# Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation
vorgelegt von

Diplom-Informatiker
Mario Schölzel

geboren am 17. November 1975 in Lübben.

| | |
|---|---|
| Gutachter: | Prof. Dr. rer. nat. habil. Peter Bachmann |
| Gutachter: | Prof. Dr. Rolf Drechsler |
| Gutachter: | Prof. Dr.-Ing. Sorin A. Huss |

Tag der mündlichen Prüfung: 09.10.2006

**Danksagung**

Diese Arbeit ist während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl von Professor Peter Bachmann entstanden. Bei ihm möchte ich mich für die fachliche Betreuung der Arbeit, seine konstruktive Kritik, die eingeräumten Freiräume und seine entgegengebrachte Geduld insbesondere gegen Ende der Arbeit bedanken. Allen meinen Kolleginnen und Kollegen am Lehrstuhl möchte ich für die angenehme Arbeitsatmosphäre danken.

Professor Vierhaus hat mich durch fachliche Hinweise und durch Diskussionen im Rahmen des gemeinsam durchgeführten Lehrstuhlseminars unterstützt. Dafür danke ich ihm sehr.

Ohne Thomas König und Felix Krüger, die beide den größten Teil der prototypischen Implementierung von DESCOMP durchgeführt haben, hätte die Arbeit nicht gelingen können. Ihnen gilt deshalb mein besonderer Dank. Ebenso Andy Heinig, Sebastian Scholz, Katja Winder, Christian Köhler und Michael Vogel, die durch ihre Studienarbeiten, Diplomarbeiten und ihre Tätigkeit als studentische Hilfskraft Beiträge und Anregungen zu dieser Arbeit geliefert haben.

Besonderer Dank gebührt Susann, die mich in den letzten Monaten so großartig unterstützt hat und mit unendlicher Geduld viele Rechtschreibfehler gefunden hat, die ich zuvor mit großer Sorgfalt eingearbeitet habe.

# Inhaltsverzeichnis

# 1 Einleitung

In einigen Einsatzbereichen digitaler Systeme sind die Anforderungen an die Verarbeitungsgeschwindigkeit bei gleichzeitig niedrigem Platz- und Stromverbrauch so hoch, dass diese Anforderungen nur durch die Verwendung spezialisierter Hardware erfüllt werden können. Zugleich sind die besonders effizient abzuarbeitenden zeitkritischen Algorithmen in solchen Systemen oft fixiert [39]. Eine spezielle Hardwarelösung für jeden Algorithmus bietet im Allgemeinen die höchste Ausführungsgeschwindigkeit, ist aber unflexibel und sehr zeitintensiv in der Realisierung, was im Widerspruch zu den geforderten kurzen Entwicklungszeiten vieler digitaler Systeme steht. Kleinere Modifikationen an den Algorithmen sind bei einer Hardwarelösung in einer späteren Entwurfsphase des Systems ebenfalls kaum noch möglich. Anwendungsspezifische Prozessoren bieten in dieser Hinsicht einen guten Kompromiss zwischen Flexibilität und Ausführungsgeschwindigkeit [73], da sie in der Lage sind, leicht modifizierte Varianten der zeitkritischen Algorithmen und andere zeitunkritische Teile der Anwendung abzuarbeiten. Die zeitkritischen Algorithmen müssen dabei nicht immer so schnell wie möglich, sondern nur innerhalb gegebener Zeitschranken ausgeführt werden, so dass auch ein Kompromiss zwischen Ausführungsgeschwindigkeit sowie Hardwarekosten und Stromverbrauch des anwendungsspezifischen Prozessors zu finden ist. Die unter diesen Optimierungszielen durchgeführte Bestimmung der Parameter eines Prozessors wird als Design-Space-Exploration (DSE) bezeichnet. Bekannte Verfahren zur DSE weisen eine hohe Komplexität auf, die dadurch entsteht, dass die gegebene Anwendung mit einem Compiler für eine Architektur mit festen Parametern übersetzt wird. Die Parameter der Architektur sowie der Compiler werden bei diesen Ansätzen in einem iterativen Prozess immer wieder modifiziert, um eine geeignete Parameterkonfiguration zu finden. Die so zu untersuchende Menge von Parameterkonfigurationen eines Prozessors ist extrem groß. Der im Rahmen dieser Dissertation entwickelte DESCOMP[1]-Ansatz vermeidet die hohe Komplexität bekannter DSE-Verfahren, indem er die gegebenen Algorithmen wie ein Compiler übersetzt, aber ohne dass die genauen Parameter der Zielarchitektur bekannt sind. Aus dem Zielcode der so übersetzten Algorithmen werden dann die Parameter des anwendungsspezifischen Prozessors, wie beispielsweise die Anzahl der Registerbänke und die Anzahl und Art parallel ausführbarer Operationen, abgeleitet.

[1] DESign by COMPilation

## 1.1 Einsatzbereiche

Die Bereiche, in denen anwendungsspezifische Prozessoren eingesetzt werden, sind eingebettete Systeme und SoCs (*System on Chip*), wie sie beispielsweise in Fahrzeugen, Mobiltelefonen, Geräten für das digitale Fernsehen und der Datenübertragung vorkommen [85] und in denen große Datenmengen zu verarbeiten sind. Diese Systeme werden immer komplexer, da immer höhere Anforderungen an sie gestellt werden [56]. Oft bestehen sie aus mehreren Prozessorkernen mit lokalen und globalen Speichern, die durch Busse verbunden sind. Anwendungsspezifische Prozessoren sind oft nur eine Komponente darin [60, 85]. Vor dem Entwurf solcher Systeme ist das auszuführende Anwendungsspektrum fixiert, so dass der Entwurf unter anderem die Zerlegung der Anwendung in Teile, die durch Hardware bzw. Software zu realisieren sind, umfasst [39]. Weiterhin ist die Auswahl der genutzten Hardwarekomponenten und die Zuordnung der in Software zu realisierenden Teile zu den Prozessorkernen im System zu treffen. Werkzeuge, die den Entwickler bei dieser Arbeit unterstützen sind z. B. in [11, 62, 82] zu finden. Sowohl in eingebetteten Systemen als auch in SoCs werden häufig seriengefertigte Prozessoren verwendet, da diese unmittelbar verfügbar und kostengünstig sind [60, 63, 85]. Entsprechende Prozessoren sind beispielsweise der TMS320C6x, ADSP2106x-SHARC oder TriMedia [1, 3, 31]. Allerdings sind diese Prozessoren in einem großen Anwendungsspektrum einsetzbar, weshalb sie einen relativ großen Funktionsumfang bieten. Da SoCs und eingebettete Systeme sehr häufig in großer Stückzahl gefertigt werden und/oder in mobilen Systemen zum Einsatz kommen, muss beim Entwurf besonders auf einen geringen Platz- und Stromverbrauch geachtet werden. Seriengefertigte Prozessorkerne führen deshalb oft zu einem höheren Platz- und Stromverbrauch als erforderlich, da sie nicht optimal an die Anforderungen der auszuführenden Software angepasst sind.

Der Einsatz von Prozessoren, die sich an die Anforderung der Anwendung anpassen lassen, erlaubt eine hohe Ausführungsgeschwindigkeit und das Einsparen von Platz und Strom. Unter anderem auch deshalb, weil verschiedene Algorithmen, die sonst durch unterschiedliche Komponenten im System abgearbeitet werden, dann auf einem Prozessor ausgeführt werden können, der an die Anforderungen dieser Algorithmen angepasst ist. Somit kann durch Synergieeffekte nicht nur Hardware sondern auch ein Teil des Verbindungsnetzwerkes im SoC eingespart werden. Neben den oben bereits genannten Prozessoren werden von der Industrie auch Prozessorkerne angeboten, deren Busbreite und Registeranzahl angepasst werden kann [27] oder die auch umfangreichere Anpassungen der Architektur zulassen, wie beispielsweise der Tensilica Xtensa LX Prozessorkern [84] und die Lx-Architektur von Hewlett-Packard und STMicroelectronics [73].

## 1.2 Motivation

Anhand des TriMedia32 Prozessors [2] wird eine Motivation für die Zielstellung dieser Dissertation geliefert. Der Prozessor wurde speziell für Multimediaanwendungen entworfen. Er verfügt insgesamt über 23 Operatoren zum Ausführen von elf verschiedenen Operationstypen. Abhängig vom Typ einer Operationen können bis zu fünf Operationen gleichzeitig ausgeführt werden, jede davon in einer funktionalen Einheit. Die aus [48] entnommene Tabelle 1.1 gibt einen Überblick über die mögliche Anordnung der Operationen in den funktionalen Einheiten.

| **Operator-typ** | **funktionale Einheiten** | | | | | **Latenz** | **Beschreibung** |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | |
| *ALU* | x | x | x | x | x | 1 | Integer- und Bitoperationen |
| *DMem* | | | | x | x | 3 | Lade- und Speicheroperationen |
| *DMemSpec* | | | | | x | 3 | Speicher- und Cache-Management |
| *Shifter* | x | x | | | | 1 | Schiebe- und Rotieroperationen |
| *DSPALU* | x | | x | | | 2 | DSP-Operationen |
| *DSPMul* | | x | x | | | 3 | Multiplikation und Addition |
| *Branch* | | x | x | x | | 4 | Sprünge |
| *FALU* | x | | | x | | 3 | Gleitkommaarithmetik |
| *IFMul* | | x | x | | | 3 | Gleitkommamultiplikation |
| *FComp* | | | x | | | 1 | Gleitkommavergleiche |
| *FTough* | | x | | | | 17 | Division und Wurzelberechnung |

Tabelle 1.1: Übersicht über die 11 verschiedenen Operatortypen des TriMedia32 Prozessors.

Jede Operation muss von einer funktionalen Einheit ausgeführt werden, in der ein Operator existiert, der die entsprechende Operation ausführen kann, verdeutlicht durch ein Kreuz in der entsprechenden Spalte in Tabelle 1.1. Alle Operatoren in einer funktionalen Einheit verwenden dieselben Lese- und Schreibports der Registerbank. Deshalb kann in jeder funktionalen Einheit nur eine Operation gleichzeitig ausgeführt werden. Einige Kombinationen von Operationen können somit nicht parallel verarbeitet werden, wie beispielsweise zwei *Shift*- und zwei *DSPALU*-Operationen. In diesem Fall müsste die funktionale Einheit 1 zwei Operationen gleichzeitig abarbeiten. Der Grund für die in Tabelle 1.1 angegebene Auswahl und Zuordnung von Operatoren zu funktionalen Einheiten ist unter anderem das Anwendungsspektrum, das typischerweise vom TriMedia32 Prozessor ausgeführt wird und für das sich

diese Zuordnung als vorteilhaft erwiesen hat. Es existieren bis zu 216.000 Möglichkeiten, die in Tabelle 1.1 angegebenen Operatoren den fünf funktionalen Einheiten zuzuordnen. Wird nicht festgelegt, in wie vielen funktionalen Einheiten ein Operatortyp vorhanden sein muss, so gibt es sogar bis zu $2^{55}$ Möglichkeiten. Noch mehr Möglichkeiten gibt es, wenn auch die Anzahl der funktionalen Einheiten nicht festgelegt ist. Für den TriMedia64 Prozessor [45] wurde eine Design-Space-Exploration durchgeführt, durch die eine geeignete Zuordnung von Operatoren zu funktionalen Einheiten für ein gegebenes Anwendungsspektrum gefunden werden sollte [30]. Auf den entsprechenden Ansatz für diese sehr komplexe Aufgabe wird im Abschnitt 3.2.3 detaillierter eingegangen.

## 1.3 Zielstellung

In dem vorangegangenen Beispiel wurde angedeutet, wie groß der Suchraum für die DSE eines anwendungsspezifischen Prozessors werden kann. Bekannte DSE-Ansätze nutzen das in der Einleitung beschriebene iterative Vorgehen zur Bestimmung der Prozessorparameter, was neben einem hohen zeitlichen Aufwand oft auch eine Interaktion mit dem Entwickler bedingt, um beispielsweise Bereiche des Suchraums während der DSE auszuschließen. Das Ziel dieser Arbeit ist es, ein methodisches Vorgehen und die dafür benötigten Techniken zu entwickeln, so dass für ein fest vorgegebenes Spektrum von Algorithmen und zugehörigen Zeitschranken die DSE für Prozessoren von datenflussdominierten Anwendungen vollständig automatisiert und ohne mehrfaches Übersetzen der Anwendung durchgeführt werden kann. Der erzeugte Prozessor muss dabei noch flexibel genug sein, um auch andere Algorithmen, für die er nicht optimiert wurde, abzuarbeiten. In Abschnitt 2 werden die genutzten Modelle spezifiziert und der prinzipielle DESCOMP-Ansatz vorgestellt, der sich in eine lokale und globale Optimierungsphase gliedert. Beide Optimierungsphasen werden in den Abschnitten 4 und 5 beschrieben. In Abschnitt 6 wird die Qualität der vorgestellten Techniken durch einen Vergleich mit einem existierenden automatisch arbeitenden DSE-Ansatz überprüft.

# 2 Modellbildung

In diesem Abschnitt wird das Prozessormodell beschrieben, dessen Parameter an eine Anwendung angepasst werden. Die Zusammenhänge, die zwischen Strom- und Platzverbrauch sowie Leistungsfähigkeit eines solchen Prozessors bestehen, werden zueinander in Beziehung gesetzt. Daraus leiten sich die Gründe für die Wahl der in dieser Arbeit verwendeten Optimierungsstrategie ab. Neben der Definition des Prozessormodells wird auch eine formale Definition für Programmdarstellung und Zusammenhang zwischen Programmdarstellung und Prozessormodell gegeben.

## 2.1 Prozessormodell

Durch die DSE werden die Parameter eines Prozessors so angepasst, dass eine gegebene Anwendung oder Teile davon innerhalb gewisser Zeitschranken ausgeführt werden können. Die Hardwarekosten und/oder der Stromverbrauch des Prozessors sollen dabei minimiert werden. Außerdem muss sich der angepasste Prozessor leicht in Hardware synthetisieren lassen. Es gibt mehrere Möglichkeiten, die erforderliche Ausführungsgeschwindigkeit der Anwendung auf einem Prozessor zu erreichen:

- durch eine Anpassung der Taktfrequenz des Prozessor,
- durch Optimierungen, die vom Prozessor während der Programmabarbeitung durchgeführt werden,
- durch eine Anpassung der Datenwortbreite des Prozessors,
- durch die parallele Ausführung von Operationen.

Der Erhöhung der Taktfrequenz ist durch die Verzögerung in den Pfaden der Prozessorlogik eine Grenze gesetzt. Um die Taktfrequenz darüber hinaus zu erhöhen, müssen die Verzögerungen auf den Pfaden verkleinert werden, was durch kleinere Strukturgrößen oder Pipelines erreicht werden kann. Kleinere Strukturgrößen erlauben schnellere Transistoren, wodurch sich die Verzögerungen auf den Pfaden verkleinern [91]. Allerdings ist die Strukturgröße ein Parameter der oft vorgegeben ist und nicht im Rahmen einer DSE bestimmt wird. Skalare Architekturen besitzen eine Pipeline, wodurch lange Pfade in der Prozessorlogik in mehrere kurze Pfade zerlegt werden. Dadurch wird die Verzögerungen auf diesen Pfaden verkleinert und die Möglichkeit geschaffen, Operationen überlappend mit einer höheren Taktfrequenz auszuführen. Mit der Verwendung von Pipelines sind auch

Änderungen an der Steuerlogik des Prozessors verbunden, um beispielsweise Konflikte innerhalb der Pipelinestufen zu behandeln. Diese Änderungen führen zu einem irregulären Steuer- und Datenpfad des Prozessors. Ebenso wie die folgenden Optimierungen, die viele superskalare Prozessoren, die gewisse Operationen parallel ausführen können, während der Programmabarbeitung durchführen:

- Sprungvorhersagen, um die Instruktionen von der richtigen Programmposition in den Befehlsdekoder zu laden,
- das Umordnen von Operationen während der Programmausführung, um den Datenpfad und vorhandene Pipelines besser auszulasten, und
- die spekulative Ausführung von Operationen, um Zeitverluste bei bedingten Sprüngen und Speicherzugriffen zu verringern.

Alle diese Maßnahmen zur Erhöhung der Ausführungsgeschwindigkeit bedingen einen sehr komplexen und irregulären Steuer- und Datenpfad, womit die automatische Adaption eines solchen Steuer- und Datenpfades an die Anforderungen einer Anwendung sehr schwierig ist.

Eine Anpassung der Datenwortbreite ist eine weitere Möglichkeit, die Leistungsfähigkeit eines Prozessors an die Anforderungen der Anwendung anzupassen [27, 68]. So kann durch die Wahl einer geeigneten Wortbreite die gewünschte Präzision bei arithmetischen Operationen erreicht werden. Allerdings bietet die alleinige Verwendung dieser Art der Adaption nur eine sehr eingeschränkte Möglichkeit zur Anpassung eines Prozessors an die Anforderungen der Anwendung.

Eine Erhöhung der Parallelität im Prozessor kann durch zusätzliche Operatoren, z. B. weitere ALUs (*Arithmetic Logical Unit*), erreicht werden. Ein Ansatz, der sich mit der Erhöhung der Parallelität in superskalaren Prozessoren beschäftigt, ist in [75] vorgestellt. Allerdings ist, wie oben bereits beschrieben, das Integrieren weiterer Operatoren in einen irregulären Steuer- und Datenpfad einer superskalaren Architekturen schwierig. In [63, 64] werden zur Erhöhung der Parallelität zusätzliche Operatoren in den Datenpfad eines skalierbaren DSPs (*Digital Signal Processor*) eingebunden. Der existierende DSP-Kern wird so durch zusätzliche Hardware für rechenintensive Anwendungen erweitert. In [38] wurde als zusätzlicher Operator ein FPGA (*field programmable gate array*) in den Datenpfad des LEON2-Prozessors integriert, um durch dynamische Rekonfiguration des FPGAs rechenintensive Aufgaben auf dieses auszulagern. Ansätze, die zusätzliche Operatoren in die Datenpfade vorhandener Prozessorkerne integrieren, können von dem optimierten Prozessorkern profitieren [64], sind aber bei der Bereitstellung zusätzlicher Parallelität begrenzt, weil die zusätzlichen Operatoren an die vorhandene Registerbank und das Speicherinterface des Prozessors angebunden werden müssen oder zu sehr irregulären Verbindungsnetzwerken führen, die die Nutzbarkeit der zusätzlichen Operatoren für andere Algorithmen erschweren. Die Konzeption

einer massiv parallelen Architektur, die die Unterbringung mehrerer Dutzend Prozessorkerne auf einem Chip unterstützt, ist die EDGE-Architektur (*Explicit Data Graph Execution*) [25]. Diese Architektur ist allerdings eher für ein sehr breites Spektrum von Anwendungen gedacht und somit nicht zur Adaption an eine spezifische Anwendung geeignet. Darüber hinaus wurde in [69] gezeigt, dass Architekturen mit dieser hohen Parallelität bei vielen Anwendungen aus dem Signalverarbeitungsbereich, was ein Haupteinsatzgebiet anwendungsspezifischer Prozessoren ist, nicht zu einer deutlich besseren Ausführungszeit führen, während Architekturen mit weniger Prozessorkernen, die dafür besser an die Anwendung angepasst sind und ein geeignetes Verbindungsnetzwerk besitzen, höhere Ausführungsgeschwindigkeiten erlauben.

### 2.1.1 Datenpfadmodell

Als Prozessormodell wird in dieser Arbeit ein Very-Long-Instruction-Word-Prozessor (VLIW-Prozessor) gewählt, der sich gut an die algorithmischen Anforderungen einer Anwendung anpassen lässt, indem im Datenpfad die Anzahl der parallel arbeitenden Operatoren und damit die Anzahl der parallel ausführbaren Operationen variiert wird. Außerdem ist der Daten- und Steuerpfad dieser Architektur sehr regelmäßig aufgebaut, so dass eine automatisierte Synthese des an eine Anwendung angepassten Steuer- und Datenpfades möglich ist. Wie in [50] beschrieben, kann zur Synthese des Datenpfades beispielsweise auf fertige Komponenten aus einer VHDL-Bibliothek zurückgegriffen werden, aus denen der Datenpfad zusammengesetzt wird. Solche Komponenten sind z. B. Registerbänke, Addierer und Shifter. Die Abbildung 2.1 zeigt den schematischen Aufbau des Steuer- und Datenpfades eines VLIW-Prozessors.

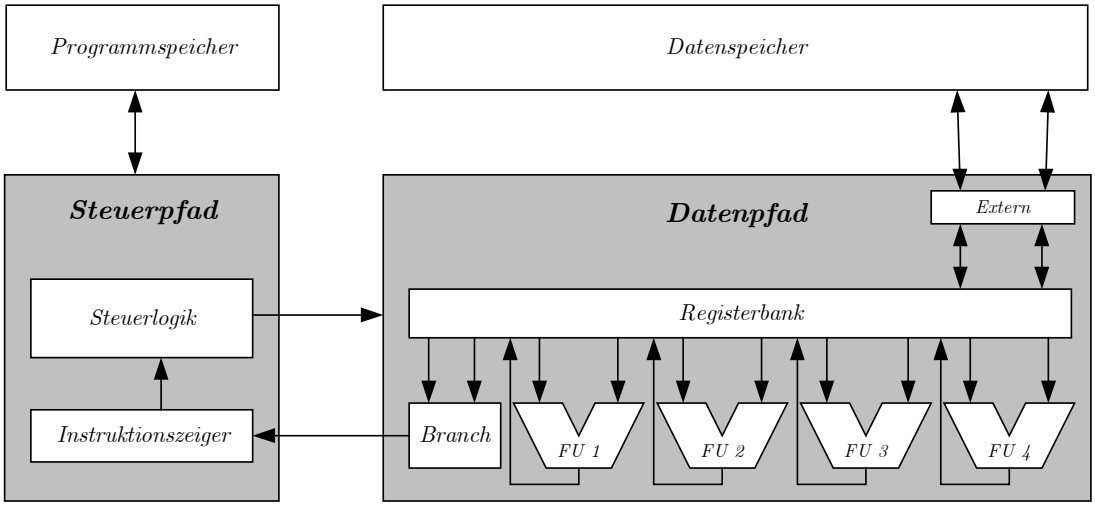



Abbildung 2.1: Steuer und Datenpfad einer einfachen VLIW-Architektur mit einer zentralen Registerbank.

Der Datenpfad enthält eine Registerbank mit allgemeinen Registern. Jede **funktionale Einheit** (FU) im Datenpfad stellt die Hardware bereit, um gewisse Operationen auszuführen. Die zur Ausführung einer Operation benötigte Hardware in einer FU wird **Operator** genannt. Jeder Operator kann nur Operationen eines bestimmten Typs ausführen, z. B. eine Addition oder eine Multiplikation. Der **Typ** einer Operation $v$ wird mit $type(v)$ bezeichnet. Die Menge aller Operationstypen, für die ein Operator im Prozessor bereitgestellt werden kann, wird mit $\mathcal{O}$ bezeichnet. Für jeden Operationstypen $t \in \mathcal{O}$ sind die Hardwarekosten $opCost(t)$ bekannt, die zur Implementierung des entsprechenden Operators in Hardware benötigt werden. Diese Hardwarekosten repräsentieren die benötigte Chipfläche, die mit der Anzahl der benötigten Transistoren korreliert. Zur Vereinfachung werden keine Synergieeffekte berücksichtigt. Diese treten beispielsweise auf, wenn in einer FU bereits ein Operator eine Addition implementiert und noch ein Operator für eine Subtraktion implementiert werden soll. Die Subtraktion verursacht dann nur noch geringe zusätzliche Hardwarekosten. Um solche Synergieeffekte zu berücksichtigen, können für die Design-Space-Exploration alle Additions- und Subtraktionsoperationen in Operationen desselben Typs umgewandelt werden, der dann die Kosten des Operators hat, der sowohl eine Addition als auch eine Subtraktion ausführen kann. Jede funktionale Einheiten fasst somit die Operatoren zusammen, die gemeinsam dieselben Ports der Registerbank nutzen[1]. Alle FUs arbeiten parallel und können gleichzeitig auf die Registerbank zugreifen. Jede FU führt aber nur eine Operation gleichzeitig aus. Der Zugriff auf die Registerbank geschieht über die internen Lese- und Schreibports. Jede FU ist durch zwei Leseports und einen Schreibport mit der Registerbank verbunden. Zwei weitere Leseports in der Registerbank werden von der Branch-Einheit verwendet, um abhängig von einem Registerinhalt einen bedingten Sprung durch das Laden eines Wertes in den Instruktionszeiger durchzuführen. Neben den Verbindungen der Registerbank zu den funktionalen Einheiten gibt es auch noch Verbindungen zwischen der Registerbank und externen Speichermedien. Diese greifen über externe Lese- und Schreibports auf die Registerbank zu. In der in Abbildung 2.1 angegebenen einfachen VLIW-Architektur ist das nur der Speicher. Der mit *Extern* bezeichnete Teil des Datenpfades enthält die benötigten Operatoren für Lese- und Schreibzugriffe auf den Speicher. Dadurch können Daten zwischen der Registerbank und dem Speicher kopiert werden. Es ist zulässig, mehrere Speicherzugriffe gleichzeitig durchzuführen, wie es beispielsweise auch beim Gepard-DSP [27], dem ADSP-21061 [3] und der Lx-Architektur in [73] der Fall ist. Der gleichzeitige Speicherzugriff erfolgt entweder auf getrennte Speicherbänke oder auch auf dieselbe Speicherbank, in der der Daten- und Adressbus mehrfach vorhanden ist. Die Auswahl eines geeigneten Speichermodells wird in dieser Arbeit nicht behandelt.

[1] In der englischen Literatur auch als *issue slot* bezeichnet.

Gesteuert wird der Datenpfad in jedem Takt durch ein Instruktionswort, wie es in Abbildung 2.2 beispielhaft dargestellt ist.


Abbildung 2.2: Instruktionswort für einen VLIW-Prozessor mit einer Branch-Einheit und vier funktionalen Einheiten im Datenpfad.

In einem solchen Instruktionswort wird an einer festen Position für jede FU im Datenpfad die auszuführende Operation (*op1*, ..., *op4*) sowie deren Quell- und Zielregister (*src*, *dst*) kodiert. Alle Operationen verwenden Prozessorregister als Operanden. Sollen Werte im Speicher manipuliert werden, dann müssen diese über Ladeoperationen zuvor in Register geholt und anschließend zurückgeschrieben werden. Die VLIW-Instruktionen sind statisch geplant. Das bedeutet, dass sie in der Reihenfolge ausgeführt werden, in der sie auftreten. Ein Umordnen der Operationen in den Instruktionen während der Abarbeitung des Programms findet nicht statt. Auch Sprungvorhersagen und die spekulative Ausführung von Operationen werden nicht durchgeführt. Im Steuerpfad des Prozessors müssen die Instruktionen somit nur dekodiert und die Steuersignale an den Datenpfad gegeben werden. Durch die Planung der Operationen in VLIW-Instruktionen während der Erzeugung des Zielprogramms durch den Compiler muss sichergestellt sein, dass jede Operation einer FU zugeordnet ist, die sie auch ausführen kann. Da Programm- und Datenspeicher getrennt sind, kann es auch nicht zu Konflikten beim Laden einer Programminstruktion und eines Datums aus dem Datenspeicher kommen. Es entfällt somit ein großer Teil der bei superskalaren Prozessoren notwendigen Steuerlogik. Durch diese Eigenschaften bietet die VLIW-Architektur gegenüber einer superskalaren Architektur folgende Vorteile:

- Die Ausführungsgeschwindigkeit kann durch zusätzliche FUs im Datenpfad erhöht werden, sofern die Anwendung genügend parallel ausführbare Operationen enthält.
- Daten- und Steuerpfad sind sehr regelmäßig aufgebaut, wodurch sich der angepasste Datenpfad sowie der zugehörige Steuerpfad gut in Hardware synthetisieren lässt.
- Das Zeitverhalten einer statisch geplanten VLIW-Architektur ist gut vorhersagbar, weil die zur Ausführung eines Programms benötigte Anzahl der Takte mit der Instruktionsanzahl korreliert.

Nachteilig wirken sich bei einer statisch geplanten Architektur unvorhersehbare Latenzzeiten von Lade- und Speicheroperationen aus. Beispielsweise wenn ein Wert nicht aus dem Cache geholt werden kann. Dadurch muss die Ausführung aller Operationen – auch der, die nicht datenabhängig von diesem Wert sind – angehalten werden, bis der Wert

verfügbar ist. Aufgrund solcher Verzögerungen kann die Ausführungszeit eines Programms nicht immer exakt aus der Anzahl der abzuarbeitenden Instruktionen bestimmt werden. Das Problem kann aber beispielsweise durch einen an die Speicherzugriffsreihenfolge der Anwendung angepassten Cache oder durch die Unterbringung des Speichers auf dem Chip abgemindert werden [46].

Ein weiteres Problem in VLIW-Architekturen stellt die von der Portanzahl abhängige Komplexität der Registerbank dar, die mit wachsender Parallelität stark zunimmt und den Platz- und Stromverbrauch in einem VLIW-Prozessor dominiert [4, 87]. Um trotz hoher Parallelität die Portanzahl in der Registerbank niedrig zu halten, kann der Datenpfad in **Cluster** zerlegt werden [49]. Eine schematische Darstellung eines solchen Prozessors ist in Abbildung 2.3 angegeben.

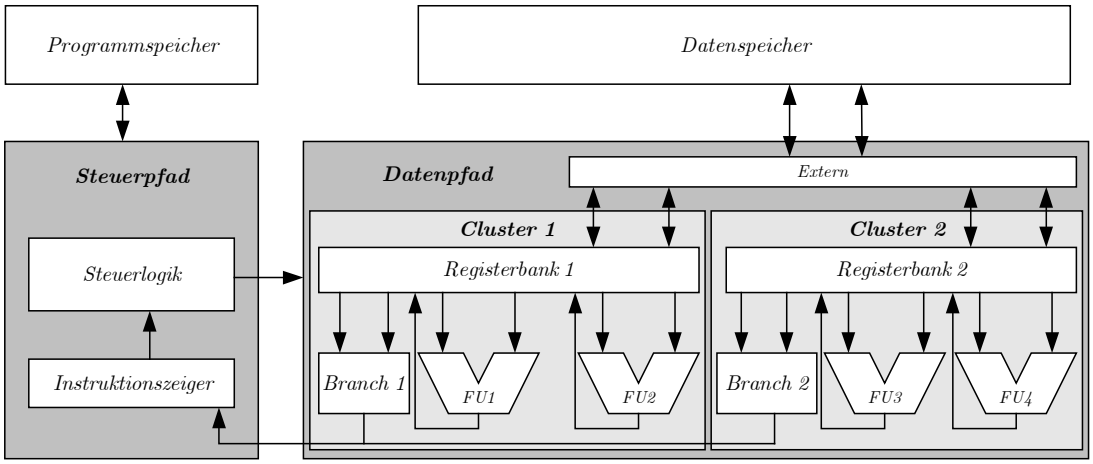



Abbildung 2.3: Steuer- und Datenpfad einer geclusterten VLIW-Architektur.

In einem geclusterten Datenpfad sind die funktionalen Einheiten in Clustern gruppiert. Jeder Cluster besitzt eine lokale Registerbank, auf die nur die funktionalen Einheiten dieses Clusters zugreifen können. Somit bleibt die vom Prozessor bereitgestellte Parallelität hoch, die Anzahl der FUs, die auf dieselbe Registerbank zugreifen, wird aber deutlich reduziert und somit auch die Anzahl der Ports in jeder Registerbank. Der Datenaustausch zwischen verschiedenen Clustern erfolgt über ein Verbindungsnetzwerk, das durch Kopieroperationen gesteuert wird, die im Instruktionswort mit kodiert sind. Da das Kopieren von Daten über dieses Verbindungsnetzwerk Zeit kostet, müssen Operationen in anderen Clustern, die diese Daten benötigen, entsprechend später ausgeführt werden. Somit ist eine geclusterte Architektur nur vorteilhaft, wenn durch die erforderlichen Kopieroperationen die Ausführungszeit der Anwendung nicht beeinträchtigt wird oder die eintretenden Verzögerungen durch eine höhere Taktung des Prozessors ausgeglichen bzw. durch geschickte Anordnung der Kopieroperationen minimiert werden können.

Die Ports der Registerbank, die für den Datenaustausch zwischen den Clustern verwendet werden, sind dieselben, über die auch Speicherzugriffe durchgeführt werden. Die Operatoren für Lade-, Speicher- bzw. Kopieroperationen werden **externe Operatoren** genannt. Jeder externe Operator bildet eine eigene externe funktionale Einheit. Da die Kopieroperationen in den VLIW-Instruktionen mit kodiert werden, ist es die Aufgabe des Compilers, diese in die VLIW-Instruktionen einzufügen. Wie die externen Ports einer Registerbank durch Speicher- und Kopieroperationen gemeinsam genutzt werden, ist in der Abbildung 2.4 dargestellt. Zugrunde liegt die Idee einer Architektur, die TTA (*Transport Triggered Architechture*) genannt wird [40, 47]. Ein Port der Registerbank (*erp*, *ewp*) ist nicht einem Kopieroperator oder einem Speicheroperator fest zugeordnet, sondern stellt nur den Wert eines Registers bereit.

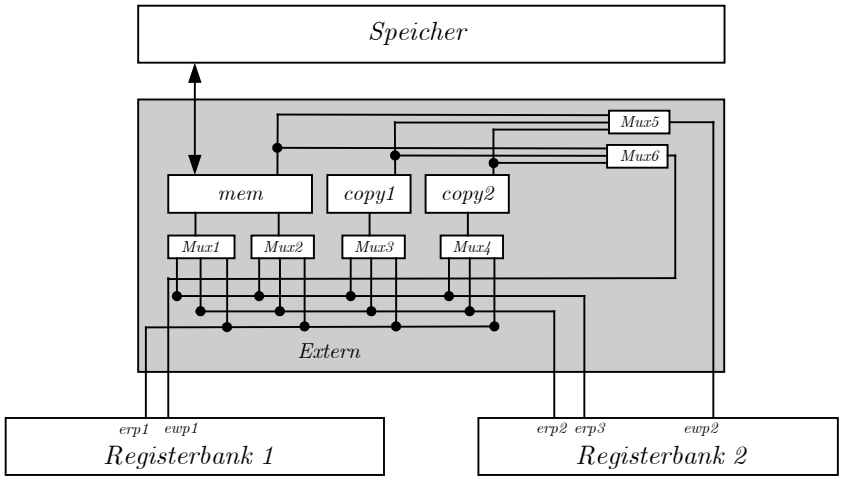



Abbildung 2.4: Externes Verbindungsnetzwerk für die Kommunikation mit externen Speichermedien und anderen Clustern.

Jeder externe Leseport ist mit allen externen funktionalen Einheiten verbunden, so dass der bereitgestellte Wert von einer oder mehreren externen FUs gleichzeitig verwendet werden kann. Durch einen Multiplexer (*Mux*) am Eingang jeder externen FU kann der Wert eines Leseports ausgewählt werden. Analog dazu kann durch einen Multiplexer für jeden externen Schreibport der Registerbank ein Wert eines Kopier- oder Speicheroperators ausgewählt werden, der in ein Register geschrieben werden soll. Für jeden externen Port muss die Nummer der anzusprechenden Quell- bzw. Zielregister in der Registerbank kodiert werden. In Abbildung 2.5 ist beispielhaft der Teil des Instruktionswortes dargestellt, der der Steuerung der externen Ports und funktionalen Einheiten dient. Für jede externe FU existieren Operationscodes, die die auszuführende Operation kodieren (*mem*, *copy1*, *copy2*). Zu jedem Operationscode werden die Ports angegeben, von denen die Operanden gelesen werden (*Mux1*, ..., *Mux4*) und die externen Operatoren, deren Ergebnisse geschrieben werden sollen (*Mux5*, *Mux6*).

Außerdem muss mit jedem externen Leseport (*erp1*, ..., *erp3*) und jedem externen Schreibport (*ewp1*, *ewp2*) ein Register in der Registerbank assoziiert werden.

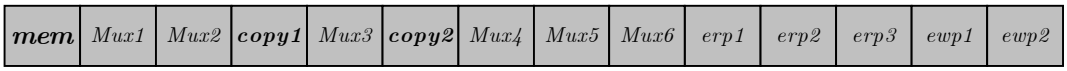



Abbildung 2.5: Teil des Instruktionswortes zur Steuerung des externen Verbindungsnetzwerkes.

Für funktionale Einheiten, die nicht gleichzeitig auf dieselbe Registerbank zugreifen oder die eine unterschiedliche Anzahl von Lese- und Schreibports benötigen, ist ein solches auf TTAs basierendes Verbindungsnetzwerk von Vorteil. In diesen Fällen kann derselbe Port in der Registerbank von verschiedenen externen funktionalen Einheiten verwendet werden und nicht jede externe FU benötigt ihre eigenen Ports. Dadurch kann die Anzahl der externen Ports in den Registerbänken verringert werden. Für die Verbindung der internen funktionalen Einheiten mit der Registerbank wird diese Technik nicht genutzt, da gewährleistet sein soll, dass alle FUs gleichzeitig auf die Registerbank zugreifen können und somit die vorhandene Parallelität auch von Anwendungen genutzt werden kann, für die der Prozessor nicht optimiert ist.

### 2.1.2 Pipelinemodell und Bypass

Als Pipelinemodell für den VLIW-Prozessor wird eine einfache 4-stufige Pipeline genutzt, deren Stufen in Abbildung 2.6 dargestellt sind.

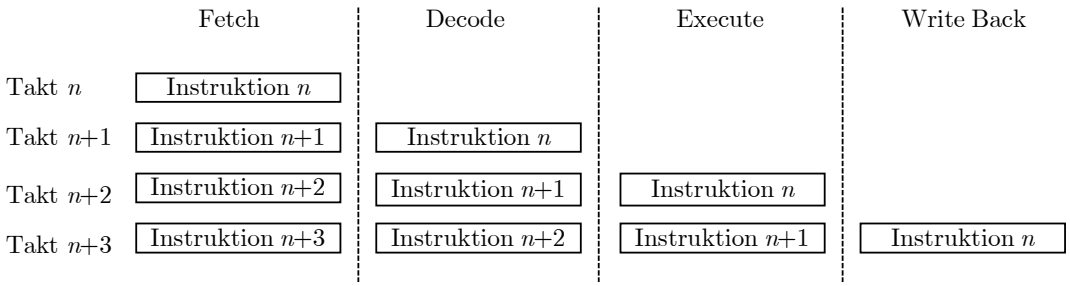



Abbildung 2.6: Schematische Darstellung des verwendeten 4-stufigen Pipelinemodells.

