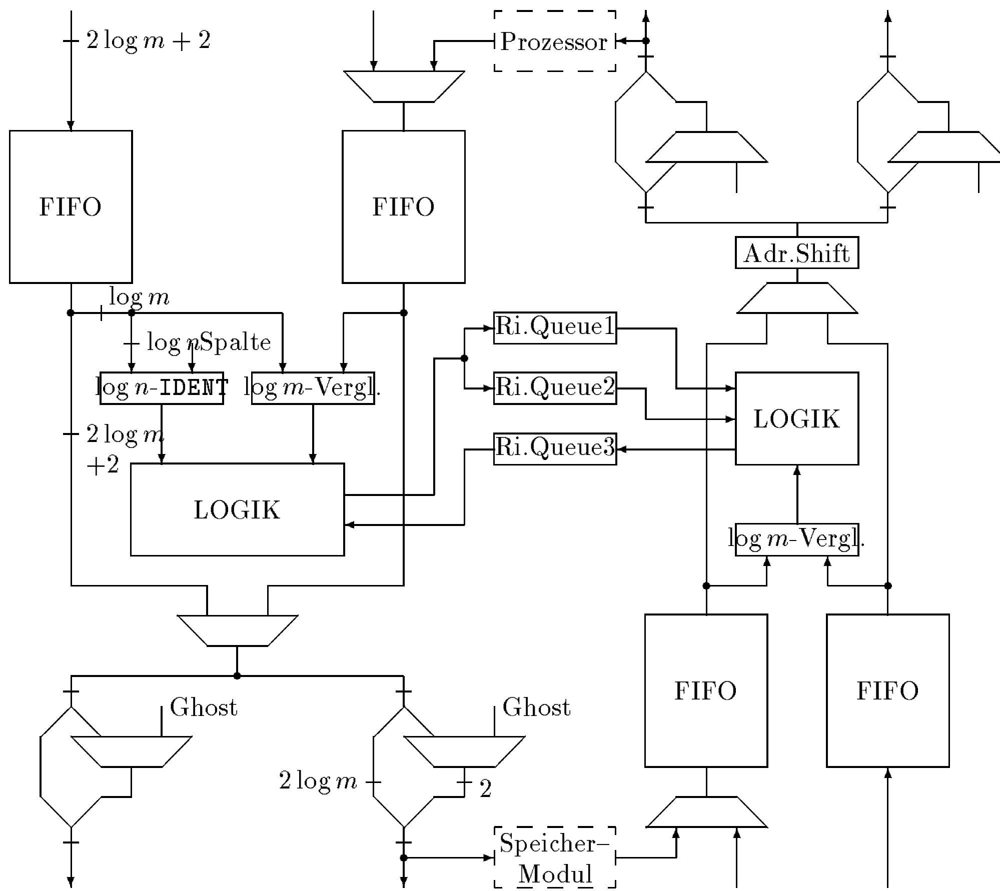

Abbildung 3.7: Datenpfade des Netzknotens

Phase 3: Es wird nur der Eingang des geraden Links benutzt. Der Ausgang des kreuzenden Links ist mit dem Speichermodul verbunden. Ankommende Pakete werden mit der Spaltennummer des Knotens auf Gleichheit verglichen. Ist der Vergleich erfolgreich, so werden sie über den Ausgang des kreuzenden Links weitergeschickt, ansonsten über den Ausgang des geraden Links. Über den jeweils anderen Ausgang wird ein GHOST verschickt, den man durch Ersetzen des Modus am entsprechenden Ausgang erhält. Hat ein zu versendendes Paket den Modus READ, so wird zusätzlich die Richtungsqueue 3 gesetzt.

Phase 5: Es werden beide Ein– und Ausgänge benutzt. Das Weiterleiten von ankommenden Paketen erfolgt in der von Richtungsqueue 2 vorgegebenen Reihenfolge.

Die Knoten des umgekehrten Butterfly Netzwerkes arbeiten wie folgt:

Phase 2: Es werden beide Eing– und Ausgänge benutzt. Die Adressen der Pakete an der Spitze der Eingangspuffer werden verglichen, das Paket mit der kleineren Adresse über den MUX selektiert. Die Entscheidung über den Ausgang, über den das Paket weitergeleitet wird, erfolgt aufgrund des untersten Adressbits. Am anderen Ausgang wird ein GHOST Paket verschickt. Um die einfache Überprüfung des richtigen Ausgangs zu ermöglichen, wird die Adresse des ausgewählten Paketes nach der Selektion logisch um ein Bit nach rechts geshiftet. Hat das ausgewählte Paket den Modus READ, so wird bei Entfernen des Paketes die Richtungsqueue 2 besetzt.

Phase 4: Der Eingang des kreuzenden Links ist mit dem Ausgang des Speichermoduls verbunden, es werden beide Eingänge benutzt, aber nur der Ausgang des geraden Links. Die Auswahl der Pakete erfolgt durch die Vorgabe der Richtungsqueue 3.

Phase 6: Es wird nur der Eingang des geraden Links benutzt, aber beide Ausgänge. Der Ausgang des kreuzenden Links ist mit dem Prozessor verbunden. Das Versenden der Pakete erfolgt nach Vorgabe der Richtungsqueue 1.

Das Verhalten der Hardware wird durch die Kontrollsignale für FIFO Clocks und Multiplexer spezifiziert; sie sind in Anhang B.2 beschrieben. Die Kosten der Logik werden pauschal mit 1000 beziffert, die Verzögerung mit 5. Damit ergeben sich folgende Gesamtkosten eines Knotens, wenn die Länge der Eingangspuffer $k = 2$ und die Länge der Richtungsqueues $k' = 11n$ gewählt werden:

$$\begin{aligned}
\mathrm{c}(\texttt{KNOTEN}) &= 4\mathrm{c}((2\log m + 2, 2) - \texttt{FIFO}) + 3\mathrm{c}((k', 3) - \texttt{FIFO}) \\
&\quad + \rho_A \cdot (4\mathrm{c}(\texttt{MUX}_{2\log m + 2}) + 2\mathrm{c}(\texttt{VGL}_{\log m}) + 2\mathrm{c}(\texttt{IDENT}_{\log m}) \\
&\quad + \mathrm{c}(\texttt{IDENT}_{\log n}) + 4\mathrm{c}(\texttt{MUX}_2)) + \rho \cdot \mathrm{c}(\mathrm{LOGIK})) \\
&= 9382 + 1069.2n + 6\log n < 9544 + 1069.2n \qquad (3.30)
\end{aligned}$$

Die Länge eines Taktes wird durch folgenden Pfad bestimmt: Nach dem Beginn des Taktes müssen die neuen Pakete an den Spitzen der Eingangspuffer erscheinen, die Adressen müssen verglichen werden. Abhängig vom Vergleich entscheidet die Logik über die Selektion. Das selektierte Paket muß 3 Multiplexer (bei Selektion, beim Verändern des Modus und bei der Auswahl zwischen Prozessoren bzw. Speichermodulen und Link vor dem nächsten Eingangspuffer) bis zum nächsten Eingangspuffer durchlaufen. Damit ergibt sich die Zykluszeit des Knotens zu

$$
\begin{aligned}
& \mathrm{d}(ck_{out} \rightarrow D_{out}) + \mathrm{d}(\mathtt{VGL}_{\log m}) + \mathrm{d}(\mathrm{LOGIK}) + 2 \cdot \mathrm{d}(\mathtt{MUX}_{2\log m+2}) + \mathrm{d}(\mathtt{MUX}_2) + \delta \\
= \; & \mathrm{d}(\mathrm{LOGIK} + 2 \log\log m + 1.5 \log k + 15 \\
= \; & 30 \qquad (3.31)
\end{aligned}
$$

3.4.3 Analyse

Die Kosten der Fluent Machine ergeben sich zu

$$c_0 = N(\mathrm{c}(\mathtt{PROZ}) + \mathrm{c}(\mathtt{KNOTEN})) = N \cdot (46939.7 + 1069.2n) \qquad (3.32)$$

Die Zeit zur Ausführung eines Befehls ergibt sich aus den 7 Prozessortakten, die benötigt werden, um ein Paket ins Netzwerk zu schicken, und der Anzahl von Netzwerktakten, die das Paket bis zur Rückkehr benötigt. Die Anzahl der Netzwerktakte bis zur Rückkehr ist mit $11n$ angegeben [RBJ88]. Diese Zahl wurde aus Simulationen gewonnen. Eigene Simulationen (siehe auch Abschnitt 4.3) bestätigen sie. Die Zeit zur Ausführung eines Befehls errechnet sich somit zu

$$t_0 = 7\mathrm{d}(\mathtt{PROZ}) + 11n \cdot \mathrm{d}(\mathtt{KNOTEN}) = 210 + 330n \qquad (3.33)$$

Diese Zeit muß nicht nur für den LOAD Befehl benutzt werden, sondern für alle Befehle, da ein Prozessor nicht weiß, ob ein anderer gerade einen LOAD ausführt oder nicht. Wäre dies der Fall, und der Prozessor wartet nicht, so wäre der synchrone Betrieb der Maschine gestört.

Die errechneten Werte für c_0 und t_0 werden in den folgenden Kapiteln als Referenz genommen, mit denen wir Veränderungen als Verbesserungen oder Verschlechterungen bewerten.

Kapitel 4

Veränderungen

RANADE's Fluent Machine ist ohne prinzipielle Änderungen des Aufbaus asymptotisch nicht zu verbessern. Man kann allerdings durch verbesserten Aufbau versuchen, Kosten zu sparen und/oder die Leistung der Maschine zu erhöhen.

4.1 Pipelining

Ein erster Ansatz zur Leistungssteigerung ist das Prinzip des *Pipelining*. Es wird benutzt, um Funktionseinheiten besser auszulasten. Folgende Beobachtung liegt dem Pipelining zugrunde: Komplexe Berechnungen erfordern oft Schaltkreise sehr großer Tiefe. Allerdings wird jeder Teil dieser Schaltkreise nur eine kurze Zeit lang benutzt. Setzt man nun an die Ausgänge eines jeden Teils Register und clockt diese, sobald die Ausgänge des Teils stabil sind, so kann man diesen Teil des Schaltkreises im nächsten Takt bereits für neue Berechnungen benutzen. Um das Prinzip anwendbar zu machen, muß man die Register so plazieren, daß alle Teilschaltkreise möglichst gleiche Tiefe haben, da die Zykluszeit der Clock vom tiefsten Teilschaltkreis abhängt. Die Teilschaltkreise bezeichnet man als *Stufen* der Pipeline.

Es ergibt sich folgende Rechnung: Hat ein Schaltkreis die Tiefe t und kann er in x gleichtiefe Stufen zerlegt werden, so kann die Pipeline mit Zykluszeit $t/x + \delta$ betrieben werden. Der Schaltkreis ohne Pipeline benötigt zur Berechnung von y Ergebnissen ty Gate Delays. Die Pipeline benötigt Zeit x Takte, bis das erste Ergebnis am Ausgang der Pipeline erscheint, danach $y - 1$ Takte für die restlichen Ergebnisse. Damit benötigt die Pipeline $(x + y - 1)(t/x + \delta)$ Gate Delays. Pipelining ist also nützlich, wenn mit jedem Takt neue Eingabedaten geliefert werden können und wenn viele gleichartige Operationen ausgeführt werden sollen. Eine Pipeline lohnt nicht, wenn die Bereitstellung neuer Eingabedaten direkt von unmittelbar vorher berechneten Ergebnissen abhängt. Dann kann nämlich nur alle x Takte ein neues Eingabedatum erzeugt werden. Man spricht von „Bubbles“ in der Pipeline. Kommt dieser Fall nur selten vor, so kann anhand des Modells aus Kapitel 3 die Rentabilität

der Pipeline geprüft werden.

Die zeitintensivste Operation der Fluent Machine ist der Zugriff auf den Speicher. Diese kann allerdings nicht direkt als Pipeline betrieben werden, weil für 6 Phasen nur 2 Netzwerke vorhanden sind. Als ersten Ansatz kann man erwägen, 6 Netzwerke vorzusehen. Nur zwei dieser Netzwerke werden allerdings als Butterfly Netzwerke benutzt, dies sind die Phasen 2 und 5. Die Phasen 3 und 4 dienen nur zum Verteilen der Pakete auf n Module und zum Wiederaufsammeln der Antworten. Phase 3 kann auch durch einen Baum von Treibern realisiert werden, wenn jedes Modul prüft, ob ein angekommenes Paket für dieses Modul bestimmt ist. Phase 4 kann durch einen Baum von ODER Gattern realisiert werden, da in jedem Takt höchstens ein Paket die Module verläßt.

Anmerkung: Obige Behauptung gilt nur unter der Voraussetzung, daß jeder Speicherzugriff gleichlange dauert. Ist dies auch in realen dynamischen RAM Bausteinen nicht der Fall, so kann jeder Zugriff doch künstlich auf eine bestimmte Zeit verlängert werden. Dies beeinflußt die Geschwindigkeit der Maschine nicht, da in der Zeitanalyse sowieso vom schlechtesten Fall ausgegangen werden muß.

Die Phasen 3 und 4 werden nicht langsamer, da ein Durchgang durch den Treiber– bzw. ODER–Baum nur $\log n \cdot \mathrm{d(Treiber)}$ bzw. $\log n \cdot \mathrm{d}(\texttt{OR})$ dauert. Diese Zeit entspricht jeweils einem Netzwerktakt, da $\mathrm{d}(\texttt{KNOTEN})$ von $\log\log m$ und damit von $\log n$ abhängt und die Verzögerung von Treibern und ODER Gattern jeweils 1 ist.

Auch die Phasen 1 und 6 benötigen kein Butterfly Netzwerk. In Phase 1 werden lediglich die n eingespeisten Pakete nach ihren Adressen sortiert und eventuell kombiniert, in Phase 6 wird dies rückgängig gemacht. Phasen 1 und 6 könnte man also aus „abgespeckten“ Netzwerkknoten realisieren.

Die neue Anordnung hat außerdem den Vorteil, daß alle Datenpfade der Phasen 4 bis 6 statt $2\log m + 2$ nur $\log m$ Bits breit sein müssen, da auf dem Rückweg nur Daten transportiert werden. Die Netzwerkknoten der Phasen 2 und 5 ergeben sich dann aus den Netzwerkknoten der Fluent Machine, indem die Datenpfade des Knotens für Phase 1,3,5 verschmälert werden. Die Multiplexer vor den Eingangspuffern fallen weg, nur noch eine Richtungsqueue bleibt. Beim Knoten für Phase 1,3,5 fallen die Vergleicher und die Multiplexer zum Einfügen von Ghosts weg.

Unsere Pipeline hat die Tiefe $11n$, wenn ein Netzknoten als Komponente gerechnet wird. Wir müssen allerdings dafür sorgen, daß stets genügend Eingabedaten geliefert werden können. Dabei fällt auf, daß die Prozessoren nach dem Einspeisen von Paketen ins Netzwerk $O(n)$ Schritte nichts tun, da sie auf die Rückkehr dieser Pakete warten müssen. Das Ergebnis eines LOAD Befehls muß vor der Ausführung des nächsten Befehls vorhanden sein, da der gelesene Wert eventuell im nächsten Befehl benutzt wird.

Es bietet sich deshalb an, die n Prozessoren einer Zeile zu einem *physikalischen Prozessor* (pP) zusammenzufassen, der dann die n Prozessoren einer Zeile als *virtuelle Prozessoren* (vP) simuliert. In Abschnitt 4.1.1 wird eine Hardware für einen physikalischen Prozessor

vorgestellt, der n virtuelle Prozessoren in einer Pipeline simuliert und nacheinander n Pakete erzeugt.

Die Idee, eine Netzwerkverzögerung durch die Simulation mehrerer Prozessoren durch einen einzigen Prozessor zu kaschieren, wird von VALIANT in [Val90a] theoretisch diskutiert. Die benötigte Anzahl unabhängiger Eingabedaten (hier die Pakete) wird dort *parallel Slackness* genannt. VALIANT geht allerdings auf konkrete Realisierungen eines solchen Prozessors nicht ein.

Da ein physikalischer Prozessor nicht alle Pakete gleichzeitig erzeugt, muß eine andere Realisierung von Phase 1 und 6 gewählt werden. Wir benutzen zur Realisierung von Phase 1 ein Sortierfeld für n Pakete, wie es in Abschnitt 4.1.2 geschildert wird. Phase 6 wird durch ein Rücksortierfeld realisiert, das von den Entscheidungen aus Phase 1 über eine Richtungsqueue gesteuert wird.

4.1.1 Virtuelle Prozessoren

Wir werden im folgenden alle Veränderungen so beschreiben, als ob cn virtuelle Prozessoren simuliert würden. Virtuelle Prozessoren werden durch ihre Registersätze repräsentiert. Die Hardware des Prozessors aus Abschnitt 3.4.1 wird demzufolge zunächst so geändert, daß das Dual Port RAM auf Größe $32cn \times \log m$ vergrößert und die Register ST, PC, SP_0, SP_1 jeweils durch $cn \times \log m$ RAMs ersetzt werden. Die Ausführung eines Befehls eines vP soll in jedem zweiten Takt beginnen. Dies ist notwendig, da eventuell während eines STORE Befehls drei Werte aus dem Dual Port RAM gelesen werden, was mindestens zwei Takte erfordert. Um den Datenfluß im Prozessor zu gewährleisten, müssen nun überall dort, wo Daten länger als zwei Takte benötigt werden, zusätzliche Register eingebaut werden. Dies sind folgende Stellen:

- Am Adressbus zum Programmspeicher (1 Latch).
- Bei der Rückkopplung von Program Counter und Stack Pointer zum Inkrementer (5 Register).
- Auf L und R Bus zwischen Eingängen und Ausgängen (1 Register).
- Vor den Eingängen des Multiplizierers (1 Register).
- Hinter Shifter, Addierer und bitorientierten Operationen (3 Register).
- Auf der Verbindung parallel zur ALU (4 Register).

Die Anzahl der Register ist in Klammern angegeben. Sie hängt von der Anzahl der Takte an, die ein Signal verzögert werden muß. Als Beispiel dient die Rückkopplung des Programm Counters. Der PC eines virtuellen Prozessors wird im 3. Takt selektiert und im 8. Takt

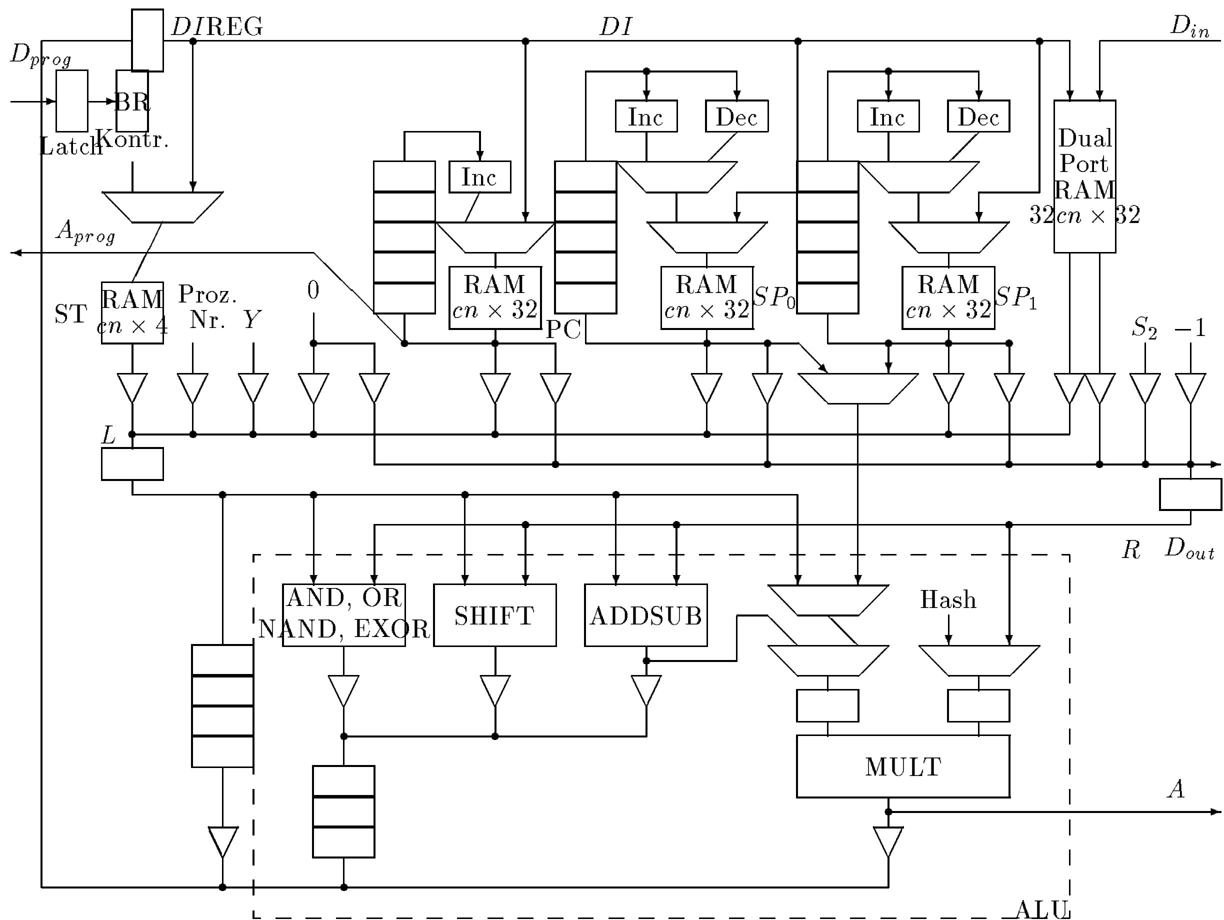

Abbildung 4.1: Datenpfade des veränderten Prozessors

geschrieben. Um dies zu gewährleisten, muß das rückzukoppelnde Signal 5 Takte verzögert werden. Dies erfolgt mittels 5 hintereinandergeschalteter Register.

Am Adressbus zum Programmspeicher wird statt eines Registers ein Latch benutzt. Ein Latch ähnelt einem Register und wird auch mit gleichen Kosten und Verzögerungen gewertet. Statt einer Clock hat ein Latch allerdings ein Select Signal. Ist dieses 1, so ist das Latch *transparent*, das heißt, am Eingang anliegende Daten werden sofort zum Ausgang weitergeleitet. Wird das Signal zu 0, so werden die zu diesem Zeitpunkt anliegenden Daten gespeichert *(gelatcht)*, bis zur nächsten steigenden Flanke liegen diese Daten am Ausgang an.

Die veränderten Datenpfade sind in Abbildung 4.1 zu sehen.

Der Befehlsfluß eines physikalischen Prozessors kann damit bei Ausführung von STORE Befehlen folgendermaßen schematisch dargestellt werden:

Es zeigt sich, daß das Dual Port RAM in ungeraden Takten auf beiden Ports gelesen wird, in geraden Takten auf einem Port bei STORE gelesen bei LOAD geschrieben und auf dem anderen nur geschrieben wird (bei COMPUTE). Die RAMs für Stackpointer und Program Counter werden in ungeraden Takten gelesen und in geraden geschrieben.

Anmerkung: Um einen reibungslosen Datenfluß zu gewährleisten, muß die Aufteilung der Takte geschickt geplant werden. Dies erfordert nicht nur die Kenntnis des formalen Modells, sondern auch etwas Erfahrung im Hardware Design. Dies ist kein Widerspruch zu einer rein formalen Vorgehensweise. Auch die Analysis stellt zum Beispiel zur Berechnung von Folgen und Reihen einige Hilfsmittel zur Verfügung, die Berechnung einer konkreten Reihe erfordert dennoch Übung und Erfahrung.

Auch die Kontrollogik muß als Pipeline arbeiten. Die Kontrollsignale werden weiterhin alle gleichzeitig erzeugt, aber eventuell durch Register verzögert. Die Anzahl der Kontrollsignale erhöht sich nicht, da die zusätzlichen Register in jedem Takt geclockt werden.

4.1.2 Sortierfelder

Ein Sortierfeld für n Pakete besteht aus n Sortierknoten, die hintereinandergeschaltet sind. Die Pakete werden nacheinander am ersten der Sortierknoten eingespeist. Das einfachste Verfahren zum Sortieren der Pakete benutzt in jedem der Knoten folgende Strategie [LL90]: Falls sich im Knoten bereits ein Paket befindet und ein neues Paket kommt an, so wird das mit der kleineren Adresse zum nächsten Knoten weitergeschickt, das andere bleibt im Knoten. Sobald das erste Paket den letzten Knoten erreicht hat, befindet sich in jedem der Knoten genau ein Paket und die Pakete sind nach Adressen sortiert. Sie können dann nacheinander aus dem Sortierfeld ins Netzwerk der Phase 2 geschoben werden.

Dieses Sortierverfahren hat allerdings einen Nachteil. Nach dem Einspeisen des letzten Paketes in den ersten Sortierknoten können bis zu n Schritte bis zum Ende der Sortierung vergehen. Dieser Fall tritt dann auf, wenn das letzte eingespeiste Paket die kleinste Adresse unter den n zu sortierenden Paketen besitzt.

Erst wenn das Ausspeisen der Pakete aus dem Sortierfeld beginnt, können parallel dazu wieder weitere Pakete eingespeist werden. Da allerdings der physikalische Prozessor, zu dem das Sortierfeld gehört, ständig Pakete erzeugt, wenn alle virtuellen Prozessoren LOAD Befehle erzeugen, muß man mindestens zwei Sortierfelder betreiben, die abwechselnd gefüllt und geleert werden.

Wir werden eine Methode benutzen, die zwar ebenfalls zwei Sortierfelder braucht, bei der aber nach dem Einspeisen des letzten Paketes direkt das Ausspeisen beginnen kann. Die zusätzliche Verzögerung von bis zu n Takten entfällt also. Bei dieser Methode wird die Strategie der Sortierknoten folgendermaßen geändert: Solange nicht alle Pakete eingespeist sind, wird von zwei Paketen in einem Sortierknoten nicht das Paket mit kleinerer Adresse, sondern das Paket mit größerer Adresse zum nächsten Sortierknoten weitergeschickt. Sobald das letzte Paket eingespeist ist, wird von zwei Paketen in einem Sortierknoten das Paket

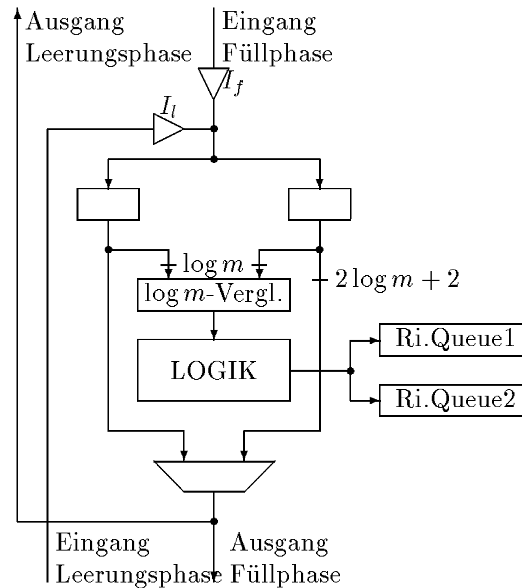

Abbildung 4.2: Datenpfade des Sortierknotens der Phase 1

mit kleinerer Adresse zum Vorgängerknoten geschickt.

Da gewährleistet ist, daß das Paket mit kleinster Adresse sich im ersten Sortierknoten befindet, kann das Ausspeisen beginnen, sobald das Einspeisen der Pakete abgeschlossen ist. Da das Ausspeisen aber bis zum Ende den ersten Sortierknoten benötigt, kann das Einspeisen der nächsten n Pakete nicht mehr gleichzeitig mit dem Ausspeisen erfolgen. Man benötigt daher zwei Sortierfelder.

Die Realisierung der Phase 6 erfolgt ebenfalls über Sortierknoten, die wie im Netzwerk mittels Richtungsqueues gesteuert werden. Beim Ein– und Ausspeisen werden verschiedene Richtungsqueues benötigt, deshalb müssen zwei Richtungsqueues vorhanden sein.

Die Datenpfade eines Sortierknotens der Phase 1 sind in Abbildung 4.2 zu sehen. Die Datenpfade eines Knotens der Phase 6 unterscheiden sich nur dadurch, daß kein Vergleicher benötigt wird, alle Datenpfade nur Breite $\log m$ haben und aus den Richtungsqueues der Phase 1 gelesen wird.

Die Sortierung erfolgt in einer Füllphase (Einspeisen der Pakete) und in einer Leerungsphase (Ausspeisen der Pakete). Für jede dieser Phasen existiert eine eigene Richtungsqueue. In der Füllphase arbeitet der Knoten wie folgt: Der Eingangstreiber von I_f ist geöffnet, der von I_l geschlossen. Sind beide Eingangsregister mit Paketen belegt, so werden die Adressen dieser Pakete verglichen. Das Paket mit größerer Adresse wird zum Ausgang geleitet. Haben beide Pakete gleiche Adresse, so werden sie kombiniert, d.h. das rechte Paket wird gelöscht. Sind nicht beide Eingangsregister belegt, so wird kein Paket verschickt. Kommt über den Eingang ein Paket an, so wird dieses am Ende des Taktes in ein freies bzw. durch Versenden oder Löschen freiwerdendes Eingangsregister geclockt. Beim Versenden oder Kombinieren

von LOAD Paketen wird eine Richtungsqueue gefüllt.