In jedem Takt wird die Abarbeitung einer Instruktion in jeder Pipelinestufe abgeschlossen. Eine Instruktion benötigt somit vier Takte um vollständig abgearbeitet zu werden. Aufgrund der vier parallel arbeitenden Pipelinestufen wird im Idealfall pro Takt die Ausführung einer Instruktion beendet. Die vier Stufen der Pipeline sind

- *Fetch*, zum Laden einer Instruktion aus dem Speicher,
- *Decode*, zum Dekodieren der Instruktion,
- *Execute*, zum Ausführen der Operationen durch die FUs und
- *Write-Back*, zum Zurückschreiben der Ergebnisse der Operationen in die Register.

Die funktionalen Einheiten selbst verfügen nicht über Pipelines. Somit kann eine FU mit der Ausführung einer Operation erst beginnen, wenn die vorangegangene Operation vollständig abgearbeitet wurde. Die Ausführung einer Operation durch eine FU kann mehrere Takte in Anspruch nehmen, wobei die Anzahl dieser Takte als **Latenzzeit** bezeichnet wird. Die Latenzzeit einer Operation ist durch ihren Typ festgelegt und durch die Funktion $lat : \mathcal{O} \rightarrow \mathbb{N}$ gegeben. Führt eine funktionale Einheit eine Operation $v$ mit der Latenzzeit $lat(type(v))$ aus, dann muss sichergestellt sein, dass in den folgenden $lat(type(v)) - 1$ Instruktionen diese FU keine weitere Operation ausführen muss. Das sicherzustellen ist ebenfalls eine Aufgabe des Compilers. Dadurch können die folgenden Instruktionen problemlos durch die Pipeline laufen, so dass die übrigen funktionalen Einheiten weiterhin mit Operationen versorgt werden.

Die vorgestellte 4-stufige Pipeline schreibt in der Write-Back-Stufe das Ergebnis einer Operation in die Registerbank zurück. Als Konsequenz kann dieses Ergebnis erst im darauf folgenden Takt als Eingangswert einer FU genutzt werden. Durch diese Restriktion kann sich die Ausführungszeit eines Programms erheblich verlängern. *Forwarding* ist eine wichtige Technik, um dieses Problem in Pipelinearchitekturen zu vermeiden [48]. Dafür werden mit einem *Bypass*, wie er in Abbildung 2.7 dargestellt ist, Werte direkt vom Ausgang einer FU an den Eingang einer anderen oder derselben FU weitergeleitet, ohne dass sie zuvor in die Registerbank geschrieben werden müssen. Innerhalb eines Clusters ist immer ein vollständiger Bypass vorhanden. Zwischen FUs verschiedener Cluster gibt es keinen Bypass.

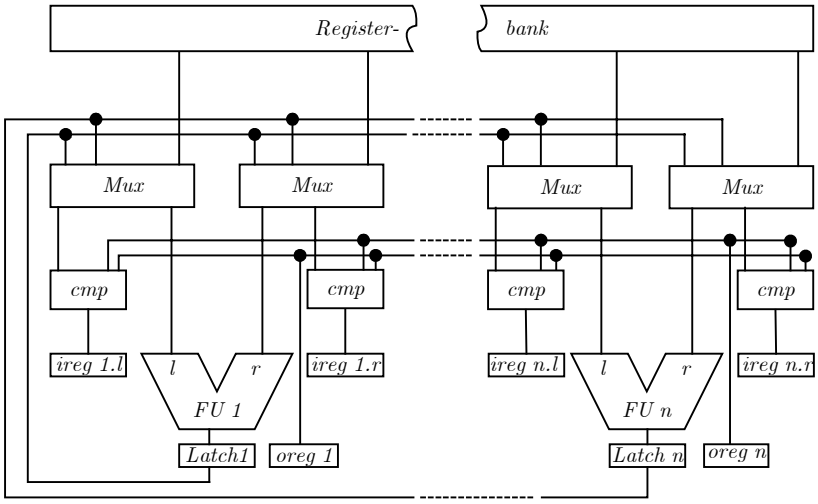



Abbildung 2.7: Schematische Darstellung des Bypasses in einem Cluster.

Im Allgemeinen übernimmt die Hardware des Prozessors die Steuerung des Bypasses. Dafür sind Informationen über die Nummern der Register notwendig, in die die Ergebnisse in der Write-Back-Phase geschrieben und deren Werte gleichzeitig als Operanden in der Execute-Phase benötigt werden. Wird ein Register als Operand benötigt, das gerade erst beschrieben wird, dann muss dieser Wert direkt vom Ausgang zum Eingang der entsprechenden funktionalen Einheiten weitergeleitet werden. Verantwortlich für die Wahl der richtigen Quelle ist ein Multiplexer (*Mux*) an jedem Eingang einer FU. Um den Multiplexer korrekt zu steuern, wird Hardware (*cmp*) zum Vergleichen der Registernummern an den linken (*l*) und rechten (*r*) Eingängen (*ireg*) und Ausgängen (*oreg*) der FUs benötigt. Die Hardware für die Vergleiche kann eingespart werden, wenn der Compiler die Steuerung des Bypasses übernimmt [41, 58]. Voraussetzung dafür ist ein statischer Ablaufplan und die genaue Kenntnis der Pipelinestufen sowie des Bypasses. Der Compiler kann dann für jede Operation festlegen, ob der Quelloperand aus der Registerbank oder dem Latchregister (*Latch*) am Ausgang einer FU stammt. Die Multiplexer und deren Verdrahtung können dadurch nicht gespart werden.

Eine Pipeline hat auch Auswirkungen auf die Genauigkeit der statischen Laufzeitabschätzung. Bei Sprüngen sind die auf den Sprungbefehl folgenden Instruktionen bereits in die Pipeline geladen worden. Wird der Sprung tatsächlich ausgeführt, müssten diese Instruktionen wieder aus der Pipeline entfernt werden, damit sie nicht ausgeführt werden. Um das zu vermeiden, verarbeitet der Prozessor noch alle Instruktionen, die sich in der Pipeline befinden, und lädt erst nach dem tatsächlichen Ausführen einer Sprungoperation die Instruktionen vom Sprungziel in die Pipeline [27]. Es ist deshalb die Aufgabe des Compilers die Instruktionen unmittelbar nach einem Sprungbefehl mit geeigneten Operationen zu füllen, da diese in jedem Fall noch abgearbeitet werden.

### 2.1.3 Modell für den Strom- und Platzverbrauch

Wesentliche Optimierungsziele im Bereich der eingebetteten Systeme sind der Strom- und Platzverbrauch. In diesem Abschnitt wird der Zusammenhang zwischen Ausführungsgeschwindigkeit, Platz- und Stromverbrauch eines VLIW-Prozessors beschrieben, woraus sich die während der DSE optimierten Parameter ergeben.

Der Platzverbrauch in einem VLIW-Prozessor wird durch die Registerbänke und den Bypass dominiert und wächst quadratisch mit der Portanzahl in der Registerbank und linear mit der Registeranzahl. Unter der Annahme, dass die Registerbank mindestens so viele Register wie Ports besitzt, ergibt sich ein kubisches Wachstum [87]. Mit steigender Portanzahl in den Registerbänken nimmt im Allgemeinen auch die Anzahl der Operatoren im Prozessor zu, so dass deren Platzbedarf ebenfalls mit wachsender Portanzahl steigt.

Der Stromverbrauch in einer VLIW-Architektur wird im Wesentlichen durch

- den statischen Stromverbrauch,
- den dynamischen Stromverbrauch,
- den Stromverbrauch der Registerbänke und
- die Taktfrequenz

beeinflusst. Auf den statischen Stromverbrauch hat insbesondere bei kleinen Strukturgrößen auch der Platzverbrauch direkten Einfluss. Bei einer 70 Nanometer Technologie entfallen bis zu 50% des Stromverbrauchs in der Schaltung auf statische Leckströme [18], so dass dieser Anteil durch eine Verringerung der Gatter, also des Platzverbrauchs, gesenkt werden kann, wie auch aus der folgenden Formel für den statischen Stromverbrauch hervorgeht [7]:

$$P_{leak} = L_g \cdot V_{dd} \cdot K_3 \cdot e^{K_4 \cdot V_{dd}} \cdot e^{K_5 \cdot V_{bs}} + \left| V_{bs} \cdot I_{Ju} \right|. \tag{1.1}$$

In dieser Formel bezeichnet $L_g$ die Anzahl der Gatter in der Schaltung und $V_{dd}$ die Versorgungsspannung. Diese beiden Größen können durch die DSE beeinflusst werden. $V_{bs}$ (*body bias voltage*), $I_{Ju}$ (*body injunktion leakage*) und $K_i$ (technologieabhängige Konstanten) sind fest vorgegeben. Der dynamische Stromverbrauch $P_{dyn}$ ist im Wesentlichen von vier Größen abhängig [7]:

$$P_{dyn} = s \cdot C_{eff} \cdot f \cdot V_{dd}^2 . \tag{1.2}$$

Hier ist $s$ die Schaltungsaktivität, $C_{eff}$ die umgeladene Kapazität, $f$ die Taktfrequenz des Prozessors und $V_{dd}$ wieder die Versorgungsspannung. Vereinfachend wird angenommen, dass $s$ und $C_{eff}$ für das gleiche Programm konstant sind. Schließlich werden immer dieselben Operationen ausgeführt,

bei verschiedenen VLIW-Architekturen im Allgemeinen in einer unterschiedlichen Reihenfolge. Unter der Annahme, dass der größte Teil der dynamischen Leistungsaufnahme in den funktionalen Einheiten, den Leitungen und der Registerbank auftritt, können die Änderungen in der Schaltungsaktivität, die durch eine andere Abarbeitungsreihenfolge der Operationen entstehen, vernachlässigt werden. Damit ist der dynamische Stromverbrauch von der Taktfrequenz und der Versorgungsspannung abhängig.

Einen sehr hohen Anteil am Stromverbrauch bei VLIW-Architekturen hat die Registerbank des Prozessors. Mit steigender Anzahl der Lese- und Schreibports nimmt der Stromverbrauch und die Zugriffszeit zu [87]. Der Stromverbrauch steigt kubisch mit der Anzahl der Ports in der Registerbank unter der Annahme, dass die Registerbank mindestens so viele Register wie Ports besitzt. Die Abhängigkeit der Zugriffszeit auf die Register von der Anzahl der Ports wird durch die Funktion

$$RFDelay(ports) = ports^{\frac{3}{2}} \tag{1.3}$$

beschrieben [87]. Die Verzögerungen gehen hauptsächlich auf die Signallaufzeiten in den Leitungen zurück. Diese Laufzeiten treten auch im Bypass des Prozessors auf [98], dessen Größe ebenfalls von der Portanzahl in der Registerbank abhängt. In [96] ist die Abhängigkeit der Verzögerung im Bypass von der Portanzahl als quadratisch angegeben. $WDelay(p)$ soll für die Portanzahl $p$ die größte Verzögerung in der Registerbank bzw. dem Bypass angeben. Die Verzögerungen in den funktionalen Einheiten ($FUDelay$) entstehen hauptsächlich durch die Verzögerungen in den Gattern. Die größte Verzögerung $maxDelay$ im Prozessor wird somit entweder durch die größte Verzögerung in den funktionalen Einheiten oder die Laufzeiten auf den Leitungen der Registerbank bzw. des Bypasses bestimmt:

$$maxDelay(p) = \max(WDelay(p), FUDelay)\,.$$

Die maximal mögliche Taktfrequenz eines Prozessors ist umgekehrt proportional zu $maxDelay(p)$ [97]. Damit lässt sie sich in Abhängigkeit von der Portanzahl einer Registerbank durch die Formel

$$fm(p) = \frac{1}{maxDelay(p)}\,. \tag{1.4}$$

beschreiben. Dabei berücksichtigt $fm$ die Verzögerungen in den Gattern der FUs, in der Registerbank und dem Bypass. Ab wann die Verzögerung der Registerbank die Verzögerung in den FUs dominiert, hängt wesentlich von der Architektur der funktionalen Einheiten, der Registerbank und der

Fertigungstechnologie ab[1]. Je niedriger die Verzögerung in den Gattern der FUs ist, desto eher dominieren die Signallaufzeiten in der Registerbank. Somit muss eine Erhöhung der Parallelität in der Architektur nicht unbedingt zu einer höheren Ausführungsgeschwindigkeit einer Anwendung führen, weil durch die erhöhte Portanzahl in der Registerbank die maximal mögliche Taktfrequenz des Prozessors sinken kann.

Der Zusammenhang zwischen der Versorgungsspannung $V_{dd}$ und der Verzögerung in einem Gatter ist durch die folgende Formel aus [97] gekennzeichnet:

$$FUDelay \cong \frac{C_L \cdot V_{dd}}{k_V \cdot W \cdot (V_{dd} - V_T - V_{DSAT})}. \tag{1.5}$$

$C_L$ bezeichnet die Kapazität. $k_v$ und $W$ sind Konstanten des Transistors. $V_T$ (*threshold voltage*) bewegt sich im Bereich von 0,5V bis 1V. $V_{DSAT}$ (*velocity saturation voltage*) ist für $V_{dd} \geq 2$V konstant und hat in etwa den Wert von $V_T$. Für $V_{dd} < 2$V nähert sich $V_{DSAT}$ asymptotisch dem Wert $(V_{dd} - V_T)$ an. Für diese in [97] beschriebenen Zusammenhänge zwischen den Größen ist in Abbildung 2.8 die Abhängigkeit der Verzögerung in den Gattern von der Versorgungsspannung grafisch dargestellt.

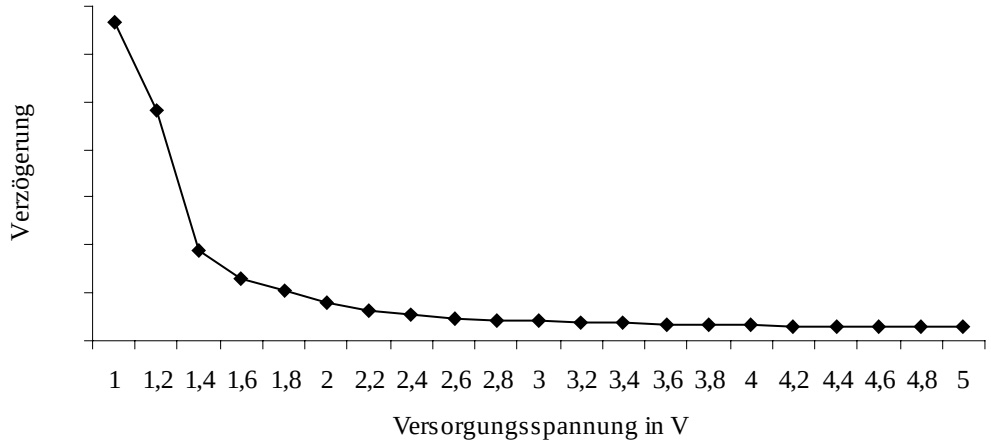



Abbildung 2.8: Abhängigkeit der Verzögerung von der Versorgungsspannung.

Mit sinkender Versorgungsspannung nimmt die Verzögerung stark zu. Dieser Zusammenhang zwischen *FUDelay* und $V_{dd}$ wird genutzt, um den Stromverbrauch des Prozessors zu verringern [97]. Durch eine erhöhte Parallelität kann die Taktfrequenz abgesenkt werden, weil der Prozessor

[1] Die Zusammenhänge zwischen diesen Größen sind vor der Design-Space-Exploration bekannt, weil sie für die Komponenten, aus denen der Datenpfad zusammengesetzt wird, ermittelt wurden [68].

trotzdem die gleiche Menge von Operationen innerhalb eines festen Zeitraums verarbeiten kann. Aufgrund der verringerten Taktfrequenz kann auch die Versorgungsspannung heruntergesetzt werden, weil jetzt eine höhere Verzögerung in den FUs akzeptiert werden kann. Insbesondere das Absenken der Versorgungsspannung geht quadratisch in die Verringerung des dynamischen Stromverbrauchs $P_{dyn}$ ein. Durch die erhöhte Parallelität wächst aber der Platz- und Stromverbrauch in den Registerbänken kubisch. Außerdem steigt der Platzverbrauch durch zusätzliche Operatoren und damit auch $P_{leak}$. Es ist deshalb nur sinnvoll, die Parallelität solange zu erhöhen, bis die Verringerung des dynamischen Stromverbrauchs geringer ist als die Erhöhung des statischen Stromverbrauchs und des Stromverbrauchs in den Registerbänken. Außerdem darf die Parallelität nur solange erhöht werden, wie trotz der dadurch sinkenden maximal möglichen Taktfrequenz des Prozessors die geforderten Ausführungszeiten eingehalten werden.

Der Platzverbrauch, Stromverbrauch und die Taktfrequenz eines VLIW-Prozessors hängen somit wesentlich von der Portanzahl in seinen Registerbänken ab, genauso wie die verfügbare Parallelität, da die Anzahl der parallel arbeitenden FUs in direktem Zusammenhang mit der Portanzahl in den Registerbänken steht. Die in dieser Arbeit gewählte Optimierungsstrategie konzentriert sich deshalb auf die Minimierung der Ports in den Registerbänken und besonders auf die Minimierung der maximalen Portanzahl der Registerbänke.

### 2.1.4 Formales Prozessormodell

Aufgrund der Unterscheidung von internen und externen Ports in der Registerbank werden auch die Operatoren unterschieden, die über die internen bzw. externen Ports mit der Registerbank verbunden sind. Interne Operatoren haben nur auf die internen Ports der Registerbank Zugriff und externe Operatoren nur auf die externen Ports. Entsprechend werden die funktionalen Einheiten, die nur interne Operatoren besitzen dürfen, interne funktionale Einheiten genannt. Die Typen der mit den internen Operatoren ausführbaren Operationen werden mit $\mathcal{O}_I$ bezeichnet. $\mathcal{O}_E$ bezeichnet die Typen der durch die externen Operatoren ausführbaren Operationen. Es gilt $\mathcal{O}_I \cap \mathcal{O}_E = \varnothing$ und $\mathcal{O} = \mathcal{O}_I \cup \mathcal{O}_E$, wobei $\mathcal{O}_E = \{copy, load, store\}$ und $\{nop\} \subseteq \mathcal{O}_I$. *copy* ist der Typ des Operators, der einen Registerinhalt aus einem Cluster in einen anderen Cluster kopiert. *load* und *store* bezeichnen die Typen der Operatoren zum Laden und Speichern von Werten aus dem bzw. in den Arbeitsspeicher und *nop* ist der Typ einer Operation ohne Wirkung. Die Latenzzeiten der externen Operatoren modellieren die Speicherzugriffszeit und die Zeit zum Kopieren von Werten zwischen verschiedenen Clustern. Ein geclusterter VLIW-Prozessor wird durch das Tupel

$$(\mathcal{C}, \mathcal{F}, cf, ft, rz, rpc)$$

beschrieben. Es ist

- $\mathcal{C} \neq \emptyset$ die Menge der Cluster, auch **Prozessorcluster** genannt, und
- $\mathcal{F} \neq \emptyset$ die Menge der funktionalen Einheiten im Prozessor.
- $cf : \mathcal{C} \to \wp(\mathcal{F})$ modelliert den Datenpfad, indem den internen Clustern eine Menge interner funktionaler Einheiten zugeordnet wird und genau einem Cluster die Menge der externen funktionalen Einheiten. Verschiedene Cluster $c$ und $c'$ bestehen aus disjunkten FU-Mengen:

$$cf(c) \cap cf(c') \neq \emptyset \Rightarrow c = c'.$$

- $ft : \mathcal{F} \to \wp(\mathcal{O})$ ordnet jeder internen FU eine Menge von internen Operationstypen zu, wodurch die in jeder FU implementierten Operatoren festgelegt sind und jeder externen FU eine Einermenge von Operationstypen. Diese Zuordnung wird **Ressourcenallokation** genannt und legt den Typ $ft(f)$ einer FU $f$ fest. Eine FU enthält für jeden Operationstypen höchstens einen Operator.
- $rz : \mathcal{C} \to \mathbb{N}$ beschreibt die Anzahl der Register in den Registerbänken der einzelnen Cluster und
- $rpc : \mathcal{C} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ modelliert die **Portkonfiguration** der Registerbank eines Clusters. Dadurch ist die Anzahl der internen Ports ($ip$) sowie der externen Lese- ($rp$) und Schreibports ($wp$) durch $rpc(c) = (ip, rp, wp)$ in einem Cluster $c$ festgelegt.

Die Anzahl der internen Ports entspricht dem Dreifachen der FU-Anzahl im Cluster. Auf die Modellierung der Branch-Einheit wird im Weiteren verzichtet, da diese in jedem Cluster zwei Ports benötigt. Ein Drittel der internen Ports sind Schreibports und zwei Drittel Leseports. Der Cluster $e$, der ausschließlich externe Operatoren enthält, besitzt keine Registerbank. Somit sind $rpc(e) = (0, 0, 0)$ und $rz(e) = 0$. Ein Prozessor $(\mathcal{C}, \mathcal{F}, cf, ft, rz, rpc)$ kann mehrere identische Cluster und jeder Cluster mehrere identische funktionale Einheiten besitzen. Die Menge aller so beschreibbaren Prozessoren zu einer festen Operationstypenmenge $\mathcal{O}$ bildet den Suchraum, in dem durch die Design-Space-Exploration ein geeigneter Prozessor für eine gegebene Anwendung gesucht wird.

Die Hardwarekosten eines VLIW-Prozessors $P = (\mathcal{C}, \mathcal{F}, cf, ft, rz, rpc)$ setzen sich aus zwei Komponenten zusammen:

$$VLIWCost(P) = DPCost(P) + RFCost(P)\,. \tag{1.6}$$

Die **Datenpfadkosten** $DPCost$ ergeben sich aus den Datenpfadkosten der Cluster in der Architektur und umfassen nur die Kosten, die durch die Operatoren in den FUs entstehen:

$$DPCost((\mathcal{C}, \mathcal{F}, cf, ft, rz, rpc)) = \sum_{c \in \mathcal{C}} ClusterCost(c, cf, ft)\,. \tag{1.7}$$

*ClusterCost* summiert die Kosten der funktionalen Einheiten in einem Cluster $c$ auf:

$$ClusterCost(c, cf, ft) = \sum_{f \in cf(c)} FUCost(f, ft). \tag{1.8}$$

Durch den Ausschluss von Synergieeffekten werden die Kosten einer funktionalen Einheit $f$ durch die Summe der Kosten der in $f$ implementierten Operatoren berechnet:

$$FUCost(f, ft) = \sum_{o \in ft(f)} opCost(o). \tag{1.9}$$

Diese Formel beschreibt annähernd die Hardwarekosten eines Prozessors, der aus Hardwaremodulen zusammengesetzt wird [50]. Wie in 2.1.3 bereits erläutert, sind die Kosten einer Registerbank quadratisch von der Anzahl der Ports ($ip + rp + wp$) und linear von der Anzahl der Register *regs* [87] abhängig. Damit können die Hardwarekosten einer Registerbank mit *regs* vielen Registern und der Portkonfiguration ($ip$, $rp$, $wp$) durch

$$RFCCost(regs, (ip, rp, wp)) = \rho \cdot (ip + rp + wp)^2 \cdot regs$$

berechnet werden, wobei $\rho$ die Kosten einer Registerbank mit einem Register und einem Port sind. Da eine Registerbank in der Praxis mindestens so viele Register wie Ports bereitstellt, wächst *RFCCost* mindestens kubisch. Die Gesamtkosten aller Registerbänke sind dann

$$RFCost((\mathcal{C}, \mathcal{F}, cf, ft, rz, rpc)) = \sum_{c \in \mathcal{C}} RFCCost(rz(c), rpc(c)). \tag{1.10}$$

Die Hardwarekosten für das Verbindungsnetzwerk zwischen den Clustern sowie den Clustern und dem Arbeitsspeicher hängen von der Anzahl der externen Operatoren ab. Die dadurch anfallenden Kosten für das Verbindungsnetzwerk werden in den Kosten der entsprechenden Operationstypen mit berücksichtigt und gehen somit durch die Kostenfunktion (1.9) in den Wert von *VLIWCost* ein.

## 2.2 Programmmodell

Nachdem im vorangegangenen Abschnitt das Prozessormodell vorgestellt wurde, dessen Parameter durch die Design-Space-Exploration angepasst werden, wird in diesem Abschnitt das Modell eingeführt, mit dem die Anwendung, an die der Prozessor angepasst werden soll, beschrieben wird. Ziel ist es, bereits durch die Formalisierung den Grundgedanken des DESCOMP-Ansatzes zum Ausdruck zu bringen, nämlich dass aus der Anwendung ein dafür speziell geeigneter Prozessor abgeleitet wird. Deshalb sollen die im vorangegangenen Abschnitt angegebenen Parameter des Prozessors auch mittels des Programmmodells beschreibbar sein.

Die Anwendung besteht aus einer Menge von Basisblöcken [8]. Weil die Leistungsfähigkeit eines VLIW-Prozessors überwiegend davon abhängt, wie gut er sich zur parallelen Abarbeitung der Operationen in den einzelnen Basisblöcken eignet, wird er für die Abarbeitung dieser Basisblöcke optimiert. Die Abarbeitungsreihenfolge der Basisblöcke wird während der DSE mit DESCOMP vernachlässigt. Es werden nur Informationen über die Ausführungshäufigkeiten der einzelnen Basisblöcke benötigt, die in einer vorangegangenen Simulationsphase ermittelt wurden. Obwohl nur Basisblöcke modelliert werden, können Basisblocktransformationen zur Erhöhung der Parallelität, wie z. B. das Entrollen von Schleifen [8] oder das Bilden von Basisblockketten, sogenannter *Traces*, wie beim *Trace-Scheduling* [51], vorher durchgeführt werden. Der Prozessor kann dann für die Basisblöcke optimiert werden, die aus den entsprechenden Transformationen entstanden sind. Ein Basisblock ist eine Menge von Operationen aus dem Quellprogramm, die beim Betreten des Basisblocks alle abgearbeitet werden. Die Reihenfolge, in der die Operationen eines Basisblocks abgearbeitet werden, ist im Allgemeinen nur partiell festgelegt. Zur Darstellung wird deshalb ein gerichteter azyklischer Graph genutzt, der die partielle Ordnung über den Operationen modelliert.

**Definition 2.1 (Basisblock)**
Ein Basisblock $b = (V, E, type)$ ist ein gerichteter azyklischer Graph mit der Knotenmenge $V$, der Kantenmenge $E \subseteq V \times V$ und der Beschriftungsfunktion $type : V \rightarrow \mathcal{O}$.

□

Ein Knoten $v \in V$ wird auch **Operationsknoten** (oder kurz Operation) genannt. Operationen können direkt von einer FU des Prozessors ausgeführt werden. In der gesamten Arbeit werden $V$ und $E$ für die Knoten- bzw. Kantenmenge eines Basisblocks verwendet. Um zu verdeutlichen, dass $V$ oder $E$ zu einem Basisblock $b$ gehören, werden beide Mengen bei Bedarf mit $b$ indiziert. Der Operationstyp eines Operationsknotens $v$ ist durch $type(v)$ bestimmt. Die Kantenmenge $E$ ist die Vereinigung zweier disjunkter Kantenmengen $D$ und $A$. Durch $D$ werden **Datenabhängigkeiten** dargestellt. Eine Kante $(u, v) \in D$ gibt es genau dann, wenn die Operation $v$ das Ergebnis der Operation $u$ benötigt. Dieser Wert muss in einem Register gehalten werden. Es kann im Programm aber noch Reihenfolgebedingungen zwischen Speicher- und Leseoperationen geben, die nicht durch eine Datenabhängigkeit verursacht werden. So muss z. B. eine Leseoperation nach einer Speicheroperation ausgeführt werden, wenn beide Operationen auf denselben Speicherplatz zugreifen und die Leseoperation den von der Speicheroperation geschriebenen Wert lesen soll. Analog dazu muss die Reihenfolge zweier Speicheroperationen beachtet werden, wenn diese denselben Speicherplatz beschreiben, damit nach der letzten Schreiboperation der richtige Wert im Speicher steht. Diese Abhängigkeiten in der Reihenfolge werden durch Kanten in der Menge $A$ modelliert. Ein Knoten $v$ ist **abhängig** vom Knoten $w$, falls $(w, v) \in E^+$, wobei $E^+$ der transitive Abschluss der Relation $E$ ist. $v$

wird auch **Nachfahre** von $w$ und $w$ **Vorfahre** von $v$ genannt. Für $(w, v) \in E$ wird $v$ **Nachfolger** von $w$ und $w$ **Vorgänger** von $v$ genannt.

**Definition 2.2 (Pfad)**
Ein Pfad $\pi = v_1 \ldots v_n$ im Basisblock $b = (V, E, type)$ mit $n \geq 1$ ist eine Folge von Knoten, so dass $\forall i \in \mathbb{N} : 1 \leq i < n \Rightarrow (v_i, v_{i+1}) \in E$.

□

**Definition 2.3 (Pfadlänge)**
Die Länge eine Pfades $\pi = v_1 \ldots v_n$ im Basisblock ist die Summe der Latenzzeiten seiner Operationen und ist definiert als

$$pl(\pi) = \sum_{i=1}^{n} lat(type(v_i)) \,.$$

□

**Definition 2.4 (kritischer Pfad, kritische Pfadlänge)**
Ein Pfad $\pi$ ist ein kritischer Pfad im Basisblock $b$, falls für alle Pfade $\pi'$ in $b$ gilt: $pl(\pi') \leq pl(\pi)$. Die Länge eines kritischen Pfades $\pi$ wird kritische Pfadlänge von $b$ genannt, kurz $cp(b)$.

□

**Definition 2.5 (Instruktion)**
Eine Menge von Operationen $i \subseteq V$ wird Instruktion genannt.

□

**Definition 2.6 (Ablaufplan)**
Ein Ablaufplan der Länge $l$ zu einem Basisblock $b$ ist eine Folge von Instruktionen. Diese Folge wird durch eine Funktion $\alpha : \{0,\ldots,l-1\} \rightarrow \wp(V_b)$ beschrieben, die den folgenden Bedingungen genügen muss:

- $V_b = \bigcup_{0 \leq i < l} \alpha(i) \,,$
- $\forall i \in \{0,\ldots,l-1\} \Rightarrow \alpha(i) \neq \emptyset$.
- Jede Operation $v$ kommt in genau $lat(type(v))$ vielen Instruktionen vor:
$$\forall v \in V_b : \left| \{\ i \mid i \in \{0,\ldots,l-1\} \wedge v \in \alpha(i)\} \right| = lat(type(v)) \,.$$
- Diese Instruktionen müssen aufeinander folgend sein:
$$\forall v \in V_b \max\{i \mid v \in \alpha(i)\} - \min\{i \mid v \in \alpha(i)\} = lat(type(v)) - 1 \,.$$
- Die Abhängigkeiten im Basisblock $b$ werden durch den Ablaufplan berücksichtigt, d. h.,
$$\forall u, v \in V : (u, v) \in E \Rightarrow \max\{j \mid u \in \alpha(j)\} < \min\{i \mid v \in \alpha(i)\} \,.$$

□

Für die Instruktion $\alpha(i)$ wird auch kurz $i$ geschrieben. Die Länge des Ablaufplans $\alpha$ wird auch mit $|\alpha|$ bezeichnet. Gemäß dem Prozessormodell wird in jeder Pipelinestufe in jedem Takt eine Instruktion abgearbeitet. Ein Ablaufplan der Länge $l$ kann mit dem vorgestellten Prozessormodell somit in genau $l + 3$ Takten abgearbeitet werden, wenn sich die erste Instruktion des Ablaufplans noch nicht in der ersten Pipelinestufe befindet. Befinden sich bei einer zyklischen Abarbeitung des Basisblocks die ersten drei Instruktionen bereits in der Pipeline, so werden genau $l$ Takte zur Abarbeitung von $\alpha$ benötigt. Von diesem Fall wird bei den weiteren Betrachtungen ausgegangen, da das vorgestellte Pipelinemodell dem Compiler erlaubt, die der Sprungoperation folgenden Instruktionen zur Planung von Operationen zu nutzen. Eine Instruktion entspricht somit genau einem Takt. Auf die Modellierung einer Sprungoperation, die im Allgemeinen einen Basisblock abschließt, wurde verzichtet, weil sie keinen wesentlichen Einfluss auf die DSE hat. Es muss lediglich sichergestellt sein, dass die Sprungoperation immer in der Instruktion $l - 3$ beginnt. Dann entspricht die Länge des Ablaufplans der benötigten Taktanzahl für seine Abarbeitung, auch wenn durch die Sprungoperation eine Verzweigung an eine andere Programmposition erfolgt. Der in Abschnitt 4.1 vorgestellte Planungsalgorithmus unterstützt die Erzeugung eines Ablaufplans, der die Sprungoperation an einer festen Position enthält.

Die Länge eines längsten Ablaufplans $\alpha$ zu einem Basisblock $b$ wird mit $lp(b)$ bezeichnet und ist definiert als $pl(v_1 \ldots v_m)$, wobei $V_b = \{v_1, \ldots, v_m\}$. Einem Basisblock $b$ können Ablaufpläne mit den Längen $cp(b)$ bis $lp(b)$ zugeordnet werden. Kürzere Ablaufpläne existieren für $b$ nicht, da dann die Datenabhängigkeiten im Basisblock nicht beachtet werden können. Längere Ablaufpläne existieren ebenfalls nicht, da es dann eine leere Instruktion $i$ gäbe. Die Latenzzeit einer Operation wird im Ablaufplan dadurch modelliert, dass die betreffende Operation in $lat(type(v))$ vielen aufeinander folgenden Instruktionen enthalten ist. Das bedeutet aber nicht, dass in jeder dieser Instruktionen mit der Ausführung der Operation begonnen wird. Vielmehr wird dadurch modelliert, dass in jeder der Instruktionen eine FU mit der Ausführung dieser Operation beschäftigt ist. Gleichzeitig ist dadurch sichergestellt, dass diese FU keine weitere Operation ausführen kann.

Um die Cluster eines VLIW-Prozessors zu modellieren, wird die Knotenmenge $V$ eines Basisblocks durch eine Äquivalenzrelation $\chi \subseteq V \times V$ in Äquivalenzklassen zerlegt. Zwei Knoten $u$ und $v$ stehen genau dann in Relation, wenn sie von der Zielarchitektur im selben Cluster ausgeführt werden sollen. Die Zerlegung $V/\chi$ wird **Clusterung** genannt. Jede Äquivalenzklasse $[v]_\chi \in V/\chi$ entspricht einem Cluster im Prozessor und wird deshalb ebenfalls als Cluster oder auch als **Basisblockcluster** bezeichnet. Um Daten zwischen zwei Clustern auszutauschen, ist eine explizite Kopieroperation im Basisblock erforderlich. Kopieroperationen werden im Basisblock durch entsprechende Knoten $v$ mit $type(v) = copy$ modelliert. Alle Kopier-, Lese- und Speicheroperationen des Basisblocks werden in einer

Klasse zusammengefasst. Um zu einem Basisblock $b$ den um die Kopieroperationen erweiterten Basisblock $b_\chi$ zu konstruieren, muss beachtet werden, dass derselbe Wert nicht mehrmals in denselben Cluster kopiert werden soll, wenn er dort von mehreren Operationen benutzt wird. Es wird deshalb die Relation $\sim_u \subseteq V \times V$ definiert, die zwei Knoten $v$ und $w$ aus demselben Cluster genau dann in Relation setzt, wenn sie einen gemeinsamen Vorgänger $u$ in einem anderen Cluster haben:

$$w \sim_u v :\Leftrightarrow [w]_\chi = [v]_\chi \wedge \exists u \in V : [u]_\chi \neq [w]_\chi \wedge (u,w) \in D \wedge (u,v) \in D\,.$$

Die Menge

$$[u,v]_\sim := \left\{ w \middle|\ v \sim_u w \right\}$$

enthält alle Knoten, die paarweise durch $\sim_u$ in Relation gesetzt werden und modelliert somit genau eine Kopieroperation. Der um die Kopieroperationen erweiterte Basisblock $b_\chi$ wird geclusterter Basisblock genannt und geht aus $b$ hervor, indem zwischen zwei Knoten $(u,\, v) \in D$, die sich in unterschiedlichen Clustern befinden, der Kopierknoten $[u,\, v]_\sim$ eingefügt wird.

**Definition 2.7 (geclusterter Basisblock)**
Ein geclusterter Basisblock $b_\chi = (V',\, E',\, type')$ entsteht aus dem Basisblock $b = (V,\, E,\, type)$ mit $E = D \cup A$ und der Clusterung $\chi$, indem

- $V$ um die Menge der benötigten Kopieroperationen erweitert wird:

$$V' = V \cup \left\{ [u,v]_\sim \middle|\ (u,v) \in D \text{ und } (u,v) \notin \chi \right\} \text{ und}$$

$$type'(v) = \begin{cases} type(v), \text{ falls } v \in V \\ copy, \text{ sonst} \end{cases}$$

- und die Kante $(u,\, v) \in E$ entfernt wird, wenn $[u]_\chi \neq [v]_\chi$ und die Datenabhängigkeit zwischen $u$ und $v$ über den eingefügten Kopierknoten $[u,\, v]_\sim$ hergestellt wird:

$$\begin{aligned} E' = E - \left\{ (u,v) \middle|\ u,v \in V \text{ und } [u,v]_\sim \in V' \right\} \cup \\ \left\{ (u,[u,v]_\sim) \middle|\ u,v \in V \text{ und } [u,v]_\sim \in V' \right\} \cup \\ \left\{ ([u,v]_\sim, w) \middle|\ [u,v]_\sim \in V' \wedge w \in [u,v]_\sim \right\}. \end{aligned}$$

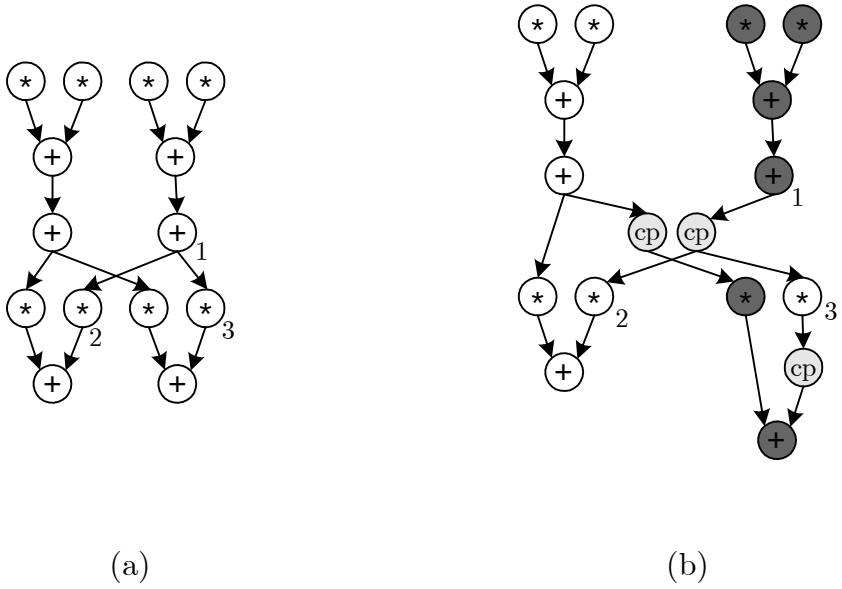
□

Jeder Wert wird so nur einmal kopiert. Alle eingefügten Kopieroperationen werden bei den folgenden Betrachtungen wie normale Operationsknoten behandelt. Die Ausführungszeit einer Kopieroperation wird durch ihre Latenzzeit modelliert. Dadurch ist sichergestellt, dass das Kopieren eines Wertes in einen Cluster abgeschlossen ist, bevor er dort verwendet wird. Der Begriff des Ablaufplans und der Instruktion lässt sich somit ohne Probleme auf geclusterte Basisblöcke anwenden. Allerdings muss die Clusterung $\chi$ für

den Basisblock $b$ auf die Kopieroperationen im geclusterten Basisblock ausgeweitet werden. Alle Kopieroperationen werden zusammen mit den *load*- und *store*-Operationen in dem Cluster für externe Operationen ausgeführt. Es ergibt sich somit die erweiterte Clusterung:

$$\chi \coloneqq \chi \cup \left\{(u,v) \middle|\ u,v \in V' \wedge type(u), type(v) \in \mathcal{O}_E\right\}.$$

Der so modellierte Cluster, der alle externen Operationen des Basisblocks enthält, wird mit $[e]_\chi$ bezeichnet. Wenn in dem geclusterten Basisblock keine Lade-, Speicher- oder Kopieroperationen auftreten, dann ist die Menge $[e]_\chi$ leer und nicht in der Zerlegung $V/\chi$ enthalten. Ein Beispiel ist in der folgenden Abbildung 2.9 angegeben.