Nachdem n Pakete in das Sortierfeld eingespeist wurden, endet die Füllphase und es beginnt die Leerungsphase:

Treiber I_f ist nun geschlossen, I_l ist geöffnet. Enthalten beide Eingangsregister Pakete, so wird das mit der kleineren Adresse über den Multiplexer zum Ausgang geleitet. Sind die Adressen gleich, werden die Pakete kombiniert, d.h. eines wird gelöscht und eines weitergesandt. Bei LOAD Paketen wird eine zweite Richtungsqueue gefüllt. Enthält nur eines der beiden Eingangsregister ein Paket, so wird dieses zum Ausgang geleitet. Kommt ein Paket an, so wird dieses am Ende des Taktes in ein freies oder freiwerdendes Register geclockt.

Beide Phasen benötigen je n Takte, da die Füllphase gerade dann abgeschlossen wird, wenn n Pakete aufgenommen wurden, und die Leerungsphase endet, wenn alle n Pakete das Feld verlassen haben; in jedem Takt verläßt ein Paket das Feld. Die Füllphase benötigt tatsächlich sogar $2n$ Takte, da ein physikalischer Prozessor nur in jedem zweiten Takt ein Paket erzeugt.

Da ein Sortierfeld in 2 Phasen arbeitet, werden pro Zeile zwei Sortierfelder eingesetzt, die stets in unterschiedlichen Phasen arbeiten. Damit ist gewährleistet, daß stets Pakete eingespeist werden können, da die Leerungsphase weniger Takte als die Füllphase benötigt.

4.1.3 Änderungen des Routingalgorithmus

Der Routing Algorithmus der Fluent Machine benutzt End of Stream Ghosts, um eine Runde abzuschließen. Da in unserem veränderten Modell mehrere Runden hintereinander in einer Pipeline durch das Netzwerk laufen sollen, benötigt man statt einem Abschluß eine Trennung. Diese wird erreicht durch sogenannte *End of Round* Pakete (EOR). Dies sind Pakete mit gleich großer Adresse wie EOS Ghosts, werden aber im Unterschied zu Ghosts nicht gelöscht, wenn sie nicht versendet werden können. EOR Pakete werden damit erst selektiert, wenn an den Spitzen beider Eingangspuffer EOR Pakete stehen. Es findet also stets eine Kombinierung von EOR Paketen statt. Im Unterschied zu anderen Paketen wird dann allerdings nicht über einen Ausgang ein EOR Paket und über den anderen ein Ghost versandt, sondern über beide Ausgänge ein EOR Paket. Das Versenden und damit Entfernen aus den Eingangspuffern darf erst erfolgen, wenn die Eingangspuffer beider nachfolgender Knoten nicht voll sind.

Die bisherigen Veränderungen haben bewirkt, daß nur noch 2^n physikalische Prozessoren existieren. Jeder dieser bearbeitet die Befehle von n virtuellen Prozessoren in einer Pipeline. Die Verzögerungszeit des Netzwerkes ist aber ein Vielfaches von n. Um die Pipeline ständig arbeiten zu lassen, muß also auch eine entsprechende Anzahl cn von (virtuellen) Prozessoren simuliert werden. Dies ist in der Hardware des physikalischen Prozessors durchaus möglich, in dem man die Register RAMs entsprechend vergrößert.

Es stellt sich allerdings die Frage, ob Netzwerk und Routing Algorithmus dies verkraften. Hierzu bewiesen CHANG und SIMON [CS86], daß es eine Konstante y gibt, so daß das Routing von Paketen durch den n–dimensionalen Würfel auch dann $O(n)$ Schritte dauert, wenn jeweils nach y Takten an jedem Knoten des Würfels ein neues Paket eingespeist wird. Ein Wert für die Konstante y wird nicht angegeben. In unserem Fall bietet sich die Wahl von $y = 2$ an, da Prozessoren und Netzwerkknoten gleiche Zykluszeit haben, in den Prozessoren aber nur in jedem zweiten Takt ein Paket erzeugt wird.

Die Frage, ob $y = 2$ die Routing Zeit von Paketen um einen konstanten Faktor vergrößert, muß durch Simulation geklärt werden, da schon die zuvor angenommene Zeit durch Simulationen bestimmt worden war.

Gleichzeitig stellt sich die Frage nach der Länge der Sortierfelder. Werden diese nicht verlängert, kann sich die Modulbelastung um den Faktor c erhöhen, da von cn Paketen einer Zeile nur jeweils n kombiniert werden können. Verlängert man die Sortierfelder auf cn Knoten pro Feld, so verlängert man die Routingzeit der Pakete. Eine geeignete Länge cn/z, wobei z die Zahl cn teilt, muß ebenfalls durch Simulationen bestimmt werden. Beides erfolgt in Abschnitt 4.3.

4.2 Delayed LOAD

Die zu bestimmende Zahl c sollte möglichst klein sein. Dies läßt sich mit Hilfe der Bewertungsfunktion für unsere Maschine begründen. Ein Schritt aller virtuellen Prozessoren ist nach cn Takten abgeschlossen. c muß so groß sein, daß normalerweise kein Prozessor auf die Antwort eines READ Paketes warten muß. Machen wir c um einen Faktor c' größer, so erhalten wir eine Maschine mit einer um den Faktor c' vergrößerten Prozessorzahl. Enthält das zu lösende Problem genügend Parallelität, um es auf $c'cN$ Prozessoren mit gleicher Effizienz zu parallelisieren wie auf cN, so verringern wir die Anzahl der Befehle pro Prozessor um den Faktor c', da jetzt mehr Prozessoren zur Verfügung stehen. Gleichzeitig wird die Zeit zur Ausführung eines Befehles um den gleichen Faktor größer. Wir gewinnen in der Gesamtausführungszeit nichts mehr, erhalten für die zusätzlichen Registersätze aber weitere Kosten. Unsere Bewertung wird also schlechter.

Könnten wir c verkleinern, ohne daß ein Prozessor auf die Antwort eines READ Paketes warten muß, so könnten wir die Bewertung unserer Maschine verbessern. Um dies zu erreichen, benutzen wir ein Merkmal, daß auch einige Prozessoren in Einprozessorsystemen benutzen, um Wartezeiten wegen Speicherzugriffen zu verkleinern: wir erwarten die Antwort eines LOAD Befehls nicht am Ende dieses Befehls, sondern am Ende des nächsten. Man nennt dies *delayed LOAD*.

Die Benutzung von delayed LOAD erlaubt es uns, c um einen Faktor 2 zu verkleinern. Wird allerdings diese Antwort schon im nächsten Befehl als Argument benötigt, so muß der Compiler einen anderen, von diesem Argument unabhängigen Befehl einschieben, falls dies

möglich ist. Ist dies nicht möglich, so muß ein „Dummy–Befehl" eingeschoben werden, was unsere Bewertung wieder verschlechterte. Nach Angaben von HENNESSY und PATTERSON [HP90] ist dies aber nur in sehr wenigen und deshalb vernachlässigbaren Fällen notwendig. Um diese Behauptung nachzuprüfen, wurde in [AKP91b] ein Algorithmus zur Berechnung von Zusammenhangskomponenten handcompiliert. Bis auf einen einzigen Befehl, der auch nur einmal während des Programmlaufs erreicht wird, konnten überall andere Befehle eingeschoben werden. Wir werden deshalb im Weiteren nicht mehr auf Dummy–Befehle achten.

4.3 Simulationen

Ziel der Simulationen ist die Bestimmung der Konstanten c und z. Um diese zu erhalten, wurden mehrere Simulationen durchgeführt. Eine Beschreibung des Simulators und weitere Simulationsergebnisse sind in [Eng92] zu finden.

In den Simulationen wurde wie in den Berechnungen des letzten Kapitels angenommen, daß Sortier– und Netzwerkknoten pro Zyklus einen Verarbeitungsschritt durchführen und daß Prozessoren in jedem zweiten Zyklus ein Paket erzeugen können. Die Speicherzugriffszeit wurde zu 2 Zyklen berechnet. Der Speicher hat eine maximale Größe von $m = 2^{32}$ Worten und ist auf $n \cdot 2^n$ Module verteilt. Jedes Modul hat demnach eine Größe von $m' = 2^{32-n}/n$ Worten. Die Zugriffszeit auf ein Speichermodul beträgt $t_m = 1.5 \cdot \log m'$. Es ergibt sich $t_m = 48 - 1.5 \cdot n - 1.5 \cdot \log n$. Zusätzlich muß noch die Zeit zum Durchqueren des Treiber– und des ODER–Baumes addiert werden, die $t_b = \log n \cdot (\mathrm{d}(\texttt{OR}) + \mathrm{d}(\text{Treiber})) = 2 \log n$ beträgt. Damit erhält man $t_m + t_b = 48 - 1.5 \cdot n + 0.5 \cdot \log n$. Dieser Wert ist für $n \leq 12$ größer als 30 Gate Delays, womit man 2 Zyklen erhält.

Die Simulationen maßen auch den Anteil der Zyklen, in denen Prozessoren angehalten wurden. c wurde so gewählt, daß dies nur in einem sehr kleinen Teil der Befehle auftrat (weniger als 1 Promille).

4.3.1 Einfluß von Hashing

ENGELMANN führte in [Eng92] auch Untersuchungen zum Einfluß von Hashing auf die Modulbelastung und auf die Laufzeit von Paketen durch. Als Eingabedaten wurden sowohl gewürfelte Pakete als auch aus Simulationen von Programmen gewonnene Pakete benutzt. Die Ergebnisse unterschieden sich im wesentlichen nicht. Es zeigte sich, daß für moderate Maschinengrößen ($n \leq 8$) schon quadratische Funktionen aufgrund ihrer Nichtbijektivität eine größere Modulbelastung verursachten als lineare Funktionen.

4.3.2 Bestimmung der Konstanten c und z

Die Bestimmung der Konstanten c und z kann nicht unabhängig voneinander vorgenommen werden, da eine Änderung von z die Verzögerung eines Paketes im Netzwerk beinflußt und damit auch c.

Für gegebenes z kann c direkt aus Simulationen und Messungen der Verzögerung im Netzwerk errechnet werden. Für gegebenes c kann z mittels folgender Überlegung gewonnen werden. Die Sortierfelder dienen dazu, Pakete einer Zeile, die zur gleichen Adresse wollen, zu kombinieren. Verkürzt man die Sortierfelder um einen Faktor z, so verlassen maximal z Pakete mit gleicher Zieladresse eine Zeile. Die Modulbelastung steigt um maximal einen Faktor z. Gleichzeitig wird allerdings die Routingzeit durch die verkürzten Sortierfelder verringert. Um ein geeignetes z zu finden, führt man deshalb Simulationen für zwei Eingabemengen durch. Die erste Eingabemenge beinhaltet zufällige Adressen. Die zweite Eingabemenge ist so gewählt, daß alle virtuellen Prozessoren eines physikalischen Prozessors Pakete mit gleicher Zieladresse ins Netz senden. Bei der ersten Eingabemenge wirkt sich die Verkürzung am besten aus, die Modulbelastung steigt hier kaum, da die Wahrscheinlichkeit, daß in einer Reihe zwei gleiche Adressen gewürfelt werden, sehr klein ist. Sie berechnet sich zu

$$1 - \frac{\prod_{i=0}^{cn-1}(m-i)}{m^{cn}}$$

Bei der zweiten Eingabemenge steigt die Modulbelastung um den Faktor z. Für gegebenes c simuliert man nun beide Eingabemengen für verschiedene z und wählt den Wert für z, bei dem beide Eingabemengen gleiche Verzögerung haben.

Mittels dieser Verzögerung v berechnet man nun ein neues c mittels der Gleichung $4cn - 8 = v$, also $c = (v+8)/4n$. Die linke Seite der ersten Gleichung beschreibt die Anzahl der Prozessortakte, nach denen die Antwort zurückerwartet wird. Die Subtraktion von 8 muß erfolgen, da erst nach 8 Takten ein Paket ins Netz gegeben wird.

Unterscheidet sich dieses c vom vorherigen Wert, so iteriert man die Simulation. Ein Konvergenzbeweis des Verfahrens kann nicht gegeben werden.

Die Simulationen ergaben $c = 3$, $z = 4$ bereits nach einer Iteration.

4.4 Analyse der Änderungen

4.4.1 Änderung der Hardwarekosten

Prozessor

Die Kosten des Prozessors erhöhen sich um die zusätzlichen Register zum Betrieb der Pipeline, die Differenz zwischen altem und neuem Registerfeld und die Differenz zwischen den Registern für ST, PC, SP_0, SP_1 und den Registerfeldern. Die Erhöhung beträgt damit

$$\begin{aligned} & 15 \cdot \mathrm{c}(\mathrm{REG}_{\log m}) + 3(\rho_K \cdot \mathrm{c}(cn \times \log m - \mathrm{RAM}) - \mathrm{c}(\mathrm{REG}_{\log m})) \\ + \; & \rho_G \cdot \mathrm{c}(32cn \times \log m - \text{Dual Port RAM}) - \rho_k \cdot \mathrm{c}(32 \times \log m - \text{Dual Port RAM}) \\ + \; & \rho_K \cdot \mathrm{c}(cn \times 4 - \mathrm{RAM}) - \mathrm{c}(\mathrm{REG}_4) \\ = \; & 24468n - 6499.2 \end{aligned} \tag{4.1}$$

Damit hat ein physikalischer Prozessor mit cn virtuellen Prozessoren Kosten

$$c(\texttt{PROZ}') = 24468n + 30896.5 \tag{4.2}$$

Da sich die Anzahl der Prozessoren um den Faktor n verringert hat, haben sich auch die Kosten für Prozessoren verringert. Dies liegt im wesentlichen an der Einsparung der Arithmethik.

Netzwerkknoten

Die Kosten eines Netzwerkknotens für Phase 2 und 5 lassen sich wie folgt berechnen:

$$\begin{aligned} \mathrm{c}(\texttt{KNOTEN}') \; = \; & 2\mathrm{c}((2\log m + 2, k) - \texttt{FIFO}) + 2\mathrm{c}((\log m, k) - \texttt{FIFO}) + \mathrm{c}((k', 3) - \texttt{FIFO}) \\ & + \rho_A \cdot (\mathrm{c}(\texttt{VGL}_{\log m}) + \mathrm{c}(\texttt{IDENT}_{\log m}) + \mathrm{c}(\texttt{MUX}_{2\log m + 2}) \\ & + 2\mathrm{c}(\texttt{MUX}_2) + \mathrm{c}(\texttt{MUX}_{\log m})) + \mathrm{c}(\mathrm{LOGIK}) \\ = \; & 6512 + 356.4n \end{aligned} \tag{4.3}$$

Hierbei wurden die gleichen Annahmen wie beim Netzwerkknoten der Fluent Machine gemacht.

Sortierknoten

Die Kosten eines Sortierknotens für Phase 1 und 6 lassen sich wie folgt berechnen, wenn die Kosten der Logik pauschal mit 500 gewertet werden:

$$\begin{aligned}
\mathrm{c}(\texttt{SORTKNOTEN}) &= 2\mathrm{c}((k',3)-\texttt{FIFO}) + 2\mathrm{c}(\mathrm{REG}_{2\log m+2}) + 2\mathrm{c}(\mathrm{REG}_{\log m}) \\
&\quad +\rho_A \cdot (\mathrm{c}(\texttt{VGL}_{\log m}) + \mathrm{c}(\texttt{IDENT}_{\log m}) + \mathrm{c}(\texttt{MUX}_{2\log m+2}) \\
&\quad +\mathrm{c}(\texttt{MUX}_{\log m})) \\
&\quad +2\mathrm{c}(\mathrm{Treiber}_{2\log m+2}) + 2\mathrm{c}(\mathrm{Treiber}_{\log m}) + \mathrm{c}(\mathrm{LOGIK}) \\
&= 5105.2 + 712.8n
\end{aligned} \tag{4.4}$$

Verteilerbäume zum und vom Speicher

Ein Verteilerbaum zum Speicher besteht aus $n-1$ Treibern der Breite $2\log m + 2$. Ein Baum vom Speicher besteht aus $(n-1)\log m$ ODER Gattern. Damit erhalten wir für ein Speichermodul folgende zusätzliche Kosten:

$$\begin{aligned}
\mathrm{c}(\mathrm{Baum}) &= (n-1)\mathrm{c}(\mathrm{Treiber}_{2\log m+2}) + (n-1)\log m \cdot \mathrm{c}(\texttt{OR}) \\
&= 460n - 460
\end{aligned} \tag{4.5}$$

Gesamtkosten

Die veränderte Maschine hat folgende Gesamtkosten:

$$c_1 = 2^n \cdot (\mathrm{c}(\texttt{PROZ}') + \mathrm{c}(\mathrm{Baum})) + n2^n \cdot (\mathrm{c}(\texttt{KNOTEN}') + \frac{2c}{z}\mathrm{c}(\texttt{SORTKNOTEN})) \tag{4.6}$$

$$= N \cdot (39097.8 + 1425.6n) + 2^n \cdot 30436.5 \tag{4.7}$$

4.4.2 Änderung des Zeitverhaltens

Die Zykluszeiten der einzelnen Teile ändern sich nicht. Ein Schritt aller virtuellen Prozessoren ist nach $2cn$ Takten beendet. Damit beträgt die Zeit zur Ausführung eines Befehls

$$t_1 = 2cn \cdot \mathrm{d}(\texttt{PROZ}) = 180n \tag{4.8}$$

4.4.3 Vergleich der Maschinen

Die veränderte Maschine hat c mal soviele Prozessoren wie die Fluent Machine. Benutzt man ein Benchmark, das auf der Fluent Machine T Befehle benötigt und das genug Parallelität besitzt, um auf der veränderten Maschine T/c Befehle zu benötigen, so erhält man als Güten

$$\begin{aligned} G_0 &= 1/(c_0 \cdot T \cdot d_0) \\ G_1 &= c/(c_1 \cdot T \cdot d_1) \end{aligned}$$

Der Quotient G_1/G_0 ergibt sich zu

$$\frac{G_1}{G_0} = \frac{c \cdot c_0 \cdot t_0}{c_1 \cdot t_1}$$

Die Analyse des Quotienten kann nur im Bereich $n \leq 27$ erfolgen, da wir die Größe $N = n2^n$ unserer Maschinen auf $N < 2^{32}$ eingeschränkt haben und somit $n \leq 27$ folgt. Die Analyse wird unterteilt in die Untersuchung der Terme c_0/c_1 und $c \cdot t_0/t_1$. Der Verlauf beider Quotienten abhängig von n und der Verlauf des Gütenquotienten ist in Abbildung 4.3 zu sehen.

Betrachten wir zuerst den Quotienten c_0/c_1. Dieser ergibt sich aus den Gleichungen 3.32 und 4.7 zu

$$\frac{c_0}{c_1} = \frac{N \cdot (46939.7 + 1069.2n)}{N \cdot (39097.8 + 1425.6n) + 2^n \cdot 30436.5}$$

Asymptotisch werden die Kosten durch Terme der Größenordnung $O(n \cdot N)$ bestimmt, die von den Richtungsqueues herrühren. Die konstanten Faktoren hierbei sind allerdings so klein, daß die Terme für $n \leq 32$ keine dominierende Rolle spielen können. Sie bewirken lediglich für große n ein leichtes Absinken des Quotienten auf 0.96. Für kleine n im Bereich von 4 bis 6 verkleinert der Term der Größenordnung $O(2^n)$ im Nenner den Bruch c_0/c_1 auf 0.98. Der Term rührt von den 2^n ALUs her. Er zeigt, daß man sich bei der Einsparung der ALUs einen Overhead eingehandelt hat, so daß sich die Einsparung erst bei größeren n voll bezahlt macht. Der maximale Wert des Quotienten ist im Bereich von $n = 7$ bis $n = 12$ mit 1.02 erreicht. Die Schwankungen im Quotienten der Kosten sind allerdings sehr klein. Die Abweichung vom Maximalwert beträgt nur 5.9%.

Das Verhalten des Quotienten der Güten wird deshalb durch den Term $c \cdot t_0/t_1$ bestimmt. Dieser ergibt sich aus den Gleichungen 3.33 und 4.8 zu

$$\frac{c \cdot t_0}{t_1} = \frac{3 \cdot (330n + 210)}{180n} = 5.5 + 3.5/n$$

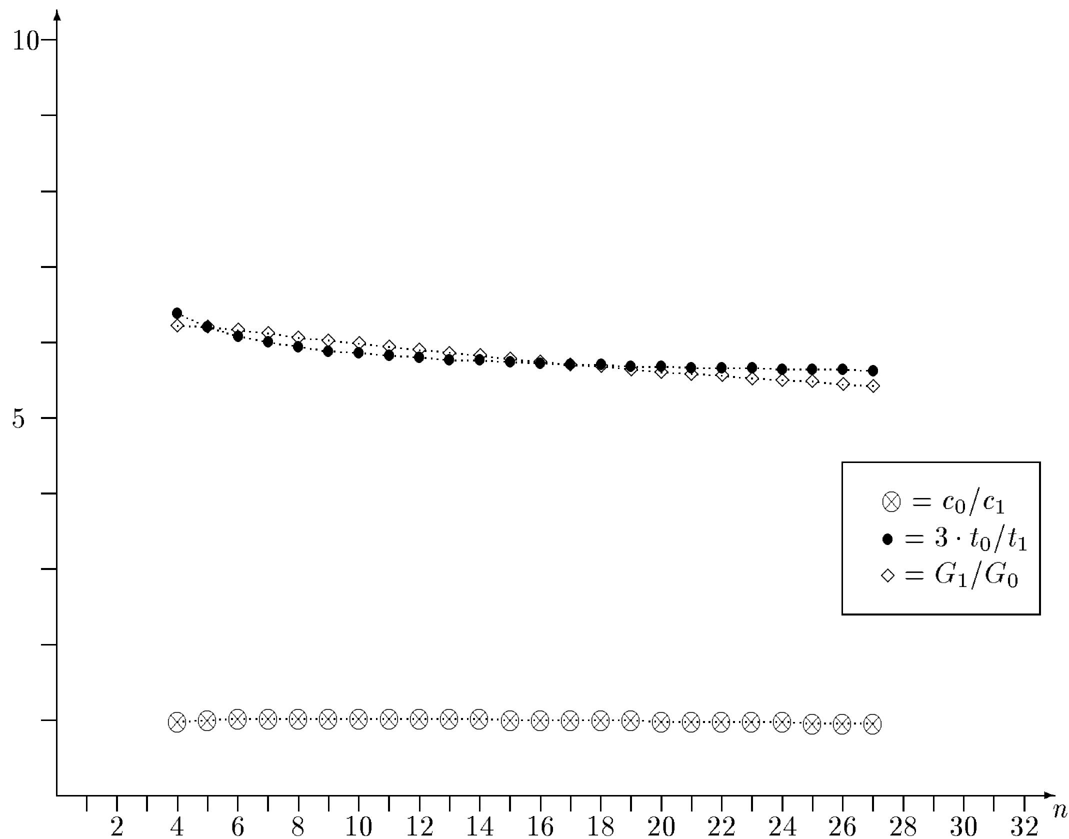

Abbildung 4.3: Quotienten der Kosten, Zeit– und Gütenfunktionen

Asymptotisch erreicht dieser Quotient den Wert 5.5. In dem hier betrachteten Bereich fällt der Quotient von 6.375 bei $n = 4$ ab bis auf 5.63 bei $n = 27$.

Der Quotient der Güten bewegt sich, bedingt durch den Abfall des Terms $c \cdot t_0/t_1$, zwischen 6.23 bei $n = 4$ und 5.42 bei $n = 27$.

Man kann also allein durch Anwendung von Architekturmaßnahmen die Fluent Machine um einen Faktor von ungefähr 6 beschleunigen und gleichzeitig die Kosten auf gleicher Höhe halten.

Kapitel 5

Design Varianten

In den vorhergehenden Kapiteln wurde gezeigt, wie man eine Architektur zur PRAM Emulation, die Fluent Machine, durch Anwendung von Wissen der Rechnerarchitektur unter Berücksichtigung des Preis/Leistungs Verhältnisses um einen konstanten Faktor verbessern kann. In diesem Kapitel soll untersucht werden, ob durch Wegnahme oder Hinzufügen von Eigenschaften der Maschine eine weitere Gütensteigerung erreicht werden kann.

Ein erster wichtiger Punkt hierbei ist die Unterstützung konkurrierender Zugriffe durch Hardware. Die Untersuchung der Notwendigkeit dieser Unterstützung ist von Bedeutung, weil nicht alle PRAM Programme eine CRCW PRAM benötigen, weil die Hardware durch den Wegfall dieser Unterstützung vereinfacht wird und weil es auch Softwarelösungen für die Unterstützung gibt. Zur Beantwortung dieser Frage werden wir einen minimalen Anteil von konkurrierenden Zugriffen bestimmen, ab dem sich die Unterstützung durch Hardware lohnt. Außerdem wird die Unterstützung nur von konkurrierenden Lesezugriffen untersucht.

Als zweites wird die Unterstützung paralleler Präfixberechnung untersucht. Paralleles Präfix ist als wichtige Unterprozedur vieler paralleler Algorithmen ebenso wie in einem parallelen Betriebssystem von Bedeutung. Auch hier werden wir einen minimalen Anteil von Aufrufen bestimmen, ab dem sich paralleles Präfix in Hardware lohnt. Wir untersuchen weiterhin, wo paralleles Präfix in Form von Betriebssystemaufrufen in Programmen vorkommt, die es nicht als Unterprozedur benutzen.

Eine letzte Untersuchung betrifft die Unterstützung vieler Prozesse. Da das Programmiermodell der PRAM beliebig viele Prozessoren in einem Programm zuläßt, eine konkrete Maschine aber nur konstant viele Prozessoren hat, ist die Frage nach der schnellen Simulation großer Prozessorzahlen sehr wichtig. Wir stellen eine Hardwarelösung vor und untersuchen ihre Güte.

5.1 Unterstützung konkurrierender Zugriffe

Um CRCW und EREW PRAMs zu vergleichen, werden wir zuerst aus der gegebenen Maschine D1 eine EREW PRAM D2 konstruieren. Danach werden wir berechnen, wie lange die Simulation eines konkurrierenden Zugriffs durch Software auf einer EREW PRAM dauert. Die Analyse der Güten liefert den gesuchten minimalen Anteil von konkurrierenden Zugriffen, ab dem eine CRCW PRAM lohnt.

5.1.1 Konstruktion einer EREW PRAM

Um eine EREW PRAM zu konstruieren, entfernen wir aus dem Netzwerk alle Teile, die nur der Kombinierung von konkurrierenden Zugriffen dienen. Dadurch ändert sich die Verzögerung des Netzwerkes. Die Prozessoren bleiben unverändert, lediglich die Konstante c verändert sich durch die neue Netzwerkverzögerung zu c'.

Die größte Veränderung besteht im Wegfall der Sortiernetzwerke für Phase 1 und 6. Diese dienten dazu, cn/z Pakete einer Reihe nach den Adressen zu sortieren und dabei Pakete, die zur gleichen Adresse wollten, zu kombinieren. Dies war notwendig, um die Modulbelastung und damit die Netzwerkverzögerung auf $O(n)$ zu halten.

Dietzfelbinger und Meyer auf der Heide beweisen in [DM90], daß selbst bei zufälliger Verteilung von unterschiedlichen 2^n Adressen (eine pro physikalischem Prozessor), ein Ausgang der Phase 2 eine Belastung von $\Omega(n/\log n)$ erhält. Würden nun alle virtuellen Prozessoren eines physikalischen Prozessors Pakete mit gleichen Adressen ins Netz geben und fände in Phase 1 keine Kombinierung statt, so würde obiger Versuch cn mal durchgeführt und ein Ausgang des Butterfly Netzwerkes von Phase 2 würde von $O(cn^2/\log n)$ Paketen benutzt, was eine höhere Belastung und damit eine höhere Verzögerung als die geforderte Zeit $O(n)$ bedeuten würde.

Ist nun keine Kombinierung mehr notwendig wie in einer EREW PRAM, so dient die Phase 1 nur noch der Sortierung von Paketen mit verschiedenen Adressen. Sie ist weiterhin notwendig, um das Funktionieren der Phase 2 zu garantieren. Diese Sortierung kann aber sehr viel einfacher erzeugt werden, wenn man der gehashten Adresse eines Paketes als *Priorität* die Nummer des virtuellen Prozessors innerhalb eines physikalischen Prozessors voranstellt, der das Paket erzeugt hat. Damit sind die Pakete einer Runde sortiert, die verschiedenen Runden werden weiterhin durch EOR Pakete getrennt. Die Sortierfelder der Phasen 1 und 6 werden somit nicht mehr benötigt.

Die Netzwerkknoten der Phase 2 ändern sich dahingehend, daß der $\log m$–bit Vergleicher auf Identität entfällt, da keine Kombinierung mehr erkannt werden muß. Die beiden Eingangspuffer, der $2\log m + 2$–bit Multiplexer und der $\log m$–Bit Vergleicher auf Vorzeichen verbreitern sich um $c'n$ Bit, um die zusätzliche Priorität zu transportieren. Die Richtungsqueue verschmälert sich von 3 auf 2 Bit, da nun keine Kombinierung von READ Paketen mehr vorgenommen werden müssen. Sie wird außerdem kürzer, da die Verzögerung im

Netzwerk kleiner wird.

Die Phasen 3 bis 5 ändern sich nicht.