(a) (b)

Abbildung 2.9: (a) Nicht geclusterter Basisblock; (b) Geclusterter Basisblock mit eingefügten Kopieroperationen.

In Abbildung 2.9 (a) ist ein nicht geclusterter Basisblock dargestellt. In Abbildung 2.9 (b) ist derselbe in zwei Cluster zerlegte Basisblock dargestellt. Durch $\chi$ sind alle gleichfarbigen Knoten miteinander in Beziehung gesetzt. Kanten, die zwei Knoten aus verschiedenen Clustern verbinden, wurden entfernt und durch Kopieroperationen (*cp*) ersetzt. Die Kopieroperationen bilden selbst einen eigenen Cluster. Der Wert, der durch die Operation 1 erzeugt wird, wird nur einmal kopiert. Es ist deshalb $[1,2]_\sim = [1,3]_\sim = \{2, 3\}$ der entsprechende Kopierknoten. Die kritische Pfadlänge des Basisblocks in (b) ist in dem angegebenen Beispiel um die Latenzzeit von zwei Kopieroperation angewachsen.

Analog zur Clusterung wird jetzt die Zuordnung der Operationen zu funktionalen Einheiten modelliert. Durch die Relation $\phi \subseteq V \times V$ werden zwei Knoten $u$ und $v$ genau dann in Relation gesetzt, wenn sie in der

Zielarchitektur von derselben funktionalen Einheit ausgeführt werden sollen. Alle Knoten einer Äquivalenzklasse $[v]_\phi \in V/\phi$ sind bei ihrer Ausführung an die gleiche funktionale Einheit gebunden. Die Zerlegung $V/\phi$ wird deshalb auch **Bindung** genannt. Da jede Äquivalenzklasse $f \in V/\phi$ einer funktionalen Einheit im Prozessor entspricht, wird der Begriff funktionale Einheit auch für die Äquivalenzklasse $f$ verwendet. Da Operationen, die der gleichen FU zugeordnet sind, auch im gleichen Cluster liegen, gilt $\phi \subseteq \chi$.

Bereits aus einer Clusterung und einem Ablaufplan lassen sich Mindestanforderungen an die Parameter eines Prozessors, der den Ablaufplan abarbeiten soll, ableiten. So kann neben der Anzahl der internen Cluster, die $|V/\chi - \{[e]\}_\chi|$ beträgt, die Anzahl der durch eine Instruktion $\alpha(i)$ beanspruchten funktionalen Einheiten in jedem Cluster $c \in V/\chi$ durch die **Instruktionsbreite**

$$FUCIWidth(\alpha(i), c) = \left|c \cap \alpha(i)\right|$$

bestimmt werden. Daraus lässt sich die **Clusterbreite** berechnen, die durch die größte Instruktionsbreite in einem Cluster festgelegt ist:

$$FUCWidth(\alpha, c) = \max\left\{FUCIWidth(\alpha(i), c) \middle|\ 0 \leq i < |\alpha|\right\}.$$

Als **Ablaufplanbreite** wird die größte Anzahl funktionaler Einheiten in einem Cluster bezeichnet:

$$FUWidth(\alpha, \chi) = \max\left\{FUCWidth(\alpha, c) \middle|\ c \in (V\,/\,\chi - \{[e]_\chi\})\right\}.$$

Aus der Clusterbreite lässt sich auch unmittelbar die Anzahl der benötigten internen Lese- und Schreibports in der Registerbank, die **interne Registerbankgröße**, des Clusters $c$

$$iPCWidth(\alpha, \chi, c) = \begin{cases} 3 \cdot FUCWidth(\alpha, c), \text{ falls } c \neq [e]_\chi \\ 0, \text{ sonst} \end{cases}$$

ableiten. Die **Größe einer Registerbank** $PCWidth$ eines Clusters berücksichtigt auch noch die Anzahl der externen Lese- und Schreibports $erPWidth$ und $ewPWidth$ in der Registerbank:

$$PCWidth(\alpha, \chi, c) = iPCWidth(\alpha, \chi, c) + erPWidth(\alpha, \chi) + ewPWidth(\alpha, \chi).$$

Obwohl die Architektur eine unterschiedliche Anzahl externer Lese- und Schreibports in jeder Registerbank zulässt, wird diese Eigenschaft nicht genutzt, da das Verbindungsnetzwerk dadurch unflexibler und irregulär wird. Jede Registerbank stellt deshalb dieselbe Anzahl externer Lese- ($erPWidth$) und Schreibports ($ewPWidth$) zur Verfügung, die sich aus der maximal verwendeten Anzahl externer Lese- ($rPC$) bzw. Schreibports ($wPC$) in einer der Registerbänke ergibt:

$$erPWidth(\alpha, \chi) = \max\left\{rPC(\alpha, \chi, c) \middle|\ c \in V\,/\,\chi - \{[e]_\chi\}\right\}$$

und

$$ewPWidth(\alpha,\chi) = \max\left\{wPC(\alpha,\chi,c)\middle|\ c \in V/\chi - \{[e]_\chi\}\right\}.$$

Die von $\alpha$ maximal verwendete Anzahl externer Lese- und Schreibports in der Registerbank des Clusters $c$ ist

$$rPC(\alpha,\chi,c) = \max\left\{rPCI(\alpha(i),\chi,c)|\ 0 \le i < |\alpha|\right\} \text{ und} \tag{1.11}$$

$$wPC(\alpha,\chi,c) = \max\left\{wPCI(\alpha(i),\chi,c)|\ 0 \le i < |\alpha|\right\}, \tag{1.12}$$

wobei *rPCI* und *wPCI* die Anzahl der benötigten externen Lese- bzw. Schreibports in der Instruktion $i$ im Cluster $c$ ist. Die Anzahl der in Instruktion $i$ gleichzeitig benutzten externen Leseports in der Registerbank des Clusters $c$ kann aus den Operationen, die in dieser Instruktion im externen Cluster ausgeführt werden und einen Wert aus dem Cluster $c$ lesen, durch

$$rPCI(\alpha(i),\chi,c) = \sum_{\substack{u\in(\alpha(i)\cap[e]_\chi)\wedge\\ \exists w\in c:(w,u)\in D}} readPorts(type(u))$$

berechnet werden, wobei $readPorts(t)$ die Anzahl der von einer Operation des Typs $t$ verwendeten externen Leseports ist. Jede Kopieroperation benötigt dabei einen Lese- und einen Schreibport. Zur Ausführung einer *load*-Operation wird ebenfalls ein Leseport, zum Laden der Speicheradresse aus einem Register, und ein Schreibport, zum Speichern des gelesenen Wertes in ein Register, benötigt. Eine *store*-Operation dagegen benötigt keinen Schreib-, dafür aber zwei Leseports, um die Adresse und den zu schreibenden Wert aus der Registerbank zu lesen. Es ist deshalb

$$readPorts(copy) = writePorts(copy) = readPorts(load) = writePorts(load) = 1$$

und

$$readPorts(store) = 2 \text{ und } writePorts(store) = 0.$$

Analog zur Anzahl der Leseports wird die Anzahl der externen Schreibports in Instruktion $i$ im Cluster $c$ berechnet:

$$wPCI(\alpha(i),\chi,c) = \sum_{\substack{u\in(\alpha(i)\cap[e]_\chi)\wedge\\ \exists w\in c:(u,w)\in D}} writePorts(type(u)).$$

Die insgesamt vorhandene Anzahl externer Lese- und Schreibports in jeder Registerbank wird mit

$$ePWidth(\alpha,\chi) = erPWidth(\alpha,\chi) + ewPWidth(\alpha,\chi)$$

bezeichnet. Die Portanzahl in der größten Registerbank des Prozessors ist

$$PWidth(\alpha,\chi) = \max\left\{PCWidth(\alpha,\chi,c)\middle|\ c \in V/\chi - \left\{[e]_\chi\right\}\right\}.$$

Daraus lässt sich die maximal mögliche Taktfrequenz, mit der ein Ablaufplan $\alpha$ abgearbeitet werden kann, durch

$$fm(PWidth(\alpha, \chi))$$

berechnen. Die Kosten der Registerbänke im Prozessor sind ebenfalls durch den Ablaufplan $\alpha$ und die Clusterung $\chi$ festgelegt:

$$RBCost(\alpha, \chi) = \sum_{c \in (V/\chi - \{[e]_\chi\})} RBCCost(\alpha, \chi, c)\,.$$

Dabei bezeichnet *RBCCost* die Kosten der Registerbank im Cluster $c$ unter der Annahme, dass jede Registerbank mindestens so viele Register wie Ports besitzt:

$$RBCCost(\alpha, \chi, c) = RFCCost(PCWidth(\alpha, \chi, c), (PCWidth(\alpha, \chi, c), 0, 0))\,.$$

Die Anzahl der Operationen des Typs $t$, die in einer Instruktion $i$ in einem Cluster $c \in V/\chi$ gleichzeitig ausgeführt werden, ist

$$TICWidth(\alpha(i), c, t) = \left|\left\{w \middle|\ w \in (\alpha(i) \cap c) \wedge type(w) = t\right\}\right|.$$

Es müssen also mindestens

$$TCWidth(\alpha, c, t) = \max\left\{TICWidth(\alpha(i), c, t) \middle|\ 0 \leq i < |\alpha|\right\}$$

viele funktionale Einheiten den Operationstypen $t$ im Cluster $c$ implementieren. Aus diesen Werten für jeden Cluster und jeden Operationstypen ergibt sich eine untere Schranke für die Datenpfadkosten des Prozessors, der benötigt wird, um $\alpha$ abzuarbeiten:

$$\sum_{t \in \mathcal{O}} \sum_{c \in V/\chi} opCost(t) \cdot TCWidth(\alpha, c, t)\,. \tag{1.13}$$

Ein Tripel ($\alpha$, $\chi$, $\phi$) bestehend aus einem Ablaufplan, einer Clusterung und einer Bindung wird **Zielprogramm** genannt. Ein solches Tripel stellt bis auf die Registerallokation das vom Prozessor auszuführende Programm dar. Aus ihm lassen sich die Mindestanforderungen an die Parameter eines VLIW-Prozessors, der dieses Zielprogramm abarbeiten kann, herleiten. Einem Zielprogramm ($\alpha$, $\chi$, $\phi$) wird ein VLIW-Prozessor ($\mathcal{C}$, $\mathcal{F}$, *cf*, *ft*, *rz*, *rpc*) zugeordnet mit:

- $\mathcal{C} \coloneqq V \,/\, \chi\,,$
- $\mathcal{F} \coloneqq V \,/\, \phi\,,$
- $\forall c \in \mathcal{C} : cf(c) \coloneqq c \,/ (\phi \cap (c \times c))\,,$
- $\forall f \in \mathcal{F} : ft(f) \coloneqq \{type(v) |\ v \in f\}\,,$
- $\forall c \in \mathcal{C}$ :

$$rpc(c) = \begin{cases} (0,0,0), \text{ falls } c = [e]_\chi \\ (PCWidth(\alpha,\chi,c), erPWidth(\alpha,\chi), ewPWidth(\alpha,\chi)), \text{ sonst.} \end{cases}$$

Die Anzahl $rz(c)$ der mindestens benötigten Register in der Registerbank eines Clusters $c$ wird während der DSE nicht minimiert. Sie kann aus der maximalen Anzahl gleichzeitig lebendiger Werte in einer Instruktion im Cluster $c$ berechnet werden. Im Ablaufplan $\alpha$ und Cluster $c$ ist die Anzahl der in Instruktion $i$ gleichzeitig lebendigen Werte

$$|\,\{v \mid \exists (v,w) \in D\ \exists j,k \in \mathbb{N} : j \leq i < k \wedge v \in \alpha(j) \wedge v \notin \alpha(i+1) \wedge w \in \alpha(k) \wedge (v \in c \vee w \in c)\}\,|\,.$$

Weiterhin müssen noch Register zum Speichern globaler Werte vorhanden sein. Diese globalen Werte werden in dem Basisblockmodell nicht dargestellt und es wird vorausgesetzt, dass genügend Register vorhanden sind, um diese Werte zu speichern. Sollten dafür nicht genügend Register vorhanden sein, dann wäre Programmcode zum Ein- und Auslagern von Registerwerten erforderlich. Dieser zusätzliche Programmcode erhöht im Allgemeinen die Ausführungszeit des Programms, weswegen es nicht sinnvoll ist, ihn in die Ablaufpläne von zeitkritischen Basisblöcken einzufügen. Die Konsequenz ist, dass die Registerbänke des Prozessors genügend Register zur Verfügung stellen müssen, um Auslagerungscode zu vermeiden und das Speichern globaler Werte in Registern während der Abarbeitung der zeitkritischen Basisblöcke zu ermöglichen.

Mit diesem Programmmodell können die Parameter eines VLIW-Prozessors vollständig durch ein für ihn erzeugtes Zielprogramm beschrieben werden. Das grundsätzliche Vorgehen bei der DSE mit DESCOMP wird im folgenden Abschnitt beschrieben.

## 2.3 Prinzip der DSE mit DESCOMP

Mit DESCOMP können vollständig automatisiert die folgenden Parameter eines geclusterten VLIW-Prozessors bestimmt werden:

- Die Anzahl der Cluster,
- für jeden Cluster eine geeignete FU-Anzahl und
- für jede FU die bereitzustellenden Operatoren.

Das schließt die Kopieroperatoren mit ein, da diese zum externen Cluster gehören. Der prinzipielle Ablauf der DSE mit den erforderlichen Eingaben, um diese Parameter zu bestimmen, ist in Abbildung 2.10 dargestellt.

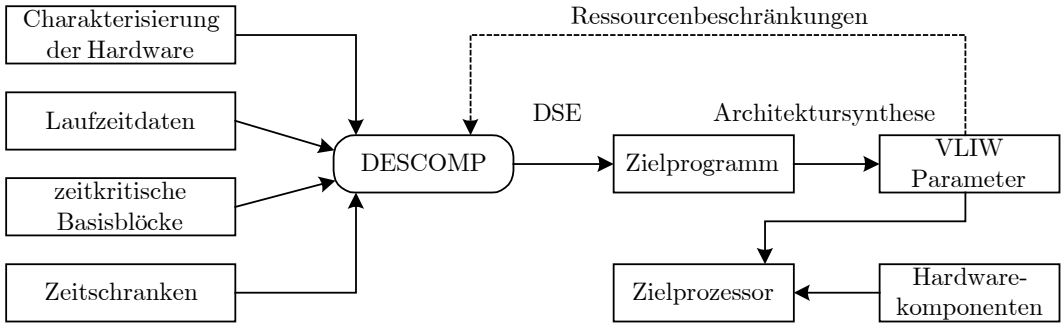



Abbildung 2.10: Prinzip der Architekturoptimierung mit DESCOMP.

Die *Charakterisierung der Hardware* umfasst die Latenzzeiten, Platz und Stromverbrauch der Operatoren, sowie das Zeitverhalten, den Platz- und Stromverbrauch einer Registerbank in Abhängigkeit von der Portanzahl. Als weitere Eingabe erhält DESCOMP die *zeitkritischen Basisblöcke* der Anwendung, für die die Architektur optimiert werden soll und *Zeitschranken*, die sich jeweils auf einen oder eine Menge dieser Basisblöcke beziehen und die bei der Ausführung dieser Basisblöcke nicht überschritten werden dürfen. Bei einer der DSE vorangegangenen Simulation der Anwendung wurden die Ausführungshäufigkeiten der zeitkritischen Basisblöcke bei repräsentativen Eingabedaten ermittelt. Diese *Laufzeitdaten* werden ebenfalls während der DSE benötigt.

Aus diesen Eingabedaten wird durch die Übersetzung der Basisblöcke ein *Zielprogramm* für die Basisblöcke erzeugt, das – ausgeführt von einem Prozessor mit einer geeigneten Taktfrequenz – die Zeitanforderungen einhält. Bei dieser ersten Übersetzung der Basisblöcke existieren keine Ressourcenbeschränkungen. Aus dem Zielprogramm lassen sich dann die *Parameter* eines VLIW-Prozessors, wie es am Ende des Abschnitts 2.2 bereits beschrieben wurde, ableiten. Daraus, und aus der Spezifikationen vorhandener *Hardwarekomponenten*, beispielsweise in VHDL, kann der Zielprozessor erzeugt werden.

Der Compiler für einen mit DESCOMP erzeugten Prozessor entsteht durch die *Ressourcenbeschränkung* der in DESCOMP zur DSE verwendeten Algorithmen. Diese Ressourcenbeschränkungen ergeben sich aus den VLIW-Parametern, die während der DSE berechnet wurden. Werden damit dieselben Basisblöcke erneut übersetzt, dann entsteht auch dasselbe Zielprogramm, wie bereits bei der ersten Übersetzung während der DSE. Entsprechende Untersuchungen sind in [67] durchgeführt worden. Dort wurden die in DESCOMP verwendeten Algorithmen so modifiziert, dass für Operationen mit einer Latenzzeit von eins vorgegebene Ressourcenbeschränkungen bei der Ablaufplanung für einen nicht geclusterten Prozessor eingehalten wurden. Somit kann DESCOMP auch als Compiler verwendet werden. Damit Zielcode für die gesamte Anwendung erzeugt werden kann, muss es FUs im Prozessors geben, die die Operationstypen unterstützen, die in der Anwendung aber nicht in den zeitkritischen Basisblöcken vorkommen. Die Zuordnung dieser Operationstypen zu Clustern und funktionalen Einheiten wird nicht mit DESCOMP optimiert, sondern in einem der Optimierung nachgelagerten Schritt, der in dieser Arbeit nicht behandelt wird.

Diese Dissertation beschäftigt sich ausschließlich mit dem Schritt der DSE in Abbildung 2.10, die während der ersten Übersetzung der zeitkritischen Basisblöcke durchgeführt wird und bei der keine Ressourcenbeschränkungen existieren. Das Zielprogramm wird dabei so erzeugt, dass primär die Hardwarekosten oder der Stromverbrauch, des zur Ausführung des Zielprogramms benötigten Prozessors, minimiert werden. Erreicht wird das für die Hardwarekosten durch die Minimierung der Portanzahl in der größten Registerbank und die Gesamtanzahl der Ports in den übrigen Registerbänken. Eine Minimierung der Portanzahl in den Registerbänken bedeutet außerdem eine Minimierung der bereitgestellten Parallelität im Prozessor, was sich auf die Anzahl der benötigten Operatoren auswirkt, wodurch ebenfalls der Platz- und Stromverbrauch sinkt. Der Stromverbrauch kann weiter minimiert werden, indem durch eine Erhöhung der bereitgestellten Parallelität die Taktfrequenz verringert wird, was zur Absenkung der Versorgungsspannung genutzt werden kann, wodurch der dynamische Stromverbrauch sinkt. Zur Erzeugung des Zielprogramms sind grundsätzlich die zwei Phasen

- **lokale Optimierung** mit Ablaufplanung, Clusterung und Ressourcenallokation sowie
- **globale Optimierung** mit Selektion der Basisblockkandidaten und Ressourcenallokation

zu unterscheiden. Ein Überblick über die Phasen und deren Zusammenhang ist in Abbildung 2.11 gegeben.

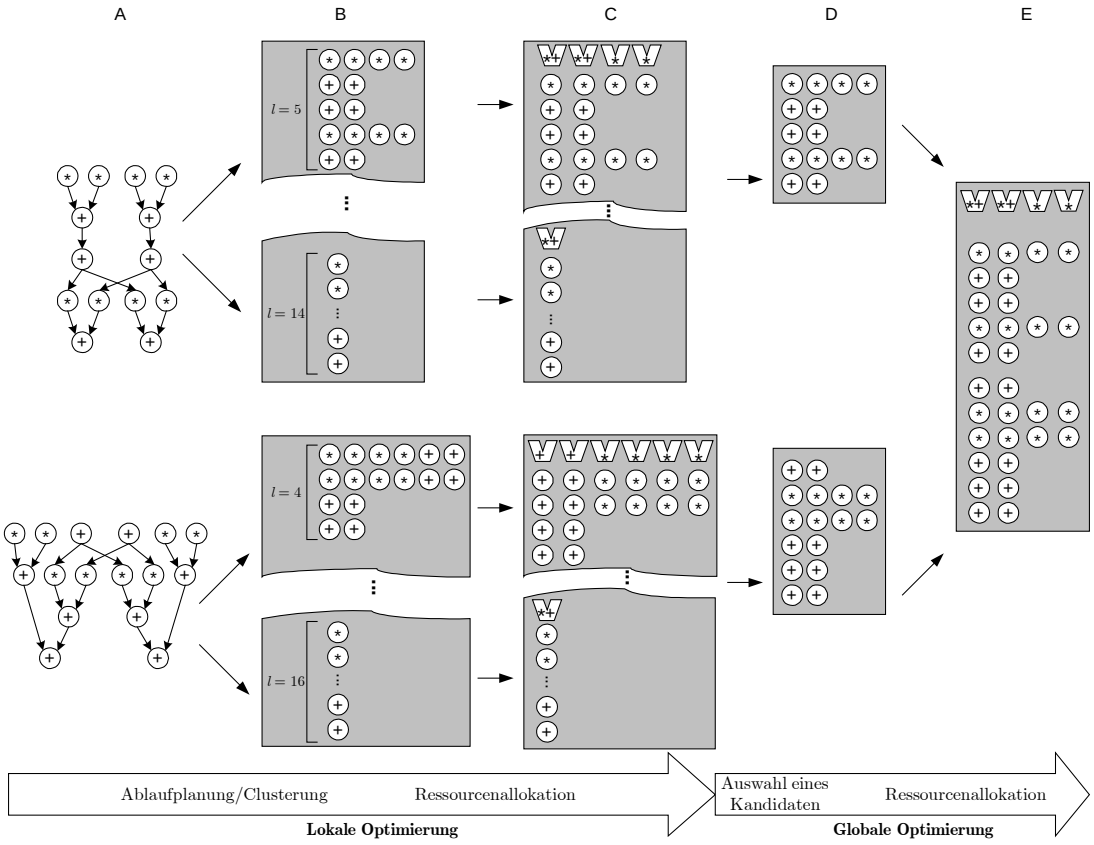



Abbildung 2.11: Erzeugung des Zielprogramms.

Während der lokalen Optimierungsphase wird jeder Basisblock einzeln betrachtet, dargestellt durch die zwei Basisblöcke in Spalte A in Abbildung 2.11. Für jeden zeitkritischen Basisblock $b$ der Anwendung werden Ablaufpläne aller zulässigen Längen $l$, d. h. $cp(b) \leq l \leq lp(b)$, erzeugt. Der dafür verwendete Planungsalgorithmus arbeitet mit geclusterten Basisblöcken und ist mit dem Algorithmus zur Bestimmung einer Clusterung gekoppelt. Dieser Clusterungsalgorithmus erzeugt bei einer fest vorgegebenen Ablaufplanlänge $l$ und Clusteranzahl aus einem Basisblock einen geclusterten Basisblock, so dass dessen kritische Pfadlänge die vorgegebene Länge $l$ nicht überschreitet. Damit ist der Planungsalgorithmus in der Lage, einen Ablaufplan zu erzeugen, der genau die Länge $l$ hat. Planung und Clusterung werden für denselben Basisblock bei derselben vorgegebenen Länge und Clusteranzahl abwechselnd ausgeführt, um den Ablaufplan schrittweise zu verbessern und die gegenseitigen Wechselwirkungen aufgrund der in den Basisblock eingefügten Kopieroperationen zu beachten. Es entsteht dadurch für jeden Basisblock und jede zulässige Ablaufplanlänge $l$ eine Menge von Ablaufplänen, die zu ihrer Ausführung unterschiedlich viele Cluster benötigen, in Spalte B nur für Ablaufpläne mit einem Cluster dargestellt. Damit ist für jeden Ablaufplan die Anzahl der benötigten Cluster und FUs in jedem Cluster sowie die Portanzahl in jeder Registerbank festgelegt. Aufgrund der Portanzahl in der größten

Registerbank lässt sich für jeden Ablaufplan auch die maximal mögliche Taktfrequenz des zu seiner Ausführung benötigten Prozessors bestimmen.

Im nächsten Schritt wird eine Ressourcenallokation, d. h. die Zuordnung der Operatoren zu funktionalen Einheiten, berechnet, indem für einen Ablaufplan $\alpha$ eine Bindung $\phi$ bestimmt wird. Mit der Bindung ist ein Zielprogramm $(\alpha, \chi, \phi)$ mit der Ablaufplanlänge $|\alpha|$ zu einem Basisblock vollständig bestimmt und somit auch die Architektur des VLIW-Prozessors, der dieses Zielprogramm abarbeiten kann. Planung und Bindung sind gekoppelt, indem die Auswirkungen von Entscheidungen während der Planung auf die spätere Bindung berücksichtigt werden. Damit sind am Ende der lokalen Optimierungsphase für jeden Basisblock $b$ und für jede zulässige Ablaufplanlänge von $b$ Zielprogramme berechnet worden, in Spalte C für Ablaufpläne mit einem Cluster dargestellt. Für eine DSE werden diese Zielprogramme nur einmal berechnet und eine mehrfache Übersetzung der Basisblöcke für VLIW-Prozessoren mit unterschiedlichen Parameterkonfigurationen ist nicht notwendig.

Erst in der globalen Optimierungsphase wird ein Prozessor erzeugt, der alle zeitkritischen Basisblöcke innerhalb der gegebenen Zeitschranken ausführen kann. Weil sich eine Zeitschranke auf mehrer Basisblöcke beziehen kann, muss deshalb neben den Parametern des Prozessors auch noch für jeden Basisblock eine Zeit festgelegt werden, die für seine Ausführung zur Verfügung steht. Beides wird durch die Auswahl eines Kandidaten für jeden Basisblock aus der Menge aller für den Basisblock erzeugten Zielprogramme berechnet, dargestellt in Spalte D. Da die ausgewählten Kandidaten auf demselben Prozessor ausgeführt werden, sollten sie ähnliche Anforderungen an die Parameter des Prozessors haben, damit die Ressourcen des Prozessors von allen Kandidaten gut genutzt werden können. Kandidaten mit ähnlichen Anforderungen an die Ressourcen eines Prozessors können während der Auswahl der Kandidaten gesucht werden, wenn die Zeitschranken zulassen, dass nicht für jeden Basisblock der kürzeste Ablaufplan als Kandidat ausgewählt werden muss bzw. zwischen unterschiedlich geclusterten Ablaufplänen mit derselben Länge ausgewählt werden kann. Eine Aussage über die Ausführungszeit eines Ablaufplans lässt sich bei den ausgewählten Kandidaten aufgrund der bekannten Ablaufplanlängen, Ausführungshäufigkeiten und Portanzahl in der größten Registerbank treffen. Mit den so bestimmten Kandidaten für jeden Basisblock wird eine abschließende Bindung zur Bestimmung der Ressourcenallokation im Zielprozessor (Spalte E) durchgeführt. Diese Phase der Bindung arbeitet mit allen ausgewählten Ablaufplänen gleichzeitig und ist notwendig, da zuvor die Bindungen nur lokal für jeweils einen Ablaufplan optimiert wurden.

Bevor die lokale und globale Optimierungsphase detailliert erläutert werden, wird im nächsten Kapitel ein Überblick über vorhandene DSE-Ansätze für VLIW-Prozessoren und Techniken zur Erzeugung anwendungsspezifischer Datenpfade gegeben.

# 3 Verwandte Arbeiten

Es gibt in der Literatur zahlreiche Arbeiten, die sich mit dem Entwurf anwendungsspezifischer Datenpfade beschäftigen. In der High-Level-Synthese (HLS) werden Datenpfade für ASICs (*application specific integrated circuits*) erzeugt. Entsprechende Verfahren werden im Abschnitt 3.1 behandelt. DSE-Ansätze zur Erzeugung des Datenpfades eines anwendungsspezifischen VLIW-Prozessors werden in Abschnitt 3.2 vorgestellt. Aus beiden Bereichen werden grundsätzliche Strategien sowie deren Vor- bzw. Nachteile herausgearbeitet.

## 3.1 High-Level-Synthese

Bereits seit den 70er Jahren des vergangenen Jahrhunderts beschäftigt sich die HLS intensiv mit der Synthese von ASICs [32]. Aufgrund der zunehmenden Komplexität der Schaltkreise und der zu realisierenden Funktionen wird der Entwurf mit einer Verhaltensbeschreibung begonnen, die die zu berechnende Funktion charakterisiert. Aus dieser Verhaltensbeschreibung wird dann automatisiert, meistens jedoch manuell mit Werkzeugunterstützung, ein integrierter Schaltkreis erzeugt, der genau diese Funktion berechnet. In einem ersten Schritt, der Architektursynthese, wird dafür aus der Verhaltensbeschreibung eine Architekturbeschreibung des ASICs erzeugt, die die benötigten Operatoren, Register und deren Verdrahtung spezifiziert. Daran schließt sich die Logiksynthese an, die diese Architekturbeschreibung in Logikgatter mit Verdrahtung und ein Layout transformiert. In diesem Abschnitt wird nur die Architektursynthese betrachtet.

Die Verhaltensbeschreibung ist oft in Form eines 3-Adress-Code ähnlichen Formats [92], VHDL oder spezieller Sprachen wie z. B. HardwareC [33] oder SILAGE [37] gegeben, die zur Architektursynthese in einen Datenflussgraphen (DFG) oder einen kombinierten Steuer- und Datenflussgraphen (CDFG), der auch hierarchisch aufgebaut sein kann, transformiert werden [9, 32]. In hierarchischen Graphen kann ein Knoten wieder einen hierarchischen Graphen repräsentieren, wodurch sich Funktionsaufrufe und Schleifen modellieren lassen [32]. Parallel oder im gegenseitigen Ausschluss abzuarbeitende Teile der Verhaltensbeschreibung sind teilweise explizit gekennzeichnet, wie beispielsweise in [92, 95]. Der durch die HLS erzeugte ASIC besteht aus Operatoren, von denen jeder eine festgelegte Menge von Operationen, meist jedoch nur eine, realisieren kann. Die Operatoren sind

untereinander verdrahtet. Die Verdrahtung ist speziell der zu realisierenden Funktion angepasst und kann somit sehr irregulär sein. Register zum Speichern von Daten sind nur mit den Funktionsblöcken verdrahtet, die auf diese Werte zugreifen müssen. Eine zentrale Registerbank gibt es nicht. Die Schaltung besteht aus einem Steuer- und einem Datenpfad. Im Datenpfad werden die Berechnungen ausgeführt. Der Steuerpfad steuert die Zustandsübergänge des ASICs in jedem Takt. In jedem Zustand sind gewisse Operatoren der Schaltung aktiv, um die jeweilige Operation auszuführen. Die Zustände sind damit den Instruktionen im Ablaufplan eines Prozessor gleichzusetzen.

### 3.1.1 Allgemeines Vorgehen in der High-Level-Synthese

Die Schritte, um aus einer Verhaltensbeschreibung die im Datenpfad benötigten Funktionsblöcke und deren Verdrahtung zu bestimmen, können in Planung[1] und Ressourcenallokation unterteilt werden. In einigen Ansätzen werden für die Operationen in der Verhaltensbeschreibung noch geeignete Operatoren ausgewählt, die die betreffende Operation im ASIC ausführen [9, 71]. Dadurch können teure und schnelle Operatoren gegen langsame und kostengünstigere ausgetauscht werden. Dem liegt die Annahme zugrunde, dass die Geschwindigkeit des ASICs durch die Gatter in den Operatoren dominiert wird. Bei der Planung werden die im DFG vorkommenden Operationen den einzelnen Zuständen, in denen sich der ASIC befinden kann, zugeordnet. Die Ressourcenallokation legt Art und Anzahl der Operatoren in der Schaltung fest. Das geschieht unter der Maßgabe, dass Operationen desselben Typs, die nicht gleichzeitig ausgeführt werden, möglichst vom selben Operator ausgeführt werden, um die Anzahl der Operatoren im ASIC zu minimieren.

Zahlreiche Techniken zur Planung und Ressourcenallokation sind in der Literatur beschrieben [9, 22, 32, 57, 70, 77, 81]. Die Planungsverfahren lassen sich in zeit- und ressourcenbeschränkte Verfahren unterteilen. Zeitbeschränkte Verfahren minimieren den Ressourcenbedarf (Operatoren, Register, Busse) bei einer gegebenen Ausführungszeit, während ressourcenbeschränkte Verfahren die Ausführungszeit bei gegebenen Ressourcen minimieren. Die zeitbeschränkten Planungsverfahren verwenden die Anzahl der Zustände, die zur Planung der Operationen zur Verfügung stehen, oder relative Abstände zwischen den Operationen [22] als Zeitbeschränkung. Die Anzahl der für die Planung verfügbaren Zustände wird dabei oft durch den längsten Pfad im DFG bestimmt [9, 17, 76]. Die ressourcenbeschränkte und zeitbeschränkte Planung sind $\mathcal{NP}$-vollständige Probleme, selbst für den Fall, dass es nur einen Ressourcentyp gibt [66]. Für beide Probleme werden deshalb oft heuristische Planungsverfahren genutzt.

[1] In der Literatur immer mit Scheduling bezeichnet.

Heuristische ressourcenbeschränkte Verfahren basieren dabei oft auf dem Prinzip des List-Scheduling [32, 52], bei dem die Operationen in der durch eine Prioritätsfunktion festgelegten Reihenfolge verplant werden, wobei eine Operation in eine Instruktion geplant wird, sobald alle ihre Vorgänger verplant wurden und genügend Ressourcen zur Ausführung der Operation vorhanden sind. Die einfachsten Verfahren zur Erzeugung eines zeitbeschränkten Ablaufplans sind die ASAP- und ALAP-Planung[1] [17, 52], bei denen jeder Knoten so zeitig wie möglich bzw. so spät wie möglich geplant wird. Ein oft verwendetes Verfahren zur zeitbeschränkten Planung, das im Gegensatz zur ASAP- und ALAP-Planung die Ressourcenverwendung ausbalanciert, ist das Force-Directed-Scheduling [70], das zunächst für jede Operation den frühesten und spätesten Ausführungszeitpunkt berechnet. Eine Operation wird innerhalb ihres Zeitintervalls in die Instruktion geplant, in der die Auslastung der benötigten Ressource bezogen auf die durchschnittliche Auslastung dieser Ressource innerhalb des gesamten Zeitintervalls am kleinsten ist. Dabei werden die Ressourcenauslastungen der unmittelbaren Vorgänger und Nachfolger der Operation mit in die Betrachtungen einbezogen. Ziel des Verfahrens ist es, Operationen desselben Typs in jedem Zustand sowie benötigte Register und Verbindungen zwischen den Operatoren zu minimieren [79]. Dass in jedem Zustand außerdem auch möglichst wenig Operationen – egal welchen Typs – ausgeführt werden sollen, wird nicht berücksichtigt. Der Grund ist, dass vorhandene Operatoren in einem ASIC ohne weiteres parallel arbeiten können, weil sie sich nicht gegenseitig beim Zugriff auf die Registerbank als gemeinsam genutzte Ressource ausschließen. In Abbildung 3.1 soll das verdeutlicht werden.

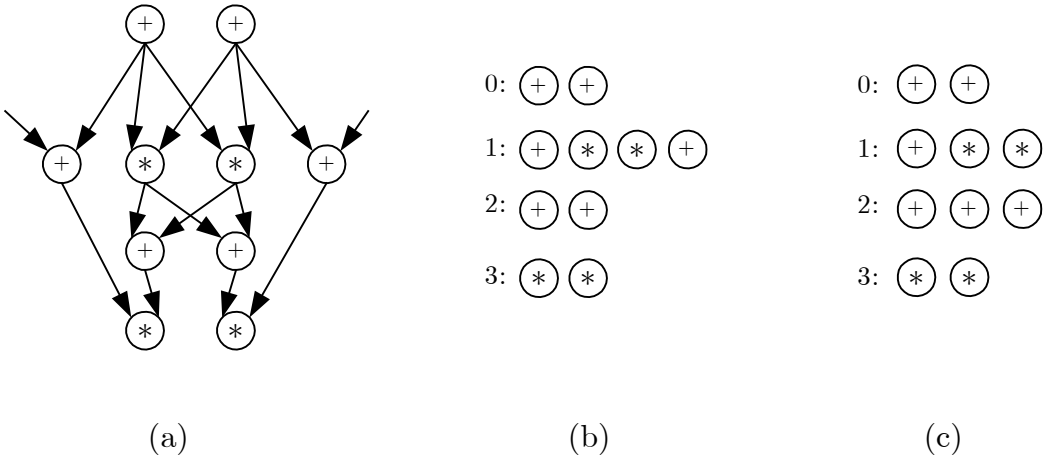



Abbildung 3.1: (a) Datenflussgraph; (b) Ablaufplan zum Datenflussgraphen optimiert für einen ASIC; (c) Ablaufplan optimiert für einen VLIW-Prozessor.