Wir berechnen nun Kosten und Geschwindigkeit der Maschine D2. Durch den Wegfall der Phasen 1 und 6 wird die Netzwerkverzögerung kleiner. Die Simulationen in Kapitel 4 ergeben eine Verzögerung von weniger als $6n$ bei kleiner Varianz, d.h. der Faktor $c = 3$ bei $12n$ Verzögerung reduziert sich zu $c' = 1.5$. Die Kosten der Prozessoren ändern sich nur durch den Wert für c' und betragen nun

$$\text{c}(\texttt{PROZ}'') = 12234n + 30896.5 \tag{5.1}$$

Die Kosten der Netzwerkknoten verändern sich wie oben vorgegeben. Hierbei ist k'' die neue Länge der Richtungsqueue, es gilt $k'' = 6n$.

$$\begin{aligned} c(\texttt{KNOTEN}'') &= c(\texttt{KNOTEN}') + \rho_A \cdot (-\text{c}(\texttt{IDENT}_{\log m}) \\ &\quad +\text{c}(\texttt{MUX}_{2\log m+2+c'n}) - \text{c}(\texttt{MUX}_{2\log m+2}) \\ &\quad +\text{c}(\texttt{VGL}_{\log m+c'n}) - \text{c}(\texttt{VGL}_{\log m})) \\ &\quad +2\text{c}((2\log m + 2 + c'n, k) - \texttt{FIFO}) - 2\text{c}((2\log m + 2, k) - \texttt{FIFO}) \\ &\quad +\text{c}((k'', 2) - \texttt{FIFO}) - \text{c}((k', 3) - \texttt{FIFO}) \\ &= 278.3n + 6297.6 \end{aligned} \tag{5.2}$$

Die Geschwindigkeit des Prozessors ändert sich nicht. Die Verzögerung eines Netzwerkknotens verringert sich durch den Wegfall des Vergleichers auf Identität nicht, da der parallel arbeitende Vergleicher auf Vorzeichen größere Tiefe hat. Die Verbreiterung der Eingangspuffer und des Multiplexers bewirkt ebenfalls keine Änderung. Die Vergrößerung der Verzögerung durch den breiteren Vergleicher auf Vorzeichen beträgt höchstens 2, da die Breite sich nicht mehr als verdoppelt. Wir werden dies vernachlässigen.

Die Anzahl der virtuellen Prozessoren von D2 werden wir mit $\tilde{N} = c'n2^n$ bezeichnen. Die Gesamtkosten der EREW PRAM und die Zeit für einen Schritt aller virtuellen Prozessoren betragen damit

$$\begin{aligned} c_2 &= 2^n \cdot (\text{c}(\texttt{PROZ}'') + \text{c}(\text{Baum})) + N \cdot \text{c}(\texttt{KNOTEN}'') \\ &= N \cdot (278.3n + 18991.6) + 2^n \cdot 30436.5 \end{aligned} \tag{5.3}$$

$$\begin{aligned} t_2 &= 2c'n \cdot \text{d}(\texttt{PROZ}'') \\ &= 90n \end{aligned} \tag{5.4}$$

5.1.2 Simulation konkurrierender Zugriffe

KARP und RAMACHANDRAN zeigen in [KR90], wie man konkurrierende Zugriffe auf einer EREW PRAM simuliert. Sie bedienen sich folgender Methode:

Möchte ein Prozessor $P_i, i \in \{0, \ldots, \tilde{N}-1\}$ auf Adresse $j \in \{0, \ldots, m-1\}$ zugreifen, so schreibt er das Tupel (i, j, Modus) in die Speicherzelle i des globalen Speichers. Prozessoren, die in diesem Schritt nicht auf den globalen Speicher zugreifen, schreiben $(i, -1, 0)$. Die Inhalte der Zellen 0 bis $\tilde{N}-1$ werden nun nach dem zweiten Eintrag aufsteigend sortiert. Prozessor P_i liest nun die Inhalte der Zellen i und $i-1$ und vergleicht die zweiten Einträge. Sind diese unterschiedlich, so führt P_i einen Zugriff auf die in i mit (i', j', Modus) angegebene Adresse j' aus und schreibt bei einem lesenden Zugriff die Antwort anstatt der Adresse als zweiten Eintrag in die Zelle i. Waren bei beiden Zellen die zweiten Einträge gleich, so tut P_i nichts, bis die Speicherzugriffe beendet sind. Falls der Modus in Zelle i lesend war, dann dupliziert er das Ergebnis des Lesezugriffs als zweiten Eintrag in Zelle i. Nun liest jeder Prozessor P_i den Inhalt der Zelle i, (i', j', Modus) und schreibt ihn in die Zelle i'. Damit ist die Umsortierung rückgängig gemacht und jeder Prozessor P_i kann als zweiten Eintrag der Zelle i das Ergebnis seines Lesezugriffs abholen.

Der aufwendigste Teil dieser Simulation ist das Sortieren. Optimale parallele vergleichsbasierte Sortieralgorithmen für N Zahlen benötigen $O(\log N)$ Schritte und N Prozessoren. Der erste optimale Sortieralgorithmus wurde von AJTAI, KOMLÓS und SZEMERÉDI vorgestellt [AKS83], weitere stammen von COLE [Col88] und PATERSON [Pat90], ein randomisierter Algorithmus wurde von REIF und VALIANT vorgeschlagen [RV87]. Alle diese Algorithmen besitzen allerdings sehr große konstante Faktoren. Wir werden deswegen BATCHER's *bitonischen Sortierer* [Bat68] benutzen, der zwar Zeit $O((\log N)^2)$ benotigt und damit nicht optimal ist, der aber aufgrund eines sehr kleinen konstanten Faktors für $N < 2^{32}$, worauf wir uns beschränken, schneller ist.

Der bitonische Sortierer kann als Programm formuliert werden, siehe Abbildung 5.1. Ist die Anzahl der Prozessoren fest, können die beiden inneren Schleifen ausgerollt werden. In Anhang A werden Programmstücke für eine Schleife, den Test einer Bedingung und für eine Zuweisung angegeben.

Die Anzahl der Instruktionen zum Sortieren beträgt mit der dortigen Analyse $9.5(\log \tilde{N})^2 + 11.5 \log \tilde{N}$, falls $\tilde{N}$ eine Zweierpotenz ist. Wir wollen dies zugunsten der EREW PRAM annehmen.

Wir werden die Anzahl der restlichen Instruktionen pauschal mit 20 angeben. Insgesamt benötigen wir dann zur Simulation eines Schrittes mit konkurrierendem Zugriff

$$t_{sim} = 9.5(\log \tilde{N})^2 + 11.5 \log \tilde{N} + 20$$

Befehle.

```
for pnum ∈ {0, ..., Ñ − 1} pardo
  for i := 1 to log Ñ do
    for k := i − 1 to 0 do
      if bit k of pnum = 0 then
        if bit i of pnum = 0 then
          A[pnum] := min(A[pnum], A[pnum + 2^k])
        else
          A[pnum] := max(A[pnum], A[pnum + 2^k])
        fi
      else
        if bit i of pnum = 1 then
          A[pnum] := min(A[pnum − 2^k], A[pnum])
        else
          A[pnum] := max(A[pnum − 2^k], A[pnum])
        fi
      fi
    od
  od
od;
```

Abbildung 5.1: Bitonischer Sortier Algorithmus

5.1.3 Analyse

Ein Benchmark, das auf Maschine D1 T Befehle benötigt, wird auf Maschine D2 $2T$ Befehle benötigen, da diese nur halb soviele virtuelle Prozessoren hat wie D1. Sind von den T Befehlen auf D1 $\alpha \cdot T$ Befehle mit konkurrierendem Zugriff, so müssen bei $2\alpha \cdot T$ Befehlen auf D2 diese Zugriffe simuliert werden. Die Maschine D2 benötigt also insgesamt

$$T' = 2\alpha \cdot T \cdot t_{sim} + 2T(1-\alpha)$$

Befehle. Dies gilt allerdings nur, wenn der Compiler ermitteln kann, in welchen Schritten, konkurrierende Zugriffe vorkommen können. Kann er das nicht, so muß in jedem Schritt ein möglicher konkurrierender Zugriff simuliert werden. Die Zeit T' erhöht sich dann auf $T' = 2\alpha T t_{sim}$. Wir wollen zugunsten der EREW PRAM den ersten Fall annehmen. Da der Anteil α keine Konstante sein wird, soll er im weiteren mit $\alpha(n)$ bezeichnet werden.

Interessant ist nun zu wissen, für welche Werte von $\alpha(n)$ die Güte der Maschine D1 größer ist als die von D2, d.h. wie groß muß der Anteil an konkurrierenden Zugriffen sein, damit sich die Unterstützung der Kombinierung in Hardware auszahlt.

Wir wollen nun indirekt die Untersuchung führen und annehmen, daß $G_2 > G_1$.

$$\begin{aligned} G_2 &> G_1 \\ \frac{1}{c_2 T' t_2} &> \frac{1}{c_1 T t_1} \\ T' &< \frac{c_1 T t_1}{c_2 t_2} \\ \alpha(n) &< \frac{\frac{c_1}{c_2} - 1}{t_{sim} - 1} \end{aligned}$$

Asymptotisch erhält man eine Grenze von $\alpha(n) < 0.43/(\log \tilde{N})^2$, da c_1/c_1 wegen der Terme der Größenordnung $O(n \cdot N)$ asymptotisch gegen $1425.6/278.3 = 5.12$ strebt und t_{sim} durch den Term $9.5(\log \tilde{n})^2$ bestimmt wird. Der gesamte Term strebt dann gegen $(5.12-1)/(9.5(\log \tilde{n})^2) = 0.43/\log \tilde{n})^2$. Bei der Einschränkung $n \leq 27$ erhält man analog $\alpha(n) < 0.19/(\log \tilde{N})^2$. Der reale Verlauf ist in Abbildung 5.2 zu sehen. Hier ist zu sehen, daß $\alpha(n)$ bei $n = 27$ gerade $0.19/(\log \tilde{N})^2$ erreicht. Bei kleineren Werten für n liegt es sogar noch sehr darunter. Als Ergebnis des Abschnitts läßt sich festhalten, daß bereits für sehr kleine Anteile von konkurrierenden Zugriffen eine CRCW PRAM kosteneffektiver ist.

5.1.4 Einfluß auf CREW

Bei der Konstruktion einer EREW PRAM fällt auf, daß die Hardware Teile, die zur Unterstützung des konkurrierenden Schreibens dienen, eine Teilmenge der Hardware Teile

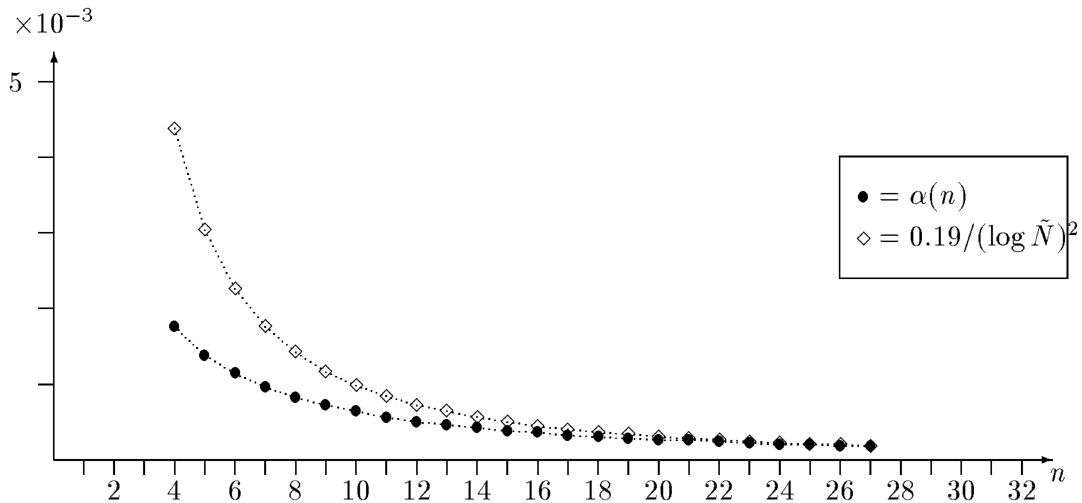

Abbildung 5.2: Minimal notwendiger Anteil von konkurrierendem Zugriff

bilden, die zum konkurrierenden Lesen benötigt werden. Möchte man lediglich das konkurrierende Lesen unterstützen, um eine CREW PRAM zu emulieren, so spart man keine Hardware, da alle Teile, die konkurrierendes Schreiben unterstützten, weiterhin zur Unterstützung des konkurrierenden Lesens benötigt werden. Andererseits spart man Hardware, wenn man sich auf die Unterstützung von konkurrierenden Schreibzugriffen beschränkt und so eine ERCW PRAM emuliert.

Bei der Betrachtung der Mächtigkeit der Berechnungsmodelle stellt man hingegen folgendes fest.

Definition 5.1 *Ein Berechnungsmodell A heißt hierarchisch schwächer als B ($A \preceq B$), falls jedes Problem, das in Modell A in Zeit T mit P Prozessoren gelöst werden kann, in Modell B in Zeit $O(T)$ mit $O(P)$ Prozessoren gelöst werden kann. Gilt $A \preceq B$ und gleichzeitig $B \not\preceq A$, so heißt A echt schwächer.*

Offensichtlich gilt $EREW \preceq CREW \preceq CRCW$.

Weiterhin gilt $CRCW \not\preceq CREW$.

Obwohl also das CREW Berechnungsmodell echt schwächer ist als CRCW, haben beide Emulationen gleiche Kosten. Hieraus folgt, daß bei unserer Art der Emulation es nicht sinnvoll ist, eine CREW PRAM zu emulieren.

Andererseits kann es Sinn machen, über die Emulation einer ERCW PRAM nachzudenken, da deren Emulation geringere Kosten hat als eine CRCW PRAM. Da allerdings das ERCW Modell unseres Wissens bisher nicht benutzt wurde, wird hierzu keine Untersuchung durchgeführt.

5.2 Unterstützung paralleler Präfixberechnung

5.2.1 Definition und Eigenschaften

Bereits in Kapitel 2 wurde das Problem der parallelen Präfixberechnung eingeführt. RANADE führt sie in folgender Form ein [RBJ88]:

Seien $P_{i_1}, \ldots, P_{i_j}$ mit $i_k < i_{k+1}$ eine Menge von Prozessoren, die in einem Schritt die Instruktion MP $a, \circ, d_{i_k}$ ausführen $(1 \leq k \leq j)$. Hierbei sei a eine Adresse des globalen Speichers, $\circ$ ein assoziativer binärer Operator und d_{i_k} ein Datum, das P_{i_k} mitgibt. Sei ferner der Inhalt der Zelle a zu Beginn des Schritts $S(a) = d$. Dann ist am Ende des Schritts der Inhalt von Zelle a $S(a) = d + \sum_{k=1}^{j} d_{i_k}$. Prozessor P_{i_l} erhält am Ende des Schritts den Wert $d + \sum_{k=1}^{l-1} d_{i_k}$ zurück. Das Summenzeichen wurde hierbei zur Abkürzung der mehrfachen Anwendung des Operators $\circ$ benutzt.

RANADE erlaubt weiterhin, daß mehrere disjunkte Prozessorgruppen dieses Kommando gleichzeitig mit verschiedenen Adressen benutzen und nennt die Operation daher *Multipräfix*.

Die Realisierung von Multipräfix funktioniert ähnlich wie das Kombinieren von Paketen bei konkurrierendem Zugriff:

In den Phasen 1 und 2 treffen sich die Pakete, die zum gleichen Präfix gehören. In der Richtungsqueue wird notiert, daß zwei Pakete des gleichen Präfix kombiniert wurden und in welche Richtung das resultierende Paket ging. Bei der Kombinierung zweier Pakete mit Daten d_1, d_2 hat das resultierende Paket das Datum $d_1 \circ d_2$, in der Richtungsqueue wird sich zusätzlich d_1 gemerkt. Die FIFO Schlange zum Speichern der Zwischenergebnisse wird im folgenden *MP–Speicher* genannt.

Kehrt in den Phasen 5 oder 6 die Antwort einer solchen Kombinierung zurück, so wird genau wie bei einem konkurrierenden Lesen die Antwort verdoppelt. Hat die Antwort das Datum d_3 und war in der Richtungsqueue d_1 gemerkt, so erhält das Antwortpaket für den linken Ausgang das Datum d_3, das Antwortpaket für den rechten Ausgang das Datum $d_1 \circ d_3$.

Durch vollständige Induktion läßt sich beweisen, daß bei dieser Vorgehensweise die obige Definition erfüllt wird. Diese Methode funktioniert nur für das Butterfly Netzwerk, jedoch nicht bei topologisch äquivalenten Netzwerken. Außerdem ist bei dieser Realisierung von paralleler Präfixberechnung auch klar, daß ohne zusätzliche Voraussetzungen mehrere Präfixe gleichzeitig berechnet werden können.

5.2.2 Hardware für Multipräfix

Wie in Abschnitt 5.2.1 gezeigt wurde, werden Operationen eines Multipräfix überall dort ausgeführt, wo die Pfade teilnehmender Pakete sich treffen und wo sie sich auf dem Rück-

weg wieder trennen. Dies bedeutet, daß Hardware zur Unterstützung von Multipräfix in jedem Netzknoten und jedem Sortierknoten in Hin- und Rücknetz zur Verfügung gestellt werden muß. Da durch die eingeschränkte Länge der Sortierfelder nicht alle teilnehmenden Pakete zu einem Paket kombiniert werden, sondern bis zu z teilnehmende Pakete übrigbleiben, muß auch in allen Speichermodulen, ja sogar in allen Speicherbänken Hardware zur Unterstützung von Multipräfix zur Verfügung gestellt werden.

Die zur Verfügung gestellte Hardware besteht im wesentlichen aus den Auswertungseinheiten für die o verschiedenen Operatoren. Man betreibt alle Auswertungseinheiten gleichzeitig und parallel zum Adressvergleich. Das gewünschte Ergebnis wählt man mittels Multiplexern aus. In Netz- und Sortierknoten benötigt man noch die MP–Speicher, die parallel zu den Richtungsqueues angelegt werden und somit gleiche Länge haben. Die Breite der MP–Speicher ist die eines Wortes. Die Kosten für zusätzliche Kontrollogik sind klein und werden vernachlässigt.

Benutzen wir als Operatoren Addition, bitweises UND, bitweises ODER und Maximum, so benötigen wir als Auswertungseinheiten einen 32 bit Carry Lookahead Addierer–Subtrahierer zum Berechnen der Summe bei Addition oder der Differenz bei Maximumbildung, 64 Gatter für bitweise Funktionen und vier 32 bit Multiplexer. Von diesen Einheiten brauchen wir $2N$ im Butterfly Netzwerk, $\frac{4c}{z}N$ in den Sortierfeldern und N in den Speicherbänken. Die MP–Speicher bestehen aus einer $(k', \log m)$–FIFO Schlange, wir brauchen N im Butterfly Netzwerk und $\frac{2c}{z}N$ in den Sortierfeldern. An Kosten für MP–Speicher müssen nur Kosten für SRAMs berechnet werden, da die Kontrolle der Richtungsqueue mitbenutzt werden kann. Die Kosten für Addierer, Multiplexer und SRAM wurden bereits in Kapitel 3.3 bestimmt. Die Kosten zur Hardwareunterstützung von Multipräfix belaufen sich damit auf

$$\begin{aligned} \mathrm{c}(\mathtt{MP}) &= (3+\frac{4c}{z})N\cdot\rho_A\cdot(\mathrm{c}(\mathtt{ADDSUB}_{\log m})+32\mathrm{c}(\mathtt{AND})+32\mathrm{c}(\mathtt{OR})+4\mathrm{c}(\mathtt{MUX}_{32})) \\ &\quad +(1+\frac{2c}{z})N\cdot\rho_K\cdot(\mathrm{c}(k'\times\log m-\mathrm{RAM})) \\ &= N(8784+4752n) \end{aligned} \tag{5.5}$$

Der Delay eines Netzwerkknotens wird nun statt durch den Vergleicher durch den Addierer bestimmt, außerdem müssen zwei Multiplexer zusätzlich passiert werden. Der Delay eines Netzwerkknotens erhöht sich damit um

$$\mathrm{d}(\mathtt{ADDSUB}_{\log m})-\mathrm{d}(\mathtt{VGL}\log m)+2\mathrm{d}(\mathtt{MUX}) = 2\log\log m+5 = 15$$

und beträgt damit 45 gate delays. Die Zugriffszeit auf den Speicher erhöht sich ebenfalls bei einem MP–Paket. Zuerst wird der bisherige Inhalt der betreffenden Speicherzelle ausgelesen. Dies benötigt Zeit t_m wie in Abschnitt 4.3 berechnet. Danach wird dieser Inhalt als Antwort zurück ins Netzwerk geschickt Gleichzeitig wird der Inhalt und der mit dem Paket eingetroffene Wert verknüpft und das Ergebnis wieder abgespeichert. Die Verzögerung

durch die Verknüpfung beträgt $t_c = \mathrm{d}(\mathtt{ADDSUB}_{\log m}) + 2\mathrm{d}(\mathtt{MUX}) = 4 \log\log m + 7$, da maximal der Addierer und 2 Multiplexer zur Auswahl des Ergebnisses durchlaufen werden müssen. Zusätzlich muß wieder die Zeit t_b zum Durchlaufen des Treiber– und ODER–Baumes hinzugerechnet werden. Der Zugriff eines Paketes auf den Speicher benötigt damit Zeit $t_m + t_b$ und ist genauso schnell wie vorher. Allerdings ist das Speichermodul noch um die Zeit $t_c + t_m$ zum Berechnen und Abspeichern des neuen Inhaltes der Speicherzelle blockiert, während es bei LOAD Befehlen direkt von neuen Paketen betreten werden konnte. Insgesamt dauert also die Benutzung des Speichers bei einem MP–Zugriff $t_b + t_c + 2t_m = 123 - 3n - \log n$, was maximal 4 Zyklen entspricht.

Durch die zusätzliche Blockierung von Speichermodulen um 2 Zyklen könnte es zu Rückstaus im Netzwerk kommen, so daß bei Dauerbetrieb der Erwartungswert der Netzwerkverzögerung steigt. Dies würde zu einem neuen Wert von c und eventuell zu einem neuen Wert von z führen.

Um dies zu überprüfen, müssen die in Abschnitt 4.3 für LOAD–Befehle durchgeführten Untersuchungen für MP–Befehle wiederholt werden. Die Untersuchungen führen zu dem Ergebnis, daß die Verzögerungszeit im Netzwerk um den Faktor 1.5 erhöht wird [Eng92]. Da zusätzlich die Netzwerkknoten um den Faktor 1.5 langsamer wurden, ergibt sich $\tilde{c} = 2.25 \cdot c$. Der Wert von z bleibt unverändert.

Die Zeit für einen Schritt der Maschine erhöht sich damit auf

$$t_3 = 2\tilde{c}n \cdot 45 = 405n \tag{5.6}$$

Durch den veränderten Wert $\tilde{c} = 6.75$ erhöhen sich die Kosten der Prozessoren um die zusätzlichen Registersätze. Die Gesamtkosten der Maschine D3 mit Unterstützung von Multipräfix betragen damit

$$c_3 = N \cdot (78466.8 + 6177.6n) + 2^n \cdot 30436.5 \tag{5.7}$$

5.2.3 Hardware–MP versus Software–MP

Um zu untersuchen, ob die Unterstützung von Multipräfix durch Hardware sinnvoll, d.h. kosteneffektiv ist, vergleichen wir sie mit einer Berechnung durch Software. Wir werden auch hier zugunsten der simulierenden Maschine D1 die Annahme machen, daß der Compiler ermitteln kann, wann Multipräfix Instruktionen ausgeführt werden. Die Simulation von Multipräfix in Software funktioniert ähnlich wie die Simulation von konkurrierendem Zugriff. Die Prozessoren schreiben ihre Pakete als Tupel (Prozessornummer, Adresse, Datum, Modus) in ein Feld, das Feld wird nach Adressen und bei gleichen Adressen nach den Zugriffsmodi sortiert. Alle Pakete eines Präfix stehen jetzt in benachbarten Zellen. Die Prozessoren, die diese Zellen verwalten, führen jetzt das Programm aus Kapitel 2, Beispiel 2 aus. Wir werden die Zeit hierfür vernachlässigen, da sie abhängig von der Anzahl der

Prozessoren eines Präfix ist. Die Zeit zur Simulation eines Multipräfix Schrittes beträgt demnach t_{sim}.

Wir berechnen nun analog zu Abschnitt 5.1.3 einen minimalen Anteil $\beta(n)$ von Multipräfix Schritten, ab dem eine Unterstützung in Hardware kosteneffektiver ist. Benötigt unser Programm auf Maschine D3 T Schritte, so braucht es auf Maschine D1 $(\tilde{c}/c) \cdot T = 2.25 \cdot T$ Schritte, da diese weniger Prozessoren hat. Bei einem Anteil von $\beta(n)$ Multipräfix Befehlen ist die Anzahl der Befehle auf D1

$$T' = 2.25 \cdot T(\beta(n)(t_{sim} - 1) + 1)$$

Hierbei beträgt $t_{sim} = 9.5(\log(cN))^2 + 11.5\log(cN) + 20$ wenn wir zugunsten von D1 annehmen, daß die Anzahl der Prozessoren cN eine Zweierpotenz ist.

Damit ergeben sich folgende Güten

$$G_1 = \frac{1}{c_1 \cdot T' \cdot t_1} \tag{5.8}$$

$$G_3 = \frac{1}{c_3 \cdot T \cdot t_3} \tag{5.9}$$

Damit ergibt sich in Analogie zu Abschnitt 5.1.3

$$\beta(n) > \frac{\frac{c_3}{c_1} - 1}{t_{sim} - 1} \Rightarrow G_3 > G_1$$

Der genaue Verlauf von $\beta(n)$ ist in Abbildung 5.3 zu sehen. $\beta(n)$ ist im Bereich $4 \leq n \leq 27$ kleiner als $0.22/(\log(cN))^2$, erreicht diese Schranke aber erst für große n. Der minimal notwendige Anteil von Multipräfix Befehlen, ab dem sich eine Unterstützung in Hardware lohnt, ist also sehr klein.

5.2.4 Einschränkungen

Die Unterstützung von Multipräfix in Hardware hat sehr hohe Kosten. Deshalb soll hier eine Einschränkung vorgestellt werden. Die Instruktion SYNC hat die gleichen Parameter wie MP, allerdings erhalten hier die Prozessoren keine Teilsummen zurück. SYNC berechnet also nur in der Zelle mit Adresse a die Verknüpfung $S(a) \circ d_{i_1} \circ \ldots \circ d_{i_l}$.

Als unterstützende Hardware benötigt SYNC nur die Auswerter der unterstützten Operationen in Phase 1 und 2 und auf den Speichermoduln. Die zusätzliche Hardware der Phasen 5 und 6 und die MP–Speicher werden nicht mehr benötigt.

Damit hat die zusätzliche Hardware die Kosten

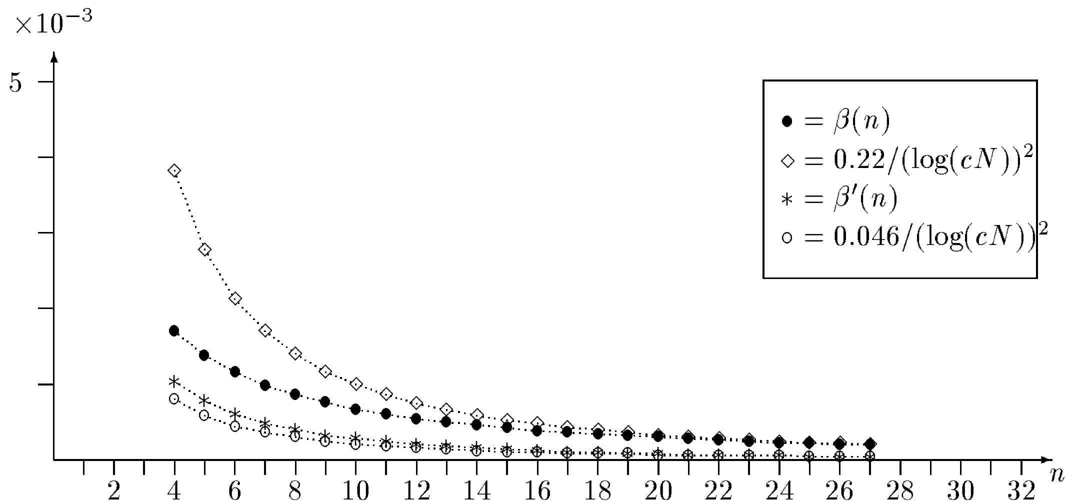

Abbildung 5.3: Minimal notwendiger Anteil von Multipräfix und Sync

$$\begin{aligned} \mathrm{c}(\texttt{SYNC}) &= (2+\frac{2c}{z})N\cdot\rho_A\cdot(\mathrm{c}(\texttt{ADDSUB}_{\log m})+32\mathrm{c}(\texttt{AND})+32\mathrm{c}(\texttt{OR})+4\mathrm{c}(\texttt{MUX}_{32})) \\ &= 5124N \end{aligned} \tag{5.10}$$

Die größere Verzögerung der Netzwerkknoten gilt zwar nur noch für die Phasen 1 und 2, wird aber für alle Phasen beibehalten, um eine synchrone Arbeitsweise des Netzwerks zu gewährleisten. Damit erhält man $t_4 = t_3$, die Konstante c ist auch hier zu $\tilde{c} = 6.75$ erhöht. Die Gesamtkosten der Maschine D4 betragen damit

$$c_4 = N\cdot(74806.8+1425.6n)+2^n\cdot 30436.5$$

Ersetzt man in der Rechnung des vorigen Abschnitts c_3 durch c_4 so erhält man ein verändertes $\beta'(n)$, das ebenfalls in Abbildung 5.3 zu sehen ist. $\beta'(n)$ erreicht einen Wert von $0.046/(\log(cN))^2$ und weicht auch nicht sehr von ihm ab. Die Einschränkung auf SYNC Befehle verringert also den minimal notwendigen Anteil von Befehlen um einen Faktor von etwa 4.8.

5.2.5 Anwendungen von Multipräfix und Sync

Die Befehle Paralleles oder Multipräfix und SYNC tauchen in vielen Algorithmen nicht auf. In diesem Abschnitt soll verdeutlicht werden, daß bei Programmen, die durch Compiler generiert werden, die beiden Befehle auch in diesen Fällen von Wichtigkeit sind.

Der Befehl SYNC wird zur Synchronisation von Prozessoren benötigt, der Befehl MP zur

Speicherverwaltung und dynamischen Allokierung. Weitere Anwendungen sind denkbar. Eine kurze Beschreibung dieser Anwendung wird auch in [ADK+91] gegeben.

Synchronisierung von Prozessoren

Sowohl FORK als auch Sprachen mit PARDO Konstrukten verlangen synchrone Ausführung von Hochsprachen Statements. Die Laufzeit des Maschinenprogramms für eine solches Statement ist aber oft nicht zur Compilezeit bestimmbar. Außerdem können Statements wie IF Anweisungen dazu führen, daß auf verschiedenen Prozessoren verschiedener Code, zum Beispiel THEN und ELSE Teil, ausgeführt wird. In diesen Fällen ist es notwendig, die Prozessoren zu synchronisieren, das heißt, die schnelleren Prozessoren solange aufzuhalten, bis die langsameren Prozessoren fertig sind.

Wir nehmen an, daß in einer Gruppe von Prozessoren, die jetzt synchron ist, später aber synchronisiert werden muß, jeder Prozessor die Adresse a einer Synchronisierungszelle im globalen Speicher kennt. Zuerst schreibt jeder Prozessor 0 in diese Zelle, dann führt jeder Prozessor SYNC $a, +, 1$ aus. a enthält jetzt die Anzahl der Prozessoren der Gruppe. Jeder Prozessor führt jetzt das Maschinenprogramm aus, dessen Laufzeit nicht bekannt ist. Danach führt er SYNC $a, +, -1$ aus. Dann liest er solange den Inhalt von a aus, bis dieser 0 ist. Jeder Prozessor weiß nun, daß alle anderen Prozessoren der Gruppe ebenfalls fertig sind, die Gruppe ist wieder synchron.

Bei dieser einfachen Version kann es allerdings passieren, daß in einem Schritt einige Prozessoren ein SYNC $a, +, -1$ ausführen, andere hingegen im gleichen Schritt den Inhalt von a lesen. Dabei könnten einige der lesenden Prozessoren Werte ungleich 0 lesen, einige könnten Werte gleich 0 lesen. Um dies zu vermeiden, vereinbaren wir, daß bei Code zum Synchronisieren nur in Schritten mit gerader Nummer SYNC Befehle ausgeführt werden, LOADs hingegen nur in Schritten mit ungerader Nummer. Um dies zu gewährleisten, erweitern wir unser Prozessor Status Wort um ein Modulo Flag, das seinen Wert nach jedem Befehl ändert. Ein bedingter Sprung kann jetzt auch vom Wert dieses Flags abhängen.

Nehmen wir an, daß der Wert der Adresse a im Register R_a gespeichert ist, so hat der Code zum Synchronisieren folgende Form:

(1) ... alle Prozessoren erreichen diesen Punkt zur gleichen Zeit

(2) STORE $R_a, R0, R0$

(3) SYNC $R_a, +, 1$

(4) ... code für das Hochsprachen Statement

(5) ... Prozessoren könnten diesen Punkt zu verschiedenen Zeiten erreichen

(6) JMP modulo clear PC,PC,R0

(7) SYNC $R_a, -, 1$

(8) LOAD $R_a, R0, R0$

Zeit	P_1		P_2	
t	(7)	SYNC		
$t+1$	(8)	LD		
$t+2$	(9)	NOP	(7)	SYNC
$t+3$	(10)	NOP	(8)	LD
$t+4$	(11)	JMP	(9)	NOP
$t+5$	(8)	LD	(10)	NOP
$t+6$	(9)	NOP	(11)	JMP
$t+7$	(10)	NOP	(12)	...
$t+8$	(11)	JMP		
$t+9$	(12)	...		

Abbildung 5.4: Synchronisierung von Prozessoren

(9) NOP

(10) NOP

(11) JMP zero clear $PC, PC, \# - 3$

(12) ... alle Prozessoren erreichen diesen Punkt zur gleichen Zeit

Der minimale Aufwand zum Synchronisieren ist dann erreicht, wenn alle Prozessoren Zeile (5) zur gleichen Zeit erreichen. Der Aufwand beträgt dann 9 Instruktionen: (2), (3), (6), (6), (7), (8), (9), (10), (11). Die NOP Instruktion in Zeile (9) ist notwendig, weil die LOAD Instruktion in Zeile (8) delayed ist. Die zweite NOP Instruktion in Zeile (10) sorgt dafür, daß in der Schleife der Befehle (8) bis (11) eine gerade Anzahl von Befehlen durchlaufen wird. Dies ist notwendig, um zu gewährleisten, daß Zeile (8) immer in einem ungeraden Zeitschritt ausgeführt wird.

Die Prozessoren laufen in Zeile (12) allerdings noch nicht vollständig synchron. Es kann passieren, daß ein Teil der Prozessoren Zeile (12) zu einer Zeit erreicht, der Rest der Prozessoren hingegen erreicht Zeile (12) erst zwei Schritte später. Der Grund dafür liegt darin, daß die Schleife zur Synchronisierung die Länge 4 Befehle hat, Prozessoren die Zeile (7) aber zu Zeiten t und $t+2$ erreichen können. In diesem Fall entsteht die in Abbildung 5.4 dargestellte Situation. Der „schnellere“ Prozessor P_1 lädt in Zeile (8) einen Wert, der ungleich Null ist, da der „langsamere“ Prozessor P_2 seinen SYNC Befehl der Zeile (7) noch nicht ausgeführt hat. Deshalb springt P_1 in Zeile (11) zurück zu (8), während P_2, der den Wert Null in Zeile (8) gelesen hat, nicht springt und Zeile (12) so zwei Schritte vor P_1 erreicht.

Diese Lücke kann geschlossen werden, indem eine zweite Synchronisierung erfolgt. Prozessoren, die die Zeile (12) erreichen führen einen SYNC $x, +, -1$ aus, wobei der vorige Inhalt von x gleich der Anzahl der Prozessoren war. Danach laden sie den Inhalt von x und vergleichen ihn mit Null. Dieser Test scheitert für die schnelleren Prozessoren und ist erfolgreich für die langsameren. Scheitert der Test, führt man 2 NOP's aus. Der Aufwand für die

vollständige Synchronisierung ist nun 16 Instruktionen: 9 Instruktionen für den ersten Teil, 7 Instruktionen für den zweiten Teil.

Jede Synchronisierung benötigt also zumindest 3 SYNC Befehle. Ist also nur alle 5000 Befehle eine Synchronisierung vonnöten, so lohnt sich schon die Hardware Unterstützung des vorigen Abschnitts.

Dynamisches Allokieren von Speicher

In einem Parallelrechner, der einen globalen Speicher zur Verfügung stellt, ist das dynamische Allokieren dieses Speichers nicht trivial, da mehrere Prozessoren gleichzeitig versuchen könnten, Speicher zu allokieren.

Eine einfache Lösung besteht darin, eine Semaphore zu benutzen, so daß stets nur ein Prozessor Speicher allokieren kann. Die Allokierung kann dann genau wie in sequentiellen Rechnern erfolgen, die Maschine kann allerdings dadurch verlangsamt werden.

Eine einfache parallele Lösung, die paralleles Präfix benutzt, hält in Zelle 0 des globalen Speichers (allgemein darf dies auch eine beliebige Zelle sein) einen Zeiger auf den Anfang des freien Speichers. Wollen nun mehrere Prozessoren P_i, $i \in I$ Speicherbereiche der Größen $m(i)$ allokieren, so führen sie die Instruktion MP $0, +, m(i)$ aus. Hat Zelle 0 zu Beginn dieses Schrittes den Inhalt $S(0) = a$, so erhält Prozessor P_i den Wert $a + \sum_{j \in I, j<i} m(j)$ zurück, der neue Inhalt der Zelle 0 ist $a + \sum_{j \in I} m(j)$.

Jeder Prozessor erhält also einen Zeiger auf einen Speicherbereich geeigneter Größe, Zelle 0 enthält vorher und auch nachher einen Zeiger auf den Beginn des freien Speichers. Diese einfache Methode benötigt nur einen Schritt. Die Rückgabe von Speicherbereichen funktioniert allerdings nur dann korrekt, wenn die Rückgabe genau in umgekehrter Reihenfolge wie die Allokierung erfolgt.

Bei Verwendung der Sprache FORK ist dies für die meisten Zwecke ausreichend. Die gegenwärtige Implementierung des FORK Compilers teilt den verfügbaren freien Speicher bei einer Aufteilung der Prozessoren in mehrere Gruppen zu gleichen Teilen auf diese Gruppen auf. Dies erfolgt zum Beispiel bei geschachtelten PARDO oder FORK Konstrukten. Innerhalb ihres Stück Speichers kann jede Gruppe nun die obige Strategie anwenden und braucht, wenn nicht unbedingt notwendig, gar keinen Speicher freizugeben. Nach dem Ende des innersten geschachtelten PARDO bzw. FORK werden die Speicherstücke aller Gruppen wieder vereint und neu vergeben. Der allokierte Speicher innerhalb einer Gruppe ist damit implizit wieder freigegeben. Beim Vereinen der Gruppen ist die richtige Reihenfolge garantiert, da alle Teilgruppen gleichzeitig nach der Synchronisierung vereint werden.

Soll eine kompliziertere Reihenfolge von Allokierungs– und Freigabevorgängen realisiert werden, so kann man ebenfalls eine einfache Lösung mithilfe von Multipräfix finden, sofern man die Speichergröße bei der Allokierung auf Vielfache einer einheitlichen Blockgröße B beschränkt. Sei die Größe des Speichers m ein Vielfaches der Blockgröße B. Dann ist die

maximale Anzahl der Blöcke $max = m/B$.

Die Zellen 1 bis max des globalen Speichers enthalten Zeiger auf Blöcke im freien Speicher, die Zelle 0 enthält max. Wollen Prozessoren $P_i, i \in I$ Speicher der Größe $m(i)$ Blöcke allokieren, so führen sie die Instruktion MP $0, +, -m(i)$ aus. Prozessor P_i erhält dann einen Zeiger auf die Zelle $x = max - \sum_{j \in I, j<i} m(j)$. Die Zellen $x, \ldots, x - m(i) + 1$ enthalten die Zeiger auf die von P_i allokierten Blöcke. Die Zelle 0 enthält nun mit $max - \sum_{j \in I} m(j)$ einen Zeiger auf den letzten noch gültigen Zeiger auf einen freien Block.

Zum Freigeben von Speicher der Größe $m(i)$ Blöcke führt P_i die Instruktion MP $0, +, m(i)$ aus. Er erhält dann einen Zeiger auf eine Zelle x und schreibt in die Zellen $x + m(i) - 1, \ldots, x$ die Zeiger auf die Blöcke, die er freigeben will.

Damit das Verfahren korrekt funktioniert, muß man verhindern, daß gleichzeitig Speicher allokiert und freigegeben wird. Hierzu kann kann man das Modulo Flag, das bei der Synchronisierung eingeführt wurde benutzen. Man legt fest, daß Allokierungen nur in Befehlen mit gerader Nummer, Freigaben nur in Befehlen mit ungerader Nummer ausgeführt werden. Das Problem, daß trotz Benutzung des Modulo Flags eine „Verzahnung“ wie bei der Synchronisierung auftauchen kann, existiert hier nicht.

Das gerade vorgestellte Verfahren verwaltet die Zeiger auf freie Blöcke in Form eines Stacks. Verwaltet man die Zeiger in Form einer FIFO Schlange, so kann man die Einschränkung durch die Benutzung de Modulo Flags fallenlassen.

Pro Allokierung und pro Freigabe ist eine Multipräfix Operation notwendig.

5.3 Simulation großer Prozessorzahlen

Wie bereits in Kapitel 2 geschildert, benutzen PRAM Programme Prozessorzahlen in Abhängigkeit von der Problemgröße. Eine feste Maschine hat allerdings nur konstant viele Prozessoren. Hat man P virtuelle Prozessoren zur Verfügung und benötigt man p Prozessoren, so simuliert jeder Prozessor p/P sogenannte *logische Prozessoren*. Ein logischer Prozessor wird repräsentiert durch den Zustand aller seiner benutzten Register einschließlich Programmzähler, Stackzeiger und Statusregister, dem sogenannten Kontext. Unter der Annahme, daß jeder logische Prozessor gleichviel Zeit t benötigt, braucht die Maschine Zeit tp/P.

Zusätzliche Zeit wird nur benötigt, wenn ein Prozessor der Maschine zwischen der Simulation zweier verschiedener Kontexte wechselt. Sind die Kontexte völlig unabhängig voneinander, so muß jeder Prozessor dies nur $p/P - 1$ mal ausführen. Oft sind solche Wechsel aber häufiger vonnöten. Als Beispiel diene folgende Situation: Alle logischen Prozessoren lesen aus einem Feld im globalen Speicher. Danach schreiben sie in dieses Feld. Der Feldindex beim Schreiben hängt vom gelesenen Wert ab.

Um eine korrekte Ausführung zu garantieren, müssen bei der Simulation erst alle Kon-

texte das Programmstück „Lesen aus dem Feld“ abarbeiten, danach erst können sie das Programmstück “Schreiben in das Feld“ abarbeiten. Obiges Beispiel entstammt einem Algorithmus zur parallelen Berechnung von Zusammenhangskomponenten eines ungerichteten Graphen [SV82].

Ähnliche Probleme existieren auch bei sequentiellen und parallelen Rechnern mit Multiuser und/oder Multitasking Betrieb. Dort sind zwar die meisten Kontexte unabhängig bis auf die gemeinsame Nutzung von Ein– und Ausgaberessourcen wie Drucker oder Scanner, die Verteilung der Rechenzeit auf die verschiedenen Kontexte sprich Benutzer oder Aufgaben ist aber schwieriger, da sie nicht nur von den Abhängigkeiten innerhalb eines Problems abhängt. Ein Benutzer möchte nicht 40 Sekunden warten, bis ein eingegebener Satz auf dem Bildschirm erscheint, nur weil ein anderer gerade gerade eine 40 Sekunden dauerndes Programm gestartet hat (Multiuser). Ein Benutzer, der einen Ausdruck startet, möchte trotz des Druckvorganges weiterhin Eingaben am Bildschirm machen können (Multitasking).

Außerdem gilt hier die obige Annahme nicht, daß jeder Kontext gleichviel Zeit braucht. Deshalb muß hier die Verteilung der Kontexte auf reale Prozessoren noch die gleichmäßige Auslastung aller Prozessoren in Betracht ziehen.

Da wir nur den Fall der PRAM Programme betrachten wollen, werden wir folgendes im weiteren voraussetzen:

- Jeder Prozessor simuliert gleichviele Kontexte. Die genaue Verteilung der Kontexte auf die Prozessoren spielt keine Rolle.
- Die Abhängigkeiten zwischen den Kontexten sind problemabhängig und können vom Compiler zur Compilezeit erkannt werden. Weitere Abhängigkeiten liegen nicht vor. Der Kontextwechsel kann deshalb explizit im Maschinenprogramm durch einen Befehl realisiert werden, es brauchen keine impliziten Wechsel durch Ablauf von Zeitscheiben (time slices) betrachtet zu werden.

5.3.1 Kontextwechsel

Ein Kontextwechsel besteht aus dem Abspeichern von 32 Registerinhalten (statt des Registers R0 wird das Statusregister gespeichert) und aus dem Laden von 32 neuen Registerinhalten. Ohne Unterstützung dauert er also mindestens 64 Befehle.

Die Unterstützung von Kontextwechseln kann nun in zwei Richtungen erfolgen: zum einen kann man versuchen, durch die Einführung spezieller Befehle zum Laden und Speichern aller Register und durch die Erweiterung der Datenpfade des Prozessors den Kontextwechsel zu beschleunigen. Zum anderen kann man versuchen, durch Bereitstellung mehrerer Registersätze pro Prozessor ein Speichern und Laden von Registern überflüssig zu machen, so daß ein Kontextwechsel nur noch aus dem Umschalten zwischen verschiedenen Registersätzen besteht. Ein dritter Ansatz besteht in einer Kombination beider Verfahren.

Der erste Ansatz hat den Vorteil, daß er kaum zusätzliche Kosten verursacht, da zusätzliche Datenpfade nur geringe Kosten für zusätzliche Treiber, Multiplexer und Kontrolle haben. Zusätzlich können damit soviele Kontexte unterstützt werden, wie im globalen Speicher Platz finden, wobei die Größe des globalen Speichers sowieso eine obere Schranke für die berechenbare Problemgröße setzt. Bei einer maximalen Speichergröße von 2^{32} Worten und einer Reservierung von 50% des Speicherplatzes für Kontexte könnten insgesamt $2^{31}/32 = 2^{26}$ Kontexte gespeichert werden, jeder virtuelle Prozessor unserer Maschine D1 könnte also $2^{26}/(cN)$ logische Prozessoren simulieren. Der Nachteil des Ansatzes liegt darin, daß beim Kontextwechsel eines virtuellen Prozessors die Pipeline angehalten werden muß und deshalb der Kontextwechsel lange dauert, wie wir bei der folgenden Analyse sehen werden. Ein zweiter Nachteil des Verfahrens liegt darin, daß es nur bei Maschinengrößen mit $cN < 2^{25}$, also für $n < 20$ anwendbar ist, da sonst der Hauptspeicher nicht ausreicht.

Der zweite Ansatz hat den Vorteil, daß kein Platz im globalen Speicher zur Abspeicherung von Kontexten benötigt wird und daß ein Kontextwechsel jetzt tatsächlich nur einen Befehl dauert. Der Nachteil besteht darin, daß die zusätzlichen Registersätze eventuell zu große Kosten verursachen und daß deswegen das Preis/Leistungsverhältnis schlecht wird.

Der dritte Ansatz soll die Geschwindigkeit des zweiten Ansatzes beibehalten, die Kosten aber drücken. Wir wollen im folgenden alle 3 Verfahren beschreiben.

5.3.2 Unterstützende Hardware für schnellen Kontextwechsel

Wir werden zur Realisierung des ersten Ansatzes nur den Fall des Schreibens betrachten. Der Fall des Ladens erfolgt analog. Um schnelles Abspeichern eines Kontextes zu unterstützen, muß man garantieren, daß nacheinander alle Register eines virtuellen Prozessors aus dem Registerfeld ausgelesen werden können, daß gleichzeitig nacheinander 32 aufeinanderfolgende Adressen erzeugt und gehasht werden können und daß Daten und gehashte Adressen als Pakete ins Netzwerk gelangen können.

Dies ist die Erweiterung eines normalen STORE Befehls. Um die zusätzlichen Register zu lesen, muß die Pipeline der virtuellen Prozessoren unterbrochen werden, da das Registerfeld sonst in den nächsten Takten bereits für die Zugriffe der folgenden virtuellen Prozessoren vorgesehen ist. Ist die Pipeline unterbrochen, kann der Multiplizierer einschließlich seiner internen Pipeline zum Hashen der Adressen benutzt werden. Die Adressen können durch ein Register mit rückgekoppeltem Inkrementierer — eine Anordnung wie bei der Realisierung des Programmzählers — erzeugt werden, wenn die zuerst berechnete Adresse $R_x + S_2$ in dieses Register geschrieben wird. Da in jedem Takt nur 1 Adresse gehasht werden kann, muß die Pipeline der virtuellen Prozessoren 31 Takte lang angehalten werden. Zum Auslesen der Register hätten 16 zusätzliche Takte genügt, da das Registerfeld zwei Ports hat.

Das Laden von Registern funktioniert ähnlich. Zur Erzeugung der Pakete, die zum Laden benötigt werden, geht man vor wie beim Schreiben und benötigt ebenfalls 31 zusätzliche Takte. Kommen die Antworten der Pakete zurück, müssen sie ins Registerfeld geschrieben

werden. Auch dies dauert 31 zusätzliche Takte, da normalerweise nur ein Paket zurückkommt. Auch hier kann nur ein Port des Registerfeldes benutzt werden, da die Pakete alle nacheinander zurückkommen. Ein Zwischenspeichern der Pakete und Benutzen beider Ports ist nicht möglich, da in den dadurch zwischenzeitlich ablaufenden Takten der normalen Pipeline eventuell Pakete erwartet würden, die nicht zurückkommen können, da die Pakete des Kontextwechsels noch teilweise im Netzwerk stecken. Käme ein Prozessor deswegen zum Stillstand, so befände sich die Maschine in einem Deadlock.

Die Unterstützung des Kontexwechsels eines virtuellen Prozessors auf diese Art und Weise benötigt also $3 \cdot 31 = 93$ zusätzliche Takte. Führt jeder virtuelle Prozessor eines physikalischen Prozessors einen solchen Kontextwechsel durch, so entstehen $93cn$ zusätzliche Takte. Da ein normaler Schritt der Maschine D1 $t_1 = 2cn$ Takte benötigt, so dauern die zusätzlichen Takte so lange wie 47 zusätzliche Befehle des Prozessors. Der Kontextwechsel dauert also insgesamt 48 Befehle statt vorher 64.

5.3.3 Unterstützende Hardware für mehrere Kontexte

Um x Kontexte pro virtuellem Prozessor zu unterstützen, benötigt man ein Registerfeld mit $x \cdot cn$ Registersätzen für die normalen Register, drei mit $x \cdot cn$ 32–bit Registern für PC, SP_0, SP_1 und eines mit $x \cdot cn$ 4–bit Registern für das Statusregister ST.

5.3.4 Kombination beider Ansätze

Um die hohen Kosten des zweiten Ansatzes zu senken, stellt man pro virtuellem Prozessor nur wenige Registersätze zur Verfügung. Da jetzt eventuell mehr logische Prozessoren benötigt werden, als Registersätze vorhanden sind, müssen Kontexte wie im ersten Ansatz gespeichert und geladen werden. Während auf einem Registersatz gearbeitet wird, ist es möglich, einen anderen zu speichern oder zu laden. Im Gegensatz zum ersten Ansatz sichert man die Kontexte aber nicht in den Hauptspeicher, sondern in einen speziellen lokalen *Kontextspeicher*. Dieser hat geringere Kosten als die Registersätze, da der Kontextspeicher im Gegensatz zu den Registersätzen nicht als Dual Port RAM aufgebaut werden muß. Wir stellen für jeden virtuellen Prozessor zwei Registersätze zur Verfügung. Auf einem der beiden wird gearbeitet. Dieser ist der aktuelle Registersatz. Bei einem Kontextwechsel wird der aktuelle Registersatz gespeichert, gleichzeitig wird der andere Registersatz mit dem nächsten auszuführenden Kontext geladen und wird somit zum aktuellen Registersatz.

Wollen alle virtuellen Prozessoren im gleichen Schritt einen Kontextwechsel ausführen, der Nachladen erfordert, und soll der Kontextwechsel so schnell wie alle anderen Befehle sein, so hat man zum Laden und zum Speichern eines Kontextes jeweils nur einen Takt Zeit, da alle zwei Takte die Ausführung des nächsten virtuellen Prozessors beginnt. Im folgenden werden die Datenpfade beschrieben, die notwendig sind, Kontexte schnell zwischen Registersätzen und Kontextspeichern hin und her zu verschieben.

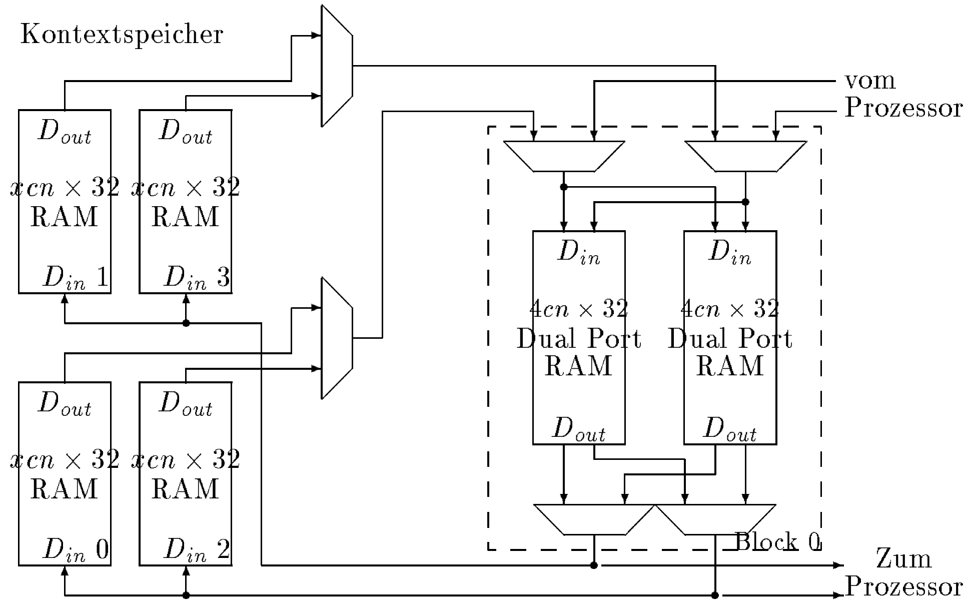

Abbildung 5.5: Datenpfade zum Wechseln von Kontexten

Um einen schnellen Kontextwechsel durchführen zu können, zerteilen wir die Registersätze in 8 Blöcke. Block $i \in \{0, \ldots, 7\}$ enthält für jeden Registersatz die Register $R_{4i}, \ldots, R_{4i+3}$. Jeder Block besteht aus 2 Dual Port RAMs. Jedes RAM enthält von jedem virtuellen Prozessor nur einen Registersatz. Der Kontextspeicher besteht aus 32 RAMs, von denen RAM $j \in \{0, \ldots, 31\}$ von jedem Kontext nur Register j enthält. Bei einem Kontextwechsel speichert im ersten Takt der aktuelle Registersatz aus jedem Block die ersten beiden Register, gleichzeitig werden vom neuen Kontext in jedem Block die letzten beiden Register geladen. Im nächsten Takt werden die übrigen beiden Register geladen bzw. gesichert.

Findet kein Kontextwechsel statt, wird in jedem Block das RAM, das gerade den aktuellen Registersatz enthält, angesprochen. Aus den 8 Blöcken wird mittels Multiplexern beim Lesen von Registern selektiert. Beim Schreiben erhält nur ein Block ein Schreibsignal.

Die Datenpfade dieser Schaltung für einen Block sind in Abbildung 5.5 zu sehen.

5.3.5 Analyse

Die zusätzlichen Kosten für den ersten Ansatz sind sehr gering, sie betragen

$$\mathrm{c}(\log m - \mathrm{REG}) + \rho_A \cdot (\mathrm{c}(\log m - \mathtt{MUX}) + \mathrm{c}(\log m - \mathtt{INC})) = 929.25$$

Die Gesamtkosten für die veränderte Maschine D5 betragen dann

$$c_5 = N \cdot (39097.8 + 1425.6n) + 2^n \cdot 31366 \tag{5.11}$$

Die Zeit für einen Schritt der Maschine beträgt $t_5 = t_1$, wenn kein Kontextwechsel ausgeführt wird. Hat ein Programm einen Anteil γ von Kontextwechseln an der Anzahl der Schritte T, so beträgt die Laufzeit auf Maschine D5 $t_5 \cdot T(48\gamma + (1-\gamma))$.

Die zusätzlichen Kosten beim zweiten Ansatz sind sehr groß, zur Unterstützung von x Kontexten pro virtuellem Prozessor betragen sie bei Kosten 24468 pro Kontext

$$24468 \cdot (x-1) \cdot N$$

Die so veränderte Maschine D6 hat dann Kosten

$$c_6 = N \cdot (24468x + 14629.8 + 1425.6n) + 2^n \cdot 30436.5 \tag{5.12}$$

Ein Schritt dieser Maschine dauert $t_6 = t_1$. Obiges Programm hat Laufzeit $T \cdot t_6$, unabhängig vom Anteil der Kontextwechsel.

Die zusätzlichen Kosten des kombinierten Ansatzes sind ebenfalls recht hoch. Zur Realisierung der Registersätze benötigt man pro physikalischem Prozessor 16 Dual Port RAMs der Größe $4cn \times 32$. Der Kontextspeicher benötigt pro physikalischem Prozessor 32 RAMs der Größe $xcn \times 32$. Zur Verbindung der beiden und zum Anschluß an die Datenpfade des Prozessors benötigt man $8 \cdot 6 + 2 \cdot 7 = 62$ Multiplexer. Es entfällt das bisherige Dual Port RAM. Packt man die RAMs mit ρ_G und die Multiplexer mit ρ_A, dann betragen die zusätzlichen Kosten

$$N \cdot (22862.7 + 11427.8x) + 2^n \cdot 8974.5$$

Die so veränderte Maschine D7 hat damit Gesamtkosten

$$c_7 = N \cdot (61960.5 + 11427.8x + 1425.6n) + 2^n \cdot 39411 \tag{5.13}$$

Ein Schritt der Maschine D7 dauert $t_7 = t_1$, unabhängig vom Anteil der Kontextwechsel.

Die Maschinen D6 und D7 sind beide gleichschnell. Abbildung 5.6 zeigt, daß für $x \geq 4$ D7 eine größere Güte hat als D6. Wir werden deshalb D6 nicht mehr weiter untersuchen.