[1] ASAP und ALAP stehen für *as soon as possible* und *as late as possible*.

Um die Operationen des in (a) dargestellten Datenflussgraphen in vier Zuständen zu planen, ist der in (b) angegebene Ablaufplan optimal für einen ASIC, weil zur Ausführung der ersten beiden Additionen bereits zwei Addierer benötigt werden und diese zwei Addierer auch im Folgezustand parallel zu den zwei Multiplizierern arbeiten können. Es werden dadurch vier Operationen parallel ausgeführt. Würde der Ablaufplan in (b) von einem VLIW-Prozessor ausgeführt, dann müsste dieser vier funktionale Einheiten bereitstellen. Darauf kann verzichtet werden, wenn eine der in Instruktion 1 ausgeführten Additionen in Instruktion 2 ausgeführt wird, wie in (c) dargestellt. Das erfordert zwar einen Addierer mehr, der kann aber kostengünstiger sein, als eine größere Registerbank, die 12 statt 9 Ports bereitstellen muss.

Die Ressourcenallokation kann bei einem gegebenen Ablaufplan durch die Bindung von Operationen an Operatoren durchgeführt und durch Färbung eines Konfliktgraphen oder Suchen einer minimalen Überdeckung mit Cliquen im Verträglichkeitsgraphen berechnet werden [32, 52, 81]. Der Verträglichkeitsgraph ist der komplementäre Graph zum Konfliktgraphen und beide Probleme sind deshalb äquivalent. Im Konfliktgraphen entsprechen die Knoten den Operationen des DFG. Zwei Knoten sind genau dann durch eine Kante verbunden, wenn sie nicht von einem gemeinsamen Operator ausgeführt werden dürfen. Das ist immer dann der Fall, wenn sie durch die vorangegangene Planung demselben Zustand zugeordnet wurden und vom selben Operator ausgeführt werden könnten. Eine Färbung des Konfliktgraphen liefert dann eine Bindung, bei der alle Knoten, die dieselbe Farbe erhalten haben, demselben Operator zugeordnet werden. In [32, 52] wird die Ressourcenallokation für alle Operationen, die denselben Typ haben, separat durchgeführt. Damit wird das Problem bei einem gegebenen Ablaufplan durch den LEFT-EDGE-Algorithmus [5] in polynomieller Zeit exakt lösbar, weil der entstehende Konfliktgraph als Intervallgraph[1] dargestellt werden kann. Für hierarchische Datenflussgraphen, wie sie in [32] verwendet werden, geht diese Eigenschaft verloren, weswegen wieder auf Heuristiken zurückgegriffen wird. In [17] sind die Kanten des Verträglichkeitsgraphen in Kategorien eingeteilt, durch die die Verträglichkeit der Operanden modelliert wird. Die Bildung von Cliquen wird in mehreren Schritten, jeweils auf die einzelnen Kategorien bezogen, durchgeführt. Dadurch wird bei der Ressourcenallokation auch das Verbindungsnetzwerk zwischen den Operatoren im ASIC minimiert.

[1] Ein ungerichteter Graph $(V, E)$ wird als Intervallgraph bezeichnet, wenn jedem seiner Knoten $v \in V$ ein Zeitintervall $[s_v, e_v)$ mit $s_v, e_v \in \mathbb{R}$ zugeordnet werden kann, so dass sich die Intervalle zweier Knoten $u$ und $v$ genau dann überlappen, wenn $\{u, v\} \in E$.

### 3.1.2 High-Level-Synthesesyteme

Existierende Systeme zur High-Level-Synthese sind unter anderem Olympus [33], V-Synth/Mimola [42], CATHEDRAL II [37], HYPER [16], EASY [95], HAL [71] und MAHA [9]. In jedem dieser HLS-Systeme werden die beschriebenen Phasen der Planung und Ressourcenallokation durchgeführt. Sie sind aber auf unterschiedliche Weise gekoppelt und auch abweichend von den oben beschriebenen Techniken realisiert worden. In vielen Fällen bieten die Systeme dem Entwickler nur eine Unterstützung beim Entwurf eines geeigneten ASICs. Wichtige Entscheidungen zur Planung bzw. Ressourcenallokation müssen durch den Entwickler getroffen werden.

Während bei EASY die Planung und somit auch die Parallelisierung der Operationen durch den Entwickler von Hand durchgeführt werden muss, berechnet V-Synth den kürzesten möglichen Ablaufplan für das gegebene Programm. In beiden Ansätzen wird die Planung vor der Ressourcenallokation durchgeführt. Da ein Ablaufplan minimaler Länge im Allgemeinen zu einer hohen Parallelität führt, kann der Entwickler diese Parallelität bei V-Synth einschränken, indem er einzelne Operationen vor der Planung an dieselbe Ressource bindet. Diese Bindung wird bei der Planung und durch die anschließende Ressourcenallokation des Mimola-Systems [76] berücksichtigt und führt somit zu einer sequentiellen Ausführung aller Operationen, die sich dieselbe Ressource teilen. Der Entwickler wird durch die Synthesewerkzeuge bei der Entwicklung des ASICs unterstützt, muss aber selbst Entscheidungen während des Syntheseprozesses treffen. Auch in CATHEDRAL II ist Nutzerinteraktion während der Synthese notwendig und gewollt, um in einem iterativen Prozess durch zusätzliche Bindungsanweisungen die Syntheseergebnisse zu verbessern. In jeder Iteration ist eine erneute Planung der Operationen unter Beachtung der zusätzlichen Bindungsanweisungen notwendig. Das CATHEDRAL II Synthesesystem nutzt als Basisarchitektur an die Anwendung angepasste Ausführungseinheiten, die unter anderem typische DSP-Operationen ausführen können. Jede dieser Ausführungseinheiten besteht aus einzelnen Funktionsblöcken, von denen jeder über eigene Registerbänke für die Eingabewerte verfügt. Die Registerbänke sind bei Bedarf untereinander durch Busse verbunden. Die Ausführungseinheiten können über spezielle Puffer Daten austauschen. Die Zielarchitektur ist somit sehr irregulär.

Im Olympus Synthesesystem wird die Anwendung in einen hierarchischen Steuer- und Datenflussgraphen übersetzt. Die Bindung der Operationen an Operatoren kann vor der Planung vom Entwickler durchgeführt werden und muss nicht vollständig sein. Für die übrigen nicht gebundenen Operationen werden mit dem HEBE Subsystem [21] durch ein Branch-and-Bound-Verfahren alle möglichen vollständigen Bindungen berechnet und für diese ein minimaler relativer Ablaufplan, der einem ASAP-Ablaufplan entspricht, berechnet [22]. Mit diesem Ablaufplan kann geprüft werden, ob die

Zeitschranken eingehalten werden können. Durch ein iteratives Vorgehen können Informationen aus der Planung zur Veränderung der Bindungen genutzt werden, wenn beispielsweise zu der durch den Entwickler vorgegebenen Bindung keine Planung der Operationen gefunden werden kann, so dass die Zeitschranken eingehalten werden können. In einem solchen Fall muss durch den Nutzer eine andere Bindung angegeben werden. Aufgrund des Branch-and-Bound-Verfahrens kann die Suche sehr aufwendig sein. Das System ist auf eine Interaktion mit dem Nutzer ausgelegt und soll ihn beim Ausprobieren unterschiedlicher Bindungsvarianten unterstützen. Es gibt aber auch die Möglichkeit, die Synthese automatisch ablaufen zu lassen [21, 33].

Im HYPER Synthesesystem ist eine Nutzerinteraktion nicht zwingend notwendig. Zu einer Funktionsbeschreibung, die in Form eines hierarchischen Steuer- und Datenflussgraphen vorliegt, wird ein ASIC erzeugt, wobei das wesentliche Optimierungsziel die Minimierung der verwendeten Operatoren und des Verbindungsnetzwerkes ist. Die Register sind fest den einzelnen Operatoren zugeordnet und mit ihnen verdrahtet. Die Datenpfadsynthese wird von einer Suchmaschine [44] durchgeführt, die auf Ressourcenverwendungstabellen und einen List-Scheduling-Algorithmus [57] zurückgreift. Der ressourcenbeschränkte List-Scheduling-Algorithmus berechnet bei einer gegebenen Menge von Hardwareressourcen für jeden Subgraphen des hierarchischen Steuer- und Datenflussgraphen und einer vorgegebenen Anzahl von Zuständen einen Ablaufplan. Schlägt die Planung fehl, so stehen der Suchmaschine anschließend Informationen der Ablaufplanung darüber zur Verfügung, welche Ressourcen fehlten und welche Ressourcen nicht ausgelastet wurden [44]. Fehlende Ressourcen können dann ergänzt bzw. nicht ausgelastete Ressourcen verringert werden. Eine bessere Auslastung der Ressourcen kann auch durch mehr Zustände, die zur Abarbeitung eines Subgraphen zur Verfügung gestellt werden, erreicht werden. Die eingesetzte Suchstrategie greift dabei auf probabilistische Techniken zurück [44].

Ein HLS-Ansatz, der die Schritte der Planung und Ressourcenallokation miteinander koppelt, ist in [92] beschrieben. Dort wird Simulated Annealing genutzt, um die kostengünstigste Hardwarearchitektur zu finden. Eine Planung und Ressourcenallokation wird gefunden, indem die Knoten des DFG in einer zweidimensionalen Matrix zufällig verschoben werden. Die Spalten entsprechen dabei den Funktionsblöcken, während die Zeilen den Zuständen der Schaltung entsprechen. Eine Kostenfunktion entscheidet über die Qualität einer gefundenen Lösung. Um die benötigte Ausführungszeit gegen die Kosten abzuwägen, wird jede verwendete Zeile in der Matrix mit zusätzlichen Kosten bestraft. Durch Gewichtungsfaktoren kann deren Einfluss auf das Gesamtergebnis bestimmt werden. Weitere Ansätze, die die Ressourcenallokation und Planung miteinander koppeln und exakte Lösungen berechnen, nutzten ganzzahlige lineare Optimierung. Entsprechende Verfahren sind zum Beispiel in [15, 32, 52] angegeben. Sie weisen aber eine exponentielle Laufzeit auf und sind nur für kleine DFGs geeignet.

## 3.2 Design-Space-Exploration für VLIWs

Um den Datenpfad eines VLIW-Prozessors zu bestimmen, wird in den hier vorgestellten Arbeiten eine Design-Space-Exploration durchgeführt, die als Grundprinzip das von vielen HLS-Werkzeugen genutzte iterative Vorgehen zur Berechnung alternativer Datenpfade [16, 37] verwendet. Dieses Prinzip ist in Abbildung 3.2 schematisch dargestellt:

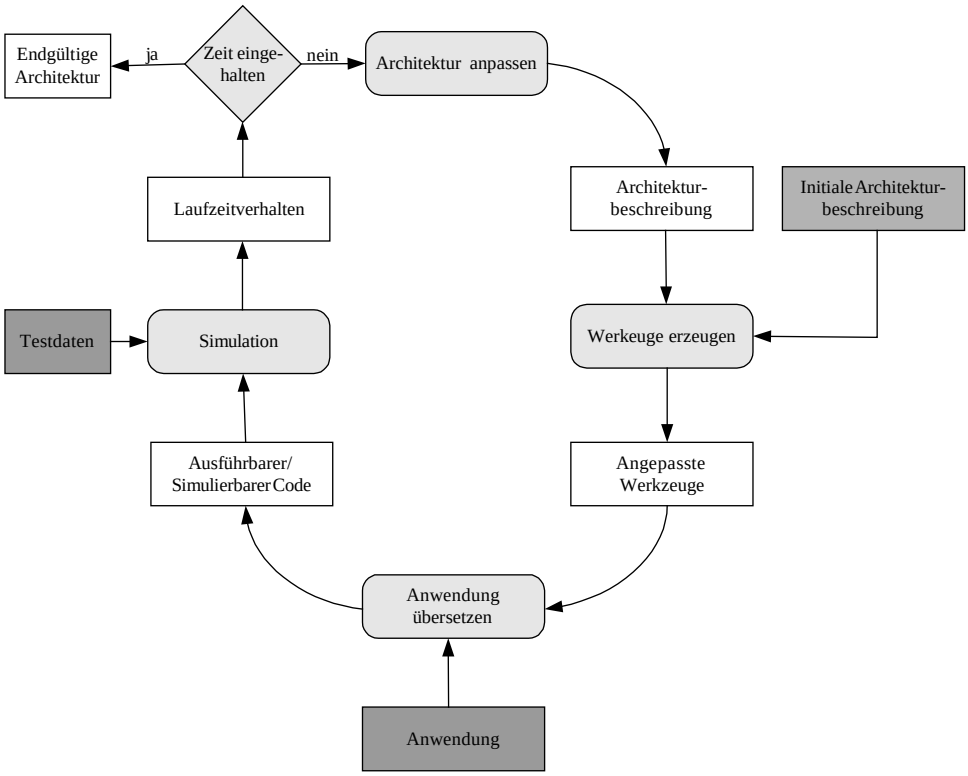



Abbildung 3.2: DSE-Zyklus.

Eine Architektur ist durch eine initiale Architekturbeschreibung gegeben. Zur Beschreibung werden teilweise existierende Architekturbeschreibungssprachen [61, 94] oder proprietäre Formate benutzt. Die benötigten Werkzeuge, wie Compiler und Simulator, werden basierend auf der Beschreibung generiert und an die Architektur angepasst. Anschließend wird die Anwendung mit dem erzeugten angepassten Compiler übersetzt. Der erhaltene ausführbare Code wird dann mit dem ebenfalls angepassten Simulator auf einer gegebenen Menge von Eingabedaten simuliert. Diese Simulation erfolgt nach Möglichkeit taktgenau. Das bedeutet, dass Wartezyklen in Pipelinestufen und bei Speicherzugriffen ebenfalls mit berücksichtigt werden. Abhängig von der so ermittelten Ausführungszeit wird die Architektur bei Bedarf noch weiter angepasst, indem die Architekturbeschreibung modifiziert wird. Mit der modifizierten Architekturbeschreibung beginnt eine neue Iteration des DSE-Zyklus.

Wenn die Modifikation der Architekturbeschreibung manuell durchgeführt wird, dann ist es die Aufgabe der DSE-Werkzeuge, den Entwickler bei seiner Arbeit zu unterstützen. Zahlreiche Werkzeuge, die teilweise auch eine DSE komplexer Systeme mit mehr als einem Prozessor erlauben, existieren bereits [10, 11, 20, 55, 62, 74, 78]. In den folgenden Abschnitten wird nur exemplarisch auf die Möglichkeiten zweier dieser Werkzeuge eingegangen. Anschließend werden verschiedene automatisierte DSE-Verfahren vorgestellt, bei denen teilweise noch eine Interaktion mit dem Entwickler nötig ist. Es wird erläutert wie diese Ansätze in das in Abbildung 3.2 dargestellte Schema passen, welche Architekturparameter erfasst werden und wie bei der Suche einer geeigneten Architektur in den automatisierten Ansätzen vorgegangen wird.

### 3.2.1 Manuelle Design-Space-Exploration

LISA ist eine Prozessorbeschreibungssprache, die zur Beschreibung der Architektur bei einer manuellen DSE in [10] genutzt wird, um daraus sowohl den Compiler als auch den Simulator automatisch zu erzeugen [6]. Das Vorgehen entspricht genau dem in Abbildung 3.2 dargestellten Schema. Mit LISA lassen sich taktgenaue Modelle des Prozessors erstellen und auch simulieren, da die Spezifikation von funktionalen Einheiten mit Pipelines, Registerbänken und Caches möglich ist. Konflikte zwischen Ressourcen in der Architektur können modelliert werden, so dass auch VLIW-Prozessoren beschrieben und simuliert werden können. Der Nachteil dieses Ansatzes ist die notwendige manuelle Exploration des Suchraums. Um die manuelle Suche nach einer geeigneten Architektur zu beschleunigen, wird der Suchraum häufig auf eine systematische Art und Weise eingegrenzt, wie es beispielsweise der DSE-Ansatz mit CASTLE erlaubt.

### 3.2.2 CASTLE

Die CASTLE Co-Design-Umgebung [74] ist ein Framework, das die Einbindung verschiedener Werkzeuge zur DSE einer bestimmten Architektur gestattet. Es wird ebenfalls der typische DSE-Zyklus genutzt. In [50] wurde mit diesem Framework eine DSE zur Bestimmung der Parameter eines nicht geclusterten VLIW-Prozessors für Videokomprimierungsalgorithmen durchgeführt. Die ersten Durchläufe des DSE-Zyklus dienen dabei der Bestimmung oberer Schranken einzelner Ressourcen. Bei dieser sogenannten Disjoint-Exploration wird die Verfügbarkeit einer einzelnen Ressource – z. B. die Anzahl der FUs oder Multiplizierer – kontinuierlich erhöht, bis bei der Simulation die Ausführungsgeschwindigkeit nicht mehr gesteigert werden kann. Alle anderen Ressourcen stehen während dieser Phase in unbeschränkter Anzahl zu Verfügung. Durch dieses Vorgehen sind aber noch keine Aussagen über die Auswirkungen bestimmter Ressourcenkombinationen

auf die Ausführungsgeschwindigkeit möglich. Die durch die Disjoint-Exploration ermittelten oberen Schranken werden deshalb verwendet, um den Suchraum während einer Conjoint-Exploration für unterschiedliche Kombinationen ausgewählter Ressourcen zu beschränken. Dieser Teil der Suche kann nur für wenige Ressourcenkombinationen durchgeführt werden. In Fällen, in denen es wenig Abhängigkeiten zwischen den Ressourcen gibt, kann dieses Vorgehen erfolgreich sein. Allerdings wird in [50] auch darauf hingewiesen, dass eine anschauliche Darstellung der Abhängigkeiten für eine Conjoint-Exploration von mehr als drei Ressourcen schwierig ist. CASTLE unterstützt den Entwickler durch eine grafische Oberfläche bei der DSE und die Einbindung der Werkzeuge, wie Compiler und Simulator, in das Framework. Es ist aber immer noch eine starke Interaktion zwischen dem Entwickler und dem CASTLE-System notwendig. Außerdem obliegt es vollständig dem Nutzer aus den zu einer Prozessorarchitektur ermittelten Laufzeitdaten, die Schwachstellen der Architektur zu erkennen, sowie aus der Gesamtheit der Laufzeitdaten, wie beispielsweise mehrere Conjoint-Explorationen, die richtigen Schlüsse für das weitere Vorgehen bei der Suche nach geeigneten Architekturparametern zu ziehen.

### 3.2.3 DSE für den TriMedia64 Prozessor

Für den TriMedia64 Prozessor, ein Nachfolgermodell des TriMedia32 Prozessors, der bereits im Abschnitt 1.2 vorgestellt wurde, ist während der Entwicklungsphase eine DSE durchgeführt worden, um verschiedene Parameter des Prozessorkerns so zu fixieren, dass er für ein ganzes Spektrum von Signalverarbeitungsaufgaben geeignet ist [26, 30, 45]. Bei dieser DSE war es aber nicht das Ziel, eine feste Zeitschranke bei der Ausführung der Anwendungen einzuhalten, sondern einen guten Kompromiss zwischen Hardwarekosten und Ausführungsgeschwindigkeit zu finden.

Der TriMedia64 Prozessor ist als VLIW-Prozessor mit fünf parallel arbeitenden funktionalen Einheiten konzipiert. Er verfügt über eine zentrale Registerbank mit 128 64-Bit-Registern, zehn vollwertigen Leseports, fünf Schreibports, fünf 1-Bit-Leseports zur Ausführung von bedingten Operationen[1] sowie einen vollständigen Bypass. Eine Besonderheit stellt die Möglichkeit dar, die Ports zweier funktionaler Einheiten gemeinsam für eine komplexe Operation[2] zu nutzen, die dann bis zu vier Quellregister und zwei Zielregister verwenden kann. Da bereits feststand, dass der Prozessor nicht geclustert ist und fünf funktionale Einheiten besitzt, bestand die Aufgabe im Finden geeigneter FU-Typen. Es war also zu ermitteln, welche Operatoren in welchen funktionalen Einheiten bereitgestellt werden sollen. Die dafür zur

[1] Das sind Operationen, deren Ausführung durch ein Guard-Bit unterdrückt werden kann.

[2] sogenannte *super-ops*

Verfügung stehenden 30 Operationstypen waren bereits fixiert. Für die DSE wurde eine Oberfläche verwendet, die dem Nutzer die Verwaltung der Werkzeuge erleichtern soll. Die exakte Prozessorarchitektur wurde in einer proprietären Prozessorbeschreibungssprache definiert. Um die Simulationszeit zu reduzieren, wurden zu Beginn der DSE einmalig die Ausführungshäufigkeiten einzelner Basisblöcke ermittelt. Auf den einzelnen Architekturen wurden die Benchmarkprogramme nicht erneut simuliert, sondern die benötigten Taktzyklen basierend auf den ermittelten Ausführungshäufigkeiten und dem ermittelten Ablaufplan berechnet. Die Simulationszeit für eine spezifische Prozessorarchitektur konnte so von vier Stunden auf vier Minuten reduziert werden. Die eigentliche DSE fand in drei Phasen statt. In der ersten Phase wurde für jeden Operationstypen

- sein Anteil an der Gesamtausführungszeit der Benchmarkprogramme und
- eine untere und obere Schranke

ermittelt. Zur Ermittlung der oberen und unteren Schranken wurde dieselbe Idee wie in CASTLE für die Disjoint-Exploration genutzt. Da nur 30% der Operationstypen einen Anteil von 93% an der Gesamtlaufzeit der Benchmarkprogramme haben, wurde in der zweiten Phase für diese Operationstypen eine vollständige Suche – analog zur Conjoint-Analyse in CASTLE – unter Einbeziehung der oberen und unteren Schranken durchgeführt. Das bedeutet, dass jede mögliche Anordnung der Operatoren in den FUs innerhalb dieser Schranken untersucht wurde, wofür etwa 3000 Architekturen betrachtet werden mussten. Unter all diesen Architekturen wurden sogenannte pareto-optimale-Architekturen als Ausgangspunkt in die dritte Phase übernommen. Eine Architektur ist eine pareto-optimale-Architektur, wenn es keine andere Architektur mit weniger oder gleichem Ressourcenbedarf gibt, die die Benchmarkprogramme schneller ausführen kann [32]. In der dritten Phase wurden die übrigen 70% der Operationstypen den fünf vorhandenen funktionalen Einheiten zugeordnet. Insgesamt wurden in dieser Phase noch einmal ca. 2500 Architekturen untersucht. Für die somit knapp 6000 betrachteten Architekturen ergab sich eine Zeit von ca. 400 Stunden (fast 17 Tage) zur Generierung der benötigten Werkzeuge und Berechnung der Ausführungsgeschwindigkeit der Benchmarkprogramme. Das ist ein hoher Zeitaufwand, da bei dieser Design-Space-Exploration die Anzahl der funktionalen Einheiten bereits feststand und keine geclusterte Architektur betrachtet wurde. Unter Einbeziehung dieser zusätzlichen Parameter in die DSE ist es sehr wahrscheinlich, dass dieses Vorgehen an seine Grenzen stoßen wird.

### 3.2.4 PICO

Das PICO[1]-Projekt von Hewlett-Packard [35, 90] beschäftigt sich mit der DSE einer ungeclusterten VLIW-Architektur. Dieser Ansatz arbeitet, im Gegensatz zu den bisher vorgestellten Ansätzen, vollständig automatisch und bestimmt trotzdem ein wesentlich größeres Spektrum an Parametern. So wird neben der Betrachtung von Anzahl und Typ der funktionalen Einheiten eines Prozessors auch die Anzahl der Register, eine Cache-Architektur und ein systolisches Array zur Realisierung von Schleifen in Hardware mit in die DSE einbezogen. Als weitere Architekturoptionen werden bedingte Operationen und spekulative Ausführung von Operationen betrachtet. Die Vorgehensweise während der DSE des VLIW-Prozessors entspricht in etwa dem DSE-Zyklus aus Abbildung 3.2. Zum Ausdruck kommt das durch die zyklische Generierung neuer Architekturen, für die dann die betreffende Anwendung immer wieder übersetzt und simuliert wird.

Vor Beginn des eigentlichen DSE-Zyklus wird zwischen VLIW-Architekturen unterschieden, die spekulative Ausführung unterstützen bzw. nicht unterstützen und Architekturen, die bedingte Operationen unterstützen bzw. nicht unterstützen. Für jede der vier möglichen Parameterkombinationen wird die Anwendung mit dem IMPACT Compiler [24] in hoch parallelen assemblerähnlichen Code übersetzt. Somit existiert für jede Parameterausprägung eine Codebasis, die als Grundlage für die folgende DSE genutzt und mit dem Elcor-Compiler in den endgültigen Zielcode übersetzt wird. Hier soll nur die DSE für die VLIW-Parameter betrachtet werden, die durch den sogenannten Spacewalker [35] realisiert wird. Es werden verschiedene Suchstrategien eingesetzt, die alle auf einer Kosten/Laufzeit Abschätzung basieren. Die einfachste Suchstrategie beginnt mit einer Kandidatenmenge, die anfangs nur die kostengünstigste VLIW-Architektur enthält. In jeder Iteration des DSE-Zyklus wird aus der Kandidatenmenge die kostengünstigste Architektur ausgewählt und geprüft, ob es sich um eine pareto-optimale-Architektur handelt. Dazu wird die Anwendung auf dieser Architektur simuliert. Wenn unter den bis dahin betrachteten Architekturen keine dabei war, die eine höhere Ausführungsgeschwindigkeit aufwies, dann handelt es sich um eine pareto-optimale-Architektur. Nur von solchen Architekturen werden neue Architekturen abgeleitet, die dann für den nächsten DSE-Zyklus in die Kandidatenmenge aufgenommen werden. Das Erzeugen neuer Architekturen geschieht durch eine Erhöhung der vorhandenen Ressourcen in einer pareto-optimalen Architektur. So hat eine neue Architektur, die in die Kandidatenmenge aufgenommen wird, beispielsweise eine zusätzliche FU oder weitere Register. Ein Schwachpunkt dieses Verfahrens ist die Art, wie die durch die zusätzliche FU bereitzustellenden Operatoren ausgewählt werden. Jeder Operator, dessen

[1] Program In Computer Out

Auslastung in der pareto-optimalen Architektur, relativ zu dem am stärksten ausgelasteten Operator, einen Schwellwert übersteigt, wird in die neue FU aufgenommen. Es wird hierbei nicht beachtet, ob Operationen dieses Typs parallel abgearbeitet werden können oder ob sie aufgrund von Abhängigkeiten sequentiell abgearbeitet werden müssen.

Abhängig von der gewählten Suchstrategie mussten unterschiedlich viele Architekturen simuliert werden. Als Benchmarkprogramme wurden zwei sehr kleine Anwendungen verwendet. Das war zum einen die *strcpy*-Funktion, die den Inhalt eines Zeichenfeldes in ein neues Feld kopiert, und zum anderen ein FIR-Filter[1]. In beiden Fällen waren mehrere Tage für die DSE notwendig, bei der, abhängig von der gewählten Suchstrategie, bis zu 1000 Architekturen untersucht werden mussten.

### 3.2.5 DSE nach Lapinskii

Das Ziel des DSE-Ansatzes in der Arbeit von Lapinskii [99] ist die Bestimmung einiger wichtiger Parameter eines geclusterten VLIW-Prozessors während einer frühen Phase der DSE, wie Clusteranzahl, FU-Anzahl in jedem Cluster und Kapazität des Clusterverbindungsnetzwerkes. Es wird ebenfalls noch der in Abbildung 3.2 angegebenen DSE-Zyklus zur Bestimmung dieser Parameter genutzt. Allerdings wird kein Compiler und Simulator mehr generiert. Stattdessen stellt das DSE-Werkzeug selbst einen List-Scheduling- und einen Clusterungsalgorithmus zur Verfügung, die beide ressourcenbeschränkt arbeiten und in einem Compiler verwendet werden können. Der Clusterungsalgorithmus wird vor der Planung ausgeführt und ordnet die Operationen der Anwendung den Clustern des VLIW-Prozessors zu. Bei der anschließenden Planung werden die Operationsknoten den Instruktionen und funktionalen Einheiten zugeordnet. Eine Simulation ist nicht notwendig, da als Kriterium für die Ausführungsgeschwindigkeit die Länge der so erzeugten Ablaufpläne verwendet wird.

Hervorzuheben ist bei diesem Ansatz die Heuristik, die zur Generierung der nächsten zu betrachtenden Architektur verwendet wird. Diese nutzt eine Beobachtung, nach der sich eine optimale Architektur mit $n + 1$ Clustern von einer optimalen Architektur mit $n$ Clustern nur in dem zusätzlichen Cluster unterscheidet. Alle anderen Cluster behalten ihre Konfiguration, d. h. Anzahl und Typ der funktionalen Einheiten, bei. Optimal bedeutet in diesem Zusammenhang, dass es keine Architektur mit derselben Clusteranzahl und maximalen FU-Anzahl in den Clustern gibt, die die Basisblöcke schneller abarbeiten kann. Diese Beobachtung wird verwendet, wenn bei der DSE der Architektur ein neuer Cluster hinzugefügt wird.

[1] Finite Impulse Resonanz Filter.

Die automatisierte DSE beginnt mit einer Architekturvariante, die maximal zwei FUs in jedem Cluster und zu Beginn genau einen Cluster besitzt. Eine neue Architektur wird durch das Hinzufügen eines weiteren Clusters erzeugt. Dieser neue Cluster enthält die aktuelle maximal zulässige Anzahl funktionaler Einheiten, wobei eine Hälfte davon Multiplikationen und die andere Hälfte Additionen ausführen kann. Um die beste Konfiguration des neuen Clusters zu bestimmen, wird dieses Verhältnis einmal zugunsten der Addierer und einmal zu Gunsten der Multiplizierer verändert. Für jede dieser Architekturen wird der Clusterungs- und Planungsalgorithmus ausgeführt. Die kostengünstigste Architektur, für die der kürzeste Ablaufplan erzeugt wurde, stellt eine weitere Architekturvariante dar. Diese Architekturvariante wird erneut um einen Cluster erweitert, der wieder die aktuelle maximale Anzahl funktionaler Einheiten enthält. Es werden solange neue Architekturvarianten gebildet, bis durch das Hinzufügen eines weiteren Clusters die Länge der erzeugten Ablaufpläne nicht mehr verringert werden kann. Dann wird die maximale Anzahl FUs, die ein Cluster enthalten darf, um eins erhöht und die DSE bei einer Architekturvariante mit genau einem Cluster und der neuen maximalen Anzahl FUs in diesem Cluster fortgesetzt. Das Verfahren endet, wenn trotz zusätzlicher funktionaler Einheiten keine kürzeren Ablaufpläne erzeugt werden können.

Das Verfahren von Lapinskii betrachtet nur zwei FU-Typen und beschränkt sich auf funktionale Einheiten, die entweder eine Addition oder eine Multiplikation ausführen können. Die Berücksichtigung weiterer FU-Typen gestaltet sich schwierig, da dann die Untersuchung der möglichen Konfigurationen eines Clusters nicht mehr in wenigen Schritten (höchstens die aktuelle maximale FU-Anzahl) abgeschlossen werden kann, sondern exponentiell mit der Anzahl der betrachteten FU-Typen wächst. Für dieses Problem wird in der Arbeit keine Lösung genannt. Durch die strenge Trennung zwischen den FU-Typen können Operationen mit unterschiedlichem Typ keine Ports in der Registerbank gemeinsam nutzen. Es wird argumentiert, dass diese Optimierung in einer späteren Phase der DSE durchgeführt werden sollte. Allerdings hat eine entsprechende Optimierung starke Auswirkungen auf die Ablaufpläne, so dass die während der früheren DSE-Phase ermittelten Parameter der Architektur ihre Gültigkeit verlieren können. Das gilt insbesondere dann, wenn es das Ziel ist, die maximale FU-Anzahl in jedem Cluster klein zu halten, damit die Registerbänke wenig Ports besitzen.

### 3.2.6 MOVE

In [40, 43] wird mit dem MOVE-Framework ein DSE-Ansatz für TTAs vorgestellt, der ebenfalls den DSE-Zyklus verwendet. Wie in Abschnitt 2.1.1 bereits erläutert, sind bei der TTA die Registerbänke mit den FUs über ein Netzwerk verbunden, das nicht allen FUs gleichzeitig Zugriff auf die Register

ermöglicht. Das Ergebnis einer Berechnung wird nicht implizit in der Registerbank abgelegt, sondern es muss explizit im Programmcode angegeben werden, wie die Werte durch das Verbindungsnetzwerk weiterzuleiten sind. Durch diesen explizit programmierbaren Bybass können die Werte auch direkt an andere FUs weitergereicht werden. Die Komplexität der Registerbank und des Verbindungsnetzwerkes kann somit an die Anforderungen einer Anwendung angepasst werden. Der in [40] beschriebene DSE-Ansatz für solche Architekturen ähnelt dem Vorgehen im PICO-Projekt, startet aber mit einer Architektur, die mehr FUs, Register und Verbindungen bereitstellt als benötigt werden. Eine *Nachbararchitektur* wird ermittelt, indem die Verfügbarkeit jeder Ressource einmal verringert und die Anwendung erneut übersetzt wird. Mit der Architektur, deren Performance im Verhältnis zu den Hardwarekosten am größten ist, wird die DSE fortgesetzt, bis die Anzahl der Ressourcen so gering geworden ist, dass die Anwendung nicht mehr in der vorgegebenen Zeit ausgeführt werden kann. Anschließend werden nach demselben Schema die verfügbaren Ressourcen wieder erhöht. Diese sogenannten *Sweeps*, die fünfmal wiederholt werden, sollen zu pareto-optimalen-Architektur führen, aus denen der Entwickler am Ende der DSE auswählen kann. Da ebenfalls das Verbindungsnetzwerk und die Verbindungspunkte zwischen den Registerbänken und den FUs optimiert werden, sind die erzeugten TTAs sehr stark an die Anwendung angepasst und die Architektur kann einen großen Teil an Flexibilität verlieren.

### 3.2.7 Die Lx-Plattform

Mit der Lx-Plattform [73] von Hewlett-Packard und STMicroelectronics wird ein skalierbarer und anpassbarer, statisch geplanter und geclusterter VLIW-Prozessor vorgestellt. Allerdings dürfen die Parameter der Architektur nur in einem gewissen Rahmen variiert werden. So kann die Lx-Architektur bis zu vier Cluster besitzen, wobei jeder Cluster aus maximal vier funktionalen Einheiten besteht. Es kann maximal ein Speicherzugriff gleichzeitig pro Cluster erfolgen. Zwei funktionale Einheiten können auch Gleitkommaoperationen verarbeiten. Kopieroperationen zwischen den Clustern müssen vom Compiler statisch geplant werden. Jeder Cluster verfügt über eine lokale Registerbank mit 64 32-Bit-Registern. Jede Registerbank stellt acht Lese- und vier Schreibports für 32-Bit-Werte zur Verfügung und außerdem acht 1-Bit-Leseports zur Verarbeitung bedingter Operationen. Innerhalb eines Clusters ist ein vollständiger Bypass vorhanden. Wie beim TriMedia-Prozessor auch, können die Ports zweier funktionaler Einheiten zusammengefasst werden, um komplexere Operationen mit bis zu vier Eingangs- und zwei Ausgangswerten zu realisieren. Als Compiler für diese Architektur wird der Multiflow-Compiler [72] verwendet, um bei vorgegebenen Architekturparametern ausführbaren Code zu erzeugen. In [73] wurde so die Leistungsfähigkeit unterschiedlicher Lx-Varianten für ein gegebenes Spekturm an Benchmarkprogrammen untersucht, indem die Benchmarkprogramme für

verschiedene Architekturvarianten übersetzt wurden. Für die DSE wurde somit ebenfalls der in Abbildung 3.2 dargestellte DSE-Zyklus angewendet.

### 3.2.8 AutoTIE

In [23] wird ein Ansatz vorgestellt, der es erlaubt, den rekonfigurierbaren Xtensa-Prozessor von Tensilica [84] mittels eines Compilers durch die Übersetzung einer Anwendung zu konfigurieren. Der Prozessor wird dabei um Registerbänke und Operatoren erweitert. Besonderer Wert wurde auf die Extraktion von Vektoroperationen und sogenannte *verschmolzene Operationen* gelegt. Das sind Operationen, die aus elementaren Basisoperationen zusammengesetzt sind, wie beispielsweise eine MAC-Operation, die aus einer Multiplikation und einer Addition besteht. Das Prinzip des Ansatzes, die Parameter der Zielarchitektur durch eine einmalige Übersetzung der Anwendung zu bestimmen, ist ähnlich zu dem bereits vorgestellten DESCOMP-Prinzip. Die Erzeugung einer geeigneten Architektur beruht auf der Betrachtung der Operationen in Schleifenkörpern, um daraus für jeden Schleifenkörper separat Rückschlüsse auf die erforderlichen Operatoren und deren Zuordnung zu funktionalen Einheiten zu ziehen. Eine geeignete Anzahl der Operatoren wird allerdings durch eine sehr einfache Abschätzung berechnet, die nur das zahlenmäßige Verhältnis der Operationen im Schleifenkörper zu vorhandenen Operatoren desselben Typs berücksichtigt. Für die Zuordnung der Operatoren zu funktionalen Einheiten wird die erwartete Ablaufplanlänge ebenfalls ohne Berücksichtigung der Abhängigkeiten zwischen den Operationen ermittelt. Weiterhin ist eine obere Grenze für die vorhandene FU-Anzahl vorgegeben, um die automatische Erzeugung von Datenpfaden zu ermöglichen. Aus den so erzeugten Datenpfaden für einzelne Schleifenkörper werden Datenpfade generiert, die alle Schleifenkörper der Anwendung ausführen können. Der Anwender hat die Möglichkeit einen dieser Datenpfade auszuwählen, um so den Prozessor und Compiler zu konfigurieren. Der DESCOMP-Ansatz stellt insbesondere für die Erzeugung der Datenpfade präzisere Methoden zur Verfügung, weil auch Abhängigkeiten zwischen den Operationen beachtet werden.