Bisher wurden stets zwei Maschinen miteinander verglichen und dazu in Abhängigkeit der Maschinengröße n ein minimaler Anteil, beispielsweise $\beta(n)$ an „ zusätzlichen“ Befehlen, z. B. Multipräfix Befehlen berechnet, ab dem eine Maschine eine größere Güte als die andere erreicht.

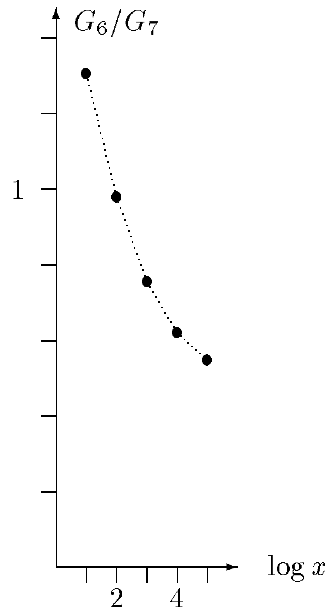

Abbildung 5.6: Quotienten der Gütenfunktionen von D7 und D6 in Abhängigkeit von x

Hier vergleichen wir drei Maschinen, deshalb ist es nicht ausreichend, einen minimalen Anteil γ von Kontextwechseln zu errechnen, ab dem D5 bzw. D7 eine höhere Güte als D1 erreichen. Der Wert von γ könnte nämlich zwar für D5 und für D7 gleich sein, D7 könnte aber bei höheren Anteilen von Kontextwechseln eine größere Güte als D5 erreichen und wäre dann vorzuziehen. Diese Information läßt sich aber nur aus dem minimalen Anteil nicht gewinnen. Außerdem ändert sich der notwendige Anteil γ nicht mit der Maschinengröße.

Deshalb berechnen wir die Güten G_5/G_1 und G_6/G_1 für $x = 4, 8, 16, 32$ bei verschiedenen Anteilen γ von Kontextwechseln an der Laufzeit. Da die Quotienten sich für verschiedene n nur minimal ändern, wählen wir $n = 7$, der der Größe des geplanten Prototyps. Abbildung 5.7 zeigt die Quotienten für verschiedene Werte von γ.

Die Kurve für D5 bewegt sich schon für minimale γ nahe der 1, entfernt sich aber nicht sehr weit (bis etwa 1.3). Dies bedeutet, daß D5 nicht viel besser wird als D1.

Die Kurven für D6 steigen erst für recht große γ (etwa bei 0.01 für $x = 8$) über die 1. Dann allerdings steigen sie recht hoch. Dies bedeutet, daß D6 erst für relativ große Anteile von Kontextwechsel an der Laufzeit besser wird als D1, dann aber sehr viel besser. Deswegen sollte man Maschine D7 wählen, wenn genügend oft Kontextwechsel stattfindet. Daß dies der Fall ist, zeigt Abschnitt 5.4.3.

Ein Vergleich der beiden Maschinen für kleinere γ zeigt deutlich, daß der erste Ansatz um so besser ist, je größer x wird. Bei genauerer Betrachtung zeigt sich aber auch, daß das Bewertungsmaß hier ungerecht gegenüber dem zweiten Ansatz ist. Im ersten Ansatz werden große Teile des Hauptspeichers zum Kontextwechsel benutzt. Diesen Teil des Hauptspei-

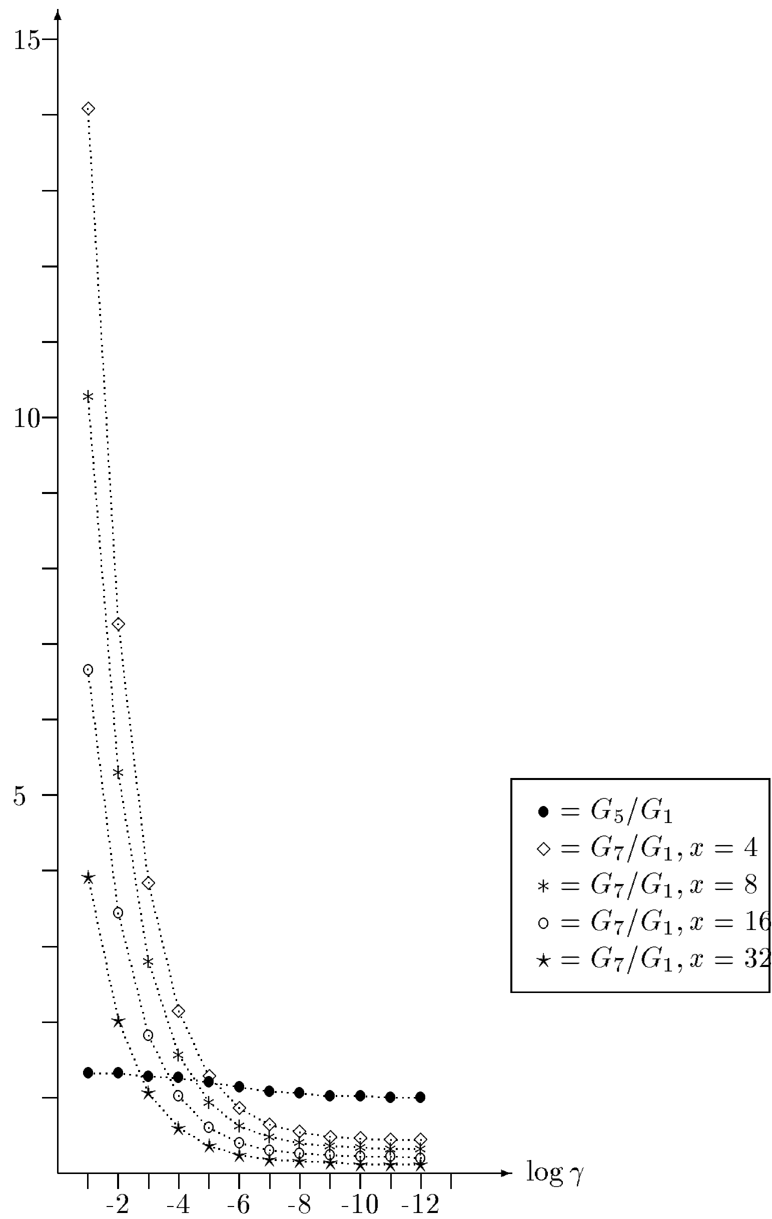

Abbildung 5.7: Quotienten der Gütenfunktionen

chers könnte man im zweiten Ansatz weglassen, da er für globale Daten nicht genutzt werden kann. Da aber der Hauptspeicher im Modell nicht mit Kosten belastet ist, bedeutet dies keine Ersparnis. Bei allen vorhergehenden Untersuchungen war dies nicht von Belang, da jede Maschine den gleichen Hauptspeicher zur Verfügung hatte.

In der Realität wird das Ergebnis also besser für die Lösungen D6 und D7 aussehen.

Aufgrund dieser Überlegungen erscheint es sinnvoll, die Alternative D7 für $x = 16$ oder $x = 8$ zu wählen, da nur dort ein Gewinn an Kosteneffektivität bei moderatem γ zu erhalten ist. Eventuell sollte die Alternative D5 zusätzlich eingebaut werden, um in Fällen, wo x logische Prozessoren nicht ausreichen, eine Ausweichmöglichkeit zu haben.

5.4 Bewertung am Beispiel

In den vorhergehenden Abschnitten wurden minimal notwendige Anteile von bestimmten Befehlen an der Laufzeit eines Programmes errechnet, die eine Unterstützung dieser Befehle durch Hardware rechtfertigen. Um festzustellen, wie die Anteile dieser Befehle in realen Programmen sind, wurde ein Benchmark ausgewählt, handcompiliert und analysiert. Als Benchmark wurde ein Algorithmus zum Berechnen der Zusammenhangskomponenten eines ungerichteten Graphen gewählt [SV82]. Da dieser Algorithmus allerdings keine Multipräfix oder SYNC Befehle benutzt, wurde zusätzlich das Programm mit einem Compiler für die Sprache PARDO compiliert [F+92] und auf einem Simulator [San90] die Ausführung simuliert. Dieses Vorgehen liefert die Anzahl von Multipräfix und SYNC Befehlen, die im Laufzeitsystem des Compilers und im Betriebssystem des Simulators ausgeführt werden.

Da weder das handcompilierte Programm noch der Compiler Kontextwechsel einsetzen, konnte die Häufigkeit von Kontextwechseln nur geschätzt werden.

Im folgenden wird zuerst der Algorithmus kurz beschrieben, dann die Anteile von concurrent read und write am handcompilierten Programm bestimmt und mit den Simulationsergebnissen verglichen. Danach wird die notwendige Anzahl von Kontextwechseln geschätzt.

5.4.1 Berechnung von Zusammenhangskomponenten

Der verwendete Algorithmus stammt von Shiloach und Vishkin [SV82], er benötigt Zeit $O(\log|V|)$ und $O(\max(|V|,|E|))$ Prozessoren bei einem ungerichteten Graphen $G = (V, E)$ mit $|V|$ Knoten und $|E|$ Kanten. Da ein sequentieller Algorithmus das Problem in Zeit $O(|V| + |E|)$ lösen kann [Meh84], ist dieser parallele Algorithmus nicht optimal. Durch seine einfache Datenstruktur erscheint er dennoch für die Praxis besser geeignet als ein optimaler Algorihtmus aus [Hag90]. Der Algorithmus benötigt eine CRCW PRAM, eine Lösung dieses Problems auf einer EREW PRAM, die nicht lediglich eine CRCW PRAM emuliert, ist nicht bekannt.

Der Graph wird repräsentiert durch 2 Abbildungen $head, tail : E \to V$. Hierbei geben $head(i)$ und $tail(i)$ die beiden Knoten an, die durch die Kante i verbunden sind. Beide Abbildungen werden durch Felder dargestellt. Berechnet wird eine Abbildung $F : V \to \{0, \ldots, |V| - 1\}$, wobei $F(u) = F(v)$ für $u, v \in V$ mit $u \neq v$ dann und nur dann wenn u und v zur gleichen Zusammenhangskomponente gehören.

Das verwendete Programm ist in Abbildung 5.8 zu sehen.

5.4.2 Analyse und Messungen

Das hier verwendete Benchmark wurde handcompiliert, der Maschinencode ist in [AKP91b] abgedruckt. In [AKP90] wird eine Analyse des Codes vorgenommen. Dabei erhält man auf einer PRAM mit P Prozessoren und bei einem Graphen mit $|V|$ Knoten und $|E|$ Kanten das folgende Ergebnis:

Befehl	Anzahl	Anteil
LOAD	$(40 \cdot \lvert V\rvert \log \lvert V\rvert + 104 \cdot \lvert E\rvert \log \lvert V\rvert)/P$	0.353
STORE	$(24 \cdot \lvert V\rvert \log \lvert V\rvert + 84 \cdot \lvert E\rvert \log \lvert V\rvert)/P$	0.265
COMPUTE	$(44 \cdot \lvert V\rvert \log \lvert V\rvert + 112 \cdot \lvert E\rvert \log \lvert V\rvert)/P$	0.382

Im Fall $|E| = |V|$ erhält man die Anteile wie in der rechten Spalte der obigen Tabelle angegeben.

Hier haben Zugriffe auf den globalen Speicher einen sehr hohen Anteil, fast alle dieser Zugriffe im Algorithmus können konkurrierend sein. Hier lohnt sich die Hardware Unterstützung von Multipräfix.

Das mittels PARDO Compiler übersetzte Programm wurde für $P = 4$ und $P = 8$ Prozessoren bei einem Graphen mit $|V| = 8$ und $|E| = 8$ simuliert. Die kleinen Werte bei Prozessoranzahl und Graphgröße mußten aus Rechenzeit– und Speicherplatzbeschränkungen gewählt werden.

Hier ergaben sich die folgenden Anteile:

Befehl	Anteil $P = 4$	Anteil $P = 8$
LOAD	0.198	0.252
STORE	0.075	0.074
MP/SYNC	0.042	0.043
COMPUTE	0.685	0.631

Durch die Einbindung eines Laufzeitsystemes und eines kleinen Betriebssystems wird der Anteil der COMPUTE Befehle fast verdoppelt. Die Anzahl der Zugriffe auf den globalen

```
       for u ∈ V pardo F[u] := u; od;
       for t := 1 to 2 log |V| do
          for u ∈ V pardo change[u] := 0; od;
          starcheck;
          for alle (u, w) mit {u, w} ∈ E pardo
(1)          if star[u] and F[w] < F[u] then
                F[F[u]] := F[w];
                change[F[u]] := 1;
                change[F[w]] := 1;
             fi;
          od;
          starcheck;
          for alle (u, w) mit {u, w} ∈ E pardo
(2)          if star[u] and not change[F[u]]
                         and F[w] ≠ F[u] then
                F[F[u]] := F[w];
             fi;
(3)          F[u] := F[F[u]]
          od
       od.

proc starcheck ;
begin
   for i ∈ V pardo
      star[i] := true;
      if F[F[i]] ≠ F[i] then
         star[F[F[i]]] := false;
      fi;
      star[i] := star[F[F[i]]]
   od
end;
```

Abbildung 5.8: Programm zur parallelen Berechnung von Zusammenhangskomponenten

Speicher ist aber immer noch hoch genug, um eine Unterstützung durch Hardware zu rechtfertigen. Gleiches gilt für den Anteil von MP und SYNC Befehlen. Dieser ist mit 4.3% erstaunlich hoch, was nur auf seine häufige Verwendung im Betriebssystem zurückgeführt werden kann. Auch diese Befehle sollten durch Hardware unterstützt werden.

5.4.3 Abschätzung der Anzahl von Kontextwechseln

Betrachtet man eine Unterstützung von mehreren Kontexten pro Prozessor, so werden bei der Ausführung der ersten beiden PARDO Schleifen des Hauptprogramms und der PARDO Schleife der Prozedur starcheck jeweils $|V|/P$ Kontexte benötigt. Bei den beiden letzten PARDO Schleifen des Hauptprogramms werden $2|E|/P$ Kontexte pro Prozessor benötigt.

Wir wollen nun für jede PARDO Schleife die notwendige Anzahl von Kontextwechseln untersuchen. Können für alle Werte der parallelen Schleifenvariablen die Schleifenrümpfe unabhängig voneinander ausgeführt werden, so muß jeder Kontext nur einmal aufgerufen werden, die Anzahl von Kontextwechseln in der PARDO Schleife entspricht dann genau der Anzahl von Kontexten, die pro Prozessor für diese PARDO Schleife benötigt werden. Müssen hingegen die Schleifenrümpfe in $b > 1$ Teile zerlegt werden, so daß alle Kontexte zuerst Teil $i < b$ durchgeführt haben müssen, bevor einer der Kontexte Teil $i+1$ durchführen darf, dann benötigt man b mal Anzahl der Kontexte viele Kontextwechsel in der PARDO Schleife. Ein Grund für solche Aufteilungen sind Datenabhängigkeiten, z. B. beim Lesen und Schreiben in globalen Feldern. Ist einer der Teile für verschiedene Kontexte verschieden lang, so muß zusätzlich synchronisiert werden. Dies ist bei unserem Benchmark allerdings nicht notwendig.

In der ersten und der zweiten PARDO Schleife des Hauptprogramms wird jeder Kontext nur einmal gestartet. Hier ist die Anzahl der Kontextwechsel jeweils $|V|/P$.

In der dritten PARDO Schleife des Hauptprogramms muß darauf geachtet werden, daß beim Befehl $F[F[u]] := F[w]$ zuerst alle Kontexte $F[u]$ und $F[w]$ lesen und danach erst in das Feld F geschrieben wird. Hier wird jeder Kontext also zweimal aufgerufen. Die Anzahl von Kontextwechseln ist hier $4|E|/P$.

In der vierten PARDO Schleife des Hauptprogrammes tritt dieser Fall sogar zweimal auf, nämlich in den Befehlen $F[F[u]] := F[w]$ und $F[u] := F[F[u]]$. Zusätzlich muß hier sichergestellt werden, daß alle Kontexte zuerst den ersten der beiden Befehle vollständig ausgeführt haben und danach erst mit dem zweiten der Befehle beginnen. Hier wird jeder Kontext also viermal aufgerufen, die Anzahl von Kontextwechseln ist hier $8|E|/P$.

In der PARDO Schleife der Prozedur starcheck muß sichergestellt werden, daß die Kontexte den IF Befehl erst ausführen, wenn alle Kontexte den Befehl $star[i] := true$ ausgeführt haben. Weiterhin muß sichergestellt werden, daß alle Kontexte den IF Befehl ausgeführt haben, bevor sie den Befehl $star[i] := star[F[F[i]]]$ ausführen. Innerhalb dieses Befehls müssen alle Kontexte erst $star[F[F[i]]]$ lesen, bevor ein Kontext $star[i]$ schreiben darf. Hier

wird also jeder Kontext viermal aufgerufen, die Anzahl von Kontextwechseln ist hier $4|V|/P$.

Wir berechnen nun für jede PARDO Schleife, wie oft sie ausgeführt wird. Danach kann man einfach die Gesamtzahl der Kontextwechsel errechnen.

Die erste PARDO Schleife des Hauptprogramms wird einmal ausgeführt. Die anderen PARDO Schleifen des Hauptprogramms werden $2\log|V|$ mal ausgeführt. Die PARDO Schleife der Prozedur starcheck wird $4\log|V|$ mal ausgeführt.

Damit ergibt sich folgendes Bild:

PARDO Schleife	# Kontextw.	# Ausführungen	Gesamt
Hauptpr. 1	$\|V\|/P$	1	$\|V\|/P$
Hauptpr. 2	$\|V\|/P$	$2\log\|V\|$	$2\|V\|\log\|V\|/P$
Hauptpr. 3	$4\|E\|/P$	$2\log\|V\|$	$8\|E\|\log\|V\|/P$
Hauptpr. 4	$8\|E\|/P$	$2\log\|V\|$	$16\|E\|\log\|V\|/P$
starcheck	$4\|V\|/P$	$4\log\|V\|$	$16\|V\|\log\|V\|/P$
GESAMT	$(\|V\| + 18\|V\|\log\|V\| + 24\|E\|\log\|V\|)/P$		

Nimmt man wie im Fall des handcompilierten Programmes an, daß $|V| = |E|$, so erhält man einen Anteil an der Gesamtlaufzeit von 9.3%. Selbst wenn man annimmt, daß dieser Anteil durch Overhead in der Verwaltung der Kontexte auf die Hälfte reduziert wird, so ist er doch hoch genug, um eine Unterstützung von Kontextwechsel in Hardware zu rechtfertigen.

Kapitel 6

Entwurf eines Prototyps

In diesem Kapitel soll skizziert werden, wie aus den Designs der letzten Kapitel ein Prototyp entwickelt werden kann. Die vorgesehene Maschinengröße ist 128 physikalische Prozessoren mit je 32 virtuellen Prozessoren. Hierbei ist eine Schwierigkeit zu beachten:

Unser Modell geht davon aus, daß wir Chip–Flächen beliebiger Größe in einheitlicher Technologie zur Verfügung haben. Damit kann man das Design beliebig partitionieren. In der Realität ist dies nicht der Fall. Verfügbare Chipgrößen sind beschränkt, die verfügbare Größe hängt zudem noch von der verwendeten Technologie ab. Gleiches gilt für die absolute Geschwindigkeit der Chips, da die Schaltzeit eines Gatters von Technologie zu Technologie variiert.

Ein erster Schritt zur Entwicklung eines Prototyps muß also die Einbettung der verschiedenen Teile unserer Maschine in verschiedene Technologien sein und die Aufteilung der Teile auf einen oder mehrere Chips der betreffenden Technologie. Weiterhin muß man unterscheiden zwischen kommerziell erhältlichen Bausteinen *(Standard Chips)* und zwischen eigenentwickelten Chips *(Application Specific Integrated Circuit, ASIC)*. Wir werden dies in den ersten beiden Abschnitten dieses Kapitels für die Eigenentwicklungen Prozessor, Netz– und Sortierknoten tun. Die restlichen Teile können durch kommerziell erhältliche Chips, vorwiegend Speicherchips realisiert werden.

Eine zweite Schwierigkeit bringt die Verdrahtung unserer Maschine. Im Modell ist Verdrahtung kostenlos und kostet keine Zeit, in einer realen Maschine hingegen kann sie nicht vernachlässigt werden. Dies gilt insbesondere für unser Netzwerk. In Abschnitt 6.3 werden 2 Alternativen zur physikalischen Anordnung unserer Maschine vorgestellt.

Eine dritte Schwierigkeit besteht in der Berechnung der Kosten. Die Kosten einer realen Maschine hängen nicht nur von der Anzahl ihrer Gatter ab, sondern auch und in besonderem Maße von den verwendeten und verfügbaren Technologien. Zusätzlich entstehen Kosten für Chips durch Gelder für Verwaltung, Entwicklung etc. Die Kosten einer Maschine im formalen Modell liefern also eine untere Schranke für die Chipkosten.

Als letzter Punkt soll in Abschnitt 6.4 die im Design gar nicht angesprochene, in der Praxis aber sehr wichtige Verbindung zur Außenwelt zur Ein– und Ausgabe von Daten diskutiert werden.

6.1 Der Prozessor

Unser Prozessor soll sich in einem ASIC Chip befinden. Eine Besonderheit unseres Prozessors stellen allerdings die vielfachen Registersätze dar. Leider gibt es derzeit keinen Fertiger, der RAM Bereiche in der Größe, die wir benötigen, auf einem ASIC Chip anbietet. Deshalb werden alle Register außer PC und ST aus dem Chip in externes SRAM ausgelagert.

Der Grund, warum ausgerechnet PC und ST im Chip gehalten werden, liegt in der Häufigkeit des Zugriffs auf diese beiden Register. Auf beide muß in jedem Befehl zwei mal zugegriffen werden, einmal lesend beim Laden des nächsten Befehls bzw. um das Modulo Bit auszulesen, und einmal schreibend, beim Wegspeichern des inkrementierten PC bzw. des invertierten Modulo Bits. Hinzu kommen noch die eventuellen Zugriffe während der Ausführung eines Befehls. Die Anzahl dieser Zugriffe beträgt maximal vier. Diese Anzahl wird beim JSR Befehl erreicht, bei dem zwei Argument Register gelesen werden, zusätzlich der Stackzeiger gelesen und der inkrementierte Stackzeiger weggespeichert wird.

Weitere Verbindungen zum Rest der Maschine werden benötigt, um einen Befehl zu laden (Anlegen der Adresse und Einlesen des Opcodes) und um Lese– oder Schreibzugriffe an das Netzwerk zu leiten (Adresse und eventuell Datum). Der Prozessor Chip ist hierzu mit der Außenwelt über mehrere Busse verbunden. Wegen der beschränkten Anzahl von Pins des Prozessor Chips (299 einschließlich der Versorgungs Pins) kann nicht für jede Verbindung ein eigener 32 Bit breiter Bus realisiert werden. Andererseits wird aber auch nicht jede Verbindung in jedem Takt genutzt. Man kann in mehreren Fällen zwei Verbindungen auf einem physikalischen Bus zusammenfassen, die diesen dann abwechselnd nutzen. Allerdings ist darauf zu achten, daß beide Verbindungen die gleiche Orientierung (Eingang/Ausgang) besitzen, da die Nutzung von bidirektionalen Bussen schwieriger ist als bei unidirektionalen Bussen. Die genaue Anordnung der Zugriffe auf das externe Register RAM und die Verteilung aller Zugriffe auf die Busse ist in Abbildung 6.1 zu sehen. Die Breite des Register Adressbusses ergibt sich aus folgender Überlegung: Der Prozessor hat 32 virtuelle Prozessoren mit je 32 logischen Prozessoren (siehe unten), jeder Registersatz hat 32 Register. Die Adresse eines Registers benötigt also $\log_2(32 \cdot 32 \cdot 32) = 15$ Bit. Da in einem Takt 2 Registerinhalte geladen werden können (je eines auf Daten– und Registerbus), benötigt man insgesamt eine Breite von 30 Bit.

Durch die Auslagerung der meisten Register kann der Prozessor Chip für verschiedene Maschinengrößen genutzt werden. Für die externen Register können je nach Maschinengröße verschieden große RAMs eingesetzt werden. Die internen Register werden in einer maximalen Anzahl zur Verfügung gestellt, je nach Maschinengröße wird nur ein Teil genutzt. Allerdings würden beliebige Größen innerhalb des durch die internen Register vorgegebe-

Bus	Nutzung in		Breite und Richtung
	geradem Takt	ungeradem Takt	
Adressbus	Adresse für Opcode	Adresse von LOAD/STORE	32, aus
Register Adressbus	Adresse Register 1,2	Adresse Register 3	30, aus
Datenbus ein	Argument Register 1	Opcode	32, ein
Registerbus ein	Argument Register 2	Argument Register 3 (SP)	32, ein
Datenbus aus		Datum von STORE	32, aus

Abbildung 6.1: Auslastung der Busse vom und zum Prozessorchip

nen Bereichs das Design komplizieren. Die Anzahl der zur Verfügung gestellten virtuellen Prozessoren wird deshalb auf 16 bzw. 32 eingeschränkt. Die Reduzierung auf Zweierpotenzen vereinfacht das Design, bringt aber die Schwierigkeit, daß die Anzahl der virtuellen Prozessoren eventuell zu groß und damit nicht optimal an die Maschinengröße angepaßt ist. Antworten auf LOAD Befehle könnten „zu früh" zurückkommen, da mehr virtuelle Prozessoren als theoretisch notwendig vorhanden sind. Die Lösung dieses Problems wird unten beschrieben.

Jedem virtuellen Prozessor werden 32 Registersätze zum Kontextwechsel zur Verfügung gestellt. Dies entspricht nicht dem Ergebnis der Untersuchung im vorigen Kapitel, läßt sich aber durch ein Preisargument rechtfertigen. Während im theoretischen Modell ein Registersatz aller virtuellen Prozessoren ungefähr soviel kostet wie der restliche Prozessor, ist der Preis für ein schnelles statisches Dual Port RAM der Größe $1K \times 32$ Bit (32 virtuelle Prozessoren mit je 32 Registern von 32 Bit Breite) mit DM 30,– gegenüber dem Preis von DM 550,– pro Prozessor Chip (Serienfertigung ohne Kosten für die Entwicklung des Prototyps) sehr gering. Die Kosten für ein göẞeres statisches RAM sind also gering im Vergleich zum Aufwand, einen Prozessor mit komplizierteren Datenpfaden zu den Registern zu designen. Die Registersätze werden wie in Design D6 beschrieben zur Verfügung gestellt. Im formalen Modell ist D7 zwar etwas günstiger und wird deshalb favorisiert, in der Praxis wäre es allerdings teuerer, viele kleine RAM Chips statt weniger größerer zu kaufen.

Ein weiterer, bereits angesprochener, wichtiger Punkt ist die Betreuung der Antworten auf LOAD Befehle. Diese haben zwar eine nur geringe Varianz um den Erwartungswert, aber auch diese sollte abgefangen werden, um Staus durch „zu früh" zurückgekehrte Pakete, die die Sortiernetze der Phase 6 blockieren, zu vermeiden. Wie bereits angemerkt, könnten diese Staus auch dadurch entstehen, daß die eingeschränkte Anzahl von Registersätzen nicht genau dem optimalen Wert von cn entspricht. Andererseits muß aber garantiert werden, daß die Antworten delayed sind. Ansonsten kann der alte Inhalt des zu ladenden Registers nicht mehr im Befehl nach dem LOAD Befehl genutzt werden, was die Anzahl der einzufügenden NOPs vergrößern würde. Die Antwort eines LOAD Befehls darf also frühestens nach dem Laden der Argumente des nächsten Befehls in das Registerfeld geschrieben werden. Um dies zu garantieren, benutzt man die Kontrolle der LOAD Antworten.

Die Kontrolle der LOAD Antworten besteht im Wesentlichen aus einer FIFO Schlange.

Ein Eintrag der FIFO Schlange besteht aus der Nummer eines virtuellen Prozessors, der Nummer eines Zielregisters und einem Flag, ob überhaupt ein LOAD Befehl ausgeführt wurde. Die Länge der FIFO Schlange beträgt $2cn-4$ wie bereits in Abschnitt 4.3 errechnet, sie wird in jedem zweiten Takt geclockt. Die Kontrolle kann nun auf zwei Arten betrieben werden. Speist jeder virtuelle Prozessor die Schlange, auch wenn er keinen LOAD Befehl ausführt, so hat die Schlange stets die maximale Länge. Sie kann dann als Pipeline von $2cn-4$ Registern realisiert werden. Der geforderte Delay ist garantiert. Antworten, die zu früh zurückkommen, können allerdings nicht behandelt werden, da der Kontrolleintrag noch nicht am Ende der Pipeline angelangt ist.

Speisen nur die virtuellen Prozessoren die Schlange, die LOAD Befehle ausführen, so kann eine Antwort eines LOAD Befehls direkt behandelt werden, da keine störenden „Fülleinträge“ den Kontrolleintrag aufhalten. Die FIFO Schlange muß hier wie in Abschnitt 3.3.8 beschrieben realisiert werden. Der Delay kann allerdings nicht garantiert werden.

Die Lösung bringt eine Kombination beider Ansätze. Die FIFO Schlange des zweiten Ansatzes wird in zwei Teilen realisiert, die ersten cn Stufen als Pipeline, die letzten $cn-4$ Stufen wie in Abschnitt 3.3.8. Nur virtuelle Prozessoren, die LOAD Befehle ausführen, speisen die Schlange. Die Pipeline garantiert den Delay, da ein Kontrolleintrag frühestens nach $2cn$ Takten den zweiten Teil erreicht und dann erst am Ausgang erscheinen kann. Nach $2cn$ Takten sind aber die Argumente des nächsten Befehls schon geladen. Nach diesem Zeitpunkt kann eine Antwort direkt behandelt werden.

Eine sehr detaillierte Beschreibung des Prozessor Chips einschließlich der Anbindung des externen Register RAMs ist bei Scheerer [Sch92] im Rahmen einer Untersuchung über die Testbarkeit großer heterogener Schaltungen zu finden. Die Angaben über Busauslastung und Verwaltung der LOAD Befehle dieses Abschnitts sind dieser Untersuchung entnommen.

6.2 Sortier- und Netzknoten

In diesem Abschnitt soll eine Realisierung des Netzwerkknotens beschrieben werden und die Änderungen, die notwendig sind, um daraus einen Sortierknoten zu machen. War beim Prozessor Chip die Anzahl der verfügbaren Gatter das beschränkende Element, so werden dies beim Netzwerk Chip die Pins sein.

Pakete in Phase 2 des Netzwerkes bestehen aus einer 32 Bit Adresse, einem 32 Bit Datum und 5 Kontroll Bits. Pakete in Phase 5 des Netzwerkes bestehen aus einem 32 Bit Datum und einem Kontroll Bit. In beiden Phasen gibt es zusätzlich zu jedem Link noch eine Rückleitung, die Auskunft drüber gibt, ob der Eingangspuffer des Knotens am Ende des Links voll ist oder nicht. Um ein ganzes Paket in einem Takt transportieren zu können, benötigt jeder Link $w = 104$ Bit, $w_1 = 70$ für Phase 2, $w_2 = 34$ für Phase 5. Ein Netzwerk Chip, der einen Netzwerkknoten wie in Abschnitt 3.4.2 beschrieben realisiert und zusätzlich Multipräfix unterstützt, benötigt 15000 Gatteräquivalente und hat 4 solcher Links. Er

benötigt damit 416 Pins plus Spannungsversorgung.

Die größten uns bekannten kommerziell erhältlichen ASIC Chips besitzen 300 Pins bei 70000 Gattern (HDC105) oder 240 Pins bei 48000 Gattern (HDC064) [Mot89a]. Man könnte also in einem Chip die Schaltungen mehrerer Netzwerkknoten unterbringen, hat aber nicht genug Pins, um die Links eines Netzwerkknotens nach außen zu führen.

Um diese Schwierigkeit zu überwinden, kann man die Schaltung des Netzwerkknotens in 2 Teile partitionieren und diese in 2 Chips unterbringen. Diese Technik wird *bit slice* genannt. Eine geeignete Partitionierung ist die Aufteilung in obere Adress–/Datenbits und untere Adress–/Datenbits. Die Aufteilung in Adresse/Modus und Datum/MP–Operator ist nicht geeignet, da die Daten des Rücknetzes, um eine gleichmäßige Auslastung zu erhalten, auf beide Chips verteilt werden müssen. Sind jetzt die Daten des Hinnetzes in einem Chip, so muß die Hälfte des MP–Speichers zum anderen Chip transferiert werden, was zu mindestens 16 zusätzlichen Pins an jedem Chip führt.

Die Auslastung der Chipfläche ist allerdings sehr schlecht. Deshalb versucht man, mehrere Netzwerkknoten samt ihren Verbindungen in Chips unterzubringen und wegen der benötigten Pinzahl dann in mehrere bit slices zu schneiden. Dies führt allerdings nur zu einer besseren Auslastung, wenn die Netzwerkknoten so gewählt werden, daß die Anzahl der Links, die die Chips verlassen, nicht genauso schnell steigt wie die Anzahl der Knoten. Wir wählen deshalb $2^x \cdot (x+1), x = 0,1,2,3$ Netzwerkknoten in Form eines Butterflies. Im weiteren wird dieses kleine Butterfly Netz als *Makro* bezeichnet. Hier ist obige Forderung erfüllt, da 2^{x+2} Links ein solches Butterfly verlassen, die Anzahl der Links also langsamer ansteigt als die Knotenzahl.

Um eine minimale Anzahl von Chips für ein Butterfly Netzwerk mit $n+1$ Stufen zu je 2^n Knoten zu bestimmen, geht man wie folgt vor: Für alle zugelassenen Werte von x, hier $0 \leq x \leq 2$ (siehe nächsten Absatz), bestimmt man die minimale Anzahl $2^{i_1(x)}$ von Slices, die man benötigt, um ein Makro mit $2^x \cdot (x+1)$ Knoten zum implementieren. Nun bestimmt man die notwendige Anzahl dieser Makros für gegebenes n und x. Das Produkt der beiden Zahlen — Anzahl der Chips pro Makro und Anzahl der Makros pro Netzwerk — ergibt die notwendige Anzahl von Chips für gegebenes n und x. Gesucht ist für gegebenes n dasjenige x, das die Anzahl der Chips minimiert.

Um 2^{x+2} Links auf Pins zu verteilen, benötigt man mindestens 2^{x+1} Chips, da maximal 2 Links zu je 104 Bit an einen Chip passen. Da das Bit Slice Verfahren aber mit zunehmender Anzahl der Slices in der Praxis immer schwieriger zu handhaben ist (Design und Debugging), schränken wir die Anzahl der Slices auf 8 ein und erhalten aus der minimalen Anzahl der Chips $x \leq 2$.

Weiterhin muß man berücksichtigen, daß mit zunehmender Anzahl der Slices zusätzliche Verbindungen zwischen den Chips benötigt werden. Bei einem Knoten, der in 2^i Slices geteilt wird, werden folgende Verbindungen benötigt:

5 Pins, so daß jedes Slice Modus und Operator bei MP Paketen hat, 2 Pins, die den Adress

Shift realisieren, 1 Pin, über den allen Slices mitgeteilt wird, ob ein Paket verschickt wird, 1 Pin, über den allen Slices mitgeteilt wird, wohin das Paket verschickt wird und 2 Pins, die allen Slices mitteilen, welches Paket versandt wird oder ob kombiniert wird.

Weiterhin benötigt der Vergleicher auf Gleichheit 4 Pins pro Slice, um seine 2^i Teilbäume zusammenzufassen, der Vergleicher auf Vorzeichen benötigt 3 Pins, ebenso die beiden Addierer für Multipräfix. Weitere i Pins pro Slice werden benötigt, um jedem Chip seine Stellung innerhalb des Makros zu verdeutlichen.

Insgesamt benötigt man also $i + 24$ zusätzliche Pins pro Knoten und Slice.

Damit Addierer und Vorzeichenvergleicher allerdings mit 3 Pins pro Slice auskommen, müssen sie kommerziell erhältliche *Carry Generators* als zusätzliche Chips verwenden. Ein Beispiel für einen Carry Generator für 4 Slices ist der Chip 74F182 von Fairchild [Fai85]. Da die Anzahl der Slices auf 8 beschränkt ist, benötigt man pro Addierer/Vergleicher 3 solcher Generatoren, insgesamt also 9. Da diese Chips sehr klein im Vergleich zu den ASIC Chips sind, wird ihre Gesamtfläche als ein Chip gewertet. Errechnet man also eine Zahl 2^i als Anzahl von Slices für ein Makro, so ist die Anzahl der Chips für dieses Makro $2^i + \mathrm{sign}(i)$, da bei mehr als einem Slice ($i > 0$) die Carry Generatoren den Platz eines zusätzlichen Chips belegen.

Um ein Makro mit $2^x \cdot (x+1)$ Knoten in 2^i Slices zusammenzufassen, benötigt man also $2^{x+2} \cdot 104$ Pins für die Links und pro Knoten und Slice je $i + 24$ Pins. Die Gesamtzahl der Pins des Makros ergibt sich damit zu

$$pin_1(x,i) = 2^{x+2} \cdot 104 + (x+1) \cdot 2^x \cdot 2^i \cdot (i+24)$$

Pins. Für jeden Wert von x bestimmt man nun die minimal notwendige Anzahl von Slices, um ein Makro mit $2^x \cdot (x+1)$ Knoten zu implementieren. Hierzu bestimmt man für jedes x die kleinste ganzzahlig positive Lösung i der folgenden Ungleichung, falls die Ungleichung für dieses x überhaupt eine ganzzahlig positive Lösung hat.

$$pin_1(x,i) \leq 2^i \cdot \#\ \text{Pins pro Chip} \tag{6.1}$$

Man definiert $i_1(x)$ als die so gefundene Lösung i bei gegebenem Wert x, falls diese für x existiert, für andere x ist $i_1(x)$ undefiniert. Wählt man als verwendeten Chip den HDC064 und setzt als Anzahl der Pins 240 ein, so erhält man $i_1(0) = 1, i_1(1) = 3$. Für größere Werte von x hat die Ungleichung keine ganzzahlige positive Lösung i, da die linke Seite mit $i \cdot 2^i$ und die rechte nur mit 2^i wächst.

Die Anzahl der Chips, die notwendig sind, um ein Butterfly Netzwerk mit 2^n Eingängen und $n+1$ Stufen durch Makros mit $(x+1) \cdot 2^x$ Knoten zu implementieren, laßt sich jetzt einfach bestimmen. Man benötigt $\lceil (n+1)/(x+1) \rceil$ Stufen von Makros, in jeder Stufe benötigt man 2^{n-x} Makros. Jedes Makro besteht aus $2^{i_1(x)} + \mathrm{sign}(i_1(x))$ Chips. Die Gesamtzahl der Chips beträgt dann

n	$chip_1(0,n)$	$chip_1(1,n)$	$chip_2(1,n)$
4	240	216	160
5	576	432	400
6	1344	1152	960
7	3072	2304	2240
8	6912	5760	5120
9	15360	11520	11520
10	33792	27648	25600
11	73728	55296	56320
12	159744	129024	122880
13	344064	258048	266240
14	737280	589824	573440
15	1572864	1179648	1228800
16	3342336	2654208	2621440

Tabelle 6.1: Benötigte Anzahlen von Netzwerkchips

$$chip_1(x,n) = \left\lceil \frac{n+1}{x+1} \right\rceil \cdot 2^{n-x} \cdot (2^{i_1(x)} + \mathrm{sign}(i_1(x)))$$

Tabellierte Werte für $x = 0,1$ und $n \in \{4,\ldots,16\}$ sind in Tabelle 6.1 zu sehen. Hier ist deutlich zu sehen, daß für alle n gilt $chip_1(0,n) \geq chip_1(1,n)$. Die Lösung $x = 1$ ist hier also vorzuziehen. Die rechte Spalte, die noch günstigere Chipzahlen zeigt, ist das Ergebnis einer Beobachtung, die im nächsten Abschnitt beschrieben wird.

Um die Ausnutzung der Chipfläche zu verbessern und so die Anzahl der benötigten Netzwerkchips und damit auch die Anzahl der Links zwischen diesen zu verringern, versucht man, Netzwerkknoten günstig zu partitionieren und auf Chips zu verteilen. Ein erster Ansatz ist das Bit Slice Verfahren, das allerdings wegen der zusätzlich benötigten Pins nur für 2 Slices funktioniert. Ein zweiter Ansatz beruht auf einer Beobachtung, die gemeinsam mit Abolhassan, Drefenstedt, Paul, Scheerer in [ADK+91] kurz beschrieben wurde.

Betrachtet man den Schaltplan eines Netzwerkknotens wie in Abbildung 6.2 dargestellt, so stellt man fest, daß trotz zweier Eingangs– und Ausgangslinks die gestrichelte Schnittlinie nur einen Link kreuzt. Dies ist keine Eigenschaft des Butterfly Netzwerkes, sondern eine Eigenschaft von Ranade's Routing Algorithmus. Die beiden Pakete, die einen Netzwerkknoten in einem Schritt verlassen, unterscheiden sich nur im Modus; das eine Paket hat den Modus GHOST (siehe hierzu auch Abschnitt 2.3.4). Deshalb benötigt man am Ausgang des Knotens nur einen Link, der dann verdoppelt wird und wo nach der Verdoppelung auf einer Seite der Modus GHOST eingefügt wird. In der „unteren" Hälfte des Knotens befinden sich nur die beiden Multiplexer, die dies bewirken. Sie werden nur über die Zielrichtung des Paketes gesteuert und benötigen deshalb keine Kontrollsignale, die die Schnittlinie kreuzen

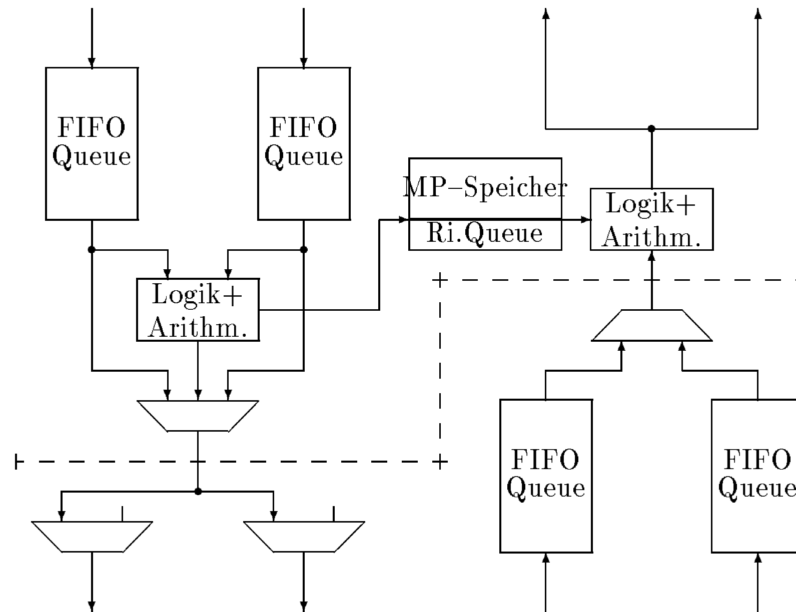

Abbildung 6.2: Schnitt durch Netzwerkknoten

müßten. Die einzigen Signale, die die Schnittlinie kreuzen müssen, sind eine Zusammenfassung der Frei–Signale der beiden Eingangspuffer an den Enden der beiden Ausgangslinks im ersten Netzwerk und ein Kontrollsignal in der Rückrichtung. Ein Link, der die Schnittlinie kreuzt, hat also eine Breite von $69 + 34 + 2 = 105$ Bit.

Damit kann man nun mittels dieser Beobachtung eine Aufteilung in Chips folgendermaßen vornehmen. Als Makro dient uns ebenfalls ein Butterfly mit 2^x Eingängen und $x + 1$ Stufen. Von den Knoten der ersten Stufe benutzt man allerdings nur die untere Hälfte, von den Knoten der letzten Stufe benutzt man nur die obere Hälfte. Damit gilt $x \geq 1$. Ein Beispiel für $x = 1$ ist in Abbildung 6.3 zu sehen. Damit enthält ein Makro $x \cdot 2^x$ Knoten, da die Knotenstücke der ersten und letzten Stufe gerade zusammen 2^x Knoten ergeben. Die Anzahl der Links, die das Makro verlassen, ist 2^{x+1}, da an jedem Eingangs– und Ausgangsknotenstück nur ein Link liegt. Bei etwa gleicher Knotenzahl hat sich also die Anzahl der Links halbiert. Um ein solches Makro in Chips unterzubringen, geht man wie oben vor und verteilt mittels Bit Slice Verfahren auf Slices. Die Anzahl der Pins ist hier bei gegebenem x und 2^i Slices

$$pin_2(x, i) = 2^{x+1} \cdot 105 + x \cdot 2^x \cdot 2^i \cdot (i + 24)$$

Die minimale Anzahl von Slices soll in Analogie zum ersten Ansatz $2^{i_2(x)}$ betragen. Setzt man in Ungleichung 6.1 statt $pin_1(x, i)$ nun $pin_2(x, i)$ ein, dann definiert man für gegebenes x $i_2(x)$ als kleinste ganzzahlig positive Lösung i der Ungleichung, falls diese für x existiert. Man erhält nun $i_2(1) = 2$. Für größere Werte von x findet sich hier kein minimales i.

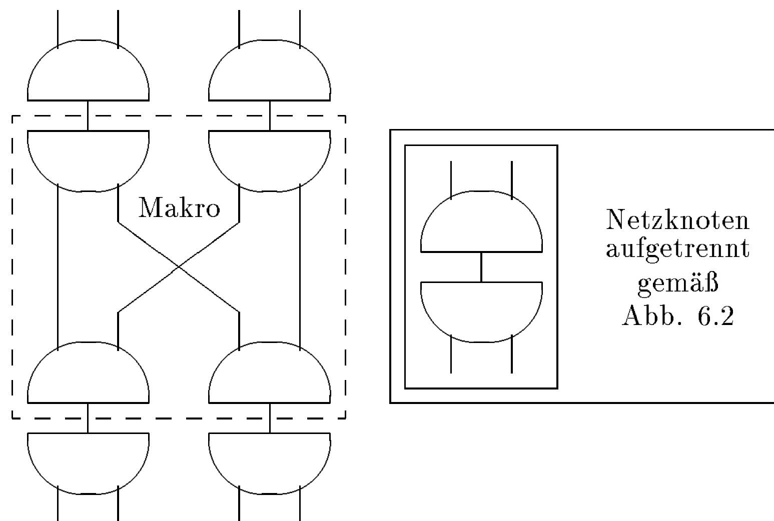

Abbildung 6.3: Aufteilung von Netzwerkknoten für $x = 1$

Die Anzahl der Chips, die notwendig sind, um ein Butterfly Netzwerk mit 2^n Eingängen und $n+1$ Stufen durch Makros mit $x \cdot 2^x$ Knoten zu implementieren, laßt sich jetzt einfach bestimmen. Man benötigt $\lceil n/x \rceil$ Stufen von Makros, da man die obere Hälfte der Knoten der ersten Stufe und die untere Hälfte der Knoten der $n+1$–ten Stufe nicht benötigt. In jeder Stufe benötigt man 2^{n-x} Makros. Jedes Makro besteht aus $2^{i_2(x)} + \mathrm{sign}(i_2(x))$ Chips. Die Gesamtzahl der Chips beträgt dann

$$chip_2(x, n) = \left\lceil \frac{n}{x} \right\rceil \cdot 2^{n-x} \cdot (2^{i_2(x)} + 1)$$

Werte für $x = 0$ und $n \in \{4, \ldots, 16\}$ sind in Abbildung 6.1, rechte Spalte, zu sehen. Die Werte hier sind günstiger als die Werte von $chip_1(x, n)$, allerdings nicht, wie erwartet, doppelt so günstig wie bei $chip(0, n)$, obwohl hier ein Makro bei gleicher Anzahl von Links die doppelte Anzahl von Knoten enthält. Dies liegt daran, daß beim zweiten Ansatz mit der doppelten Anzahl von Knoten auch die doppelte Anzahl von Kontrollsignalen anfällt. Für den Wert $i = 1$ hat die linke Seite der Gleichung 6.1 den Wert 466 im ersten Ansatz, 520 im zweiten Ansatz, die rechte Seite in beiden Fällen den Wert 480. Deshalb hat $i_2(1)$ den Wert 2 statt wie erwartet den Wert 1.

Ein weiterer wichtiger Punkt, den es zu beachten gilt, ist die Geschwindigkeit des Netzwerkes. Während man im theoretischen Modell für Prozessor und Netzknoten Zykluszeiten von 30 Gate Delays hat, so hat in der Realität der Prozessor eine Zykluszeit von $60ns$ wegen der ausgelagerten Registersätze [Sch92], der Netzwerkknoten benötigt allerdings nur $30ns$ Zykluszeit.

Beim Prozessor spielen die ausgelagerten Registersätze eine Rolle. Diese benötigen etwa $25ns$ für einen Zugriff, zu dieser Zeit muß allerdings noch die Zeit für die Verbindung vom Chip zum RAM und zurück addiert werden [Sch92]. Durch das wegen der Pinbeschränkung

x	$i'_1(x)$	$i'_2(x)$	$2^{i'_1(x)} + \text{sign}(i'_1(x))$	$2^{i'_2(x)} + \text{sign}(i'_2(x))$
0	0	–	1	–
1	2	0	5	1
2	5	2	33	5

Tabelle 6.2: Anzahl von Slices bei Halbierung der Links

notwendige Multiplexen der Busse verliert man hier zusätzlich Zeit, so daß man eine Zykluszeit von etwa $50ns$ erhält. Um eine synchrone Arbeitsweise von Prozessor und Netzwerk zu gewährleisten, wählt man $60ns$ als Zykluszeit.

Die Basistechnologie für beide Chips hat Gatterverzögerungen von etwa $1.4ns$ [Mot89a]. Bei Verwendung von Makrokomponenten für Arithmetik wie zum Beispiel 4–Bit Addierer erhält man aber günstigere Verzögerungszeiten als beim Aufbau der gleichen Komponente aus Gattern. Dies fließt allerdings im theoretischen Modell nicht ein. Der Anteil dieser Makrokomponenten am kritischen Pfad ist beim Netzwerkknoten sehr hoch. Deshalb erhält man hier eine Zykluszeit von $30ns$, die gemessen an der Basisverzögerung von $1.4ns$ und einer theoretischen Zykluszeit von 30 Gate Delays sehr klein ist.

Allerdings muß im Netzwerk noch die Leiterlaufzeit zwischen Netzwerkchips berücksichtigt werden. Diese kann zwischen Chips, die auf verschiedenen Platinen plaziert sind und durch Flachbandkabel verbunden werden, bis zu $30ns$ betragen [Dre91]. Verlängert man die Zykluszeit des Netzwerkknotens um die eventuell notwendige Übertragungszeit, so erhält man für beide wieder gleiche Zykluszeiten.

Kann man jedoch die Übertragungszeit aus dieser Berechnung ausnehmen wie im nächsten Abschnitt beschrieben, so hat man pro zu verarbeitendem Paket 2 Zyklen im Netzwerkknoten zur Verfügung. Dies kann man dazu nutzen, Pakete in 2 Teilen nacheinander zu verarbeiten. Demzufolge ist es auch ausreichend, Pakete zwischen Netzknoten in zwei Teilen zu übertragen. Dies reduziert die Breite unserer Links auf die Hälfte, nämlich auf 53 Bits.

Diese Vorgehensweise bringt zwei Vorteile. Erstens reduziert sich der Verkabelungsaufwand auf und zwischen verschiedenen Platinen. Die Bedeutung dieser Erleichterung wird in Abschnitt 6.3 bei der Beschreibung des Aufbaus der Platinen deutlich. Zweitens reduziert sich die Anzahl der Pins bei der Aufteilung eines Makros auf Chips. In den Gleichungen für pin_1 und pin_2 werden die Zahlen 104 bzw. 105 durch 52 bzw. 53 ersetzt, man erhält als neue Anzahlen von Pins pin'_1 und pin'_2. Außerdem können die 24 zusätzlichen Kontrollbits auf 12 reduziert werden. Damit ändern sich die notwendigen Anzahlen von Slices zu $2^{i'_1(x)} + \text{sign}(i'_1(x))$ bzw. $2^{i'_2(x)} + \text{sign}(i'_2(x))$. Der zusätzliche Term sign rührt von der Tatsache her, daß für $i'(x)$ auch der Wert 0 auftritt. In diesem Falle kann ein Makro in einem Chip untergebracht werden und der zusätzliche Platz für Carry Generierung entfällt. Die notwendigen Anzahlen von Slices bei halbierten Links sind in Tabelle 6.2 zu sehen.

n	$chip'_1(0,n)$	$chip'_1(1,n)$	$chip'_1(2,n)$	$chip'_2(1,n)$	$chip'_2(2,n)$
4	80	120	264	32	40
5	192	240	528	80	120
6	448	640	1584	192	240
7	1024	1280	3168	448	640
8	2304	3200	6336	1024	1280
9	5120	6400	16896	2304	3200
10	11264	15360	33792	5120	6400
11	24576	30720	67584	11264	15360
12	53248	71680	168960	24576	30720
13	114688	143360	337920	53248	71680
14	245760	327680	675840	114688	143360
15	524288	655360	1622016	245760	327680
16	1114112	1474560	3244032	524288	655360

Tabelle 6.3: Benötigte Anzahlen von Netzwerkchips bei halbierten Links

Im folgenden soll stets der Index 1 bzw. 2 den ersten bzw. zweiten Ansatz kennzeichnen, der Apostroph die Halbierung der Breite.

Die notwendige Anzahl von Chips ändert sich nun ebenfalls. Die neuen Werte sind in Tabelle 6.3 zu sehen. Aus der Tabelle geht deutlich hervor, daß der zweite Ansatz mit $x = 1$ die kleinste Anzahl von Chips verlangt, außerdem kann hier ein Makro in einem Chip untergebracht werden.

Bei der Implementierung des Sortierfeldes in Chips stößt man weder auf Pin– noch auf Gatterbeschränkungen, die zu Problemen führen würden. Jeder Knoten hat 4 Links, von denen allerdings immer nur 2 benutzt werden. Ein Wechsel der Nutzung findet nur beim Wechsel der Phasen statt. Da dies selten geschieht, kann man je zwei abwechselnd benutzte Links auf einem bidirektionalen Bus anlegen. Damit benötigt ein Chip mit einem Knoten nur etwa 210 Pins. Die Anzahl der Knoten, die in einem Chip untergebracht werden, ändert die Pinzahl nicht, da die Knoten als Feld linear verbunden sind und so stets nur ein Link des ersten Knotens und ein Link des letzten Knotens den Chip verläßt. Hier kann man soviele Knoten in einem Chip unterbringen, wie die Gatterzahl zuläßt. Benötigt man ein Feld größerer Länge, schaltet man mehrere Sortierchips hintereinander.

6.3 Physikalischer Aufbau — 2 Alternativen

Im vorigen Abschnitt wurde nach einer „besten“ Implementierung und Partitionierung von Netzwerkknoten in Chips gesucht. Als Güte der Implementierung diente die notwendige Anzahl von Chips. Diese Überlegung reicht zu einem physikalischen Aufbau allerdings nicht

aus. Alle Chips müssen auf Platinen *(printed circuit board, PCB)* partitioniert werden, die Platinen müssen so in (möglichst kommerziell erhältliche) Racks und Schränke eingebaut werden, daß man sie untereinander verkabeln, kühlen, mit elektrischer Energie versorgen kann und vieles mehr. Hier soll nur der Aspekt der Verdrahtung betrachtet werden.

Wir werden 3 Arten von PCBs benutzen: Prozessor Boards, Speicher Boards und Netzwerk Boards. Ein Prozessor Board enthält einen Prozessor Chip samt Register RAM, zugehöriger Logik, den Programm Speicher, zwei Sortierfelder in mehreren Sortierchips, eine Verbindung nach außen zum Laden des Programm Speichers u. ä. (siehe auch Abschnitt 6.4) und einen Link zum Netzwerk. Ein Speicher Board enthält n Speicher Module, die aus kommerziell erhältlichen Speicher Chips aufgebaut werden, je einen Treiber– und ODER Baum, die zugehörige Logik, eine Verbindung nach außen zum Schreiben und Lesen des globalen Speichers zwecks Ein– oder Ausgabe (siehe auch Abschnitt 6.4) und eine Verbindung zum Netzwerk. Um eine einfachere Arbeitsweise zu erhalten, wählt man als Anzahl der Speicher Module pro Speicher Board nicht genau n, sondern $2^{\lceil \log n \rceil}$.

Ein Netzwerk Board enthält eine Anzahl Netzwerk Chips und Steckverbinder (einschließlich eventuell notwendiger Treiber) für die zugehörigen Netzwerklinks zu anderen Platinen, die sogenannten *Off–Board Links*.

Prozessor und Speicher Boards sind im Aufbau nicht außergewöhnlich, die Verbindungen nach außen und zum Netzwerk können über Steckverbinder am Ende der Boards realisiert werden. Deshalb wird hier nicht weiter darauf eingegangen. Schwierig sind Aufbau und Verkabelung der Netzwerk Boards. Dies hat zwei Gründe. Erstens besitzt eine Platine nur eine beschränkte Fläche zur Plazierung von Steckverbindern, über die Links zu anderen Platinen realisiert werden. Hier wiederholt sich mit veränderten Parametern ein Problem, das bereits im letzten Abschnitt vorkam. Diese Links sind besonders kritisch, da sie wegen ihrer Länge eine große Verzögerungszeit haben und elektrische Probleme aufwerfen. Zweitens muß bei der Partitionierung der Netzwerk Chips dafür gesorgt werden, daß die verbleibenden Links zwischen Platinen diese so verbinden, daß die Platinen in handelsüblichen Einschüben und Schränken so angeordnet werden können, daß die Links nicht zu lang werden. Unter diesem Aspekt ist auch die Auswahl einer Makrogröße des letzten Abschnitts noch einmal zu überdenken.

Im folgenden wird zuerst eine maximale Anzahl von Chips und Steckverbindern berechnet, die eine Platine aufnehmen kann. Außerdem wird eine Realisierung von Off–Board Links vorgeschlagen, die es erlaubt, die Verzögerungszeit dieser Links aus der Zykluszeit der Netzwerk Chips auszuklammern. Danach wird für die verschiedenen Partitionierungen der Netzwerkknoten in Chips die Gesamtzahl der Links und der Anteil der Off–Board Links errechnet. Zuletzt wird, basierend auf den Erkenntnissen dieser Überlegungen, ein Vorschlag für die Anordnung der Netzwerk Boards in Einschüben und Schränken gemacht.