### 3.2.9 DSE für geclusterte Architekturen

Wenn in den vorgestellten Ansätzen eine DSE für geclusterte Architekturen durchgeführt wurde [73, 99], dann sind die Anwendungen immer für eine spezifizierte Architektur übersetzt worden. Es können somit ressourcenbeschränkte Clusterungsalgorithmen, wie sie auch in Compilern Anwendung finden, genutzt werden [4, 28, 34, 49, 72, 83, 93]. Weitere ressourcenbeschränkte Clusterungsverfahren, auf die hier nicht näher eingegangen werden soll, sind im Zusammenhang mit Modulo-Scheduling entwickelt worden [29, 46, 86]. Da diese Clusterungsverfahren ressourcenbeschränkt sind,

kann keines davon die Einhaltung einer zuvor festgelegten Ablaufplanlänge garantieren.

Die Clusterungsverfahren können danach unterschieden werden, ob die Clusterung vor der Planung, nach der Planung oder mit der Planung gekoppelt durchgeführt wird. Eine Clusterung nach der Planung bietet nur noch wenig Optimierungsmöglichkeiten. Eine gekoppelte Clusterung und Planung [28] führt zu sehr komplexen Abhängigkeiten während der Optimierung, weswegen beispielsweise in [83] Simulated Annealing zur Lösung des Problems genutzt wird. Die meisten Verfahren führen die Clusterung vor der Planung mittels einer Heuristik durch. In [12] werden diese in *Längste-Pfade-Verfahren* und *hierarchische Verfahren* unterteilt. Längste-Pfade-Verfahren orientieren sich bei der Clusterung stark am längsten Pfad im zu clusternden Basisblock [34, 49, 101] und vermeiden es, auf diesen Pfaden Kopieroperationen einzufügen. Da die Länge des letztlich erzeugten Ablaufplans aber oft über der kritischen Pfadlänge liegt, wäre es unproblematisch, auch in den kritischen Pfad Kopieroperationen einzufügen. Hierarchische Verfahren zerlegen den Basisblock in Gruppen von Operationen, die dann solange zusammengefasst werden, bis die Anzahl der Gruppen der gewünschten Clusteranzahl entspricht [34, 65]. Ein Nachteil dieser Verfahren besteht darin, dass gerade im letzten Schritt des Verfahrens die Zusammenfassung in etwa gleich großer Gruppen zu Clustern sehr unterschiedlicher Größe führen kann. Es sind dann in einer anschließenden Phase Korrekturen notwendig, die die Operationen gleichmäßig auf die Cluster verteilen, wie beispielsweise bei [65].

## 3.3 Zusammenfassung

Sowohl die vorgestellten HLS-Systeme als auch die DSE-Ansätze für VLIW-Prozessoren dienen der Erzeugung anwendungsspezifischer Datenpfade. Sie unterscheiden sich aber in der zu Grunde gelegten Basisarchitektur und den zu optimierenden Parametern dieser Architekturen. In der HLS wird ein ASIC für die gegebene Funktion optimiert. Es werden hauptsächlich die Anzahl der Operatoren und deren Verdrahtung optimiert, indem das Verbindungsnetzwerk speziell der zu realisierenden Funktion angepasst wird. Die Register sind fest den einzelnen Funktionsblöcken zugeordnet [37, 57], wodurch die entstehenden Datenpfade sehr irregulär sind. Eine Clusterung wird nicht berücksichtigt, da die Operatoren direkt mit den Registern verbunden werden, deren Werte sie lesen müssen. Die Optimierungsansätze in der HLS basieren zudem auf der Annahme, dass die Verzögerungen in den Schaltungen durch die Operatoren entstehen [9, 71] und ein wesentlicher Teil des Platzverbrauchs durch Operatoren und Multiplexer verursacht wird [32]. Bei VLIW-Prozessoren führt aber insbesondere die Verdrahtung zu Verzögerungen und einem hohen Platzverbrauch im Prozessor. Weiterhin soll das Verbindungsnetzwerk zwischen FUs und Registerbank vollständig sein,

um den Anwendungsbereich der Architektur nicht zu sehr einzuschränken. Die Optimierung des Verbindungsnetzwerkes wird deshalb durch eine Clusterung, eine Minimierung der vom Prozessor bereitgestellten Parallelität in den Clustern und der Zusammenfassung von Operatoren zu funktionalen Einheiten erreicht, die gemeinsam dieselben Ports der Registerbank nutzen. Weiterhin wird eine Minimierung der bereitgestellten Operatoren und der im gesamten Prozessor verfügbaren Parallelität angestrebt. Das Verbindungsnetzwerk zwischen FUs und Registerbank innerhalb eines Clusters bleibt aber vollständig. Der Einfluss der Registerbank und des Verbindungsnetzwerkes auf die Hardwarekosten, Stromkosten und die maximale Taktfrequenz eines geclusterten VLIW-Prozessors muss berücksichtigt werden, was in dieser Form bei der HLS nicht geschieht [102]. Die Optimierungsziele in der HLS unterscheiden sich somit von den Optimierungszielen bei der DSE von VLIW-Prozessoren [99].

Beim methodischen Vorgehen ähneln viele Ansätze aus der HLS dem Vorgehen bei der DSE für VLIW-Prozessoren. Die Planung und Ressourcenallokation wird mit unterschiedlichen Ressourcenbeschränkungen mehrfach wiederholt bis die Zeitschranken eingehalten werden können [21, 33, 37]. Dadurch sind bei einer automatisierten Synthese sehr viele Architekturvarianten zu untersuchen. Die Ansätze in der HLS, die die Erzeugung eines ASICs als eine Optimierung der Ressourcen bei gegebenen Ablaufplanlängen auffassen, haben das Problem, dass die Aufteilung der insgesamt zur Verfügung stehenden Ausführungszeit auf die einzelnen Basisblöcke ebenfalls mit berechnet werden muss [9, 44, 92]. Die Komplexität dieses Problems wächst mit der Anzahl der Basisblöcke stark an, so dass zur Bewältigung beispielsweise probabilistische Techniken [44, 92] genutzt werden oder die Anzahl der zur Verfügung stehenden Instruktionen durch die kritische Pfadlänge bestimmt wird [9]. Bei der Verwendung probabilistischer Techniken kann eine erneute Optimierung mit denselben Eingabedaten zu einem anderen Ergebnis führen. Durch die Beschränkung auf die kritische Pfadlänge können sehr viele Ressourcen benötigt werden. Dadurch sind viele HLS-Systeme nur zur Optimierung von Schaltungen für eine oder sehr wenige Funktionen geeignet [40].

Bei der DSE wird der Datenpfad erzeugt, indem für verschiedene Parameterkombinationen die Anwendung auf dem Prozessor simuliert wird. Am Ende des Zyklus ist somit zusammen mit der Architektur auch ein Compiler, Assembler und Simulator entstanden. Insbesondere der Compiler liefert bei einer erneuten Übersetzung der unveränderten Benchmarkprogramme denselben Ablaufplan[1], der bereits während der DSE erzeugt wurde. Es ist somit garantiert, dass die während der DSE erreichten Ausführungszeiten auch nach einer erneuten Übersetzung derselben

[1] Ausgenommen es wurden probabilistische Techniken wie beispielsweise Simulated Annealing während des Übersetzungsprozesses eingesetzt [83].

Anwendung eingehalten werden. Aufgrund der Werkzeuge, die während des DSE-Zyklus verwendet werden, lassen sich auch komplexe Anwendungen übersetzen, linken und simulieren. Wegen der langen Übersetzungszeiten einiger optimierender Compiler führt die Verwendung dieser Werkzeuge gleichzeitig zu sehr langen DSE-Phasen, die bis zu mehreren Tagen dauern können. Wird in jedem DSE-Zyklus eine Simulation der Benchmarkprogramme auf der aktuellen Architektur durchgeführt, dann übersteigt diese Simulationszeit die Übersetzungszeit sogar bei weitem. Dieses Problem kann bei statisch geplanten VLIW-Architekturen aber relativ gut durch die Abschätzungen der Latenzzeit basierend auf der Instruktionsanzahl nach der Übersetzung gelöst werden. Bei automatisierten DSE-Verfahren wird dieses Zeitproblem insbesondere dadurch verschärft, dass der DSE-Zyklus sehr oft durchlaufen werden muss und der Suchraum sehr groß wird, wenn für jede FU die in ihr implementierten Operatoren bestimmt werden sollen. Diese Komplexität wird durch die hier betrachteten DSE-Ansätze vermieden, indem sie solche FUs nicht zulassen [99], die Zuordnung der Operatoren zu FUs durch eine getrennte Betrachtung der Operationstypen vereinfachen [30, 74] oder recht einfache Strategien zur Erzeugung der FU-Typen verwenden [23, 35]. Mit zunehmender Integrationsdichte auf den Chips und wachsenden Anforderungen an eingebettete Systeme werden künftig viele Cluster mit vielen FUs realisierbar sein, wodurch der Suchraum noch um ein Vielfaches größer wird. Eine Suche mittels eines automatisierten ressourcenbeschränkten Ansatzes, der den DSE-Zyklus mehrfach durchlaufen muss, kann somit in der Praxis extrem zeitaufwendig oder gar unmöglich werden.

Bei einer manuellen oder einer manuell unterstützten DSE, wie sie auch in den meisten HLS-Systemen erforderlich ist, können aufgrund der Erfahrungen des Entwicklers viele DSE-Zyklen gespart werden, allerdings eröffnen sich dann neue Probleme. Die zunehmende Vielfalt und Komplexität der Algorithmen wird es dem Entwickler künftig erschweren, durch Auswerten von Laufzeitstatistiken die Abhängigkeiten und gegenseitigen Wechselwirkungen richtig einzuschätzen, um daraus die notwendigen Schlüsse zur Modifikation der Ressourcenbeschränkungen zu treffen. Es ist sehr wahrscheinlich, dass der Entwickler die komplexen Zusammenhänge dann kaum noch erkennen kann oder soviel Zeit dafür benötigt, dass sein Vorteil gegenüber einem automatisierten DSE-Werkzeug verloren geht.

Die Probleme der automatisch arbeitenden Ansätze werden durch das in Abschnitt 2.3 vorgestellte Prinzip überwunden. Die Einteilung in eine lokale und globale Optimierungsphase erlaubt es, in der lokalen Optimierungsphase mit Techniken, wie sie in der HLS verwendet werden, optimierte Architekturen für einzelne Basisblöcke zu erzeugen und dabei die Kombination verschiedener Operatoren in den einzelnen funktionalen Einheiten zu berücksichtigen. Durch die globale Optimierungsphase, die Informationen über diese optimierten Architekturen nutzt, wird der DSE-Zyklus mit der mehrfachen Übersetzung der Anwendung vermieden.

# 4 Lokale Optimierung mit DESCOMP

In diesem Abschnitt werden die Techniken beschrieben, mit denen für jeden Basisblock und jede seiner zulässigen Ablaufplanlängen Ablaufpläne mit unterschiedlicher Clusteranzahl berechnet werden. Zunächst wird der Planungsalgorithmus für geclusterte Basisblöcke und danach der Ressourcenallokationsalgorithmus angegeben. In beiden Fällen wird von einer fest vorgegebenen Clusterung $\chi$ ausgegangen. Anschließend wird der Algorithmus zur Berechnung einer Clusterung $\chi$ und die Kopplung des Clusterungsalgorithmus mit der Ablaufplanung beschrieben.

Um für jeden Basisblock eine Menge von Ablaufplänen für jede zulässige Ablaufplanlänge berechnen zu können, sind die Planung und Clusterung zeitbeschränkt. Für einen Basisblock wird bei einer gegebenen Ablaufplanlänge der Ablaufplan $\alpha$ so erzeugt, dass für den zur Ausführung von $\alpha$ erforderlichen Prozessor

- eine maximale Ausführungsgeschwindigkeit gewährleistet ist und
- dabei die Hardwarekosten minimal sind.

Beide Ziele werden bei einer festen Ablaufplanlänge erreicht, indem die Portanzahl in der größten Registerbank sowie die Gesamtanzahl der Ports in den übrigen Registerbänken und die Anzahl der Operatoren im Prozessor minimiert werden. Durch eine minimale Portanzahl in der größten Registerbank wird eine hohe maximal mögliche Taktfrequenz erreicht. Eine größere Portanzahl bei der gegebenen Ablaufplanlänge würde die Ausführungsgeschwindigkeit nicht erhöhen, da diese aufgrund der festen Ablaufplanlänge und maximal möglichen Taktfrequenz festgelegt ist und die maximale Taktfrequenz durch die Portanzahl in der größten Registerbank dominiert wird. Durch diese Optimierungsziele wird auch der Stromverbrauch minimiert, da dieser ebenfalls durch die Größe der Registerbänke dominiert wird. Bei fester Ablaufplanlänge sind für einen Basisblock die Optimierung des Platz- und Stromverbrauchs somit keine sich widersprechenden Optimierungsziele und es kann eine maximale Ausführungsgeschwindigkeit bei gleichzeitig niedrigem Stromverbrauch und niedrigen Hardwarekosten erreicht werden.

## 4.1 Planungsalgorithmus

In der Planungsphase wird für eine Clusterung $\chi$, einen geclusterten Basisblock $b_\chi$ und eine Ablaufplanlänge $l$ ein Ablaufplan $\alpha$ mit $|\alpha| = l$ erzeugt, indem durch den iterativen Planungsalgorithmus in jeder Iteration eine unverplante Operation $v$ im Basisblock ausgewählt und für diese Operation eine Instruktion festgelegt wird, in der mit der Ausführung von $v$ begonnen wird. Die im Datenflussgraphen $b_\chi$ eingefügten Kopieroperationen werden ebenfalls mit verplant. Der Planungsalgorithmus nutzt dafür Techniken des List-Scheduling sowie des Force-Directed-Scheduling. Da die Clusterung $\chi$ während der Planung nicht verändert wird, wird für den geclusterten Basisblock $b_\chi$ im Folgenden nur $b$ geschrieben.

Die Zuordnung einer Operation zu einem Ausführungszeitpunkt kann die möglichen Ausführungszeitpunkte anderer Operationen beeinflussen. Das kann zu hohen Hardwarekosten führen, weil möglicherweise die Planung vieler Operationen in dieselbe Instruktion erzwungen wird. Um solche Abhängigkeiten während der Planung zu berücksichtigen, wird im Folgenden der Intervallgraph eingeführt. Zur Konstruktion des Intervallgraphen muss für jede Operation ihr frühester (*eet*) und spätester (*let*) Ausführungszeitpunkt bei einer gegeben Ablaufplanlänge $l$ bekannt sein. Diese können durch die Berechnung des ASAP- bzw. ALAP-Ablaufplans bestimmt werden. Es ist

$$eet(v) := \begin{cases} 0, \text{ falls } \forall u \in V : (u,v) \notin E \\ \max\left\{eet(u) + lat(type(u)) \middle| \ (u,v) \in E\right\}, \text{ sonst} \end{cases} \tag{3.1}$$

$$let(v) := \begin{cases} l - lat(type(v)), \text{ falls } \forall u \in V : (v,u) \notin E \\ \min\left\{let(u) - lat(type(v)) \middle| \ (v,u) \in E\right\}, \text{ sonst.} \end{cases} \tag{3.2}$$

Die **Mobilität** eines Operationsknotens $v$ wird als

$$mob(v) = let(v) - eet(v)$$

definiert. Wie beim Force-Directed-Scheduling sind die Zeitpunkte, an denen mit der Ausführung einer Operation $v$ begonnen werden kann $[eet(v), let(v)]$. Sie werden als **Ausführungsintervall** bezeichnet. Aus der Definition von *let* ergibt sich

**Folgerung 4.1**
Für alle Pfade $\pi = v_1 \ldots v_n$ im Basisblock $b$ mit $n \in \mathbb{N} - \{0\}$ gilt: $pl(v_1 \ldots v_{n-1}) \leq let(v_n) - let(v_1)$, wobei für $n = 1$ gilt: $pl(\pi) = 0$.

□

**Definition 4.1 (Intervallgraph)**
Ein Intervallgraph $T = (V, S, K)$ zu einer gegebenen Ablaufplanplanlänge $l$ und einem Basisblock $b = (V, E, type)$ besteht aus:

- der Menge der Operationsknoten $V$,
- der Menge der Instruktionsknoten $S = \{0, \ldots, l - 1\}$ und
- der Kantenmenge $K \subseteq V \times S$, wobei
  $(v, i) \in K \Leftrightarrow eet(v) \leq i < let(v) + lat(type(v))$.

□

Die Menge $S$ repräsentiert die Instruktionen im Ablaufplan $\alpha$. Ein Operationsknoten ist mit einem Instruktionsknoten im Intervallgraphen durch eine Kante verbunden, wenn die Operation in der entsprechenden Instruktion ausgeführt werden kann. Mit der Ausführung eines Operationsknotens $v$ kann aber nicht in allen Instruktionen aus $S$, mit denen er adjazent ist, begonnen werden, sondern nur in den Instruktionen $i$, für die gilt:

$$eet(v) \leq i \leq let(v)\,.$$

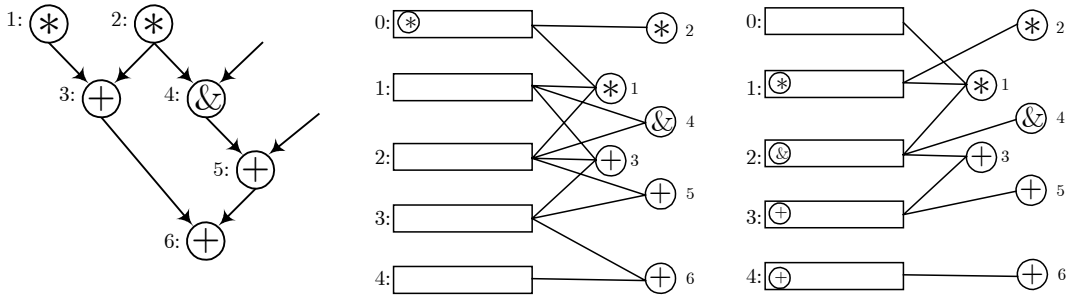
Alle Entscheidungen, die auf der Basis des Intervallgraphen getroffen werden, lassen sich auch mit Hilfe der Funktionen *eet* und *let* treffen. Der Intervallgraph wird dennoch zur besseren Veranschaulichung genutzt, da er für jede Instruktion des Ablaufplans explizit repräsentiert, welche Operationen in dieser Instruktion ausgeführt werden können. Der **Grad** eines Knotens $v \in V \cup S$ wird mit $deg(v)$ bezeichnet und entspricht der Anzahl der Kanten, mit denen $v$ adjazent ist. Ein Knoten $v \in V$ wird als **verplant** bezeichnet, falls gilt:

$$deg(v) = lat(v)\,.$$

Das bedeutet, dass dann der Zeitpunkt des Beginns seiner Ausführung festgelegt ist. Aus den verplanten Knoten in einem Intervallgraphen $T = (V, S, K)$ lässt sich ein **partieller Ablaufplan** $\sigma(T) : \{0, \ldots, l-1\} \rightarrow \wp(V)$ erzeugen, indem:

$$v \in \sigma(T)(i) \Leftrightarrow v \text{ ist verplant und } (v, i) \in S\,.$$

Der partielle Ablaufplan darf somit leere Instruktionen enthalten. In Abbildung 4.1 (b) und (c) sind zwei Beispiele für Intervallgraphen zum Basisblock in (a) bei der Ablaufplanlänge fünf angegeben. Die Instruktionsknoten sind als Rechtecke dargestellt. Bereits verplante Operationen sind zusätzlich in die Instruktionsknoten eingezeichnet. Der Basisblock in (a) hat die kritische Pfadlänge vier. In (b) ist die Operation 2 in die Instruktion 0 geplant worden und in (c) wurde die Operation 2 in Instruktion 1 geplant. Damit wurden gleichzeitig die Ausführungszeitpunkte der Operationen 4, 5 und 6 festgelegt.




(a) (b) (c)

Abbildung 4.1: (a) Basisblock mit kritischer Pfadlänge vier; (b) Intervallgraph zum Basisblock in (a) mit in Instruktion 0 verplanter Operation 2; (c) Intervallgraph mit in Instruktion 1 verplanter Operation 2.

Zuordnungen eines Knotens zu einer Instruktion, die mit hoher Wahrscheinlichkeit zu einem Ablaufplan führen, in dem viele Operationen derselben Instruktion zugeordnet werden müssen, sollen vermieden werden, um die Parallelität im Ablaufplan niedrig zu halten. Um eine entsprechende Bewertung durchzuführen, werden nach der Zuordnung eines Knotens $v$ zur Instruktion $i$ die Ausführungsintervalle der Knoten aktualisiert, die von $v$ abhängig sind bzw. von denen $v$ abhängig ist. In Algorithmus 4.1 ist das Verfahren zur Aktualisierung der spätesten Ausführungszeitpunkte der Knoten angegeben, von denen $v$ abhängig ist. Die Menge $H_k \subseteq V$ enthält Knoten, von denen $v$ abhängig ist und deren spätester Ausführungszeitpunkt nicht größer als $k$ sein darf.

**Algorithmus 4.1 (Aktualisierung von *let*)**

```
Eingabe: Instruktion i und Knoten v,
         Basisblock (V,E,type) und Ablaufplanlänge l
         Bisherige späteste Ausführungszeitpunkte let
Ausgabe: Aktualisierte späteste Ausführungszeitpunkte let

for k:=0 to l-1 do H_k:=∅ od
H_i:={v}
while(i ≥ 0) do
  if(u ∈ H_i und (w,u) ∈ E und w ist nicht verplant ) then
    H_{i-lat(type(w))}:=H_{i-lat(type(w))} ∪ {w}
  fi
  if(u ∈ H_i und let(u) > i) then let(u):=i fi
  i:=i - 1
od
```

Beginnend bei $v$ wird eine Breitensuche im Basisblock $b = (V, E, type)$ entgegen der Kantenrichtung durchgeführt und der späteste Ausführungszeitpunkt eines Knotens $u$, der Vorfahre des Knotens $v$ ist, aktualisiert, wenn die Anzahl der Instruktionen zwischen $let(u)$ und dem für $v$ gewählten Ausführungszeitpunkt $i$ nicht ausreicht, um alle Operationen auf dem längsten Pfad zwischen $u$ und $v$ auszuführen. Die früheste Ausführungszeit ändert sich für die Vorfahren von $v$ nicht. Ein analoges Verfahren muss auch auf die Nachfahren von $v$ angewendet werden, um deren früheste Ausführungszeitpunkte zu korrigieren.

**Lemma 4.1**
Wenn der Knoten $v$ der Instruktion $i$ zugeordnet wurde, dann gilt nach der Korrektur der spätesten Ausführungszeitpunkte mit Algorithmus 4.1 wieder für alle Pfade $v_1 \ldots v_n$ des Basisblocks mit $n \geq 1$: $pl(v_1 \ldots v_{n-1}) \leq let(v_n) - let(v_1)$.

**Beweis**
Es sei $let(u)$ der späteste Ausführungszeitpunkt der Operation $u$ vor der Aktualisierung mit Algorithmus 4.1 und $let'(u)$ der aktualisierte späteste Ausführungszeitpunkt. Wegen Folgerung 4.1 gilt nach der Konstruktion des Intervallgraphen für die zwei Knoten $v_1$ und $v_n$ eines Pfades $v_1 \ldots v_n$ die Behauptung $pl(v_1 \ldots v_{n-1}) \leq let(v_n) - let(v_1)$. Es wird nur noch gezeigt, dass nach der Ausführung von Algorithmus 4.1 $pl(v_1 \ldots v_{n-1}) \leq let'(v_n) - let'(v_1)$ gilt, wenn vor der Ausführung $pl(v_1 \ldots v_{n-1}) \leq let(v_n) - let(v_1)$ gegolten hat. Da $v_1$ ein Vorfahre von $v_n$ ist, sind entweder beide Knoten Vorfahren des Knotens $v$, der der Instruktion $i$ zugeordnet wurde, oder nur $v_1$ ist Vorfahre von $v$ oder keiner der beiden Knoten ist Vorfahre von $v$. Wenn beide Knoten Vorfahre von $v$ sind, dann gilt $pl(v_1 \ldots v_{n-1}) \leq let'(v_n) - let'(v_1)$ wegen der Konstruktion von $let'$ durch Algorithmus 4.1. Ist nur $v_1$ Vorfahre von $v$, dann ist $let'(v_1) \leq let(v_1)$ und $let'(v_n) = let(v_n)$, woraus mit
$pl(v_1 \ldots v_{n-1}) \leq let(v_n) - let(v_1)$ folgt: $pl(v_1 \ldots v_{n-1}) \leq let'(v_n) - let'(v_1)$. Wenn beide Knoten kein Vorfahre von $v$ sind, dann ist $let'(v_1) = let(v_1)$ und $let'(v_n) = let(v_n)$. Damit gilt auch die Behauptung.

□

Die Änderung der Ausführungsintervalle wirkt sich auch auf den Intervallgraphen $T$ aus und bewirkt dort, wegen der veränderten frühesten und spätesten Ausführungszeitpunkte, das Wegfallen von Kanten zwischen Operations- und Instruktionsknoten. Der so durch die Planung von Knoten $v$ in Instruktion $i$ aus dem Intervallgraphen $T$ entstehende Intervallgraph wird mit $T_i^v$ bezeichnet.

**Lemma 4.2**
Nach der Aktualisierung der Ausführungsintervalle mit Algorithmus 4.1 gilt für jeden Knoten $w \in V$: $eet(w) \leq let(w)$, falls $v$ in eine Instruktion $i$ mit $eet(v) \leq i \leq let(v)$ geplant wurde.

**Beweis**
Mit $let(w)$ und $eet(w)$ werden die Ausführungszeitpunkte eines Knotens $w$ vor der Aktualisierung mit Algorithmus 4.1 und mit $let'(w)$ und $eet'(w)$ nach der Aktualisierung bezeichnet. Angenommen es gibt ein $w \in V$, so dass vor der Aktualisierung $eet(w) \leq let(w)$ und nach der Aktualisierung $eet'(w) > let'(w)$ gilt, dann wurde entweder $eet(w)$ oder $let(w)$ aktualisiert. Für den Fall, dass $eet(w)$ aktualisiert wurde, ist $v$ ein Vorfahre von $w$ und deshalb $let(w) = let'(w)$. Weil $eet(w)$ aktualisiert wurde und $eet'(w) > let'(w)$, muss es einen Pfad $v...v_k w$ mit $i + pl(v...v_k) > let(w)$ geben. Wegen $let(v) \geq i$ folgt daraus $let(v) + pl(v...v_k) > let(w)$ und somit $pl(v...v_k) > let(w) - let(v)$, was ein Widerspruch zu Lemma 4.1 ist. Der Fall, dass *let* aktualisiert wird, kann analog bewiesen werden.

□

Da durch die Aktualisierung der Ausführungsintervalle die späteste Ausführungszeit nur verkleinert, die früheste Ausführungszeit nur vergrößert und für jeden Knoten immer Lemma 4.2 gilt, ergibt sich:

**Folgerung 4.2**
Jede Operation kann innerhalb ihres durch die Formeln (3.1) und (3.2) berechneten Ausführungsintervalls geplant werden.

□

**Folgerung 4.3**
Alle Operationen können innerhalb der Ablaufplanlänge $l$ verplant werden.

□

Der Planungsalgorithmus zur Erzeugung eines Ablaufplans $\alpha$ ist in Algorithmus 4.2 angegeben. Wie bei einem List-Scheduling-Algorithmus wird durch eine **Prioritätsfunktion** eine Prioritätsliste erzeugt, die die Reihenfolge festlegt, in der versucht wird, die Knoten zu verplanen. Durch eine **Zielfunktion** wird für den nächsten zu verplanenden Knoten in der Prioritätsliste eine Instruktion bestimmt, in der mit der Ausführung dieses Knotens begonnen wird. Der ausgewählte Knoten darf unverplante Vorgänger haben, muss aber in eine zulässige Instruktion geplant werden. Eine Instruktion $i$ wird als **zulässig** bezeichnet, wenn die Planung von $v$ in die Instruktion $i$ eine Anordnung der verbleibenden unverplanten Operationen erlaubt, so dass in dem entstehenden Ablaufplan jede Instruktion eine Operation ausführt.

**Algorithmus 4.2 (Planungsalgorithmus)**

**Eingabe:** Basisblock $b$
Ablaufplanlänge $l$
**Ausgabe:** Ablaufplan für $b$

Konstruiere den Intervallgraphen $T$ zum gegebenen Basisblock $b$
**while**(es gibt noch unverplante Knoten in $b$) **do**
  Erzeuge die Prioritätsliste $v_1,\ldots,v_n$
  Bestimme das kleinste $k \in \{1,\ldots,n\}$, so dass es für $v_k$ eine zulässige Instruktion gibt
  Wähle mittels Zielfunktion eine zulässige Instruktion $i \in [eet(v_k), let(v_k)]$, in der mit der Ausführung von $v_k$ begonnen wird und plane $v_k$ in Instruktion i;
  Aktualisiere $eet$, $let$
  Aktualisiere $T$ durch $T := T_i^v$
**od**
**return** $\sigma(T)$

Nachdem ein Knoten verplant wurde, wird die Prioritätsliste unter Berücksichtigung der aktualisierten Ausführungsintervalle neu berechnet. Die Notwendigkeit der Prioritätsliste ergibt sich aus dem Ausschluss von nicht zulässigen Instruktionen, wodurch es vorkommen kann, dass für einen Knoten in der Prioritätsliste keine zulässige Instruktion gefunden wird, obwohl diese existiert. In Satz 4.3 wird noch gezeigt, dass trotzdem alle Operationen in zulässige Instruktionen verplant werden können, wenn solche Operationen, für die in der aktuellen Iteration von Algorithmus 4.2 keine zulässige Instruktion gefunden wird, später verplant werden. Dadurch ist abgesichert, dass alle Instruktionen des Ablaufplans Operationen ausführen und der erzeugte Ablaufplan genau die Länge $l$ hat. Die verwendete Zielfunktion berücksichtigt bei der Berechnung des Zielwertes die Auswirkungen, die die Planung eines Knotens auf die Ausführungsintervalle aller seiner Vor- bzw. Nachfahren hat. Es wurden zahlreiche Varianten für die Prioritäts- und Zielfunktion getestet, auf die hier aber nicht weiter eingegangen werden soll.

### 4.1.1 Prioritätsfunktion

Die Prioritätsfunktion soll die Zielfunktion beim Zuordnen einer Operation zu einer Instruktion unterstützen. Die unverplanten Knoten werden in der erzeugten Prioritätsliste so angeordnet, dass Operationen mit geringer Mobilität vor Knoten mit hoher Mobilität verplant werden, um bei deren Planung die vorhandenen Freiheitsgrade noch zu nutzen. Gibt es mehrere Operationen mit derselben Mobilität, dann werden diese Operationen so angeordnet, dass Operationen mit hohen Hardwarekosten vor Operationen mit niedrigen Hardwarekosten verplant werden. Dadurch soll erreicht werden,

dass möglichst wenig teure Operationen desselben Typs in dieselbe Instruktion geplant werden müssen. Die Auswahl der nächsten zu verplanenden Operation geschieht unabhängig von dem Cluster, dem die Operation zugeordnet ist.

### 4.1.2 Test auf Zulässigkeit

Für einen zu verplanenden Knoten $v$ muss zunächst festgestellt werden, welche Instruktionen $i$ im Ausführungsintervall $[eet(v), let(v)]$ zulässig sind. Um das zu prüfen, muss in jede Instruktion, die nach der Planung von $v$ in Instruktion $i$ noch keine Operation ausführt, ein unverplanter Operationsknoten angeordnet werden können. Für den Intervallgraphen $T_i^v$, in dem $v$ probeweise in Instruktion $i$ geplant wurde, wird diese Überprüfung durch Algorithmus 4.3 vorgenommen.

Es ist $U \subseteq V$ eine Menge unverplanter Knoten in $T_i^v$, die durch $t : U \rightarrow \{0, \ldots, l-1\}$ einer noch leeren Instruktion zugeordnet werden, in der mit der Ausführung dieser Operation begonnen wird. Die durch $t$ festgelegte Zuordnung ist nur temporär, um zu prüfen, dass für jede leere Instruktion ein solcher Knoten gefunden werden kann.

**Algorithmus 4.3 (Test auf leere Instruktionen)**

1. $U := \emptyset$; $t := \emptyset$
2. Finde Instruktion $j$ mit $\sigma(T_i^v)(j) = \emptyset$, so dass
   $\forall z : 0 \leq z < j \Rightarrow (\sigma(T_i^v)(z) \neq \emptyset \vee \exists u \in U : t(u) \leq z < t(u) + lat(u))$
3. Falls kein Knoten $w \in V - U$ mit $(w, j) \in K$ existiert, dann Stopp mit Fehler.
4. Ansonsten wähle einen Knoten $w \in V - U$ mit $(w, j) \in K$ und $let(w)$ ist minimal.
5. $U := U \cup \{w\}$ und $t(w) := \min\{j, let(w)\}$
6. Falls es noch leere Instruktionen gibt, denen temporär keine Operation zugeordnet wurde, mache mit Schritt 2 weiter, sonst stoppe mit Erfolg.

Algorithmus 4.3 ermittelt in Schritt 2 die kleinste Instruktion, in der weder eine Operation im partiellen Ablaufplan $\sigma(T_i^v)$ ausgeführt wird noch eine der zu $U$ gehörenden Operationen, deren Ausführungsbeginn durch $t$ festgelegt ist. In diese Instruktion wird der Knoten $w$ temporär verplant und in die Menge $U$ aufgenommen, dessen späteste Ausführungszeit minimal ist. Während der Abarbeitung von Algorithmus 4.3 werden die Ausführungsintervalle der Knoten in $T_i^v$ nicht aktualisiert. Es gilt:

**Satz 4.1**
Wenn Algorithmus 4.3 für einen Intervallgraphen $T_i^v = (V, S, K)$ mit Erfolg stoppt, dann erlaubt die Zuordnung von $v$ zu $i$ einen Ablaufplan zu finden, in dem alle Instruktionen mindestens eine Operation ausführen und alle Operationen innerhalb ihrer Ausführungsintervalle verplant sind.

**Beweis**
$U$ ist die in Algorithmus 4.3 erzeugte Menge der temporär verplanten Knoten. Es wird gezeigt, dass die Anordnung der Knoten in $U$ so auch hätte gewählt werden können, wenn die Ausführungsintervalle nach jedem temporär verplanten Knoten aktualisiert worden wären. Damit ist jeder Knoten aus $U$ immer in seinem aktualisierten Ausführungsintervall geplant worden und zusammen mit Folgerung 4.2 folgt daraus, dass auch alle unverplanten Knoten in $V - U$ noch verplant werden können. Die Knoten $u_1, \ldots, u_n$ in der Menge $U = \{u_1, \ldots, u_n\}$ seien in dieser Reihenfolge durch Algorithmus 4.3 temporär verplant worden. Es ist klar, dass $u_1$ in seinem aktualisierten Ausführungsintervall geplant wurde. Wurde ein Knoten $u_j$ durch Algorithmus 4.3 geplant, gilt für alle Zeitpunkte $t(u_m)$ mit $1 \leq j < m \leq n$ und jeden Knoten $w \neq u_j$ mit $(w, t(u_m)) \in K : let(u_j) \leq let(w)$, da sonst $w$ durch Algorithmus 4.3 geplant worden wäre. Damit ist $w$ entweder unabhängig von $u_j$ oder $w$ ist ein Nachfahre von $u_j$. Wurde ein $u_j$ temporär geplant, dann würde die Aktualisierung der Ausführungsintervalle durch Algorithmus 4.1 für alle Nachfahren $v'$ von $u_j$ nur $eet(v')$ aktualisieren und für die Vorfahren $v$ von $u_j$ nur $let(v)$. Damit ändert sich für kein $w$ der Wert $let(w)$, was das Auswahlkriterium für den nächsten temporär zu verplanenden Knoten in Algorithmus 4.3 ist. Damit könnte Algorithmus 4.3 auch nach der Aktualisierung der Ausführungsintervalle zur temporären Planung in $t(u_{j+1})$ wieder den Knoten $u_{j+1}$ wählen, wenn $u_{j+1}$ auch nach der Aktualisierung der Ausführungsintervalle noch der Instruktion $t(u_{j+1})$ zugeordnet werden kann, d. h., für den aktualisierten frühesten Ausführungszeitpunkt muss gelten $eet(u_{j+1}) \leq t(u_{j+1})$. Das ist aber sichergestellt, da es sonst einen Pfad $u_j \ldots v\, u_{j+1}$ geben müsste und wegen $let(v) < let(u_{j+1})$ durch Algorithmus 4.3 nicht $u_{j+1}$ in die Instruktion $t(u_{j+1})$ geplant worden wäre.

□

**Satz 4.2**
Für den Fall, dass nur Operationen mit einer Latenzzeit von 1 im Basisblock enthalten sind, stoppt Algorithmus 4.3 genau dann erfolgreich, wenn zu $T_i^v$ ein Ablaufplan existiert, in dem es keine leeren Instruktionen gibt.

**Beweis**
Es wird nur noch gezeigt: Wenn ein Ablaufplan ohne leere Instruktionen existiert, dann stoppt Algorithmus 4.3 erfolgreich. Angenommen für $T_i^v$ existiert ein Ablaufplan $\alpha$ ohne leere Instruktionen und Algorithmus 4.3

stoppt mit Fehler bei der Instruktion $z$, d. h., in die Instruktion $z$ konnte durch Algorithmus 4.3 keine Operation mehr verplant werden. Die Menge

$$W := \left\{ w \middle| \ (w,z) \in K \wedge w \text{ ist in } T_i^v \text{ nicht verplant} \right\}$$

soll alle Knoten enthalten, die beim Start von Algorithmus 4.3 potentiell in Instruktion $z$ hätten ausgeführt werden können. Da in $z$ keine Operation aus $W$ geplant werden kann, gilt $W \subseteq U$ und alle $w \in W$ wurden durch Algorithmus 4.3 in Instruktionen $z' < z$ verplant, wobei $U$ wieder die Menge der temporär verplanten Knoten in Algorithmus 4.3 ist. Weiterhin sei

$$m := \min \left\{ t(w) | w \in W \right\}$$

die kleinste Instruktion, in die eine Operation aus $W$ verplant wurde und $N$ die Menge der unverplanten Knoten in $T_i^v$ zu Beginn von Algorithmus 4.3. Für die Instruktion $m$ gilt, dass es keinen Knoten $w' \in N - W$ mit $(w', m) \in K$ gibt. Gäbe es einen solchen Knoten $w'$, dann wäre $w' \notin W$ und somit $let(w') < z$ und damit wäre der Instruktion $m$ dieser Knoten durch Algorithmus 4.3 zugeordnet worden. Also kann es einen solchen Knoten $w'$, der der Instruktion $m$ nicht zugeordnet wurde, aber mit $m$ und nicht mit $z$ adjazent ist, nicht geben.