Breite	Ansatz	x		
		0	1	2
voll	1	1	0	
	2		1	
halb	1	2	0	0
	2		2	0

Tabelle 6.4: Größe von Subnetzwerken auf PCBs

6.3.1 Größe von PCBs

Verfügbare PCBs mit Standard Maßen haben die Fläche $366mm \times 340mm = 124440mm^2$ [Bic88]. Dies entspricht dem Europa Format mit 3–facher Höhe und „triple extended" Länge. Ein HDC064 Chip hat eine Fläche von $2237.3mm^2$ [Mot89a]. Ein Steckverbinder zur Aufnahme eines Links halber Breite hat eine Fläche von $94mm \times 6mm = 564mm^2$, für einen Link mit ganzer Breite wird die doppelte Fläche benötigt. Wir wollen annehmen, daß Chips und Steckverbinder zusammen etwa 80% der Fläche des PCBs belegen, da der Rest wegen Verdrahtung und Verschnitt beim Plazieren der Chips und Steckverbinder nicht genutzt werden kann. Mit diesen Daten man nun berechnen, wieviele Makros der verschiedenen Ansätze auf einem Board untergebracht werden können. Hierbei wird gefordert, daß die Makros ein Subnetzwerk bilden, um die Anzahl der Off–Board Links zu minimieren. Wenn ein Makro jeweils 2^r Ein– und Ausgänge hat, dann besteht ein solches Subnetzwerk aus $s+1$ Stufen mit je $2^{r \cdot s}$ Makros pro Stufe. Außerdem soll mindestens ein ganzes Makro auf das Board passen.

Tabelle 6.4 zeigt für alle Ansätze die maximale Anzahl s von Stufen eines Subnetzwerkes, das auf ein PCB paßt.

6.3.2 On–Board und Off–Board Links

Links zwischen Platinen werden realisiert durch Steckverbinder auf beiden Platinen, die durch ein Flachbandkabel miteinander verbunden sind. Der Delay dieser Links beträgt bis zu $30ns$. Um diese Zeit aus der Zykluszeit der Netzwerkknoten auszuklammern, geht man wie folgt vor: Vor den ersten Steckverbinder wird ein Register gesetzt. Dieses wird mit dem gleichen Clock Signal wie die Netzwerkknoten gespeist. Damit benötigt ein Paket, das diesen Link benutzen will, nun zwei Takte. Am Ende des ersten Taktes verläßt es den Netzwerkknoten, der es abschickt, und wird in das Register geclockt. Im nächsten Takt überquert es den Link und wird am Ende des zweiten Taktes in den Eingangspuffer des Netzwerkknotens am Ende des Links aufgenommen. Die Geschwindigkeit des Netzwerkes ist unbeeinflußt. Der Delay eines Paketes ist um die Anzahl von Zyklen erhöht, wie oft es einen Off–Board Link überqueren muß. Diese Anzahl ist allerdings klein im Vergleich zum Gesamt

n	$cable_1(0,n)$	$cable_1(1,n)$	$cable_2(1,n)$	$cable'_1(0,n)$	$cable'_1(1,n)$	$cable'_1(2,n)$	$cable'_2(1,n)$	$cable'_2(2,n)$
	volle Breite			halbe Breite				
4	16640	9984	6720	8320	4992	3328	3392	1696
5	39936	19968	16800	19968	9984	6656	8480	5088
6	93184	53248	40320	46592	26624	19968	20352	10176
7	212992	106496	94080	106496	53248	39936	47488	27136
8	479232	266240	215040	239616	133120	79872	108544	54272
9	1064960	532480	483840	532480	266240	212992	244224	135680
10	2342912	1277952	1075200	1171456	638976	425984	542720	271360
11	5111808	2555904	2365440	2555904	1277952	851968	1193984	651264
12	11075584	5963776	5160960	5537792	2981888	2129920	2605056	1302528
13	23855104	11927552	11182080	11927552	5963776	4259840	5644288	3039232
14	51118080	27262976	24084480	25559040	13631488	8519680	12156928	6078464
15	109051904	54525952	51609600	54525952	27262976	20447232	26050560	13893632
16	231735296	122683392	110100480	115867648	61341696	40894464	55574528	27787264

Tabelle 6.5: Notwendige Kabel bei verschiedenen Partitionierungen

Delay und kann vernachlässigt werden, da die Anzahl der virtuellen Prozessoren ohnehin aus technischen Gründen zur Zweierpotenz aufgestockt wurde und deshalb eigentlich zu hoch ist. Die genaue Zahl beträgt 8 und wird wird später in diesem Abschnitt errechnet.

Es soll nun errechnet werden, wieviele Links zwischen verschiedenen Makros verdrahtet werden müssen, entweder als Leitungen auf einer Platine oder als Kabel zwischen Platinen. Die Anzahl dieser Links ist sehr einfach zu errechnen. Im ersten Ansatz benötigt man $\lceil (n+1)/(x+1) \rceil$ Stufen von Makros, deshalb benötigt man $link_1(x,n) = \lceil (n+1)/(x+1) \rceil - 1$ Stufen von Links. Pro Stufe benötigt man 2^{n+1} Links. Im zweiten Ansatz braucht man analog $link_2(x,n) = \lceil n/x \rceil - 1$ Stufen mit je 2^n Links. Die Ein– und Ausgänge des Netzwerkes sind hierbei nicht mitgerechnet. Die Anzahl der Kabel beim ersten Ansatz mit voller Breite der Links ergibt sich dann zu

$$cable_1(x,n) = link_1(x,n) \cdot 2^{n+1} \cdot 104$$

Die Formeln für $cable_2, cable'_1$ und $cable'_2$ erhält man analog. Aus den Formeln ergibt sich, daß die Anzahl der Links mit steigendem x kleiner wird. Der zweite Ansatz braucht weniger Links. Die wenigsten Kabel benötigt man also beim zweiten Ansatz mit $x = 2$ und halbierten Links. Die genauen Zahlen sind in Tabelle 6.5 zu sehen.

Mit Hilfe der im vorigen Abschnitt berechneten Zahl s kann man nun auch den Anteil an Off–Board Links berechnen. Enthält eine Platine $s+1$ Stufen von Makros und benötigt das gesamte Netzwerk $link(x,n)$ Stufen von Links, so benötigt es $\lceil (link(x,n)+1)/(s+1) \rceil$ Stufen von Platinen, da es $link(x,n)+1$ Stufen von Makros gibt. Die Anzahl der Stufen von Off–Board Links berechnet sich dann für Ansatz $j \in \{1,2\}$ zu

n	$cable_1(0,n)$	$cable_1(1,n)$	$cable_2(1,n)$	$cable'_1(0,n)$	$cable'_1(1,n)$	$cable'_1(2,n)$	$cable'_2(1,n)$	$cable'_2(2,n)$
	volle Breite			halbe Breite				
4	6656	6656	1680	1664	3328	1664	848	848
5	13312	13312	6720	3328	6656	3328	1696	3392
6	39936	39936	13440	13312	19968	13312	3392	6784
7	79872	79872	40320	26624	39936	26624	13568	20352
8	212992	212992	80640	53248	106496	53248	27136	40704
9	425984	425984	215040	159744	212992	159744	54272	108544
10	1064960	1064960	430080	319488	532480	319488	162816	217088
11	2129920	2129920	1075200	638976	1064960	638976	325632	542720
12	5111808	5111808	2150400	1703936	2555904	1703936	651264	1085440
13	10223616	10223616	5160960	3407872	5111808	3407872	1736704	2605056
14	23855104	23855104	10321920	6815744	11927552	6815744	3473408	5210112
15	47710208	47710208	24084480	17039360	23855104	17039360	6946816	12156928
16	109051904	109051904	48168960	34078720	54525952	34078720	17367040	24313856

Tabelle 6.6: Notwendige Kabel in Off–Board Links

$$offlink_j(s_j(x), x, n) = \left\lceil \frac{link_j(x,n)+1}{s_j(x)+1} \right\rceil - 1$$

Jede Stufe von Off–Board Links enthält 2^{n+1} Links im ersten Ansatz, 2^n Links im zweiten Ansatz. Die Breite der Links ergibt sich wie oben. Die Anzahl von Kabeln in Off-Board Links für die verschiedenen Ansätze ist in Tabelle 6.6 zu sehen.

Die günstigste Alternative bietet der zweite Ansatz mit $x = 1$ und halbierten Links. Diese Aufteilung wurde in [ADK+91] bereits kurz beschrieben. Auf einer Platine wird damit ein Butterfly Netzwerk mit $x * (s'_2(x) + 1) = 3$ Netzwerk Stufen und $2^{x+s'_2(x)} = 8$ Knoten pro Stufe realisiert. Ist n nicht durch 3 teilbar, so wird im Fall $n = 3u - 1$ in der ersten Stufe der Platinen und im Fall $n = 3u - 2$ in der ersten und letzten Stufe von Platinen die letzte Stufe der Knoten weggelassen. Diese Platinen enthalten dann zwei Butterfly Netzwerke mit je 2 Stufen und 4 Knoten pro Stufe.

Damit ergibt sich die Anzahl von zusätzlichen Zyklen zum Transport eines Paketes durch das Netzwerk zu $2 \cdot (offlink_2(s'_2(1), 1, n) + 2) = 8$. Ein Paket muß vom Prozessor Board ins Netzwerk und vom Netzwerk zum Speicher Board 2 Off–Board Links überqueren, innerhalb des Netzwerkes muß es $offlink$ viele Off-Board Links überqueren, der gleiche Weg muß wieder zurück zum Prozessor erfolgen.

6.3.3 Anordnung von Boards

In der theoretischen Informatik geht man in der Regel bei der Einbettung von Butterfly Netzwerken davon aus, daß eine genügend große einheitliche Fläche zur Verfügung steht,

entweder als Chip Fläche oder als PCB. Auch das Kostenmodell nimmt dies an. In dieser vereinfachten zweidimensionalen Sicht gibt es optimale Einbettungen von Butterfly Netzwerken mit optimal kurzen Kabeln [BK89].

In der Realität muß das Netzwerk partitioniert in Chips und diese wiederum auf Platinen verteilt werden. Dies ist in den letzten Abschnitten erfolgt. Über die geometrische Anordnung der Knoten in Chips und die geometrische Anordnung der Chips auf den Platinen wurde dort keine Aussage gemacht, da die Chipfläche nicht ausgenutzt ist und deshalb dort genügend Platz zum Verdrahten vorhanden ist. Auf einer Platine ist erstens extra Fläche zum Verdrahten vorgesehen, außerdem kann die Verdrahtung in den vorhandenen 10 bis 20 Verdrahtungslagen der Platine auch dort erfolgen, wo Chips auf der Platine sitzen. Die Verbindungen in Chips und auch die auf Platinen sind sehr kurz und damit nicht zeitkritisch. Ihre Länge muß deshalb auch nicht optimiert werden.

Bei der Verbindung der Platinen untereinander kann dies nicht vernachlässigt werden. Platinen müssen in Schränken sehr regelmäßig angeordnet werden. Die Verbindungen zwischen Platinen sind sehr lang. Deshalb muß bei der Anordnung der Platinen darauf geachtet werden, daß die Verbindungen zwischen Platinen möglichst kurz sind. Diese Verbindungen sollen möglichst auch so liegen, daß die Platinen von ihnen nicht verdeckt sind. Damit soll gewährleistet werden, daß die Platinen im Wartungsfall einfach zu entfernen sind.

Platinen sitzen in genormten Einschüben, den sogenannten *Racks*, nebeneinander. An der Rückseite dieser Einschübe befindet sich eine Leiterplatte mit Steckverbindern, zu denen die Platinen mittels Steckverbindern am rückseitigen Ende der Platine Verbindung haben und so auch mit Elektrizität versorgt werden. Da eine Platine mit dreifacher Bauhöhe, wie wir sie verwenden wollen, allerdings nur drei Steckverbinder mit 128 Pins am rückwärtigen Ende haben kann, ist diese Möglichkeit nicht zum Wegführen der Off–Board Links von den Platinen zu nutzen. Off–Board Links müssen also an der Vorderseite des Racks die Platine verlassen. Hier können Steckverbinder auch so angeordnet werden, daß die Schmalseite der Steckverbinder an der Vorderseite der Platine sitzt, damit sind genügend Steckverbinder pro Platine möglich.

Steckverbinder, die nicht am vorderen oder hinteren Ende der Platine liegen, versucht man nach Möglichkeit zu vermeiden, da Kabel an diesen Steckverbindern nur entfernt werden können, wenn die Platine entfernt wird.

Mehrere Racks setzt man übereinander in Schränke. Die Kühlung erfolgt durch Gebläse von unten nach oben, auch Gebläseeinschübe in halber Höhe des Schrankes sind üblich.

Kabel zwischen Platinen verlegt man an Vorder– oder Rückseite der Racks, jedoch nicht innerhalb von Racks, da sie dort die Kühlung und die Entfernung von Platinen behindern.

Unter diesen Beschränkungen ist eine komplexe Verbindung von Platinen nur sehr schwierig zu realisieren.

Der einzige mir bekannte Artikel, der sich mit Realisierung von Butterfly Netzwerken auf

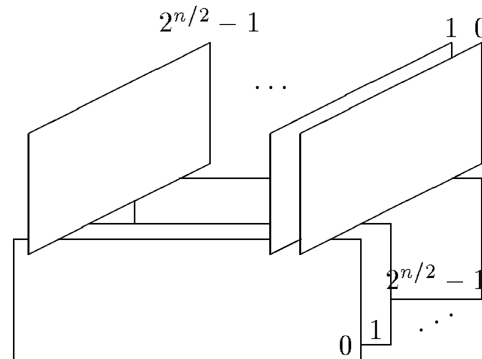

Abbildung 6.4: Anordnung von Boards nach Wise

Platinen befaßt, stammt von WISE [Wis81]. WISE wählt folgende Vorgehensweise.

Gegeben sei ein Butterfly Netzwerk mit $n+1$ Stufen und 2^n Knoten pro Stufe. Sei n ungerade, also $n+1=2u$ gerade. Man schneidet das Butterfly Netzwerk nach $u=(n+1)/2$ Stufen entzwei. Jeder der zwei Teile zerfällt in 2^u Subnetzwerke. Die Subnetzwerke sind Butterfly Netzwerke mit u Stufen und 2^{u-1} Knoten pro Stufe. WISE beweist, daß jedes der Subnetzwerke des ersten Teils mit allen Subnetzwerken des zweiten Teils verbunden ist. Die obigen Verbindungen sind überdies die einzigen zwischen verschiedenen Subnetzwerken. WISE schlägt vor, jedes Subnetzwerk auf einem PCB unterzubringen, und die Boards anzuordnen wie in Abbildung 6.4 angedeutet. Die Boards des ersten Teils sitzen über den Boards des zweiten Teils und sind gegenüber letzteren um einen Winkel von 90° gedreht. Jedes Board des ersten Teils berührt nun jedes Board des zweiten Teils. Die Off–Board Links können dort realisiert werden, so daß keine langen Verbindungen auftreten.

Dieser Aufbau hat in der Praxis allerdings mehrere Nachteile. Die Platinen lassen sich in der gezeigten Anordnung nicht in Racks anordnen. Ordnet man sie in zwei übereinanderliegenden Racks an, so hat man Kabel zwischen diesen. Um die Platinen noch aus den Racks entfernen zu können, muß man diese Kabel um die Länge zweier Boards verlängern. Um die Racks in einen Schrank einbauen zu können, muß man das obere Rack um 90° drehen. Dann ist die Verkabelung nicht mehr günstig. Die Boards sind in der Art eines vollständigen bipartiten Graphen miteinander verbunden, wobei die beiden Teile die Partitionen bilden. Nach den Berechnungen der vorigen Kapitel kann eine Platine maximal 3 Stufen von Chips mit je 4 Chips pro Stufe enthalten. Mit der Idee von WISE kann man also nur 6 stufige Netzwerke aufbauen. Dies schränkt die Anzahl der Prozessoren auf maximal 64 ein. Dies ist für den vorgesehenen Prototyp mit 128 Prozessoren nicht ausreichend. Eine Erweiterung durch mehrfache Anwendung dieses Prinzips ist nicht offensichtlich. Deshalb soll hier ein weiterer Vorschlag zur Anordnung der Boards gemacht werden.

Dieser Vorschlag beruht auf der Verbindungsstruktur von Subnetzwerken, wenn man ein Butterfly Netzwerk in 3 Teile schneidet. Auskunft hierüber gibt das folgende Lemma.

Teil	Stufen x	Knoten
1	$0,\ldots,u-1$	$\langle x, j\cdot 2^{u-1}\rangle,\ldots,\langle x,(j+1)\cdot 2^{u-1}-1\rangle$
2	$u,\ldots,2u-1$	$\langle x,\tilde{j}\rangle,\langle x,\tilde{j}+2^u\rangle,\ldots,\langle x,\tilde{j}+(2^{u-1}-1)\cdot 2^u\rangle$ wobei $\tilde{j}=\left\lfloor\frac{j}{2u}\right\rfloor\cdot 2^{2u-1}+j \bmod 2^u$
3	$2u,\ldots,3u-1$	$\langle x,j\rangle,\langle x,j+2^{2u}\rangle,\ldots,\langle x,j+(2^{u-1}-1)\cdot 2^{2u}\rangle$

Tabelle 6.7: Knoten in Subnetzwerk j

Stufen x	Subnetzw.
$0,\ldots,u-1$	$(1,\lfloor\frac{y}{2^{u-1}}\rfloor)$
$u,\ldots,2u-1$	$(2, y \bmod 2^c+\lfloor\frac{y}{2^{2u-1}}\rfloor)$
$2u,\ldots,3u-1$	$(3, y \bmod 2^{2u})$

Tabelle 6.8: Zuordnung Subnetzwerk zu Knoten y

Lemma 6.1 *Gegeben sei ein Butterfly Netzwerk mit $n+1$ Stufen und 2^n Knoten pro Stufe. Sei $n+1=3u$. Zerschneidet man das Butterfly Netzwerk nach u Stufen und nach $2u$ Stufen, so zerfällt das Netzwerk in 3 Teile. Jeder Teil besteht aus 2^{2u} Subnetzwerken. Jedes Subnetzwerk beinhaltet ein Butterfly Netzwerk mit u Stufen und 2^{u-1} Knoten pro Stufe. Sei (i,j) das Subnetzwerk $j\in\{0,\ldots,2^{2u}-1\}$ im Teil $i\in\{1,2,3\}$. Man nummeriere die Subnetzwerke jedes Teils mittels*

$$\phi:\{0,\ldots,2^{2u}-1\}\rightarrow\{0,\ldots,2^u-1\}\times\{0,\ldots,2^u-1\},\phi(j)=\left(\left\lfloor\frac{j}{2^u}\right\rfloor, j \bmod 2^u\right)$$

Fügt man die Subnetzwerke wieder zu dem ursprünglichen Netzwerk zusammen, dann ergeben sich bei obiger Nummerierung der Subnetzwerke jedes Teils die folgenden Verbindungen zwischen den Subnetzwerken:

Die Subnetzwerke $(1,(j,0)),\ldots,(1,(j,2^u-1))$ des ersten Teils sind nur mit den Subnetzwerken $(2,(j,0)),\ldots,(2,(j,2^u-1))$ des zweiten Teils verbunden.
Die Subnetzwerke $(2,(0,j)),\ldots,(2,(2^u-1,j))$ des zweiten Teils sind nur mit den Subnetzwerken $(3,(0,j)),\ldots,(3,(2^u-1,j))$ des dritten Teils verbunden.

Beweis: Sei $\langle x,y\rangle$ der Knoten y in der x–ten Stufe des Butterfly Netzwerkes. Die Tabellen 6.7 und 6.8 geben an, welche Knoten jedes Subnetzwerk enthält und zu welchem Subnetzwerk jeder Knoten gehört.

Um die Verbindungen zwischen Subnetzwerken verschiedener Teile zu ermitteln, gibt man die Menge K der Knoten der Stufe u an, mit denen die Knoten der Stufe $u-1$, die sich in Subnetzwerk $(1,j)$ befinden, verbunden sind. Danach errechnet man, zu welchen Subnetzwerken des zweiten Teils die Knoten der Menge K gehören. Dies beweist den ersten

Teil der Behauptung. Zum Beweis des zweiten Teils der Behauptung geht man analog mit Subnetzwerken des dritten und zweiten Teils des Netzwerkes vor.

Knoten $\langle u-1, y\rangle$ ist nach Definition des Butterfly Netzwerkes verbunden mit den Knoten $\langle u, y\rangle$ und $\langle u, y \oplus 2^u\rangle$. Setzt man dies nach Tabelle 6.7 für alle Knoten des Subnetzwerkes $(1, j)$ ein, so ergibt sich, daß von diesem Subnetzwerk Verbindungen zu den Knoten $\langle u, j \cdot 2^{u-1} + t\rangle$ gehen. Hierbei gilt $t \in \{0, \ldots, 2^c - 1\}$, falls j gerade, $t \in \{-2^{c-1}, \ldots, 2^{c-1} - 1\}$, falls j ungerade. Für den Fall j gerade liegen diese Knoten nach Tabelle 6.8 in Subnetzwerken

$$(2, (i \cdot 2^{u-1} + t) \bmod 2^u + \left\lfloor \frac{j \cdot 2^{u-1} + t}{2^{2u-1}} \right\rfloor) = (2, t + \left\lfloor \frac{j}{2^u} \right\rfloor)$$

Für den Fall j ungerade erhält man das gleiche Ergebnis. Dies beweist den ersten Teil der Behauptung.

Knoten $\langle 2u, y\rangle$ ist nach Definition des Butterfly Netzwerkes verbunden mit den Knoten $\langle 2u-1, y\rangle$ und $\langle 2u-1, y \oplus 2^{2u-1}\rangle$. Setzt man dies nach Tabelle 6.7 für alle Knoten des Subnetzwerkes $(3, j)$ ein, so ergibt sich, daß von diesem Subnetzwerk Verbindungen zu den Knoten $\langle 2u-1, j + t \cdot 2^{2u-1}\rangle$ gehen. Hierbei ist $t \in \{0, \ldots, 2^u - 1\}$, falls $j < 2^{2u-1}$, oder $t \in \{-1, \ldots, 2^u - 2\}$, falls $j \geq 2^{2u-1}$. Für den Fall $j < 2^{2u-1}$ liegen diese Knoten nach Tabelle 6.8 in Subnetzwerken

$$(2, (j + t \cdot 2^{2u-1}) \bmod 2^u + \left\lfloor \frac{j + t \cdot 2^{2u-1}}{2^{2u-11}} \right\rfloor) = (2, j \bmod 2^u + t)$$

Für den Fall $j \geq 2^{2u-1}$ erhält man das gleiche Ergebnis. Dies beweist den zweiten Teil der Behauptung. ∎

Ähnliche Anordnungen erhält man für die Fälle $n+1 = 3u-1$ und $n+1 = 3u-2$. Im ersten Fall hat man im zweiten Teil des Netzwerkes nur $u-1$ Stufen. Die beiden $(u-1)$–stufigen Subnetzwerke, die mit dieser fehlenden Stufe ein u–stufiges Subnetzwerk gebildet hätten, behandelt man als ein „Pseudo“–Subnetzwerk. Im zweiten Fall haben erster und dritter Teil des Netzwerkes nur $u-1$ Stufen. Hier verfährt man ebenso.

Aus dieser Beobachtung kann man folgendes geometrisches Arrangement von Subnetzwerken ableiten: die Subnetzwerke jedes Teils werden als Quadrat angeordnet, die Anordnung soll die Nummerierung ϕ repräsentieren. Die Quadrate werden nun folgendermaßen angeordnet. Das Quadrat des ersten Teil sitzt über dem Quadrat des zweiten Teils, das Quadrat des dritten Teils sitzt rechts vom Quadrat des zweiten Teils.

Nach obigen Lemma gibt es jetzt zwischen erstem und zweitem Teil nur senkrechte Verbindungen, die alle Subnetzwerke einer Spalte verbinden. Zwischen zweitem und drittem Teil gibt es nur waagerechte Verbindungen, die alle Subnetzwerke einer Zeile verbinden. Die Verbindung der Subnetzwerke ist sehr regelmäßig, die Länge einer Verbindung zwischen Subnetzwerken ist begrenzt durch das Zweifache der Seitenlänge eines Quadrates,

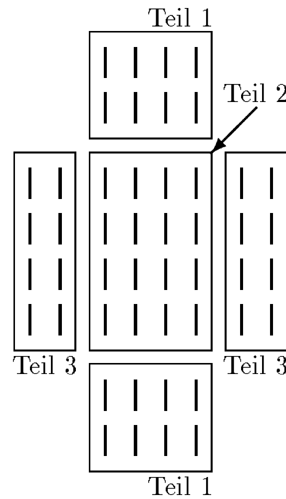

Abbildung 6.5: Anordnung der Boards in 3 Teilen

also durch $O(2^{u+1}) = O(2^{n/3})$. In der Praxis stellt sich allerdings die Frage nach der versteckten Konstante.

Für $n \leq 8$ haben wir ein Netzwerk mit höchstens 9 Stufen. Jedes Subnetzwerk hat damit höchstens 3 Stufen, paßt also auf ein Board. Diese Boards können sogar in Racks angeordnet werden. Um die Anordnung symmetrisch zu machen, teilt man das Quadrat des ersten Teils in zwei Rechtecke mit 2^u Spalten und 2^{u-1} Zeilen. Man ordnet sie oberhalb und unterhalb des Quadrates des zweiten Teils an. Analog geht man für das Quadrat des dritten Teils vor. Man ordnet zwei Rechtecke mit 2^{u-1} Spalten und 2^u Zeilen rechts und links vom Quadrat des zweiten Teils an.

Für den geplanten Prototyp hat das Netzwerk 7 Stufen. Pro Netzwerkteil gibt es damit 16 Boards. Die Anordnung dieser Boards ist in Abbildung 6.5 zu sehen.

Die Verbindungen zwischen den einzelnen Teilen haben immer noch die Form vollständiger bipartiter Graphen. Nur hatten die Graphen im Ansatz von Wise $2^{n/2}$ Knoten pro Partition, hier haben sie $2^{n/3}$ Knoten pro Partition. Für den Bereich von n, für den diese Anordnungen gedacht sind, ist dies eine deutliche Vereinfachung. Die Anordnung zeigt allerdings immer noch eine Unsymmetrie. Boards sind wesentlich höher als breit. Dadurch sind die vertikalen Leitungen länger als die horizontalen. Dies kann man umgehen, indem man mehrere Zeilen zu einer zusammenfaßt.

Ein Rack hat eine Breite von ca. $40cm$. Netzwerk Platinen sollten wegen der Steckverbinder auf den Platinen einen Abstand von mindestens $5cm$ haben. Da die Platinen eine Höhe von $34cm$ haben, sind vertikale Links um einen Faktor $34/5 = 6.8$ länger als horizontale Links. Faßt man nun a Zeilen zusammen, so reduziert sich die Höhe um den Faktor a, die Breite vergrößert sich um den Faktor a. Das Verhältnis vertikaler zu horizontalen Links reduziert

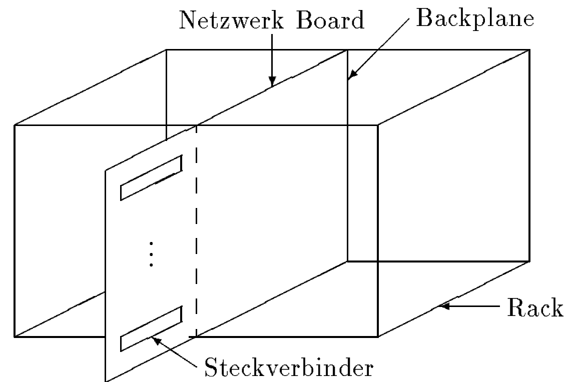

Abbildung 6.6: Anordnung von Steckverbindern auf Boards

sich damit um a^2. Die Zahl a müßte damit zu $a = \sqrt{6.8} \approx 2.6$ gewählt werden. Da a aber nur ganzzahlige Werte annehmen kann, wählt man $a = 2$. Damit hat man 16 Boards pro Zeile, die Breite dieser Zeile beträgt $16 \cdot 5cm = 80cm$. Jede Zeile paßt also in zwei Racks. Dies ist damit auch etwa die Länge der benötigten Kabel.

Im folgenden wollen wir noch angeben, wie die Kabel an den Boards befestigt werden. Netzwerk Boards ragen an der Vorderseite ein Stück aus den Racks heraus. An diesen vorstehenden Vorderseiten der Boards werden rechteckige Steckverbinder übereinander angebracht. Deren Schmalseite steht senkrecht, die lange Seite waagerecht. Damit kann man eine genügende Anzahl Steckverbinder so plazieren, daß sie gut erreichbar sind. Die Anordnung ist in Abbildung 6.6 zu sehen. Zur Verkabelung wickelt man Flachbandkabel rund. Dadurch lassen sich die notwendigen Überkreuzungen von Kabeln besser arrangieren.

Zu jeweils zwei Netzwerk Boards gehören 16 Prozessor Boards bzw. Speicher Boards. Diese passen jeweils in ein Rack. Damit kann man alle Prozessor und Speicher Boards in Schränken rechts und links des Netzwerkes unterbringen. Die Kabel zum Netzwerk verlaufen alle waagerecht und haben höchstens die Länge eines Racks.

6.3.4 Eine Alternative

Brüning schlägt vor, die Realisierung auf sehr großen Spezialplatinen der Größe $50cm \times 50cm$ vorzunehmen [AB91]. Auf diesen Platinen lassen sich 4 Prozessor Boards bzw. Speicher Boards unterbringen, zusätzlich jeweils die Subnetzwerke, die die ersten bzw. die letzten beiden Stufen des Netzwerkes realisieren. Die Kabel zwischen Prozessor bzw. Speicher Boards und Netzwerk entfallen damit. Der mittlere Teil des Netzwerkes kann wie bisher realisiert werden. Die Anordnung bleibt damit ähnlich, die Anzahl der Off–Board Links ist allerdings reduziert.