Es kann aber Knoten $w''$, die mit einer Instruktion zwischen $m$ und $z$ adjazent sind, also $m < eet(w'') \leq let(w'') < z$, geben. Diese Knoten wurden durch Algorithmus 4.3 vor den Knoten aus $W$ verplant, weil $let(w'') < let(w)$. Die Anzahl der Instruktionen, in die diese Knoten $w''$ verplant wurden, soll $k$ sein. Da Algorithmus 4.3 genau einen Knoten pro leerer Instruktion verplanen muss (die Latenzzeit aller Operationen ist 1), gab es im Intervall $[m, z-1]$ genau $|W| + k$ viele leere Instruktionen und $|W| + k$ viele Knoten, die in diese Instruktionen geplant werden konnten. Im Intervall $[m, z]$ gab es somit $|W| + k + 1$ viele leere Instruktionen aber auch nur $|W| + k$ viele verplanbare Operationen. Somit kann es keinen Ablaufplan ohne leere Instruktionen gegeben haben.

□

Existieren auch Operationen mit einer Latenzzeit größer 1, so liefert Algorithmus 4.3 nur noch ein hinreichendes Kriterium. Abbildung 4.2 gibt ein Beispiel an, in dem zwar eine Zuordnung existiert, bei der alle Instruktionen eine Operation ausführen, Algorithmus 4.3 aber mit einem Fehler stoppt. In den Instruktionen 2 und 4 sollen bereits zwei Additionen angeordnet sein. Die Operation & hat eine Latenzzeit von 1 und die Operation $*$ eine Latenzzeit von 2. Durch Algorithmus 4.3 wird der Instruktion 0 zunächst die Operation & zugeordnet und dann die Multiplikation der Instruktion 1. Für Instruktion 3 kann dann keine weitere Operation gefunden werden. Die Zuordnung der Multiplikation zur Instruktionen 1 und der Operation & zur Instruktion 3 hätte dagegen eine Lösung geliefert, bei der in jede Instruktion eine Operation ausgeführt wird.

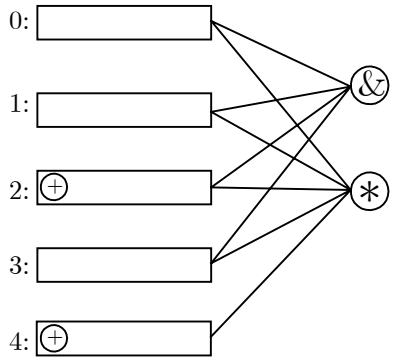



Abbildung 4.2: Beispiel bei dem der Test auf leere Instruktion fehl schlägt.

Die Konsequenz ist, dass die Zuordnung einer Operation $v$ zu einer Instruktion ausgeschlossen wird, obwohl dies nicht nötig wäre. Somit wird ein Teil der möglichen Ablaufpläne nicht vom Planungsalgorithmus berücksichtigt. Auch wenn durch Algorithmus 4.1 einige Zuordnungen verworfen werden, ist dennoch sichergestellt:

**Satz 4.3**
Bei der Planung der unverplanten Knoten durch Algorithmus 4.2 wird immer ein Knoten in der Prioritätsliste gefunden, für den Algorithmus 4.3 nicht mit einem Fehler stoppt.

**Beweis**
Es wird zunächst gezeigt, dass Algorithmus 4.3 mit Erfolg stoppt, wenn noch kein Knoten im Intervallgraphen $T$ verplant wurde. Angenommen Algorithmus 4.3 würde mit einem Fehler stoppen, dann sei $z$ die kleinste Instruktion, in die kein Knoten mehr temporär geplant werden kann und $u$ der Knoten, der temporär in $z-1$ ausgeführt wird. $u$ hat keinen Nachfolger, weil dieser sonst in $z$ geplant werden könnte. Weiterhin muss es einen Knoten $v \in V - U$ geben, für den $let(v)$ maximal ist. Wenn $v$ ein Vorfahre von $u$ ist, dann hätte $v$ vor $u$ geplant werden müssen und da $u$ keinen Nachfahren hat, sind $u$ und $v$ nicht abhängig. Weil $v$ nicht in $z$ geplant werden kann, muss $v$ einen Nachfolger $w$ haben und ist damit nicht der Knoten mit maximaler spätester Ausführungszeit gewesen. Damit ist gezeigt, dass Algorithmus 4.3 eine temporäre Knotenzuordnung findet, wenn in $T$ noch kein Knoten verplant wurde.

Als nächstes wird gezeigt, dass es einen Knoten $v$ und eine Instruktion $i$ gibt, so dass Algorithmus 4.3 für $T_i^v$ mit Erfolg stoppt, wenn bereits für $T$ mit Erfolg gestoppt wurde. Wenn Algorithmus 4.3 für $T$ mit Erfolg stoppte, dann wurde bei diesem Test auf Zulässigkeit die Menge $U$ berechnet. Der Knoten, der als erstes in diese Menge $U$ aufgenommen wurde, wird mit $v$ bezeichnet. Der Ausführungsbeginn von $v$ wurde durch Algorithmus 4.3 in

Instruktion $t(v)$ festgelegt. Spätestens wenn Algorithmus 4.2 diesen Knoten $v$ in die Instruktion $t(v)$ plant, dann stoppt Algorithmus 4.3 mit Erfolg, weil sich in $T^v_{t(v)}$ für alle Knoten $w \in U - \{v\}$ der Wert $let(w)$ nicht geändert hat, denn entweder waren diese Knoten unabhängig von $v$ oder sie waren ein Nachfahre von $v$ und dann wird nur ihr frühester Ausführungszeitpunkt aktualisiert. Damit ordnet Algorithmus 4.3 den leeren Instruktionen in $T^v_{t(v)}$ wieder diese Knoten zu und stoppt dann mit Erfolg.

□

### 4.1.3 Zielfunktion

Ist für den von der Prioritätsliste ausgewählten Knoten $v$ sichergestellt, dass es zulässige Instruktionen gibt, dann wird $v$ probeweise jeder zulässigen Instruktion $i \in [eet(v), let(v)]$ zugeordnet und für den so entstehenden Intervallgraphen $T^v_i$ die Zielfunktion

$$z(\chi, T^v_i) := \alpha \cdot RBLoad(\chi, T, T^v_i) + \beta \cdot TLoad(\chi, T, T^v_i) + \gamma \cdot TDLoad(\chi, T, T^v_i)$$

bei der fest vorgegebenen Clusterung $\chi$ berechnet. Die Zielfunktion berücksichtigt dabei die Auswirkungen, die die Planung von $v$ in Instruktion $i$ auf alle seine Vor- bzw. Nachfahren hat. Beim Force-Directed-Scheduling werden nur unmittelbare Vor- und Nachfolger in die Berechnung des Zielwertes einbezogen. Die dadurch entstehende höhere Laufzeit wird im Zusammenhang mit dem bearbeiteten Optimierungsproblem in Kauf genommen. Die Ausführung von $v$ beginnt dann in der Instruktion $i$, für die die Zielfunktion minimal ist. Die Zielfunktion ordnet jedem Intervallgraphen einen rationalen Zahlenwert zu und wird aus den drei Kostenfunktionen *RBLoad*, *TLoad* und *TDLoad* gebildet:

- *RBLoad* berücksichtigt die Veränderung der Portanzahl in der größten Registerbank eines Prozessors und den Gesamtzuwachs an Ports in allen Registerbänken des Prozessors. Durch *RBLoad* werden somit die Registerbankkosten und die maximal mögliche Taktfrequenz, die von der größten Registerbank abhängt, erfasst. Für die Abschätzung werden alle unverplanten Knoten in $T^v_i$ temporär verplant.
- *TLoad* schätzt die Hardwarekosten ab, die durch die bereitzustellenden Operatoren in den FUs entstehen, wenn im Ablaufplan Operationen desselben Typs gleichzeitig ausgeführt werden müssen. *TLoad* berücksichtigt bei der Vorhersage dieser Hardwarekosten auch unverplante Knoten im Intervallgraphen.
- *TDLoad* präzisiert den durch *TLoad* berechneten Wert, da es nicht immer ausreicht, einen Operator des Typs $t$ in genau $n$ FUs eines Clusters bereitzustellen, auch wenn höchstens $n$ Operationen dieses Typs in jeder Instruktion gleichzeitig ausgeführt werden.

Durch *TLoad* und *TDLoad* sind die Planung und Ressourcenallokation eng miteinander gekoppelt, weil die Auswirkungen von Entscheidungen in der Planungsphase auf die Ressourcenallokation berücksichtigt werden. Die Konstanten $\alpha$, $\beta$ und $\gamma$ dienen der Priorisierung der einzelnen Kostenfunktionen. Wird $\alpha$ groß genug gewählt, so dominiert *RBLoad* immer die Funktionen *TLoad* und *TDLoad*, wodurch die Größe der Registerbänke und damit die verfügbare Parallelität im Prozessor minimiert wird. Wird $\alpha$ kleiner gewählt, dann kann eine Verringerung der Operatorkosten zu Lasten einer erhöhten Portanzahl in den Registerbänken erreicht werden.

### 4.1.4 Minimierung der Ports

Die Funktion *RBLoad*($\chi$, $B$, $T$) wird bei einer gegebenen Clusterung auf zwei Intervallgraphen $B$ und $T$ angewendet, wobei $T$ aus $B$ entstanden ist, indem unverplante Operationen in $B$ verplant wurden. Die Auswirkung dieser Planung auf die Zunahme der Portanzahl in der größten Registerbank und die Zunahme der Registerbankkosten durch zusätzliche Ports in allen Registerbänken soll durch die Funktion *RBLoad* abgeschätzt werden. Um auch die noch nicht verplanten Knoten in diese Betrachtungen einzubeziehen, wird der Greedy-Algorithmus 4.4 (E-Planung) verwendet. Dieser verplant alle unverplanten Knoten in $T$ temporär und minimiert dabei die Anzahl der benötigten Ports im größten Cluster sowie die Kosten durch zusätzliche Ports in den übrigen Registerbänken. Durch die E-Planung wird zu dem Intervallgraphen $T$ ein Intervallgraph $T'$ erzeugt, in dem alle Operationen verplant sind. Mit $T'$ wird der erwartete Zuwachs an Ports durch einen Vergleich mit den in $B$ benötigten Ports abgeschätzt. Die Kombination der Operationstypen innerhalb einer Instruktion wird durch die E-Planung nicht beachtet. Um während der E-Planung für eine Operationen $v$ zu entscheiden, welcher Instruktionen $k$ sie zugeordnet wird, werden zwei Größen lokal betrachtet, d. h. ohne Beachtung der von $v$ abhängigen Knoten:

- Die Anzahl der Ports *fp*, die in Instruktion $k$ noch von Operationen benutzt werden können, ohne dass zusätzliche Ports in einer Registerbank entstehen und
- die Anzahl der Ports *mp*, die im schlechtesten Fall in Instruktion $k$ benötigt werden.

Ist die Operation $v$ dem externen Cluster zugeordnet, dann hat die Wahl einer Instruktion nur Einfluss auf die Anzahl der externen Ports der Registerbänke. Wenn $v$ nicht dem externen Cluster zugeordnet ist, dann hat die Wahl einer Instruktion nur Einfluss auf die internen Ports des Clusters, dem $v$ durch $\chi$ zugeordnet ist. Um *fp* und *mp* für eine Instruktion $k$ zu berechnen, werden zu einem Intervallgraphen $T = (V, S, K)$ und einem Cluster $c$ die zwei Graphen $T^{k,c} = (V, S, K^{k,c})$ und $T_{k,c} = (V, S, K_{k,c})$ durch

$$K^{k,c} = \big\{(v,m)\big|\ (v,m) \in K \wedge v \notin c\big\} \cup$$
$$\big\{(v,m)\big|\ (v,k) \in K \wedge v \in c \wedge s \leq m < s + lat(type(v)) \wedge s = \min(k, let(v))\big\}$$

und

$$K_{k,c} = \big\{(v,m)\big|\ (v,m) \in K \wedge v \notin c\big\} \cup$$
$$\big\{(v,m)\big|\ (v,k) \in K \wedge v \in c \wedge (v,m) \in K \wedge v \text{ ist in } T \text{ verplant}\big\}$$

gebildet. In $T^{k,c}$ sind die Operationen aus $T$, die dem Cluster $c$ zugeordnet und potentiell in der Instruktion $k$ ausführbar sind, in Instruktion $k$ verplant. Alle anderen dem Cluster $c$ zugeordneten Operationen sind mit keinem Instruktionsknoten adjazent. Dadurch kann die in der Registerbank des Cluster $c$ von Operationen, die potentiell in $k$ ausführbar sind, maximal beanspruchte Portanzahl durch

$$mp(k,c,T) := \begin{cases} ePWidth(\sigma(T^{k,c}),\chi), \text{ falls } c = [e]_\chi \\ iPCWidth(\sigma(T^{k,c}),\chi,c), \text{ sonst} \end{cases}$$

berechnet werden. Wenn $c$ der externe Cluster ist, dann wird durch $mp$ die maximale externe Portanzahl in den Registerbänken berechnet, sonst die maximale interne Portanzahl. $T_{k,c}$ enthält genau die Operationen, die bereits in $T$ in der Instruktion $k$ verplant waren. Die Anzahl der freien Ports, die durch weitere Operationen in der Instruktion $k$ im Cluster $c$ noch benutzt werden können, ohne dass dadurch mehr Ports in der Registerbank des Clusters $c$ erforderlich sind, wird durch

$$fp(k,c,T) = \begin{cases} ePWidth(\sigma(T),\chi) - ePWidth(\sigma(T_{k,c}),\chi), \text{ falls } c = [e]_\chi \\ iPCWidth(\sigma(T),\chi,c) - iPCWidth(\sigma(T_{k,c}),\chi,c), \text{ sonst} \end{cases}$$

berechnet. In Abbildung 4.3 ist ein Beispiel angegeben, mit dem das Prinzip verdeutlicht werden soll.

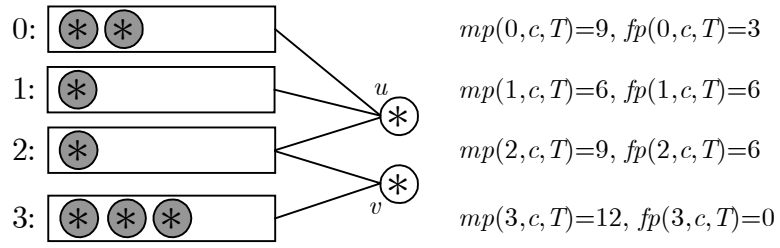



Abbildung 4.3: Berechnung der maximal belegten Ports ($mp$) und freien Ports ($fp$).

Es wird angenommen, dass in dem dargestellten Intervallgraphen $T$ alle Operationen zum Cluster $c$ gehören und jede graue Operation in die Instruktion verplant ist, in die sie eingezeichnet wurde. Die Operationen aus

anderen Clustern sind nicht dargestellt. Wegen Instruktion 3 muss die Registerbank im Cluster $c$ bereits neun Ports enthalten. Die Anzahl der unbenutzten Ports in Instruktion 0 ist deshalb drei. Maximal können in Instruktion 0 neun Ports benötigt werden. Algorithmus 4.4 plant einen Knoten $u$ in diejenige Instruktion, für die das Verhältnis von höchstens erforderlichen Ports und freien Ports am kleinsten ist.

**Algorithmus 4.4 (E-Planung)**

1. Start mit Intervallgraph $T$
2. Ein unverplanter Knoten $u$ aus $T$ mit minimaler Mobilität wird ausgewählt.
3. $u$ wird der Instruktion $k \in [eet(u),\ let(u)]$ zugeordnet, so dass für alle anderen Instruktionen $m \in [eet(u),\ let(u)]$ gilt:
$$\frac{mp(k,[u]_\chi,T)}{fp(k,[u]_\chi,T)+1} \leq \frac{mp(m,[u]_\chi,T)}{fp(m,[u]_\chi,T)+1}$$
4. $T := T_k^u$
5. Gehe zu Schritt 2, falls noch ein unverplanter Knoten in $T$ existiert.

Nach der Planung eines Knotens werden die Ausführungsintervalle der abhängigen Knoten in Schritt 4 aktualisiert. Dadurch ist abgesichert, dass durch die E-Planung die Abhängigkeiten der Knoten beachtet werden und die Entscheidung für die Planung des nächsten Knotens mittels der aktualisierten Ausführungsintervalle getroffen wird.

Es sei $T'$ der Intervallgraph der entsteht, nachdem alle Operationen durch Algorithmus 4.4 verplant wurden. Eine vergrößerte Portanzahl in der größten Registerbank verursacht eine Verringerung der maximal möglichen Taktfrequenz, die durch

$$\Delta f = fm\big(PWidth(\sigma(B),\chi)\big) - fm\big(PWidth(\sigma(T'),\chi)\big) \tag{3.3}$$

erfasst wird. Außerdem bedingt eine erhöhte Portanzahl in den Registerbänken höhere Hardwarekosten, die ebenfalls erfasst werden müssen. Die durch zusätzliche Ports entstandenen Registerbankkosten sind

$$\Delta rb = RBCost\big(\sigma(T'),\chi\big) - RBCost\big(\sigma(B),\chi\big)\,. \tag{3.4}$$

Die Funktion *RBLoad* fasst nun die Veränderung der Taktfrequenz $\Delta f$ und die zusätzlich entstandenen Hardwarekosten in den Registerbänken $\Delta rb$ zusammen:

$$RBLoad(\chi,B,T) = \alpha_1 \cdot \Delta f + \alpha_2 \cdot \Delta rb\,.$$

Durch die Konstanten $\alpha_1$ und $\alpha_2$ können beide Größen gewichtet werden. Durch die Priorisierung von $\Delta f$ wird eine Architektur erzeugt, in der die

Portanzahl in der größten Registerbank minimiert ist. Dem untergeordnet sind dann die zusätzlich entstandenen Hardwarekosten in den übrigen Registerbänken, wodurch eine maximale Ausführungsgeschwindigkeit des Basisblocks erreicht wird.

### 4.1.5 Minimierung der Operationstypen

Die Funktion *TLoad*($\chi$, $B$, $T$) wird auf zwei Intervallgraphen $B$ und $T$ angewendet, wobei $T$ entstanden ist, indem unverplante Operationen in $B$ verplant wurden. *TLoad* erfasst die zusätzlichen Datenpfadkosten, die durch die Bereitstellung von Operatoren in funktionalen Einheiten zur Ausführung der verplanten Operationen in $T$ entstehen. Zusätzliche Operatoren des Typs $t$ werden in einem Cluster $c$ immer dann benötigt, wenn aufgrund der in $T$ verplanten Operationen, die in $B$ noch nicht verplant waren, mehr als *TCWidth*($\sigma(B)$, $c$, $t$) viel Operationen des Typs $t$ in einer Instruktion im Cluster $c$ ausgeführt werden müssen. Ist das nicht der Fall, dann werden voraussichtlich keine weiteren Operatoren dieses Typs im Cluster $c$ benötigt und es entstehen auch keine zusätzlichen Hardwarekosten. Das gilt aber nur unter der Annahme, dass *TCWidth*($\sigma(B)$, $c$, $t$) viele Operatoren des Typs $t$ im Cluster $c$ auch genügen, um die maximal *TCWidth*($\sigma(B)$, $c$, $t$) vielen Operationen des Typs $t$ in jeder Instruktion auszuführen. Da dies nicht immer der Fall ist, wird die Kostenfunktion *TLoad* in Abschnitt 4.1.6.2 durch die Zielfunktion *TDLoad* präzisiert.

Neben den verplanten Operationen in $T$ sollen durch *TLoad* auch die Operationen berücksichtigt werden, die noch nicht verplant sind. Es wird deshalb ein Cluster-*load*-Wert eingeführt, der für jeden Cluster und für jeden Operationstypen $t$ die zusätzliche Belastung der Operatoren des Typs $t$ angibt und dabei sowohl die verplanten als auch die nicht verplanten Operationen berücksichtigt. Aus diesen *load*-Werten der Cluster wird der Wert von *TLoad* durch

$$TLoad(\chi, B, T) = \sum_{c \in V/\chi} \sum_{t \in \mathcal{O}} load(\sigma(T), c, t) \cdot opCost(t)$$

berechnet. Um den Cluster-*load*-Wert für einen Operationstypen $t$ in einem Cluster $c$ zu berechnen, wird jeder Kante $(v, i) \in K$ im Intervallgraphen ein Wahrscheinlichkeitswert $p$ zugeordnet:

$$p((v,i)) = \frac{1}{mob(v)+1}.$$

Dieser Wert gibt an, mit welcher Wahrscheinlichkeit der Knoten $v$ der Instruktion $i$ zugeordnet wird. Für bereits verplante Knoten ist dieser Wert 1. Weiterhin sollen beim Cluster-*load*-Wert nur solche Knoten berücksichtigt werden, die voraussichtlich zusätzliche Hardwarekosten verursachen. Deshalb sollen die Knoten, die die höchste Wahrscheinlichkeit haben in Instruktion $i$

ausgeführt zu werden und für deren Ausführung kein zusätzlicher Operator des Typs $t$ benötigt wird, nicht in die Kostenfunktion eingehen. Die Menge dieser Knoten wird mit $NAC_{i,c,t}$ bezeichnet und die Menge der Knoten, die in der Kostenfunktion berücksichtigt werden müssen mit $AC_{i,c,t}$. Für die Instruktion $i$, den Cluster $c$ und den Typ $t$ sind diese zwei Mengen im Intervallgraphen $T = (V, S, K)$ definiert als:

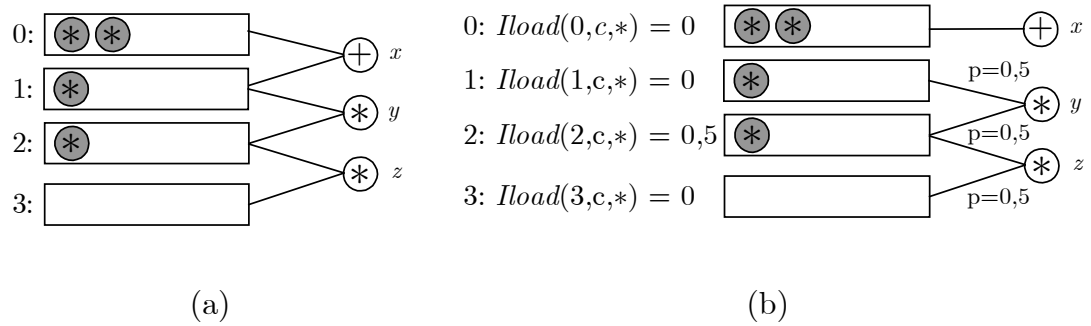
- $AC_{i,c,t} \cup NAC_{i,c,t} = \{v \mid v \in c \text{ und } (v, i) \in K \text{ und } type(v) = t\}$ und
- $AC_{i,c,t} \cap NAC_{i,c,t} = \varnothing$ und
- $|NAC_{i,c,t}| = TCWidth(\sigma(B), c, t)$ und
- $\forall v \forall v' : v \in NAC_{i,c,t} \text{ und } v' \in AC_{i,c,t} \Rightarrow p((v, i)) \geq p((v', i))$.

Die Knoten in der Menge $NAC_{i,c,t}$ sind die Knoten, die in $T$ mit der größten Wahrscheinlichkeit der Instruktion $i$ zugeordnet werden oder es bereits sind und die mit den Operatoren, die bereits zur Ausführung des partiellen Ablaufplans $\sigma(B)$ erforderlich sind, auch ausgeführt werden können. Diese Operationen haben keinen Einfluss auf das Ergebnis der Kostenfunktion, weil sie keine zusätzlichen Hardwarekosten verursachen. $AC_{i,c,t}$ enthält dann nur noch die Knoten des Typs $t$, die der Instruktion $i$ potentiell zugeordnet werden können oder es bereits sind und mit den vorhandenen Operatoren für $t$ nicht mehr ausgeführt werden können. Alle diese Knoten $v$ verursachen mit der Wahrscheinlichkeit $p(v, i)$ zusätzliche Hardwarekosten. Der Instruktions-*load*-Wert *Iload*, der einer Instruktion $i$ im Cluster $c$ angerechnet wird, summiert die Wahrscheinlichkeiten aller dieser Knoten auf:

$$Iload(i, c, t) = \sum_{v \in AC_{i,c,t}} p((v, i)) . \tag{3.5}$$

Damit beachtet der *Iload*-Wert einer Instruktion bereits vorhandene Hardware und die mögliche Planung einer Operation in eine Instruktion wird nur dann durch hohe Kosten bestraft, wenn dadurch zusätzliche Operatoren erforderlich werden.

In Abbildung 4.4 ist ein Beispiel angegeben, das die Berechnung des *Iload*-Wertes für die Instruktionen 0 bis 3 in einem Intervallgraphen für die Multiplikation verdeutlichen soll. Für das Beispiel wird angenommen, dass alle dargestellten Operationen demselben Cluster $c$ zugeordnet sind. Der Intervallgraph in (a) sei $B$ und der Intervallgraph $B_0^x$, in dem probeweise die Operation $x$ in Instruktion 0 verplant wurde, ist in (b) dargestellt und wird mit $T$ bezeichnet. Die in den Instruktionen grau dargestellten Operationen waren bereits in $B$ verplant. Es ist der Übersichtlichkeit wegen keine Kante zwischen diesen Knoten und der Instruktion eingezeichnet.




(a) (b)

Abbildung 4.4: (a) Intervallgraph $B$; (b) Intervallgraph $B_0^x$ mit zugehörigen *Iload*-Werten für die Multiplikation.

In Abbildung 4.4 (b) sind die weiß dargestellten Multiplikationen noch nicht verplant worden. Aufgrund der zwei in Instruktion 0 bereits verplanten Multiplikationen ist $TCWidth(\sigma(B), c, *) = 2$, denn es werden zur Ausführung von $\sigma(B)$ bereits zwei Multiplizierer benötigt. In Instruktion 1 kann höchstens eine weitere Multiplikation verplant werden, wodurch maximal zwei Multiplikationen parallel in Instruktion 1 abzuarbeiten sind. Deshalb ist $AC_{1,c,*} = \varnothing$ und der *Iload*-Wert für Instruktion 1 ist 0. In Instruktion 2 ist ebenfalls bereits eine Multiplikation verplant und es besteht die Möglichkeit, dass noch zwei weitere Multiplikationen dorthin verplant werden können; jede mit der Wahrscheinlichkeit 0,5. Wegen $TCWidth(\sigma(B), c, *) = 2$ ist $AC_{2,c,*} = \{z\}$, denn die Operation $y$ kann ohne einen zusätzlichen Multiplizierer in Instruktion 2 ausgeführt werden. Die Multiplikation $z$ würde dann einen zusätzlichen Multiplizierer erfordern und ist deshalb in $AC_{2,c,*}$ enthalten. Es ergibt sich deshalb für Instruktion 2 der *Iload*-Wert 0,5. In Instruktion 3 ist der *Iload*-Wert 0, weil maximal eine Multiplikation in Instruktion 3 verplant werden kann und dafür genügend Multiplizierer zur Verfügung stehen.

Der Cluster-*load*-Wert für einen Operationstypen $t$ in einem Cluster $c$ ergibt sich aus dem größten Instruktions-*load*-Wert dieses Typs im Cluster $c$:

$$load(\sigma(T), c, t) = \max\left\{ Iload(i, c, t) \,\middle|\, 0 \leq i < |\sigma(T)| \right\}.$$

Das genügt, weil durch diesen maximalen Wert bereits die Wahrscheinlichkeit angegeben ist, mit der zusätzliche Hardware für den Typ $t$ benötigt wird, d. h., in einer Instruktion mehr als $TCWidth(\sigma(B), c, t)$ viele Operationen des Typs $t$ auftreten. Die dadurch entstehenden Hardwarekosten müssen weiteren Instruktionen nicht mehr angelastet werden, da die dortigen Operationen mit der zusätzlichen Hardware ausgeführt werden können.

### 4.1.6 Minimierung von Operatorkonflikten

Die Aufgabe der Ressourcenallokation ist es, die benötigten Operatoren in jedem Cluster des Prozessors zu bestimmen und diese den FUs zuzuordnen. Wird die Planungsphase vor der Ressourcenallokation durchgeführt, dann können in der Planungsphase getroffene Entscheidungen Einfluss auf die Ressourcenallokation haben. So ist durch einen Ablaufplan $\alpha$ für jeden Typ $t$ bereits eine untere Schranke für die Anzahl der benötigten Operatoren dieses Typs in jedem Cluster $c$ durch $TCWidth(\alpha, c, t)$ festgelegt, weil eine entsprechende Anzahl Operationen dieses Typs gleichzeitig ausgeführt werden muss. Diese Abhängigkeit zwischen Planungs- und Ressourcenallokationsphase wird durch die Funktion $TLoad$ erfasst. Es gibt eine weitere Abhängigkeit, die nicht durch $TLoad$ erfasst wird und die durch Operationen desselben Typs in verschiedenen Instruktionen und die beschränkte Verfügbarkeit funktionaler Einheiten verursacht wird. Dadurch kann es vorkommen, dass mehr als $TCWidth(\alpha, c, t)$ viele Operatoren des Typs $t$ im Cluster $c$ erforderlich sind.

**Satz 4.4**
Zur Ausführung der Operationen des Typs $t$ im Ablaufplan $\alpha$, die einem Cluster $c$ mit $FUCWidth(\alpha, c)$ vielen FUs zugeordnet sind, genügen nicht immer $TCWidth(\alpha, c, t)$ viele Operatoren des Typs $t$.

**Beweis**
Das Beispiel in Abbildung 4.5 (a) liefert den Beweis. Zur Ausführung des Ablaufplans $\alpha$ stehen drei funktionale Einheiten zur Verfügung und $TCWidth(\alpha, c, t) = 1$ für alle $t \in \{+, -, *, \&\}$. Um Instruktion 1 auszuführen, muss eine FU eine Subtraktion, eine weitere FU eine Addition und die letzte FU ein logisches Und implementieren. O.b.d.A. seien dies die funktionalen Einheiten $a$, $b$ und $c$. Wenn Instruktion 2 ausgeführt wird, so müssen die Operationen + und & wieder von denselben funktionalen Einheiten ausgeführt werden wie in der ersten Instruktion. Ansonsten müsste bereits hier einer dieser Operationstypen in einer zweiten FU implementiert werden. Damit bleibt für die Multiplikationsoperation der zweiten Instruktion nur die FU $a$ übrig. In Instruktion 3 treten nun die Subtraktion und Multiplikation gemeinsam auf. Es können aber nicht beide Operationen gleichzeitig von der FU $a$ ausgeführt werden. Somit muss entweder der Operator für die Multiplikation oder das Logische Und in einer weiteren FU implementiert werden. In Abbildung 4.5 (a) ist der Operator für die Multiplikation in einer weiteren FU implementiert worden.

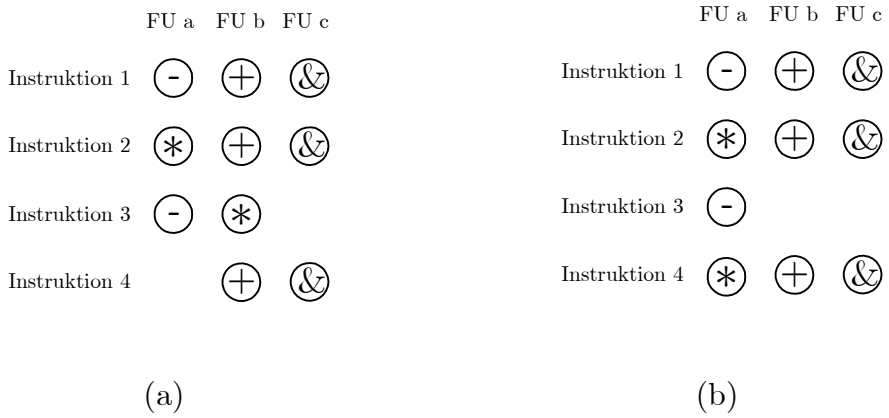



Abbildung 4.5: (a) Operatorenkonflikt in Instruktion 3 zwischen – und ∗; (b) Vermiedener Operatorenkonflikt durch die Planung der Multiplikation aus Instruktion 3 in (a) in die Instruktion 4.

□

Diese Art des Konflikts wird **Operatorkonflikt** genannt, weil zwar alle benötigten Operatoren im Prozessor vorhanden sind, aber nicht gleichzeitig eine Operation ausführen können, da sie derselben FU zugeordnet sind. Weiterhin ist in Abbildung 4.5 (b) ein Beispiel dafür angegeben, wie solche Operatorkonflikte während der Planungsphase vermieden werden können. Es soll angenommen werden, dass die Multiplikation in Instruktion 3 in Abbildung 4.5 (a) auch in die Instruktion 4 geplant werden kann. In beiden Fällen liefern die Zielfunktionen *RBLoad* und *TLoad* keine Entscheidungshilfe, da weder die Anzahl der benötigten Ports ansteigt noch die Anzahl der Operationstypen in einer Instruktion. Dennoch führt die Zuordnung der Multiplikation zu Instruktion 3 dazu, dass entweder die Multiplikation oder die Subtraktion zweimal implementiert werden muss. Wird die Multiplikation dagegen in Instruktion 4 geplant, so treten keine Operatorkonflikte auf. Operatorkonflikte sollen bereits während der Planungsphase berücksichtigt und wenn möglich vermieden werden. Dafür wird im Folgenden zunächst der Interferenzgraph eingeführt, der der Erkennung von Operatorkonflikten dient und auf dessen Basis die Zielfunktion *TDLoad* (vgl. S. 64) in der Planungsphase berechnet wird. Anschließend wird ein entfalteter Interferenzgraph eingeführt, der zur Ressourcenallokation verwendet wird.

#### 4.1.6.1 Konstruktion des Interferenzgraphen

Interferenzgraphen werden häufig zur Modellierung von Ressourcenkonflikten eingesetzt [32, 36]. Die Konflikte werden durch Kanten zwischen den Knoten des Interferenzgraphen modelliert. Bei der hier betrachteten Ressourcenallokation sind die Ressourcen die Operatoren, die durch die Knoten des

Interferenzgraphen repräsentiert werden. Derselben FU können solche Operatoren zugeordnet werden, zwischen denen es keine Konflikte gibt. Das ist immer dann der Fall, wenn sie nie gleichzeitig eine Operation ausführen müssen. Operationen, die in verschiedenen Clustern ausgeführt werden, können nie einen Konflikt zwischen Ressourcen verursachen, weil sie immer verschiedene Operatoren benutzen. Es kann deshalb für jeden Cluster ein separater Interferenzgraph konstruiert werden. Werden Operationen gleichen Typs im selben Cluster nicht gleichzeitig ausgeführt, so sollen diese Operationen nach Möglichkeit vom selben Operator und somit derselben FU ausgeführt werden, um die Datenpfadkosten zu verringern. Alle Operationen, die vom selben Operator ausgeführt werden sollen, werden über eine Äquivalenzrelation $\sim \subseteq V \times V$ in Beziehung gesetzt. Die durch $\sim$ implizierten Äquivalenzklassen bilden die Knoten des Interferenzgraphen. Der Interferenzgraph für einen Ablaufplan $\alpha$ muss mindestens $TCWidth(\alpha, c, t)$ viele Knoten für jeden Typ $t$ enthalten, weil dieser Wert eine untere Schranke für die Anzahl der funktionalen Einheiten im Cluster $c$ ist, die einen Operator des Typs $t$ bereitstellen müssen.

**Definition 4.2 (Interferenzgraph)**
Ein Interferenzgraph zu einem partiellen Ablaufplan $\alpha$ und einem Cluster $c$ ist ein ungerichteter Graph $(N, K)$ mit $N = c/\sim$ und $K \subseteq \{\{m, n\} \mid m, n \in N \text{ und } m \neq n\}$, wobei

1. $u \sim v \;\Rightarrow\; (type(u) = type(v) \wedge \forall i : \{u, v\} \subseteq \alpha(i) \Rightarrow u = v)$ und
2. $|N|$ ist minimal und
3. $\{m, n\} \in K \Leftrightarrow \exists u, v : u \in m \wedge v \in n \wedge \; \exists i : \{u, v\} \subseteq \alpha(i)$.

□

Jeder Knoten des Interferenzgraphen repräsentiert eine Menge von Operationsknoten des Basisblocks, die durch dieselbe FU ausgeführt werden und deshalb nicht in derselben Instruktion vorkommen können. Weil alle durch einen Interferenzgraphknoten repräsentierten Operationsknoten denselben Typ haben, wird jedem Knoten $[v]_\sim \in N$ des Interferenzgraphen der Typ $type([v]_\sim) := type(v)$ zugeordnet. Wegen der Forderungen (1) und (2) in Definition 4.2 beträgt die Anzahl der Interferenzgraphknoten, die den Typ $t$ haben, $TCWidth(\alpha, c, t)$. Diese Knoten werden deshalb auch mit $[t, i]$ bezeichnet, für $i \in \mathbb{N}$ und $1 \leq i \leq TCWidth(\alpha, c, t)$. Wenn die Nummer $i$ in $[t, i]$ nicht relevant ist, dann wird sie, wie in Abbildung 4.6, auch weggelassen und der Knoten nur mit seinem Typ beschriftet. Zwei Interferenzgraphknoten sind genau dann durch eine Kante verbunden, wenn jeder der zwei Interferenzgraphknoten einen Operationsknoten enthält und diese beiden Operationsknoten nicht von derselben FU ausgeführt werden dürfen, weil sie in derselben Instruktion vorkommen. Interferenzgraphknoten, die nicht in Konflikt stehen, repräsentieren Operatoren, die von derselben FU bereitgestellt werden können. Eine Ressourcenallokation kann also bestimmt

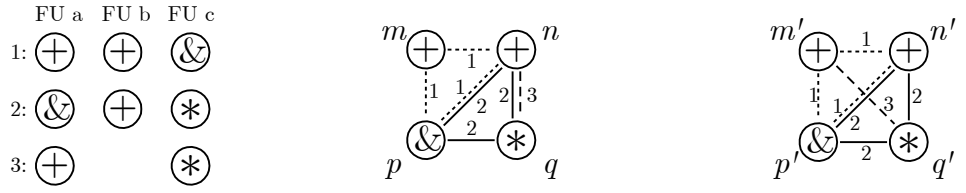
werden, indem Interferenzgraphknoten, die paarweise nicht adjazent sind, zu einer funktionalen Einheit zusammengefasst werden. Die Anzahl der so gebildeten funktionalen Einheiten darf aber die zu diesem Zeitpunkt durch die Ablaufplanung bereits festgelegte Anzahl funktionaler Einheiten $FUCWidth(\alpha, c)$ nicht überschreiten. Das entspricht einer Färbung der Knoten des Interferenzgraphen mit $FUCWidth(\alpha, c)$ vielen Farben, so dass adjazente Knoten verschieden gefärbt sind. In Abbildung 4.6 sind Interferenzgraphen für die Ablaufpläne in Abbildung 4.5 angegeben.


(a) (b)

Abbildung 4.6: (a) Interferenzgraphen zum Ablaufplan in der Abbildung 4.5 (a); (b) Interferenzgraph zum Ablaufplan in der Abbildung 4.5 (b).

Dabei ist im Interferenzgraphen (a) $N = \{m, n, p, q\}$ mit $m = \{1b, 2b, 4b\}$, $n = \{1c, 2c, 4c\}$, $p = \{2a, 3b\}$, $q = \{1a, 3a\}$, wobei $xz$ den Knoten in Instruktion $x$ und FU $z$ aus Abbildung 4.5 (a) bezeichnet. Wie in dem Beispiel anhand der Kante $(p, q)$ zu sehen ist, modelliert der Interferenzgraph die Operatorkonflikte, die zwischen den Operationsknoten im Ablaufplan bestehen. Die günstigere Zuordnung der Operation $4a$ aus Abbildung 4.5 (b) in Instruktion 4 drückt sich durch eine fehlende Kante im Interferenzgraphen in Abbildung 4.6 (b) aus. Der Interferenzgraph wird deshalb genutzt, um während der Ablaufplanung Operatorkonflikte zu erkennen.

Nach Definition 4.2 existieren bei der Konstruktion des Interferenzgraphen Freiheitsgrade bei der Festlegung der Äquivalenzrelation $\sim$, so dass er nicht eindeutig bestimmt ist, wie das Beispiel in Abbildung 4.7 zeigt.




(a) (b) (c)

Abbildung 4.7: (a) Ablaufplan; (b) Interferenzgraph zum Ablaufplan in (a) mit minimaler Kantenanzahl; (c) Interferenzgraph zum Ablaufplan in (a) mit maximaler Kantenanzahl. Kanten sind mehrfach eingezeichnet und mit Instruktionsnummern beschriftet, um zu verdeutlichen, welche Instruktionen zu der entsprechenden Abhängigkeit geführt haben.

Beide Interferenzgraphen in (b) und (c) sind korrekte Interferenzgraphen für den Ablaufplan in (a) mit $m = \{1a\}$, $n = \{1b, 2b, 3a\}$, $p = \{1c, 2a\}$ und $q = \{2c, 3c\}$. In (c) wird aber die Addition $3a$ dem Interferenzgraphknoten $m'$ und nicht $n'$, wie in (b), zugeordnet. Dadurch entsteht eine weitere Abhängigkeit zwischen $q'$ und $m'$. Möglich ist das, weil aufgrund der zwei Additionen in Instruktion 1 zwei Interferenzgraphknoten existieren und die Addition in Instruktion 3 einem beliebigen dieser beiden Knoten zugeordnet werden kann. Um die Zuordnung derart zu gestalten, dass möglichst viele bereits existierende Abhängigkeiten im Interferenzgraphen wiederverwendet werden und die Konstruktion des Interferenzgraphen eindeutig wird, wird eine Ordnung über den Knoten des Interferenzgraphen eingeführt.

**Definition 4.3 (Knotenordnung im Interferenzgraphen)**
$\lessdot \subseteq N \times N$ ist eine Knotenordnung für den Interferenzgraphen $(N, K)$ mit

$$\lessdot := \{([t, i], [t, j]) \mid t \in \mathcal{O} \text{ und } i, j \in \mathbb{N} \text{ und } i < j\}.$$

□

Der Algorithmus 4.5 zur Konstruktion eines Interferenzgraphen nutzt diese Knotenordnung, um die Operationen des Typs $t$ in einer Instruktion den Interferenzgraphknoten $[t, j]$ in aufsteigender Reihenfolge zuzuordnen. Aus dieser Zuordnung wird anschließend die Kantenmenge berechnet. Der so entstehende Interferenzgraph ist eindeutig bestimmt.

**Algorithmus 4.5: (Konstruktion des Interferenzgraphen)**

```
Eingabe: Ein Cluster c ∈ V/χ
         Ablaufplan α
Ausgabe: Interferenzgraph (N,K) zu α

N:=∅
foreach t ∈ 𝒪 do
  for i:=1 to TCWidth(α, c, t) do
    [t,i]:=∅
    N:=N ∪ {[t,i]}
  od
od
for i:=0 to |α|-1 do
  foreach t ∈ 𝒪 do
    foreach v mit v ∈ α(i) und v ∉ α(i-1) und type(v)=t do
      Finde kleinstes j mit [t,j] ∩ α(i) = ∅
      [t,j]:=[t,j] ∪ {v}
    od
  od
od
K:={{u,v} | ∃m∈u ∃n∈v ∃i∈ℕ: {m,n} ⊆ α(i)}
return (N,K)
```

Für Ablaufpläne, in denen nur Operationstypen mit einer Latenzzeit von eins vorkommen, werden alle Operationen desselben Typs aus einer Instruktion den Interferenzgraphknoten dieses Typs in aufsteigender Reihenfolge bzgl. der Knotenordnung $\lessdot$ zugeordnet. Algorithmus 4.5 liefert deshalb in diesem Fall einen kantenminimalen Interferenzgraphen, d. h., $|K|$ – die Anzahl der dargestellten Operatorkonflikte – ist minimal. Um diese Eigenschaft zu beweisen, wird zunächst festgestellt, dass für eine Instruktion $i$, in der $n$ Knoten des Typs $t$ und $m$ Knoten des Typs $t'$ vorkommen, Algorithmus 4.5 zwischen den Interferenzgraphknoten $[t, 1]$, ..., $[t, n]$ und $[t', 1]$, ..., $[t', m]$ die Kanten

$$K_i^{t,t'} = \{\{[t,k],[t',j]\} \mid 1 \leq k \leq n \text{ und } 1 \leq j \leq m\}$$

erzeugt. Offensichtlich ist die Vereinigung zweier Mengen $K_i^{t,t'}$ und $K_j^{t,t'}$ für jedes Instruktionspaar $i$ und $j$ minimal, weil abgesichert ist, dass beide Kantenmengen mit einer maximalen Anzahl gemeinsamer Knoten adjazent sind.

**Lemma 4.3**
Für zwei verschiedene Operationstypen $t$ und $t'$ ist die Menge

$$K^{t,t'} := \bigcup_{0 \le i < l} K_i^{t,t'}$$

minimal, wobei $l = |\alpha|$.

**Beweis**
In einem Interferenzgraphen sei $F_i^{t,t'}$ die Kantenmenge, die eine Kante $\{m, n\}$ genau dann enthält, wenn es in Instruktion $i$ zwei Knoten $u$ und $v$ mit $type(u) = t$, $type(v) = t'$, $u \in m$ und $v \in n$ gibt. Jede der Kantenmengen $F_i^{t,t'}$ muss dieselben Konflikte modellieren wie die Kantenmenge $K_i^{t,t'}$ und hat daher dieselbe Mächtigkeit. Angenommen es gibt Kantenmengen $F_1^{t,t'},\ldots,F_l^{t,t'}$ so dass

$$\left|\bigcup_{0 \le i < l} F_i^{t,t'}\right| < \left|\bigcup_{0 \le i < l} K_i^{t,t'}\right|.$$

Weiterhin sei $f_i$ die Anzahl der Kanten in $F_i^{t,t'}$, die in den Kantenmengen $F_0^{t,t'}$ bis $F_{i-1}^{t,t'}$ nicht vorkommen. Analog dazu ist $k_i$ für $0 \le i < l$ und die Mengen $K_i^{t,t'}$ definiert. Es ist $f_0 = k_0$ und

$$\left|\bigcup_{0 \le i < l} F_i^{t,t'}\right| = \sum_{i=0}^{l-1} f_i\,.$$

Somit muss es ein $f_i$ geben, das kleiner ist als $k_i$. Das bedeutet, dass die Menge $F_i^{t,t'}$ weniger zusätzliche Kanten enthielt als die Menge $K_i^{t,t'}$. Somit muss es ein $j < i$ geben, für das $\left|F_i^{t,t'} \cup F_j^{t,t'}\right| < \left|K_i^{t,t'} \cup K_j^{t,t'}\right|$. Das ist ein Widerspruch zu der Feststellung, dass in dem durch Algorithmus 4.5 konstruierten Interferenzgraphen die Vereinigung zweier Kantenmengen $K_i^{t,t'}$ und $K_j^{t,t'}$ minimal ist.

□

**Satz 4.5**
Der durch Algorithmus 4.5 erzeugte Interferenzgraph $I$ ist kantenminimal.

**Beweis**
Angenommen $I = (N, K)$ ist nicht kantenminimal, dann gibt es einen Interferenzgraphen $I' = (N, K')$ mit $|K'| < |K|$, der mindestens eine Kante weniger hat.

1. Fall: Diese Kante fehlt zwischen den zwei Interferenzgraphknoten $[t, i]$ und $[t, j]$ mit $1 \le i, j \le TCWidth(\alpha, c, t)$, wobei $c$ der Cluster ist, für den der Interferenzgraph konstruiert wurde. Es muss aber wegen der Minimalitätsforderung für $N$ in der Definition 4.2 eine Instruktion in $\alpha$ geben, die genau

$TCWidth(\alpha, c, t)$ viele Operationen des Typs $t$ enthält. Damit gibt es auch zwei Operationen $u \in [t, i]$ und $v \in [t, j]$, die gemeinsam in dieser Instruktion vorkommen. Demnach muss es eine Kante zwischen $[t, i]$ und $[t, j]$ geben.

2. Fall: Somit muss es zwei Operationstypen $t \neq t'$ geben, so dass in $I'$ $\left|F^{t,t'}\right| < \left|K^{t,t'}\right|$ ist, wobei $F^{t,t'}$ die Menge aller Kanten in $I'$ ist, die mit einem Interferenzgraphknoten des Typs $t$ und einem Knoten des Typs $t'$ adjazent sind. Diese Kantenmenge ist aber nach Lemma 4.3 bereits in $I$ minimal.

□

Gibt es im Ablaufplan Operationen mit einer Latenzzeit größer eins, so liefert Algorithmus 4.5 im Allgemeinen keinen kantenminimalen Interferenzgraphen. Ein entsprechendes Beispiel ist in Abbildung 4.8 angegeben.

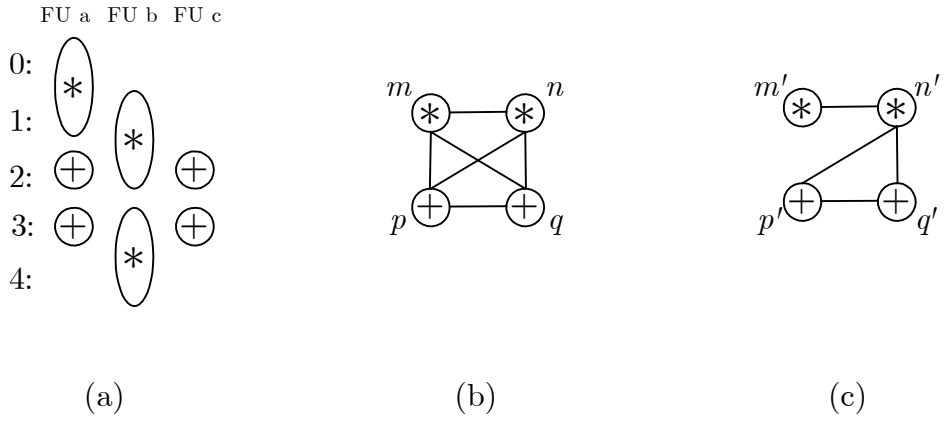



Abbildung 4.8: (a) Ablaufplan, in dem Multiplikationen die Latenzzeit 2 haben; (b) Interferenzgraph zu (a), wie er durch Algorithmus 4.5 konstruiert wird; (c) Kantenminimaler Interferenzgraph zum Ablaufplan in (a).

In Abbildung 4.8 (b) ist die Zuordnung dargestellt, wie sie durch Algorithmus 4.5 vorgenommen wird. Es ist $m = \{0a, 1a, 3b, 4b\}$, $n = \{1b, 2b\}$, $p = \{2a, 3a\}$ und $q = \{2c, 3c\}$. In Abbildung 4.8 (c) ist die kantenminimale Zuordnung $m = \{0a, 1a\}$, $n = \{1b, 2b, 3b, 4b\}$, $p = \{2a, 3a\}$ und $q = \{2c, 3c\}$ angegeben.

#### 4.1.6.2 Minimierung von Operatorkonflikten durch *TDLoad*

Die Zielfunktion $TDLoad(\chi, B, T)$ wird während der Planungsphase auf einen Intervallgraphen $T$ angewendet, in dem wenigstens die Knoten verplant sind, die bereits in $B$ verplant waren. Für jeden Cluster wird durch *TDLoad* die Kostenabschätzung der Zielfunktion $TLoad(\chi, B, T)$ präzisiert. Das wird erreicht, indem für $\sigma(T)$ und jeden Cluster ein Interferenzgraph konstruiert wird, um Operatorkonflikte zu erkennen, die aufgrund der in $T$ neu

verplanten Operationen entstanden sind. Dadurch können Operatorkonflikte bereits während der Planungsphase vermieden werden.

Für den partiellen Ablaufplan $\sigma(T)$ wird ein Interferenzgraph nach Algorithmus 4.5 für jeden Cluster berechnet. Für einen Cluster $c$ sei $I = (N, K)$ der Interferenzgraph zum Ablaufplan $\sigma(B)$ und $I' = (N', K')$ der Interferenzgraph zum Ablaufplan $\sigma(T)$. Es soll abgeschätzt werden, ob $TCWidth(\sigma(T), c, t)$ viele Operatoren des Typs $t$ im Cluster $c$ genügen, um alle Operationen des Typs $t$, die in $\sigma(T)$ gleichzeitig ausgeführt werden müssen, auszuführen. Das ist genau dann der Fall, wenn in $I'$ die Knoten zu nicht mehr als $FUCWidth(\sigma(T), c)$ vielen funktionalen Einheit zusammengefasst werden können. Da alle zusammengefassten Knoten untereinander nicht durch eine Kante verbunden sein dürfen, entspricht das Zusammenfassen dem Graphfärbungsproblem, wobei die Anzahl der nutzbaren Farben der Anzahl der funktionalen Einheiten $mF_c := FUCWidth(\sigma(T), c)$ im Cluster $c$ entspricht. Um die Abschätzung schnell zu berechnen, wird der Kantengrad der Knoten im Interferenzgraphen genutzt. Statt den Graphen mit $mF_c$ vielen Farben zu färben, wird berechnet, welche Knoten nicht färbbar sein könnten. Dafür werden die Knoten entfernt, deren Färbung sicher möglich ist. Das sind alle Knoten, die mit weniger als $mF_c$ vielen anderen Knoten adjazent sind. Diese Knoten und adjazente Kanten werden schrittweise aus dem Graphen entfernt, wie es durch die folgende Funktion *red* beschrieben ist, die einen reduzierten Interferenzgraphen erzeugt. Durch das Entfernen eines Knotens können weitere Knoten färbbar werden, weil mit dem Knoten auch Kanten entfernt werden:

$$red((N,K)) = \begin{cases} (N,K), \text{ falls } \forall n \in N : \deg(n) \geq mF_c \\ red\Big(\big(N - \{n\}, K - \big\{\{n,m\} | m \in N\big\}\big)\Big), \text{ falls } n \in N \wedge \deg(n) < mF_c. \end{cases}$$

Dasselbe Vorgehen wird auch bei der Berechnung einer Registerallokation mittels Graphfärbung angewendet [36]. Wird durch *red* der leere Graph berechnet, so konnten alle Knoten entfernt werden und der ursprüngliche Graph ist mit $mF_c$ vielen Farben färbbar. Ist nicht der leere Graph berechnet worden, dann ist für die in dem reduzierten Interferenzgraphen $red(I')$ verbleibenden Knoten mit einem Kantengrad von mindestens $mF_c$ nicht sichergestellt, dass sie färbbar sind. In $I'$ neu entstandene Kanten, die in $red(I')$ noch vorhanden sind, modellieren damit neu entstandene Operatorkonflikte. Es ist $(N'', K'') = red(I')$ und

$$NK = K'' - K$$

die Menge dieser neu entstandenen Kanten. Unter Verwendung der Kantenmenge $NK$ können jetzt alle Knoten in $I'$ und damit die durch diese Knoten repräsentierten Operatoren bestimmt werden, deren Kantengrad durch einen neu entstandenen Operatorkonflikt zugenommen hat:

$$W = \big\{u \big|\ \{u,v\} \in NK\big\}.$$

Jeder dieser Knoten $w \in W$ ist mit einer gewissen Wahrscheinlichkeit $p$ nicht färbbar und verursacht somit zusätzliche Hardwarekosten. Je größer dabei der Kantengrad von $w$ ist, desto höher ist diese Wahrscheinlichkeit. Daher wird für jeden Knoten in $W$ ein Gewicht $wnc$ definiert, das den Kantengrad berücksichtigt:

$$wnc(w) = \left|\left\{u \middle|\ \{w,u\} \in NK\right\}\right|.$$

Dieses Gewicht berücksichtigt nur die Kanten, die neu entstanden sind und dazu beitragen, dass $w$ einen Kantengrad größer als $mF_c$ in $I'$ hat. Für einen Cluster $c$ wird die Summe dieser Gewichte durch

$$TDLoad_c(B,T) = \frac{\sum_{w \in W} opCost(type(w)) \cdot wnc(w)}{z} \cdot p$$

gebildet. Die Kosten werden durch $z$ relativiert, wobei $z$ die Anzahl der Knoten ist, die in $T$ verplant sind und in $B$ noch nicht verplant waren. Die Gesamtkosten ergeben sich aus den Kosten der zusätzlich entstandenen Operatorkonflikte in den einzelnen Clustern:

$$TDLoad(\chi,B,T) = \sum_{c \in (V/\chi - [e]_\chi)} TDLoad_c(B,T).$$

## 4.2 Ressourcenallokation

Durch den Planungsalgorithmus ist zu einem Basisblock, einer Clusterung und einer Ablaufplanlänge ein Ablaufplan berechnet worden. Aus ihm geht die Anzahl der benötigten funktionalen Einheiten in jedem Cluster und die Zuordnung der Operationen zu Instruktionen hervor. Damit sind auch die Registerbankkosten und eine untere Schranke für die Datenpfadkosten des Prozessors bekannt (vgl. (1.13)). Es ist aber noch nicht festgelegt worden, welche Operatoren jede funktionale Einheit bereitstellen muss und welche Operationen durch welche FU in jeder Instruktion ausgeführt werden. Ziel ist deshalb die Berechnung der Bindung $\phi$, aus der dann die benötigte Ressourcenallokation $ft : \mathcal{F} \to \wp(\mathcal{O})$ abgeleitet werden kann. Da die Ressourcenallokation keinen Einfluss auf die Registerbankkosten in den einzelnen Clustern hat, können die Datenpfadkosten minimiert werden, indem die Funktion (1.8) für jeden Cluster separat minimiert wird. Ziel ist somit die Minimierung der Operatorkosten, indem in jedem Cluster für jeden Operationstyp möglichst wenig Operatoren in den funktionalen Einheiten implementiert werden müssen. Folgende Bedingungen sind dabei einzuhalten:

- Die Operationen jeder Instruktion müssen sich in jedem Cluster mit der durch die Ablaufplanung festgelegten Anzahl funktionaler Einheiten ausführen lassen und

- Operationen, die eine Latenzzeit größer als eins haben, müssen in jeder Instruktion von derselben FU ausgeführt werden.

Mit dem in Definition 4.2 angegebenen Interferenzgraphen lassen sich die Operatorkonflikte während der Ablaufplanung gut modellieren. Er eignet sich aber weniger gut zur Berechnung der Ressourcenallokation, weil große Cliquen während der kantenminimalen Konstruktion entstehen können und Entscheidungen zur Ressourcenallokation bereits während der Konstruktion des Interferenzgraphen getroffen werden. Ein Beispiel dafür ist in der Abbildung 4.9 angegeben.

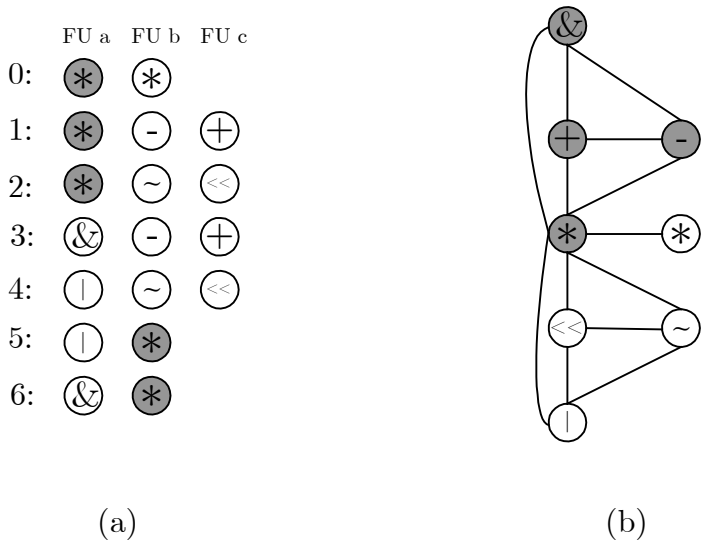



Abbildung 4.9: (a) Ablaufplan; (b) Interferenzgraph zum Ablaufplan in (a).

Der Interferenzgraph in (b) enthält eine grau dargestellte 4-Clique, die eine Färbung des Interferenzgraphen mit drei Farben unmöglich macht. Trotzdem können die Operationen im Ablaufplan so angeordnet werden, dass bis auf die Multiplikation jeder Operationstyp einmal und die Multiplikation zweimal implementiert wird. In Abbildung 4.9 (a) ist diese Zuordnung bereits dargestellt. Die Ursache für die 4-Clique ist die durch die Knotenordnung (vgl. Definition 4.3) erzwungene Zuordnung aller grau dargestellten Multiplikationen in (a) zu dem grau dargestellten Multiplikationsknoten in (b). Existierende Freiheitsgrade bei der Bindung einer Operation an eine FU werden so bereits bei der Konstruktion eingeschränkt, indem für die Ausführung einzelner Operationen gleichen Typs aus verschiedenen Instruktionen derselbe Operator und damit dieselbe funktionale Einheit festgelegt wird. Dieses Zuordnungsproblem entsteht immer dann, wenn in der Instruktion, in der eine Operation des Typs $t$ ausgeführt wird, weniger Operationen des Typs $t$ vorhanden sind als in einer anderen Instruktion. In Abbildung 4.9 (a) ist das der Fall für die Multiplikation in Instruktion 6, die mit $v$ bezeichnet werden soll, und die zwei Multiplikationen in Instruktion 0,

von denen eine mit $u$ und die andere mit $w$ bezeichnet werden soll. Es muss entschieden werden, ob $v$ von der FU ausgeführt wird, die $u$ ausführt oder von der FU die $w$ ausführt. Diese Entscheidung hat aber nicht nur lokale Konsequenzen, weil durch die Bindung von $v$ an eine FU andere Operationen in der Instruktion 6 nicht mehr an diese FU gebunden werden können. Dadurch wird an dieser Stelle eine Zuordnung von Operatoren zu funktionalen Einheiten ausgeschlossen, die sich an anderer Stelle nachteilig auswirken kann. Zur Berechnung der Bindung $\phi$ soll diese Entscheidung nicht während der Konstruktion des Interferenzgraphen getroffen werden sondern erst bei dessen Färbung.

Der in der HLS zur Lösung der Ressourcenallokation verwendete Konfliktgraph (vgl. 3.1.1) ist nur bedingt zur Modellierung des Ressourcenallokationsproblems für VLIW-Prozessoren geeignet, da für seine Konstruktion bekannt sein muss, welche Operationen von einer FU ausgeführt werden können. Die Operationen, die eine FU ausführen kann, müssen aber erst bestimmt werden. Für die Berechnung der Bindung wird deshalb ein entfalteter Interferenzgraph genutzt.

### 4.2.1 Konstruktion des entfalteten Interferenzgraphen

Die Knoten des entfalteten Interferenzgraphen, der zum Cluster $c$ konstruiert wird, entsprechen den Operationsknoten des Clusters $c$. Die Ressourcenallokation wird durch Färbung dieser Knoten berechnet. Die verfügbaren Farben entsprechen dabei den funktionalen Einheiten im Cluster $c$, deren Anzahl $FUCWidth(\alpha, c)$ aus der Planungsphase bekannt ist. Es werden zwei Arten von Konflikten unterschieden und durch zwei Kantenmengen $H$ und $S$ modelliert. Das sind zum einen Konflikte zwischen Operationen, die gleichzeitig ausgeführt werden und die unbedingt beachtet werden müssen. Diese Konflikte werden durch $H$-Kanten modelliert. Es ist $\{u, v\} \in H$, falls die zwei Operationen $u$ und $v$ gleichzeitig ausgeführt werden. Zum anderen werden Konflikte zwischen Operationen, die nicht gleichzeitig ausgeführt werden und trotzdem nicht von derselben FU ausgeführt werden sollen, durch $S$-Kanten modelliert. Diese Konflikte können unberücksichtigt bleiben, führen dann aber zu einem erhöhten Operatorbedarf im Prozessor. Durch die $S$-Kanten wird das oben beschriebene Zuordnungsproblem, das bei der Konstruktion des Interferenzgraphen nach Definition 4.2 auftritt, in die Färbungsphase verschoben. In dem Ablaufplan in Abbildung 4.10 (a) zum Beispiel gibt es einen solchen Konflikt zwischen der Addition in Instruktion 3 und der Multiplikation in Instruktion 2. Wird dieser Konflikt bei der Färbung mit drei Farben beachtet, dann erhält die Addition in Instruktion 3 dieselbe Farbe, die eine der Additionen in Instruktion 2 bekommen hat.

Um die Kantenmenge $S$ zu bestimmen, wird in jedem Cluster $c$ für jeden Operationstypen $t$ eine beliebige Instruktion aus $\alpha$, in der die meisten

Operationen des Typs $t$ gleichzeitig ausgeführt werden, als **maximale Instruktion** festgelegt und mit $m_t$ bezeichnet. Für $m_t$ muss also gelten:

$$TCWidth(\alpha, c, t) = TICWidth\big(\alpha(m_t), c, t\big).$$

Zur Ausführung der Instruktion $m_t$ werden die meisten funktionalen Einheiten, die den Operrationstyp $t$ unterstützen, benötigt. Kommt in einer anderen Instruktion eine Operation $v$ des Typs $t$ vor, dann sollte eine dieser funktionalen Einheiten zur Ausführung von $v$ verwendet werden. Das bedeutet, dass $v$ nicht von einer FU ausgeführt werden soll, die in $m_t$ eine Operation mit einem Typ verschieden von $t$ ausführt. Aus diesem Grund wird die Operation $v$ mit allen Operationen in $m_t$, die nicht den Typ $t$ haben, durch eine $S$-Kante verbunden. Damit alle funktionalen Einheiten, die in $m_t$ keine Operation des Typs $t$ ausführen, zur Ausführung von $v$ ausgeschlossen werden, muss die Instruktion $m_t$ so viele Operationen enthalten, wie es funktionale Einheiten im Cluster $c$ gibt. Ist das nicht der Fall, wird $m_t$ mit *nop*-Operationen aufgefüllt. Es entsteht ein neuer Ablaufplan $\alpha'$ mit

$$\alpha'(i) = \begin{cases} \alpha(i), \text{ falls } \forall t \in \mathcal{O} : i \neq m_t \\ \alpha(i) \cup \{h_{i,1}, \ldots, h_{i,n}\} \text{ mit } n = FUCWidth(\alpha, c) - FUCIWidth(\alpha(i), c) \end{cases}$$

und $type(h_{i,k}) = nop$ für alle Hilfsknoten $h_{i,k}$. In Abbildung 4.10 (b) ist der um die *nop*-Operation erweiterte Ablaufplan aus Abbildung 4.10 (a) für den Fall angegeben, dass $m_{\&} = 1$. Stehen beispielsweise drei funktionale Einheiten zur Verfügung, wie im Ablaufplan in Abbildung 4.10 (a), und $m_{\&} = 1$ enthält nur eine Operation des Typs & und es würden keine *nop*-Operation eingefügt werden, dann würde für die &-Operation in Instruktion 3 nur die FU der Multiplikation in Instruktion 1 ausgeschlossen werden. Es muss aber auch eine zweite FU ausgeschlossen werden, um sicherzustellen, dass die &-Operation in Instruktion 3 dieselbe Farbe erhält wie die &-Operationen in Instruktion 1.

**Definition 4.4 (Entfalteter Interferenzgraph)**
Ein entfalteter Interferenzgraph $I = (c, H \cup S)$ zu einem um *nop*-Operationen erweiterten Ablaufplan $\alpha'$ und einem Cluster $c \in V/\chi$, wobei $V$ die Knotenmenge des Basisblocks ist, ist ein ungerichteter Graph mit der Knotenmenge $c$ und den zwei disjunkten Kantenmengen
$H, S \subseteq \{\{v, w\} \mid v, w \in c \text{ und } v \neq w\}$, wobei

- $H := \big\{\{u,v\} \big| u \neq v \wedge \exists i : \{u,v\} \in \alpha'(i)\big\}$
- $S := \{\{u,v\} \big|\, u \in \alpha'(m_t) \wedge \exists i \neq m_t : v \in \alpha'(i) \wedge type(v) = t \wedge type(u) \neq t \wedge type(v) \neq nop \wedge \{u,v\} \notin H\}$.

□

Für *nop*-Operationen wird keine maximale Instruktion $m_{nop}$ bestimmt. Bei der Konstruktion des entfalteten Interferenzgraphen gibt es nur noch bei der Wahl der maximalen Instruktionen Freiheitsgrade. Dadurch wird, anders als beim Interferenzgraphen, erreicht, dass während der Konstruktion des entfalteten Interferenzgraphen keine Entscheidungen zur Ressourcenallokation getroffen werden müssen. Obwohl bei der Wahl der maximalen Instruktionen Freiheitsgrade existieren können, beeinflusst deren Wahl nicht die Ressourcenallokation, sofern sichergestellt ist, dass eine optimale Färbung des entfalteten Interferenzgraphen gefunden wird. Das Konstruktionsverfahren für den entfalteten Interferenzgraphen ist im Folgenden Algorithmus 4.6 angegeben.

**Algorithmus 4.6: Konstruktion des entfalteten Interferenzgraphen**

1. Für jeden Operationstypen $t \neq nop$ wird eine beliebige maximale Instruktion $m_t$ bestimmt.
2. Für jeden Operationstyp $t$ wird die Instruktion $m_t$ mit *nop*-Operationen aufgefüllt, so dass $m_t$ genau $FUCWidth(\alpha, c)$ viele Operationen enthält.
3. Für jede Instruktion $i$ im Ablaufplan und jedes Paar von Operationen $\{u, v\} \subseteq \alpha(i)$ mit $u \neq v$ wird die Kante $\{u, v\}$ in $H$ aufgenommen.
4. Für jede Instruktion $i$ und jede Operation $v \in \alpha(i)$ wird eine Kante $\{u, v\}$ in $S$ aufgenommen, wenn $type(v) = t$ und $type(u) \neq t$ und $u \in \alpha(m_t)$ und $\{u, v\} \notin H$.

In Abbildung 4.10 (c) ist der so konstruierte entfaltete Interferenzgraph zum Ablaufplan in Abbildung 4.10 (a) angegeben.

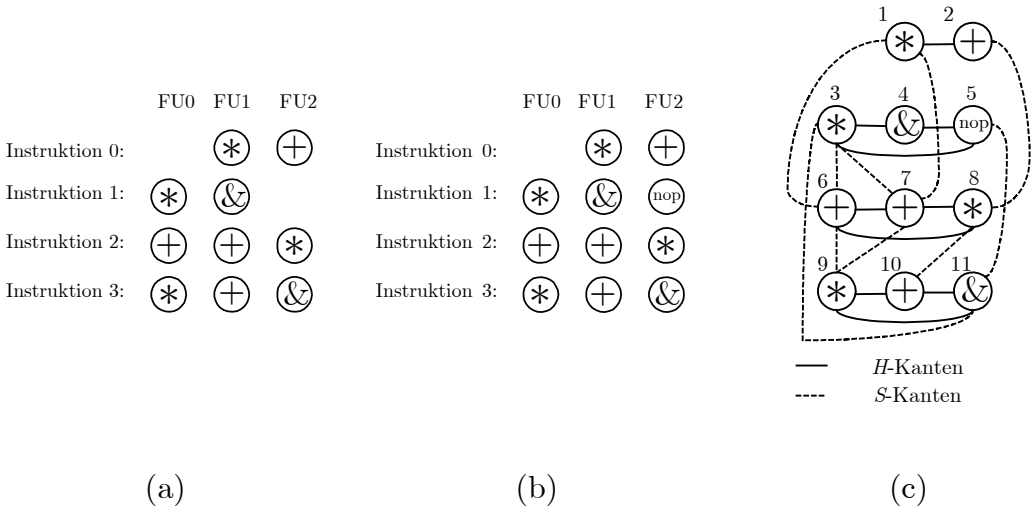



Abbildung 4.10: (a) Ablaufplan; (b) Maximale Instruktionen im Ablaufplan aus (a) mit *nop*-Operationen aufgefüllt, wobei $m_+ = 2$, $m_* = 2$ und $m_\& = 1$; (c) Entfalteter Interferenzgraph zum Ablaufplan in (b).

Operationen mit einer Latenzzeit größer eins werden durch einen Knoten im entfalteten Interferenzgraphen repräsentiert, auch wenn die entsprechende Operation in mehreren aufeinander folgenden Instruktionen ausgeführt wird. Entsprechend wird die Operation durch *H*-Kanten mit jeder Operation verbunden, mit der sie gleichzeitig ausgeführt wird. In Abbildung 4.11 ist ein Beispiel angegeben. Die Multiplikationen haben dort eine Latenzzeit von zwei. Die maximale Instruktion für die Multiplikation ist Instruktion 2, in der gleichzeitig die Multiplikationen 3 und 8 ausgeführt werden müssen. Die übrigen Multiplikationen 1 und 9 sind deshalb durch eine *S*-Kante mit den beiden Additionen 6 und 7 verbunden. Auch wenn die Multiplikationen 3 und 8 außer in Instruktion 2 noch in den Instruktionen 1 bzw. 3 als Operation vorkommen, ist eine *S*-Kante von Multiplikation 3 bzw. 8 zu den Additionen 6 und 7 in Instruktion 2 nicht notwendig, da beide Multiplikationen bereits durch eine *H*-Kante mit diesen Additionsoperationen verbunden sind. Weiterhin ist jede Multiplikation durch *H*-Kanten mit allen Operationen verbunden, die mit ihr in einer gemeinsamen Instruktion vorkommen. So ist beispielsweise die Multiplikation 1 mit den Operationsknoten 2, 3, 4 und 5 adjazent.

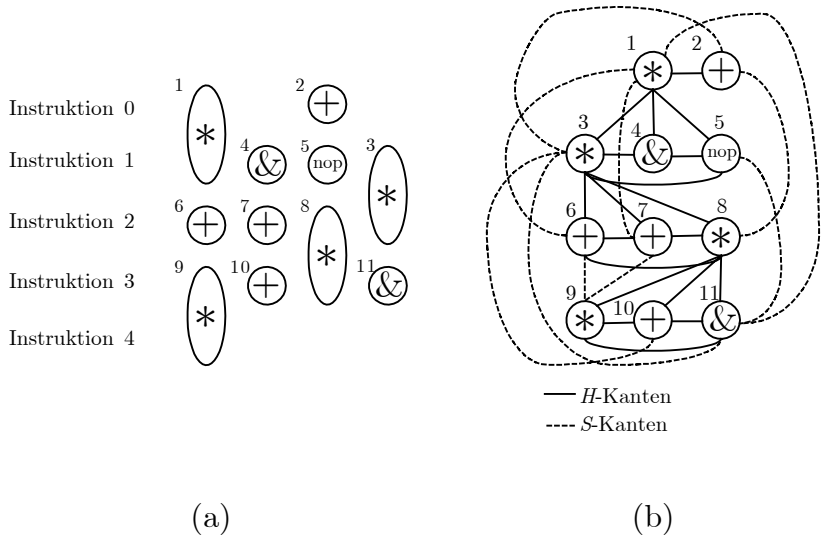



Abbildung 4.11: (a) Ablaufplan mit Multiplikationsoperationen, deren Latenzzeit 2 beträgt; (b) Interferenzgraph zum Ablaufplan in (a). Maximale Instruktionen sind: $m_* = 2$, $m_+ = 2$, $m_\& = 1$.

### 4.2.2 Färben des entfalteten Interferenzgraphen

Nach der Konstruktion des Interferenzgraphen $I = (c, S \cup H)$ für einen Cluster $c$ wird die Bindung durch die Färbung der Knoten im Interferenzgraphen berechnet. Jede Farbe entspricht dabei einer funktionalen

Einheit im Cluster $c$. Die gesuchte Färbung ist dann eine Funktion $col : c \to \{1, \ldots, K\}$ mit $K = FUCWidth(\alpha, c)$ und der Eigenschaft

$$\{u, v\} \in H \Rightarrow col(u) \neq col(v)$$

sowie einer maximalen Menge $S' \subseteq S$ mit

$$\{u, v\} \in S' \Rightarrow col(u) \neq col(v).$$

Wird die Färbungseigenschaft bei einer Kante $\{u, v\} \in S$ mit $type(u) = t$ und $type(v) = t'$ verletzt, so konnte entweder $u$ oder $v$ nicht mit einer der $K$ Farben gefärbt werden. Die Konsequenz ist, dass entweder der Operator $t$ von mehr als $TCWidth(\alpha, c, t)$ bzw. $t'$ von mehr als $TCWidth(\alpha, c, t')$ vielen funktionalen Einheiten unterstützt werden muss. Aus der Färbung $col$ wird für den Cluster $c$ die Bindung $\phi_c$ erzeugt durch

$$(u, v) \in \phi_c \Leftrightarrow col(u) = col(v)\,.$$

Die Bindung $\phi$ für alle Operationen ist die Vereinigung der Bindungen $\phi_c$ der einzelnen Cluster. Damit ist auch sichergestellt, dass $\phi \subseteq \chi$. Für den externen Cluster $[e]_\chi$ wird kein Interferenzgraph konstruiert. Die Bindung wird direkt aus dem Ablaufplan berechnet. Die Anzahl der benötigten Lade- bzw. Speicheroperatoren ist

$$TCWidth\big(\alpha, [e]_\chi, load\big) \text{ bzw. } TCWidth\big(\alpha, [e]_\chi, store\big)$$

und die Anzahl der benötigten Kopieroperatoren ist

$$TCWidth\big(\alpha, [e]_\chi, copy\big)\,.$$

Damit sind in jeder Instruktion für jede Lade-, Speicher- und Kopieroperation genügend Operatoren vorhanden, um die Operationen auszuführen, so dass alle Operationen, die demselben Operator zugeordnet sind, durch $\phi$ in Relation gesetzt werden.