6.4 Anbindung an die Außenwelt

Der Prototyp des entworfenen Designs muß nicht nur rechnen, sondern auch mit der Außenwelt kommunizieren. Eingabedaten müssen von Massenspeichern oder Terminals zum globalen Speicher transportiert werden, Ausgabedaten müssen in der Gegenrichtung transportiert werden. Um dies als „Minimallösung" zu realisieren, haben alle Speicher Boards einen Bus Anschluß. Dieser Bus Anschluß ist auch an Prozessor Boards vorhanden, um den Programmspeicher zu laden und in der Testphase des Prototyps die Register RAMs der Prozessoren auslesen zu können. Die Backplane jedes Racks enthält einen Bus für die 16 Boards dieses Racks. Die Busse der Racks werden hierarchisch zu einem Bus zusammengefaßt. Über diesen hierarchischen Bus kommuniziert ein Host Rechner mit dem Prototyp und steuert die Ein– und Ausgabeaktivitäten. Diese Lösung ist allerdings minimal und bei großen Datenmengen langsam, da sie sequentiell arbeitet. Effiziente parallele Ein– und Ausgabe mittels verteilter Massenspeicher ist ein eigenes, schwieriges und noch sehr junges Forschungsgebiet und wird deshalb nicht behandelt.

Kapitel 7

Zusammenfassung

Die vorliegende Arbeit hatte zum Ziel, Kosten und Zeitverhalten einer PRAM Emulation möglichst genau zu bestimmen. Dies ist für RANADE's Fluent Machine erfolgt. Durch Reengineering konnte das Preis/Leistungs–Verhältnis des Ansatzes um einen Faktor, der etwa 6 beträgt, verbessert werden. Diese Verbesserung rückt den Ansatz in den Leistungsbereich konventioneller Parallelrechner. Ein gemeinsam mit ABOLHASSAN und PAUL publiziertes Ergebnis [AKP91a] besagt, daß die Güte des verbesserten Ansatzes bei 2^n physikalischen Prozessoren im schlimmsten Falle, wenn die Eigenschaft eines globalen Speichers nicht ausgenutzt wird, nur um einen Faktor $1.21n$ schlechter sein kann als die Güte einer optimalen Distributed memory Machine. Wird die Eigenschaft des globalen Speichers massiv ausgenutzt, kann die Güte des verbesserten Designs sogar um einen Faktor 438 besser sein als die Güte einer Distributed Memory Machine, die die shared Memory Emulation in Software ausführt. Diese Beobachtung und ihre Auswirkungen auf den Vergleich der Kosteneffektivität von PRAMs und anderen Parallelrechnern wird bei ABOLHASSAN [Abo92] ausgiebig diskutiert.

Ein zweiter Teil der Arbeit behandelt den Nutzen von verschiedenen Design Varianten. Er untersucht für mehrere Varianten, ob sie besser in Hardware oder in Software unterstützt werden sollten. Das Bewertungsmaß ist auch hier die im ersten Teil definierte Güte des Preis/Leistungs–Verhältnisses. Hierbei stellt sich heraus, daß die Unterstützung von konkurrierenden Zugriffen und die Unterstützung paralleler Präfixberechnung in Hardware beide bei bereits minimalen Anteilen dieser Aktionen an der Laufzeit zu einer besseren Güte führen als die Unterstützung in Software. Bei der Unterstützung mehrerer Kontexte pro Prozessor in Hardware ist der minimal notwendige Anteil höher, die Unterstützung einer moderaten Zahl von Kontexten, z. B. 8, scheint dennoch geraten, um die Zeit zum Kontextwechsel klein zu halten.

Der letzte Teil der Arbeit beschreibt die Schwierigkeiten beim Übergang vom Design zum Prototyp. Um vom abstrakten Modell zu einer Prototyp Realisierung zu gelangen, müssen mehrere Schwierigkeiten überwunden werden. Während das Modell eine große VLSI Fläche zur Einbettung der Maschine zur Verfügung stellt, werden in der Realität viele relativ kleine

Chips auf mehreren Platinen benutzt. Hierzu müssen die Teile des Designs so aufgeteilt werden, daß man möglichst wenige Arten von Chips entwickeln muß und daß die Chips so auf möglichst wenige Platinen verteilt werden können, daß die Platinen mit möglichst kurzen Kabeln möglichst regelmäßig verbunden werden können. Die Einschränkung der Kabellänge ist notwendig, da im Gegensatz zum Modell in der Realität Leiterlaufzeiten auf Kabeln nicht vernachlässigt werden können.

Für die Problemkreise Einbettung der verschiedenen Teile der Maschine in zu entwickelnde Chips und die Verteilung dieser Chips auf Platinen wurden Lösungen angegeben. Für die Anordnung dieser Platinen mit einer möglichst einfachen Verkabelung wurde ein Ansatz vorgestellt.

Viele Aspekte des parallelen Rechnens konnten in dieser Arbeit nicht berücksichtigt werden. Als Beispiel sei Fehlererkennung, Fehlerkorrektur und Fehlertoleranz sowohl im Netzwerk als auch im gesamten Design genannt. Zur Realisierung dieser Features gibt es bereits Ansätze. Da diese Probleme allerdings bei allen Parallelrechnern auftauchen, sollte ihre Lösung vom Entwurf eines PRAM Designs abgekoppelt werden können. Beim Bau eines Prototyps müssen jedoch hier Entscheidungen getroffen werden. Ein weiterer wichtiger Punkt ist die I/O–Bandbreite. Hier ist die Lösung einer PRAM–Host Verbindung über einen Bus für Probleme, die außer einer hohen parallelen Rechenleistung auch sehr viele Daten benötigen, wohl nicht ausreichend. Das Gebiet der parallelen Ein– und Ausgabe steckt allerdings noch in den Kinderschuhen. Hier muß also insgesamt noch viel Forschungsarbeit investiert werden, von deren Früchten auch der Prototyp der PRAM profitieren kann.

Literaturverzeichnis

[AB91] Ferri Abolhassan and Ulrich Brüning. Arrangements of network boards. Personal Communication, Oktober 1991.

[Abo92] Ferri Abolhassan. *PRAMs und DMMs.* Manuskript, Universität des Saarlandes, FB Informatik, 1992.

[ADK+91] Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of PRAMs. In Johannes Buchmann, Harald Ganzinger, Wolfgang J. Paul, editors, *Informatik. Festschrift zum 60. Geburtstag von Günter Hotz.* Teubner Verlag, 1992.

[AHMP87] Helmut Alt, Torben Hagerup, Kurt Mehlhorn, and Franco P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16(5):808–835, October 1987.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[AKP90] Ferri Abolhassan, Jörg Keller, and Wolfgang J. Paul. Überblick über PRAM–Simulationen und ihre Realisierbarkeit. In Theo Härder, Hartmut Wedekind, and Gerhard Zimmermann, editors, *Entwurf und Betrieb verteilter Systeme. Fachtagung der Sonderforschungsbereiche 124 und 182. Proceedings*, Informatik Fachberichte, pages 15–40. Springer, September 1990.

[AKP91a] Ferri Abolhassan, Jörg Keller, and Wolfgang J. Paul. On the cost–effectiveness of PRAMs. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9. IEEE, December 1991.

[AKP91b] Ferri Abolhassan, Jörg Keller, and Wolfgang J. Paul. On the cost–effectiveness and realization of the theoretical PRAM model. FB 14 Informatik, SFB-Report 09/1991, Universität des Saarlandes, May 1991.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Annual Symposium on Theory of Computing*, pages 1–9, New York, 1983. ACM.

[Ale82] R. Alelunias. Randomized parallel communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 60–72, August 1982.

[Bat68] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference, Vol. 32*, pages 307–314, Reston, Va., 1968. AFIPS Press.

[BH88] Gianfranco Bilardi and Kieran T. Herley. Deterministic simulations of PRAMs on bounded degree networks. In *Proceedings of the 26th Annual Allerton Conference on Communication, Control and Computation*, September 1988.

[Bic88] Bicc Vero Electronics. *Elektronik Handbuch*, 1988.

[BK89] Richard Beigel and Clyde P. Kruskal. Processor networks and interconnection networks without long wires. In *Proceedings of the 1989 Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 42–51. ACM, 1989.

[Clo53] Charles Clos. A study of non–blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[Cra84] Cray Research Inc., Mendota Heights, Minnesota. *The Cray X–MP series of Computer Systems Publication MP–2101*, 1984.

[CS86] Y. Chang and J. Simon. Continuous routing and batch routing on the hypercube. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 272–281, 1986.

[CW79] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[Die90] Martin Dietzfelbinger. Hashing modulo powers of two. Personal Communication, October 1990.

[Die91a] Martin Dietzfelbinger. On limitations of the performance of universal hashing with linear functions. Reihe Informatik Bericht Nr. 84, Universität–GH Paderborn, June 1991.

[Die91b] Martin Dietzfelbinger. Universal hashing in sequential, parallel, and distributed computation. Manuskript, September 1991.

[DM90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. Reihe Informatik Bericht Nr. 67, Universität–GH Paderborn, April 1990.

[DR86] P.W. Dymond and W.L. Ruzzo. Parallel RAMs with owned global memory and deterministic context–free language recognition. In *ICALP 1986, Automata, Languages and Programming*, pages 96–104, 1986.

[Dre91] Reinhard Drefenstedt. Untersuchung elektrischer Links für Parallelrechner. Manuskript, Universität des Saarlandes, FB Informatik, December 1991.

[DS87] William J. Dally and Charles L. Seitz. Deadlock–free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C–36(5):547–553, May 1987.

[Eng92] Curd Engelmann. Simulationen von PRAM's. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1992.

[F+92] Mathias Ferdinand et al. Pardo compiler. FOPRA Beschreibung, 1992.

[Fai85] Fairchild Semiconductor GmbH, Garching b. München. *FAST Data Book*, 1985.

[FOP+92] Arno Formella, Alexander Obe, Wolfgang J. Paul, Thomas Rauber, and Dietmar Schmidt. The SPARK 2.0 system — a special purpose vector processor with a vector PASCAL compiler. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*. IEEE, 1992.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[GGK+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin McAuliffe, Larry Rudolph, and Marc Snir. The NYU ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C–32(2):175–189, February 1983.

[GP83] Zvi Galil and Wolfgang J. Paul. An efficient general–purpose parallel computer. *Journal of the ACM*, 30(2):360–387, April 1983.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[Hag90] Torben Hagerup. Optimal parallel algorithms on planar graphs. *Information & Computation*, 84:71–96, 1990.

[HB84] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw–Hill, 1984.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[HSS91] Torben Hagerup, Arno Schmitt, and Helmut Seidl. FORK: A high–level–language for PRAMs. In *Proceedings of the Parallel Architectures and Languages Europe91*, 1991.

[KK79] Parviz Kermani and Leonard Kleinrock. Virtual cut–through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[Kni89] Thomas F. Knight Jr. Technologies for low latency interconnection switches. In *Proceedings of the 1989 Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 351–358. ACM, 1989.

[KR90] Richard M. Karp and Viaya L. Ramachandran. A survey of parallel algorithms for shared–memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. Elsevier, 1990.

[KRT88] Danny Krizanc, Sanguthevar Rajasekaran, and Thanasis Tsantilas. Optimal routing algorithms for mesh–connected processor arrays. In *Proceedings of the Aegan Workshop on Computing: VLSI Algorithms and Architectures*, pages 411–422. Springer, 1988.

[KT89] Manfred Kunde and Thomas Tensi. Multi–packet–routing on mesh connected arrays. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 336–343, 1989.

[KU88] Anna R. Karlin and Eli Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, October 1988.

[Kun88] Manfred Kunde. Routing and sorting on mesh–connected arrays. In *Proceedings of the Aegan Workshop on Computing: VLSI Algorithms and Architectures*, pages 423–433. Springer, 1988.

[LF80] Richard E. Ladner and Michael J. Fisher. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[LL90] F.T. Leighton and Charles E. Leiserson. Theory of parallel and VLSI computation. Lecture notes, research seminar series, Massachusetts Institute of Technology, May 1990.

[LMR88] F.T. Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[LMRR90] F.T. Leighton, Bruce Maggs, Abhiram Ranade, and Satish Rao. Randomized routing and sorting on fixed–connection networks. unpublished manuscript, 1990.

[LMT89] F.T. Leighton, Fillia Makedon, and G. Ioannis Tollis. A $2n - 2$ step algorithm for routing in an $n \times n$ array with constant size queues. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, 1989.

[LPV81] Gavriela Freund Lev, Nicholas Pippinger, and Leslie G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C–30(2):93–100, 1981.

[Mas90] Werner Massonne. Schnelle Routing–Algorithmen auf Butterfly Netzwerken. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1990.

[McM88] F.H. McMahon. The livermore fortran kernels test of the numerical performance range. Technical report, Lawrence Livermore National Laboratory, 1988.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms*, volume 2, Graph Algorithms and NP–Completeness. Springer, 1984.

[Mot89a] Motorola, Inc. ASIC Division, Chandler, Arizona. *Motorola High Density CMOS Array Design Manual*, July 1989.

[Mot89b] Motorola, Inc., Literature Distribution, Phoenix, Arizona. *MC 68040 User's Manual*, 1989.

[MP90] Silvia M. Müller and Wolfgang J. Paul. Towards a formal theory of computer architecture. In *Proceedings of PARCELLA 90, Advances in Parallel Computing.* North–Holland, 1990.

[Mül89] Silvia M. Müller. Die Auswirkung der Startup–Zeit auf die Leistung paralleler Rechner bei numerischen Anwendungen. Diplomarbeit, Universität des Saarlandes, FB Mathematik, 1989.

[Mül91] Silvia M. Müller. *RISC und CISC: Optimierung und Vergleich von Architekturen.* Dissertation, Universität des Saarlandes, FB Informatik, 1991.

[MV84] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[Pat90] M.S. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5:75–92, 1990.

[Pip84] Nick Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 127–136, 1984.

[PM86] Robert Perron and Craig Mundie. The architecture of the ALLIANT FX/8 computer. In *Proceedings of the IEEE Computer Society International Conference, Spring 86*, pages 390–393. IEEE, May 1986.

[PS82] David A. Patterson and Carlo H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, 1982.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison Wesley, 2nd edition, 1985.

[Ran87] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[Ran91] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[RBJ88] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnson. The Fluent Abstract Machine. In *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, pages 71–93, 1988.

[RV87] J.H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[San90] Georg Sander. PRAM Simulator — Systembeschreibung des PRAMSIM. Interner Bericht, FB Informatik, Universität des Saarlandes, October 1990.

[Sch80] J.T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, October 1980.

[Sch92] Dieter Scheerer. *Entwurf eines testbaren PRAM Prozessors.* Manuskript, Universität des Saarlandes, FB Informatik, 1992.

[Spa76] Otto Spaniol. *Arithmethik in Rechenanlagen.* Teubner, 1976.

[SV82] Y. Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Tan89] Andrew S. Tanenbaum. *Computer Networks.* Prentice–Hall, 2nd edition, 1989.

[Tro88] Ulrich Trottenberg, editor. *Proceedings of the 2nd International SUPRENUM Colloquium*, Parallel Computing Vol. 7 No. 3. North–Holland, September 1988.

[Upf82] Eli Upfal. Efficient schemes for parallel communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 55–59, August 1982.

[Upf89] Eli Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st ACM Annual Symposium on Theory of Computing*, pages 241–250, May 1989.

[Val82] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[Val90a] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Val90b] Leslie G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 943–971. Elsevier, 1990.

[Vui83] Jean Vuillemin. A very fast multiplication algorithm for VLSI implementation. *Integration, The VLSI Journal*, 1(1):39–52, 1983.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions.* Teubner, 1987.

[WF80] C.-L. Wu and T.-Y. Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, C–29(8):694–702, August 1980.

[Wis81] David S. Wise. Compact layouts of Banyan/FFT networks. In H. T. Kung, B. Sproull, and G. Steele, editors, *Proceedings of the CMU Conference on VLSI Systems and Computations*, pages 186–195, 1981.

Anhang A

Code für Bitonisches Sortieren

Wir werden in diesem Kapitel folgende Register verwenden:

Register	Inhalt
R_i	i
R_k	k
R_{bitk}	2^k
R_{biti}	2^i
R_{pnum}	Prozessornummer
R_A	Anfangsadresse Feld A
R_{x1}, R_{x2}	Zwischenergebnisse

Um eine Schleife zu programmieren, zum Beispiel die innere Schleife, müssen vor dem ersten Durchlauf die Register R_k, R_{bitk} gesetzt werden, vor jedem weiteren Durchlauf werden sie geändert. Die Verwaltung einer aufgerollten Schleife benötigt also $2 \cdot$Anzahl der Durchläufe Schritte.

Die IF Anweisungen haben folgende Form:

if bit k of $pnum = 1$ **then**

Dies läßt sich folgendermaßen programmieren:

CMP (R_{bitk}, R_{pnum})
JMP zeroset PC,#(Länge else Teil)+1
code für else Teil
JMP PC,#(Länge then Teil)+1
code für then Teil
NOP

Die Verwaltung einer IF Anweisung benötigt also 3 Schritte. Im Programm benötigen then und else Teil gleichviele Schritte. Die Codes für Tests auf Null und für Tests mit i haben

analoges Aussehen.

Die Zuweisungen haben folgende Form:

$A[pnum] := max(A[pnum], A[pnum + 2^k])$

Dies läßt sich folgendermaßen programmieren:

```
LD   R_A, R_pnum, R_x1          ; R_x1 := A[pnum]
ADD  R_pnum, R_bitk, R_x2
LD   R_A, R_x2, R_x2            ; R_x2 := A[pnum + 2^k]
NOP
CMP  (R_x1, R_x2)               ; Teste auf Maximum
JMP  negset PC,#3
ST   R_A, R_pnum, R_x2
JMP  PC,#3
ST   R_A, R_pnum, R_x1
NOP
```

Die Zuweisung benötigt also 8 Schritte. Da aber Prozessornummer und Modus auch vertauscht werden müssen, erhöht sich die Anzahl um 3 auf 11.

Eine geschachtelte IF Anweisung mit Zuweisung benötigt damit 17 Schritte.

Die Anzahl dieser Anweisungen ist gleich der Anzahl der Durchläufe der inneren Schleife und ergibt sich zu

$$I = \sum_{j=1}^{\log \tilde{N}} j = \frac{\log \tilde{N} \cdot (\log \tilde{N} + 1)}{2}$$

Die Anzahl der Durchläufe der äußeren Schleife ist $\log \tilde{N}$. Damit ergibt sich die Gesamtlaufzeit der Sortierung zu

$$I \cdot (17 + 2) + 2 \log \tilde{N}$$

Anhang B

Kontrolle der Fluent Machine

In diesem Anhang soll die prinzipielle Vorgehensweise bei der Konstruktion von Kontrollsignalen in hardwired Logik skizziert werden. Danach werden beispielhaft einige der Kontrollsignale der Fluent Machine beschrieben werden. Die weiteren Kontrollsignale lassen sich analog konstruieren.

B.1 Kontrollsignale des Prozessors

Für jedes Kontrollsignal des Prozessors muß man abhängig vom gerade geladenen Befehl (und eventuell abhängig vom Zustand anderer Kontrollsignale) in einem sogenannten *Startterm* entscheiden, ob es aktiv wird oder nicht. Zusätzlich muß man noch spezifizieren, ab welchem Takt des gerade auszuführenden Befehls das Singal aktiv sein soll. Soll das Signal länger als einen Takt aktiv sein, so muß man auch dieses mittels sogenannter *Halteterme* programmieren.

Um den Starttakt eines Signales zu spezifizieren, gibt es zwei Möglichkeiten: entweder erzeugt man alle Signale im ersten Takt nach dem Laden des Befehls und verzögert sie durch Registerstufen bis zum benötigten Takt oder man benutzt einen Zähler, der in jedem Befehl wieder bei 0 gestartet wird und spezifiziert den Anfangstakt mittels eines Zählerstandes.

Damit es in späteren Kapiteln möglich ist, die Architektur als Pipeline zu betreiben, wird hier die erste Methode benutzt.

Eine Schaltung für ein Kontrollsignal hat also folgende Form:

if (Startterm(Signal)=**true**) **and** (erster Takt)
  or (Halteterm(Signal)=**true**) **and** (Signal=aktiv im letzten Takt)
then Signal=aktiv
else Signal=inaktiv
fi

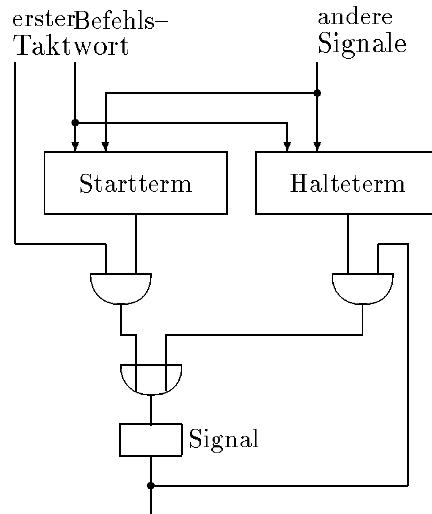

Abbildung B.1: Berechnung eines Kontrollsignales

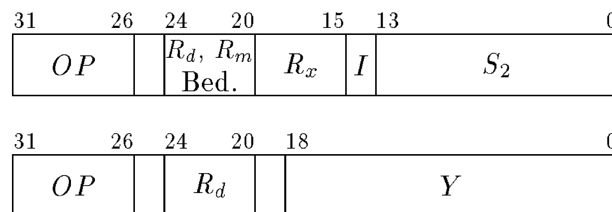

Abbildung B.2: Instruktionsformate

Findet man Schaltkreise für Startterm und Halteterm dieses Signales, so kann man das Signal mittels der in Abbildung B.1 gezeigten Schaltung berechnen.

Um einen Schaltkreis für Startterm bestimmen zu können, muß man die Kodierung des Instruktionssatzes kennen. Wir wollen uns weitestgehend an die Kodierung des Berkeley RISC Instruktionssatzes halten [PS82].

Dieser kennt nur zwei Instruktionsformate, die in Abbildung B.2 zu sehen sind. Das untere Format wird nur beim Befehl LDHI benutzt, das obere für alle anderen. OP bezeichnet den *Opcode*, der den Befehl mit 6 Bits spezifiziert. Die Opcodes werden so gewählt, daß die oberen beiden Bits des Opcodes zwischen den Befehlsgruppen LOAD, STORE, COMPUTE und SPRUNG Befehlen selektieren während die unteren 4 Bits die Nummer eines Befehles innerhalb seiner Gruppe angeben. Die anderen Felder spezifizieren die benötigten Register und Konstanten. Hierbei gibt I an, ob S_2 die Nummer eines Registers oder eine Konstante enthält. Stehen in einem Feld mehrere Bezeichner, so wird abhängig vom Befehl einer dieser in diesem Feld gespeichert. Benutzt ein Befehl nicht alle Felder, so können die unbenutzten Felder beliebigen Inhalt haben.

Da sich die einzelnen Befehle und ihre Argumente sehr einfach aus dem Opcode eines Befehles ergeben, können wir die Startterme in einer übersichtlichen Weise formulieren, aus der sich der gesuchte Schaltkreis in einfacher Weise ergibt.

Um Schaltkreise für Halteterme zu bestimmen, muß man lediglich wissen, wie lange ein Signal aktiv sein soll. Hierzu benutzt man einen Zähler, der bei der Aktivierung des Kontrollsignales bei Null anfängt zu zählen. Man hält das Signal dann solange, bis ein bestimmter Zählerstand erreicht ist. Betreibt man den Prozessor als Pipeline, so erscheint in jedem Takt ein neuer Befehl, Halteterme werden dort nicht benutzt. Wir beschreiben also Halteterme durch einen Vergleich mit dem Zählerstand. Der gesuchte Schaltkreis ist einfach zu erhalten.

Beispielhaft sollen drei Kontrollsignale beschrieben werden: die Clock eines Registers, das *oe* Signal eines Treibers, das select Signal eines Multiplexers. Wir wählen als Register den SP_0, als Treiber den Treiber vom SP_0 zum L Bus und als Multiplexer den Multiplexer vor dem SP_0.

Clock des SP_0 Der SP_0 wird geclockt bei allen Befehlen der Gruppen LOAD und COMPUTE und bei POPi, falls $R_d = SP_0$ und bei den Befehlen JSR0, POP0, PUSH0. Es gibt keine Halteterme.

MUX Select Der linke Eingang des Multiplexers wird bei den Befehlen JSR0, POP0, PUSH0 gewählt, um den inkrementierten oder dekrementierten Inhalt des Stackpointers zu clocken. Bei allen anderen Befehlen wird der rechte Eingang gewählt. Es gibt keine Halteterme.

Treiber oe Der Treiber wird enabled bei LD, ST JMP, JSRi, PUTPSW und Compute Befehlen, falls $R_x = SP_0$ und bei PSH0 und POP0 Befehlen. Der Treiber bleibt insgesamt 4 Takte aktiv.

Damit können die Startterme der 3 Signale beschrieben werden, die Anzahl der Halteterme wurde bereits angegeben.

```
SPOCK:=(((OP in LOAD) OR (OP in COMPUTE) OR (OP=POPi)) AND (Rd=SP0))
       OR (OP=JSR0) OR (OP=PSH0) OR (OP=POP0)

MUXSELECT:=(OP=JSR0) OR (OP=PSH0) OR (OP=POP0)

SP0LOE:=(((OP=LD) OR (OP=ST) OR (OP=JMP) OR (OP=JSRi) OR (OP=PUTPSW)
           OR (OP in COMPUTE)) AND (Rx=SP0))
         OR (OP=PSH0) OR (OP=POP0)
```

In den Datenpfaden des Prozessors gibt es 10 Multiplexer, 6 Register, 20 Treiber und ein Dual Port RAM. Das Dual Port RAM benötigt 12 Steuersignale (für jeden Port 5 Adressleitungen und ein r/w–Signal), alle anderen Komponenten ein Steuersignal. Rechnet man

durchschnittlich 40 Gatteräquivalente für ein Steuersignal, so erhält man als Gesamtkosten der Logik $48 \cdot 40 \approx 2000$. Die Annahme von 40 Gatteräquivalenten erscheint bei der Komplexität der oben betrachteten Steuersignale und der Möglichkeit zur Vereinfachung von Termen durchaus realistisch.

B.2 Kontrollsignale des Netzknotens

Im Netzwerkknoten hängen die Startterme der Kontrollsignale nicht von einem Befehl, sondern von der Phase, von den Ergebnissen der Adressvergleicher, vom Modus eines selektierten Paketes und/oder vom Inhalt einer Richtungsqueue ab. Halteterme gibt es nicht, da in jedem Takt ein neues Paket versendet werden kann.

Wir wollen hier ähnlich vorgehen wie beim Prozessor und beispielhaft drei Kontrollsignale auswählen: das ck_{in} Signal der Richtungsqueue 3 und die Select Signale zweier Multiplexer des rechten Knotens, direkt vor dem Adress Shift zur Auswahl des zu versendenden Paketes und direkt hinter dem Adress Shift rechts zum Einfügen des GHOST Modus, falls das betreffende Paket nach links will.

ck_{in} Die Richtungsqueue 3 wird nur in Phase 2 beschrieben, wenn das selektierte Paket den Modus READ hat und tatsächlich verschickt werden kann, wenn also der Eingangspuffer am Ende des Links in der Richtung, in die das Paket will, nicht voll ist.

Select Paket In diesem MUX wird in Phase 2 das linke Paket selektiert, wenn der Vergleicher auf Vorzeichen eine 1 liefert und der linke Eingangspuffer nicht leer ist. In Phase 4 bzw. 6 wird das durch die Richtungsqueue 2 bzw. 1 bestimmte Paket selektiert.

Select Ghost Dieser MUX hat nur in Phase 2 eine Funktion, da in den Phasen 4 und 6 Pakete nur noch aus einem Datum bestehen. Der linke Eingang wird gewählt, wenn das selektierte Paket über den rechten Ausgangslink verschickt werden soll, wenn also das niedrigstwertige Bit seiner Adresse den Wert 1 hat. Sonst wird der rechte Eingang gewählt.

Damit kann man die Kontrollsignale in einer Notation ähnlich der im vorigen Abschnitt darstellen. Der Struct PAKET repräsentiert das selektierte Paket vor dem Adress Shift. VOLL bezeichnet die Rückleitungen, die informieren, ob die folgenden Eingangspuffer voll sind. LEER bezeichnet den Zustand der eigenen Eingangspuffer.

```
CKIN:=(PHASE=2) AND (PAKET.MODUS=READ)
      AND ((PAKET.ADRESSE[0] AND NOT VOLL[RECHTS])
           OR (NOT PAKET.ADRESSE[0] AND NOT VOLL[LINKS]))
```

```
SELPAKET:=   ((PHASE=2) AND (VGL=1) AND (NOT LEER[LINKS]))
          OR ((PHASE=4) AND (RICHTUNGSQUEUE2=LINKS))
  OR ((PHASE=6) AND (RICHTUNGSQUEUE1=LINKS))

SELGHOST:=(PHASE=2) AND (PAKET.ADRESSE[0]=1)
```

Ein Netzwerkknoten enthält 8 Multiplexer und 7 FIFO Schlangen. Die FIFO Schlangen benötigen zwei Kontrollsignale (ck_{in} und ck_{out}), die Multiplexer benötigen ein Select Signal. Setzt man als Kosten pro Kontrollsignal 40 Gatteräquivalente wie beim Prozessor, so erhält man $22 \cdot 40 = 880$. Da zusätzlich noch die Phasen verwaltet werden müssen, setzt man die Kosten mit 1000 fest.