Zur Berechnung der Färbung $col$ des Interferenzgraphen kann jeder Algorithmus verwendet werden. Hier wird eine Strategie verwendet, die auch bei der Registerallokation genutzt wird [36] und immer erfolgreich ist, wenn alle $S$-Kanten des Interferenzgraphen außer Acht gelassen werden[1]. Die Anzahl der verfügbaren Farben ist durch $K = FUCWidth(\alpha, c)$ gegeben. Das Färben wird in zwei Phasen durchgeführt:

- In der **Reduktionsphase** wird der Interferenzgraph $I$, wie in der Funktion $red$ (vgl. S. 79), zum leeren Graphen reduziert, indem

[1] In dem Fall, dass alle $S$-Kanten unbeachtet bleiben, entspricht der entfaltete Interferenzgraph dem Konfliktgraphen, der in der HLS zur Ressourcenallokation verwendet wird. Die funktionalen Einheiten sind dann der einzige Ressourcentyp, der jede Operation ausführen kann.

schrittweise jeder Knoten und die mit dem Knoten adjazenten Kanten aus $I$ entfernt werden.

- In der **Färbungsphase** werden die Knoten zusammen mit den Kanten in der umgekehrten Reihenfolge wieder in $I$ eingefügt. Dabei wird jeder Knoten so gefärbt, dass seine Farbe verschieden von der Farbe seiner bis zu diesem Zeitpunkt schon wieder eingefügten Nachbarn ist.

Dieses Vorgehen ist in Algorithmus 4.7 dargestellt. Dort ist $deg(v)$ der Kantengrad eines Knotens $v \in c$ im Interferenzgraphen, der der Anzahl der Kanten in $H \cup S$, mit denen $v$ adjazent ist, entspricht.

**Algorithmus 4.7 (Färbungsalgorithmus für den entfalteten Interferenzgraph)**

```
Eingabe: Entfalteter Interferenzgraph (c, H ∪ S)
         Anzahl der Farben K
Ausgabe: Bei Erfolg Färbung der Knoten: col
         Bei Fehler der nicht färbbare Knoten v_j

j:=0; E:=H ∪ S
while(c ≠ ∅) do                                  // Reduktionsphase
  F:={u | u ∈ c und deg(u) < K}
  if(F = ∅) then
    U:={u | |{v | {u,v} ∈ E ∩ H}| < K}
    Wähle einen Knoten v ∈ U aus
  else
    Wähle einen Knoten v ∈ F aus
  fi
  c:=c - {v}; E:=E - {{u,v} | u ∈ c}
  v_j:=v; j++;
od
while(j > 0) do                                  // Färbungsphase
  j--;
  c:=c ∪ {v_j}; E := E ∪ {{v_j,u} | {v_j,u} ∈ H ∪ S und u ∈ c};
  if(es existiert eine Farbe für v_j) then
    col(v_j):=f, wobei f durch Algorithmus 4.8 berechnet wird
  else return v_j fi
od
return col
```

**Definition 4.5 (sicher färbbar)**
Wenn in der Reduktionsphase von Algorithmus 4.7 ein Knoten mit weniger als $K$ Nachbarn aus $I$ entfernt wird, dann wird dieser Knoten als sicher färbbar bezeichnet.

□

In Algorithmus 4.7 enthält in jeder Iteration der ersten *while*-Schleife die Menge $F$ alle Knoten, die in dieser Iteration mit weniger als $K$ anderen Knoten unter Berücksichtigung der $H$- und $S$-Kanten verbunden sind.

**Lemma 4.4**
Wird ein Knoten $v$ durch Algorithmus 4.7 aus $I$ entfernt und ist dieser Knoten sicher färbbar, dann kann $v$ in der Färbungsphase eine der $K$ Farben zugeordnet werden, so dass für alle Nachbarn $w$ von $v$, die zu diesem Zeitpunkt schon wieder in $I$ eingefügt wurden, gilt: $col(v) \neq col(w)$.

**Beweis**
Wenn $v$ aus $I$ entfernt wird, hat $v$ die Nachbarn $\{w_1, \ldots, w_n\}$ mit $n < K$. Wird $v$ wieder in $I$ eingefügt, so sind zu diesem Zeitpunkt auch nur diese $n$ Nachbarn in $I$ vorhanden. Selbst wenn jeder Nachbar mit einer anderen Farbe gefärbt ist, wurden nur $n < K$ viele Farben verwendet. Es gibt also noch mindestens eine Farbe mit der $v$ gefärbt werden kann.

□

In Abbildung 4.12 (a) ist der Interferenzgraph aus Abbildung 4.10 (c) nach dem Entfernen der Knoten 2, 1, 4, und 5 dargestellt.

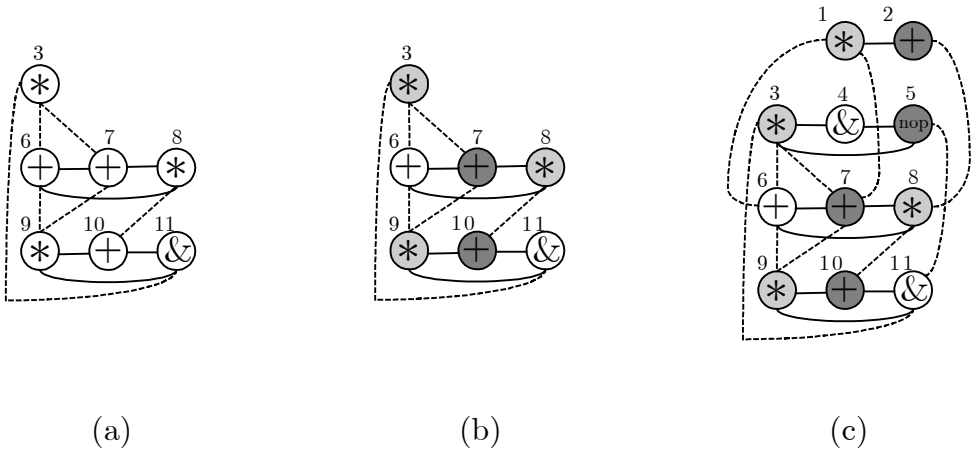



Abbildung 4.12: (a) Interferenzgraph aus Abbildung 4.10 (c) nach dem Entfernen der Knoten 2, 1, 4 und 5 in dieser Reihenfolge; (b) Gefärbter Interferenzgraph nach dem Einfügen der Knoten 3, 7, 6, 8, 9, 10 und 11; (c) Gefärbter Interferenzgraph nach dem Einfügen der verbleibenden Knoten 5, 4, 1 und 2.

Falls es während der Reduktionsphase in Algorithmus 4.7 keinen Knoten mit weniger als $K$ Nachbarn gibt, so existieren unterschiedliche Strategien. Hier wird die optimistische Strategie, die auch in [80] bei der Registerallokation verwendet wird, genutzt. Diese Strategie nimmt an, dass Nachbarknoten eines Knotens die gleiche Farbe erhalten können. Somit ist auch ein Knoten mit $K$ oder mehr Nachbarn potentiell färbbar. Da das aber erst in der Färbungsphase festgestellt werden kann, wird dieser Knoten zunächst

genauso aus dem Graphen entfernt wie Knoten, die sicher färbbar sind. Beispielsweise gibt es in Abbildung 4.12 (a) keinen Knoten mit weniger als drei Nachbarn. Trotzdem wird zunächst Knoten 11 entfernt. Anschließend können die übrigen Knoten entfernt werden, da dann immer wieder ein Knoten mit weniger als drei Nachbarn zu finden ist. Das Verfahren zur Auswahl des Knotens, der als nächstes aus $I$ entfernt wird, ist im Folgenden Abschnitt 4.2.3 näher erläutert. Nach dem Einfügen und Färben der so entfernten Knoten entsteht der gefärbte Interferenzgraph in Abbildung 4.12 (b). Dort kann auch Knoten 11 gefärbt werden, weil die Knoten 3 und 9 dieselbe Farbe bekommen haben.

Es muss beachtet werden, dass ein eingefügter Knoten immer unter Berücksichtigung aller $H$-Kanten färbbar sein muss. Das kann garantiert werden, weil Algorithmus 4.7 während der Reduktionsphase immer einen Knoten finden kann, der mit weniger als $K$ Nachbarknoten durch eine $H$-Kante verbunden ist. Das heißt, die Menge $U$ in Algorithmus 4.7, die nur dann gebildet wird, wenn die Menge $F$ leer ist, enthält immer mindestens einen Knoten.

**Satz 4.6**

In einem entfalteten Interferenzgraphen $I = (c, H \cup S)$ zu einem Ablaufplan $\alpha$ und einem Cluster $c$ gibt es in jeder Iteration während der Reduktionsphase in Algorithmus 4.7 immer einen Knoten $v \in c$, so dass $v$ mit weniger als $K$ Knoten durch eine $H$-Kante verbunden ist.

**Beweis**

Aus der Konstruktion des entfalteten Interferenzgraphen folgt, dass ein Knoten $u$ im Interferenzgraphen immer mit einem Knoten im Ablaufplan korrespondiert und genau dann durch eine $H$-Kante mit einem anderen Knoten $v$ verbunden ist, wenn eine Instruktion $i$ mit $\{u, v\} \subseteq \alpha(i)$ existiert. Außerdem kommen in jeder Instruktion höchstens $K$ viele Operationen vor. Es gibt immer eine Operation in $\alpha$, die in jeder Iteration der ersten *while*-Schleife in Algorithmus 4.7 mit weniger als $K$ Knoten in $c$ durch eine $H$-Kante verbunden ist. Das ist die Operation $v$, so dass für alle anderen Operationen $u$ gilt: $let(v) \leq let(u)$. Angenommen $v$ wäre doch mit $K$ oder mehr Knoten in $c$ durch eine $H$-Kante verbunden, dann muss in den Instruktionen, in denen $v$ ausgeführt wird, mit der Ausführung von mindestens $K$ anderen Operationen begonnen werden. Jede dieser Operationen wird aber auch noch in der Instruktion $let(v)$ ausgeführt, weil sonst $let(v)$ nicht minimal wäre. Damit enthält die Instruktion $let(v)$ aber mindestens $K + 1$ Operationen, was ein Widerspruch ist.

□

Aus Satz 4.6 und Lemma 4.4 ergibt sich unmittelbar die

**Folgerung 4.4**
Unter Verwendung von Algorithmus 4.7 kann jeder Interferenzgraph unter ausschließlicher Berücksichtigung der *H*-Kanten gefärbt werden.

□

### 4.2.3 Knotenauswahl in der Reduktionsphase

In der Reduktionsphase von Algorithmus 4.7 ist in jeder Iteration ein Knoten auszuwählen, der als nächstes aus dem Interferenzgraphen entfernt wird. Dort ist $F$ die Menge, die die Knoten enthält, die mit weniger als $K$ vielen Nachbarn adjazent sind. Ist $F \neq \emptyset$, dann wird der Knoten $v \in F$, für den $opCost(v)$ maximal ist, entfernt. Diese teuren Knoten werden in der folgenden Färbungsphase erst spät eingefügt. Da sie aber sicher färbbar sind, bleibt für sie eine Farbe übrig und alle Konflikte im Interferenzgraphen werden berücksichtigt. Durch das späte Einfügen haben diese Knoten dann aber schon wieder viele gefärbte Nachbarn, wodurch die Auswahl einer Farbe stark eingeschränkt ist. Auf diese Weise können Fehler bei der Farbauswahl für teure Operationsknoten vermieden werden, die dazu führen können, dass Knoten, die nicht sicher gefärbt werden können, tatsächlich nicht gefärbt werden können.

Ist $F = \emptyset$, dann muss ein Knoten ausgewählt werden, für den nicht sichergestellt ist, dass er in der Färbungsphase gefärbt werden kann. Es wird in Algorithmus 4.7 die Menge $U$ gebildet, die alle Knoten des Interferenzgraphen enthält, die mit weniger als $K$ vielen Knoten durch eine *H*-Kante verbunden sind. Nach Satz 4.6 gibt es immer einen solchen Knoten. Es wird dann der Knoten $v \in U$ als nächstes entfernt, für den $opCost(v)$ minimal ist. Das heißt, teure Operationsknoten verbleiben im Interferenzgraphen. Diese können durch das Entfernen eines billigen Operationsknoten $v$ eventuell wieder sicher färbbar werden, so dass nur der billige Operationsknoten in der Färbungsphase eventuell nicht gefärbt werden kann.

### 4.2.4 Behandlung der Knoten in der Färbungsphase

In der Färbungsphase werden die Knoten wieder zum entfalteten Interferenzgraphen zusammengesetzt und dabei wird jedem Knoten eine Farbe zugeordnet. Ein Knoten $v$ ist in der Färbungsphase **färbbar**, wenn

$$\left|\left\{col(u) \middle| \{u, v\} \in H \cup S\right\}\right| < K\,.$$

Das heißt, die zu diesem Zeitpunkt gefärbten Nachbarn von $v$ benutzen noch nicht alle der $K$ möglichen Farben. Einem Knoten können oft mehrere Farben zugeordnet werden, ohne dass die Färbungseigenschaft verletzt wird.

Durch eine ungünstige Färbung kann es vorkommen, dass Knoten, die nicht sicher färbbar waren, tatsächlich nicht gefärbt werden können, obwohl dies eigentlich möglich gewesen wäre. Deshalb werden für jeden Operationstypen die Farben ermittelt, die bereits zum Färben von Knoten dieses Typs verwendet wurden, damit sie, falls möglich, wiederverwendet werden. Es sei $usedCol : \mathcal{O} \to \wp(\{1, \ldots, K\})$ definiert als

$$usedCol(t) := \left\{ col(v) \middle|\ v \in c \text{ und } type(v) = t \right\}.$$

Die Menge der davon noch verwendbaren Farben $availCol : V \to \wp(\{1, \ldots, K\})$ für den nächsten zu färbenden Knoten $v$ ist

$$availCol(v) := usedCol(type(v)) - \left\{ col(w) \middle|\ \{v, w\} \in H \cup S \right\}.$$

Falls ein Knoten $v$ färbbar ist, dann wird für $v$ durch Algorithmus 4.8 eine Farbe aus $availCol(v)$ gewählt oder, falls $availCol(v) = \varnothing$, dann eine Farbe, mit der noch keiner der Nachbarknoten von $v$ gefärbt wurde.

**Algorithmus 4.8 (Auswahl einer Farbe für einen Interferenzgraphknoten $v$)**

1. $NB := \{u \mid \{u,v\} \in H \cup S$ und $u$ wurde noch nicht in den Interferenzgraphen eingefügt und $u$ ist nicht sicher färbbar$\}$.
2. Bestimme für alle Farben $f$ den Wert $count(f)$ durch
$$count(f) := |\{u \mid u \in NB \wedge \exists w : \{u,w\} \in H \cup S \wedge col(w) = f\}|.$$
3. $$F = \begin{cases} availCol(v), \text{ falls } availCol(v) \neq \varnothing \\ \{1,\ldots,K\} - \left\{ col(w) \middle|\ \{v,w\} \in H \cup S \right\}, \text{ sonst} \end{cases}$$
4. Färbe $v$ mit der Farbe $f \in F$, so dass $count(f)$ maximal ist.

In Algorithmus 4.8 enthält die Menge $NB$ alle mit $v$ adjazenten Knoten, deren Färbung sich während der weiteren Abarbeitung der Färbungsphase als schwierig erweisen kann, weil sie nicht sicher färbbar sind und noch nicht in den Interferenzgraphen eingefügt wurden. $F$ enthält alle Farben, die zur Färbung des Knotens $v$ noch benutzt werden können. Durch $count$ wird für jede dieser Farben ermittelt, wie viele Knoten aus $NB$ einen Nachbarn besitzen, der bereits mit dieser Farbe gefärbt wurde. Die Färbbarkeit eines Nachbarn von $v$ aus der Menge $NB$ wird nicht eingeschränkt, wenn $v$ auch mit dieser Farbe gefärbt wird. Es wird deshalb für $v$ die Farbe gewählt, die die Färbbarkeit der meisten seiner noch ungefärbten Nachbarn in $NB$ nicht einschränkt.

Falls ein Knoten $v$ in der Färbungsphase nicht färbbar ist, dann müsste ihm eine Farbe zugewiesen werden, die bereits einer seiner Nachbarn erhalten hat. Diese Farbe muss aber verschieden von den Farben sein, die die Nachbarn, mit denen er über eine $H$-Kante verbunden ist, erhalten haben. Dieses Vorgehen liefert zwar eine Bindung, birgt aber ein Problem in sich.

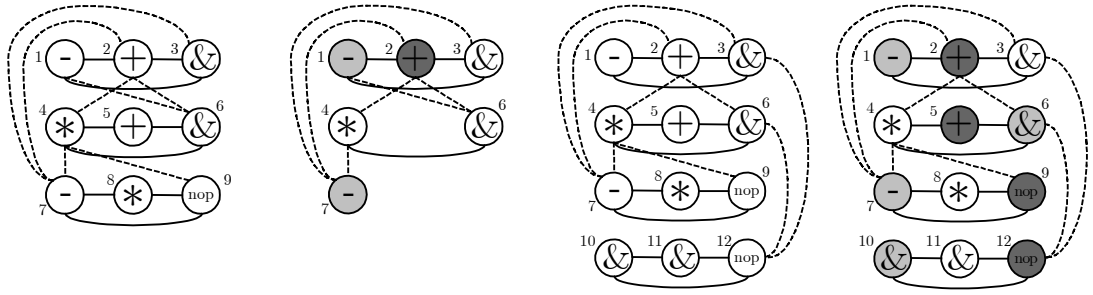
Durch die Verwendung der zusätzlichen Farbe muss eine weitere funktionale Einheit den betreffenden Operator bereitstellen. Damit steht für den gesamten Ablaufplan diese zusätzliche FU zur Ausführung von Operationen des betreffenden Typs zur Verfügung. In der Konsequenz könnten viele Konflikte im Interferenzgraphen, die durch *S*-Kanten modelliert werden, entfallen. Das passiert aber nicht, wenn die Färbungsphase an dieser Stelle fortgesetzt wird, da sie auf dem ursprünglichen, jetzt mit überflüssigen Konflikten versehenen, Interferenzgraphen beruht. Um diese überflüssigen Konflikte zu eliminieren, wird der Interferenzgraph neu erzeugt. Bei dieser Konstruktion muss berücksichtigt werden, dass der betreffende Operationstyp $t = type(v)$ jetzt von mehr als $TCWidth(\alpha, c, t)$ vielen funktionalen Einheiten unterstützt wird. Modelliert wird das durch eine zusätzliche Instruktion $\zeta_t$ im Ablaufplan, die nur für die erneute Konstruktion des Interferenzgraphen benötigt wird und als

$$\zeta_t = \{v_1,\ldots,v_m\} \cup \{v_{m+1},\ldots,v_n\}$$

definiert ist, wobei

- $m-1$ die Anzahl der funktionalen Einheiten ist, die den Operationstypen $t$ im zuletzt konstruierten entfalteten Interferenzgraphen unterstützt haben,
- $n = FUCWidth(\alpha, c)$,
- $v_1, \ldots v_n \notin V$, $type(v_1) = t, \ldots, type(v_m) = t$ und $type(v_{m+1}) = nop, \ldots, type(v_n) = nop$.

$\zeta_t$ wird als neue Instruktion dem Ablaufplan $\alpha'$ hinzugefügt oder ersetzt eine eventuell bereits vorhandene Instruktion $\zeta_t$. In $\zeta_t$ gibt es jetzt einen Knoten mehr, der den Typ $t$ hat, als in der bisherigen maximalen Instruktion des Typs $t$. Alle Knoten des Typs $t$, die nicht zur maximalen Instruktion von $t$ gehören, haben damit im neuen Interferenzgraphen eine *S*-Kante weniger. In Abbildung 4.13 ist ein Beispiel angegeben. In dieser Abbildung ist in (a) der Interferenzgraph für die ersten drei Instruktionen des Ablaufplans in Abbildung 4.5 dargestellt. Um den Interferenzgraphen mit drei Farben zu färben, werden zunächst die Knoten 5, 8 und 9 entfernt. Danach haben alle Knoten noch mindestens drei Nachbarn. Der nächste Knoten, der entfernt wird, ist der Knoten 6. Anschließend können alle weiteren Knoten als sicher färbbar entfernt und wieder eingefügt werden, bis sich der in (b) dargestellte, teilweise gefärbte Interferenzgraph ergibt, in dem der Knoten 6 nicht färbbar ist. Für den Operationstyp & wird deshalb eine neue maximale Instruktion, bestehend aus den Knoten 10, 11 und 12, erzeugt und ein neuer Interferenzgraph konstruiert, der in (c) dargestellt ist. Dieser Graph kann dann problemlos reduziert und gefärbt werden, womit sich der in (d) dargestellte Interferenzgraph ergibt.




(a) (b) (c) (d)

Abbildung 4.13: (a) Interferenzgraph zu den ersten drei Instruktionen aus Abbildung 4.5 (a); (b) Teilweise gefärbter Interferenzgraph, in dem Knoten 6 keine Farbe zugewiesen werden kann; (c) Interferenzgraph aus (a), der um die Knoten 10, 11 und 12 für die neue maximale Instruktion $\zeta_{\&}$ erweitert wurde; (d) Fertig gefärbter Interferenzgraph aus (c).

Die erneute Konstruktion des Interferenzgraphen kann mehrmals erforderlich sein. In Algorithmus 4.9 ist das Verfahren zusammenfassend dargestellt.

**Algorithmus 4.9 (Ressourcenallokation)**

**Eingabe**: Ablaufplan $\alpha$
**Ausgabe**: gefärbter entfalteter Interferenzgraph

$\alpha'$:=$\alpha$ wird um nop-Operationen erweitert.
Konstruiere entfalteten Interferenzgraphen I für $\alpha'$
Färbe I mit Algorithmus 4.7
**while**(Algorithmus 4.7 stoppt mit nicht färbbarem Knoten v) **do**
  Bestimme einen Knoten w mit w = v oder {v,w} ∈ S, so dass
  TICWidth($\alpha'$($m_{type(w)}$),[w]$_\chi$,type(w)) < FUCWidth($\alpha'$,[w]$_\chi$)
  Erzeuge eine Instruktion $\zeta_{type(w)}$
  Füge $\zeta_{type(w)}$ in $\alpha'$ ein und ersetze die eventuell bereits
  vorhandene Instruktion $\zeta_{type(w)}$
  Konstruiere entfalteten Interferenzgraphen I für $\alpha'$
  Färbe I mit Algorithmus 4.7
**od**

Wenn Algorithmus 4.7 mit dem nicht färbbaren Knoten $v$ stoppt, dann muss es nicht immer zweckmäßig sein, eine neue maximale Instruktion für $type(v)$ einzufügen. Eventuell ist es sinnvoller, einen mit $v$ adjazenten Knoten dafür auszuwählen. In der verwendeten Implementierung wird jedoch $v$ ausgewählt oder, wenn für $v$ bereits eine maximale Instruktion $\zeta_{type(v)}$ ohne *nop*-Operationen existiert, ein Knoten $w$, der mit $v$ über eine $S$-Kante adjazent ist. Wenn $v$ nicht färbbar ist, dann muss $v$ wegen Folgerung 4.4 mit einer

*S*-Kante adjazent sein. Außerdem können nicht $v$ und $w$ Knoten sein, so dass $\zeta_{type(v)}$ und $\zeta_{type(w)}$ keine *nop*-Operationen mehr enthalten, weil es dann keine *S*-Kante zwischen $v$ und $w$ geben würde. Auch wenn die Konstruktion des entfalteten Interferenzgraphen mehrfach wiederholt werden muss, so gilt

**Lemma 4.5**
Im Cluster $c$ wird höchstens $FUCWidth(\alpha, c) - 1$ mal für einen Knoten des Typs $t$ eine maximale Instruktion $\zeta_t$ erzeugt.

**Beweis**
Damit überhaupt eine Instruktion $\zeta_t$ erzeugt wird, muss $TCWidth(\alpha, c, t) \geq 1$ sein. Spätestens nachdem $FUWidth(\alpha, \chi) - 1$ mal eine Instruktion $\zeta_t$ neu erzeugt wurde, enthält $\zeta_t$ genau $FUCWidth(\alpha, c)$ viele Operationen des Typs $t$ und in Algorithmus 4.9 wird für diesen Typ nicht erneut eine maximale Instruktion erzeugt.

□

Daraus ergibt sich

**Satz 4.7**
Algorithmus 4.9 stoppt.

**Beweis**
Aus Lemma 4.5 folgt, dass für jeden Operationstypen $t$ in Cluster $c$ höchstens $FUCWidth(\alpha, c)$ mal eine neue Instruktion $\zeta_t$ eingefügt wird. Spätestens dann, haben alle maximalen Instruktionen die Form $\zeta_t = \{v_1, ..., v_n\}$, wobei $type(v_k) = t$ für alle $1 \leq k \leq n$ und $n = FUCWidth(\alpha, c)$. Für alle Instruktionen $0 \leq i < l$ gibt es somit keine *S*-Kanten zwischen den Knoten in Instruktion $i$ und den Knoten in den Instruktion $\zeta_t$. Damit existieren im konstruierten Interferenzgraphen keine *S*-Kanten mehr. Aus Folgerung 4.4 ergibt sich damit, dass der Interferenzgraph spätestens jetzt färbbar ist.

□

## 4.3 Clusterung

Bisher wurde eine Clusterung der Operationsknoten in den Basisblöcken als gegeben vorausgesetzt. Dieser Abschnitt beschäftigt sich mit der Bestimmung einer Clusterung $\chi$ für einen Basisblock $b$ bei einer vorgegebenen Ablaufplanlänge $l$ und einer vorgegebenen Clusteranzahl *maxC*. Da die in Abschnitt 3.2.9 angegebenen Clusterungstechniken ressourcenbeschränkt sind und somit die Einhaltung einer vorgegebenen Ablaufplanlänge $l$ nicht garantieren, wurden verschiedene Varianten für eine zeitbeschränkte Clusterung betrachtet:

- Clusterung nach der Ablaufplanung,
- Clusterung vor der Ablaufplanung,
- Gekoppelte Clusterung und Ablaufplanung.

Das Optimierungsziel bei allen Clusterungsvarianten ist die Minimierung der Portanzahl in der größten Registerbank und die Minimierung der Anzahl aller Ports in allen Registerbänken. Zur Lösung dieses $\mathcal{NP}$-vollständigen Problems [12] werden Heuristiken genutzt.

### 4.3.1 Clusterung nach der Ablaufplanung

Diese Variante der Clusterung wurde in zwei Studienarbeiten untersucht [19, 54]. Der Ablaufplan wurde für eine ungeclusterte Architektur mit dem Planungsalgorithmus aus Abschnitt 4.1 erzeugt. In den Studienarbeiten wurden Heuristiken entwickelt, um die Operationen im Ablaufplan in eine vorgegebene Anzahl von Clustern zu zerlegen, ohne dass ihre Ausführungszeitpunkte verändert werden. Obwohl bei diesem Ansatz die Ablaufplanung und Clusterung völlig ungekoppelt blieben, waren die erzielten Ergebnisse ähnlich gut wie die, in der hier als Vergleich zu Grunde gelegten Arbeit von Lapinskii [99], in der die Clusterung vor der Ablaufplanung durchgeführt wird. Die Ergebnisse der Studienarbeiten konnten durch die anderen beiden untersuchten Clusterungsvarianten nochmals verbessert werden. Der Vorteil, die Clusterung erst nach der Ablaufplanung durchzuführen, liegt in der geringen Laufzeit des Clusterungsalgorithmus.

### 4.3.2 Clusterung vor der Ablaufplanung

In der Diplomarbeit [12] wurden Techniken entwickelt, um eine zeitbeschränkte Clusterung vor der Ablaufplanung durchzuführen. Für den so geclusterten Basisblock wurde dann mit dem vorgestellten Planungsalgorithmus ein Ablaufplan erzeugt. Der in [12] entwickelte KC-Algorithmus zur Clusterung berechnet eine minimale Kettenzerlegung der Knoten des Basisblocks. Diese Ketten werden dann auf eine vorgegebene Anzahl von Clustern verteilt. Dabei werden die benötigten Kopieroperationen eingefügt und somit sichergestellt, dass eine vorgegebene Ablaufplanlänge auch vom geclusterten Basisblock eingehalten wird. Der KC-Algorithmus untersucht alle möglichen Anordnungen der Ketten in Clustern. Um die Qualität der Anordnung zu bestimmen, wurde eine Schätzfunktion entwickelt, die in kurzer Zeit die Anzahl der benötigten Ports in jeder Registerbank bei einer gegebenen Kettenanordnung abschätzt. Damit die berechnete Kettenzerlegung die Möglichkeiten der Clusterung nicht zu sehr dominiert, werden durch den Clusterungsalgorithmus auch Anordnungen betrachtet, bei denen die Ketten in Teilketten zerlegt und diese Teilketten unterschiedlichen Clustern zugeordnet werden.

Die mit diesem Ansatz erzeugten Architekturen waren kostengünstiger als die in der Arbeit von Lapinskii [99]. Weil aber alle Kettenanordnungen betrachtet werden, hat der Clusterungsalgorithmus eine exponentielle Laufzeit, die von der Cluster- und der Kettenanzahl abhängt. Trotzdem konnten Ergebnisse für die meisten der in Abschnitt 6.2 angegebenen Basisblöcke mit bis zu drei Clustern in wenigen Minuten ermittelt werden [13].

### 4.3.3 Gekoppelte Clusterung und Ablaufplanung

Bei der gekoppelten Clusterung und Ablaufplanung werden die Wechselwirkungen zwischen Clusterung und Ablaufplanung beachtet, indem die Clusterung und Ablaufplanung abwechselnd für eine gegebene Ablaufplanlänge und Clusteranzahl durchgeführt werden. Informationen, die aus dem erzeugten Ablaufplan gewonnen werden, werden genutzt, um die Clusterung zu verbessern. Beim Entwurf des Clusterungsalgorithmus wurde neben den bereits genannten Optimierungszielen darauf geachtet, dass

- auch für sehr kurze Ablaufplanlängen eine Clusterung gefunden wird und
- ohnehin benötigte Kopieroperatoren genutzt werden, um Werte zwischen Clustern zu kopieren, wenn dadurch Operatoren in einigen Clustern eingespart werden können.

Um beide Ziele umzusetzen wurde ein Clusterungsalgorithmus entworfen, der sich von den in Abschnitt 3.2.9 vorgestellten Verfahren dadurch unterscheidet, dass er zeitbeschränkt ist. Das heißt, es ist eine feste Ablaufplanlänge vorgegeben, die der Ablaufplan des geclusterten Basisblocks nicht überschreiten darf. Die Zerlegung der Knoten des Basisblocks in $maxC$ viele Cluster wird in zwei Stufen durchgeführt:

- In der **ersten Stufe** wird eine bestehende Clusterung verbessert, indem vollständig Pfade des Basisblocks aus einem Cluster in einen anderen Cluster verschoben werden.
- In der **zweiten Stufe** wird eine bestehende Clusterung verbessert, indem nur noch Teile eines Pfades des Basisblocks aus einem Cluster in einen anderen Cluster verschoben werden.

Die in beiden Stufen genutzten Verfahren unterscheiden sich nur an wenigen Stellen und beide Verfahren nutzen als Grundprinzip die iterative Verbesserung einer bestehenden Clusterung und die gekoppelte Ausführung der Clusterung mit der Ablaufplanung. Die Kopplung wird erreicht, indem nach einer Clusterung eine Ablaufplanung durchgeführt wird, um im Ablaufplan die Stellen zu identifizieren, an denen sich Operationen befinden, die in einen anderen Cluster verschoben werden. Die zwei Schritte der Planung und Clusterung werden wiederholt ausgeführt. In Abbildung 4.14 ist dieses

Grundprinzip, das in der ersten und zweiten Stufe der Clusterung Anwendung findet, dargestellt.

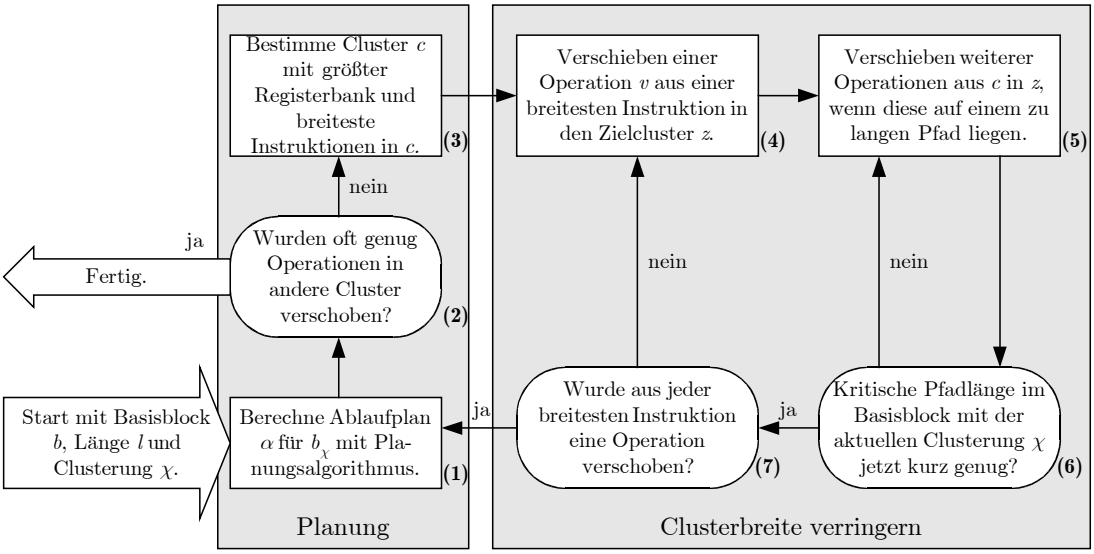



Abbildung 4.14: Schematische Darstellung der Kopplung von Ablaufplanung und Clusterung. Die einzelnen Schritte sind durch die Zahlen in runden Klammern nummeriert.

Nach dem Start wird in Schritt 1 zunächst ein Ablaufplan $\alpha$ für die gegebene Ablaufplanlänge $l$ berechnet. Wenn $l$ die kritische Pfadlänge im ungeclusterten Basisblock $b$ ist, dann sind bei dieser Planung alle Knoten demselben Cluster zugeordnet. Ist $l$ nicht die kritische Pfadlänge, dann existiert bereits eine Clusterung $\chi$ für $b$, die zur Ablaufplanlänge $l-1$ ermittelt wurde und es werden in Schritt 1 die Operationen des bereits geclusterten Basisblocks $b_\chi$ verplant. Ausgehend von dem so erstellten Ablaufplan $\alpha$ wird in Schritt 3 der Cluster $c$ bestimmt, für den $FUCWidth(\alpha, c)$ am größten ist. Dieser Cluster besitzt die größte Registerbank und es wird die Menge $BI$ der Instruktionen bestimmt, in denen die meisten Operationen gleichzeitig in $c$ ausgeführt werden:

$$BI = \left\{ \alpha(i) \,\middle|\, FUCIWidth\big(\alpha(i), c\big) = FUCWidth\big(\alpha, c\big) \right\}.$$

Aus jeder dieser Instruktionen wird mindestens eine Operation in einen anderen Cluster verschoben, um die Größe der Registerbank im Cluster $c$ zu verringern. Dafür werden die Schritte 4, 5, 6 und 7 iterativ ausgeführt. In Schritt 4 wird eine Instruktion aus $BI$ ausgewählt und eine der darin vorkommenden Operationen $v$ in einen Zielcluster $z$, der verschieden von $c$ sein muss, verschoben. Es entsteht eine neue Clusterung $\chi'$, die zu neuen Kopieroperationen in $b_{\chi'}$ und somit auch zu längeren Pfaden in $b_{\chi'}$ geführt haben kann. Solange es in $b_{\chi'}$ noch Pfade gibt, deren Länge größer als $l$ ist, werden durch wiederholte Ausführung der Schritte 5 und 6 weitere Operationen, die auf diesen Pfaden liegen, in den Zielcluster $z$ verschoben. Dadurch werden diese zu langen Pfade verkürzt, weil Kopieroperationen

entfallen. Um in Schritt 4 die zu verschiebende Operation $v$ aus einer Instruktion in $BI$ und den besten Zielcluster $z$ für $v$ zu bestimmen, wird aus der gewählten Instruktion jede Operation $v$ probeweise in jeden möglichen Zielcluster verschoben und für jede dieser Operationen die bereits beschriebenen Schritte 5 und 6 solange ausgeführt, bis alle Pfade in $b_{\chi'}$ höchstens die Länge $l$ haben. Bevor zu Schritt 7 übergegangen wird, wird für jede so berechnete Clusterung $\chi'$ eine Zwischenbewertung vorgenommen. Mit der Clusterung, die die beste Zwischenbewertung erhalten hat, wird in Schritt 7 fortgefahren. Da die in Schritt 5 und 6 mit $v$ gemeinsam verschobenen Operationen zu Instruktionen aus $BI$ gehört haben können, wird nach Schritt 7 nur dann Schritt 4 ausgeführt, wenn es noch eine Instruktion in $BI$ gibt, aus der noch keine Operation von $c$ in einen anderen Cluster verschoben wurde. Erst wenn aus jeder Instruktion in $BI$ mindestens eine Operation in einen anderen Cluster verschoben wurde, wird zu der so entstandenen Clusterung eine erneute Ablaufplanung in Schritt 1 durchgeführt. Die Bewertung der neuen Clusterung ergibt sich aus der Bewertung des erzeugten Ablaufplans mittels der Zielfunktionen *RBLoad*, *TLoad* und *TDLoad*. Außerdem werden in dem Ablaufplan die Stellen identifiziert, an denen die Verbesserung der Clusterung fortgesetzt wird. Unabhängig davon, ob die so entstandene neue Clusterung besser ist als die Clusterung mit der die Iteration in Schritt 1 startete, wird mit der neuen Clusterung fortgefahren. Konnte innerhalb mehrerer Iterationen die bis dahin beste gefundene Clusterung nicht mehr verbessert werden, dann stoppt der Clusterungsalgorithmus in Schritt 2 mit der besten bis dahin gefundenen Clusterung als Ergebnis. Die dargestellte Iteration wird sowohl in der ersten als auch in der zweiten Stufe durchlaufen. Mit der Clusterung, mit der nach der ersten Stufe in Schritt 2 gestoppt wird, wird die zweite Stufe in Schritt 1 wieder gestartet. Beide Stufen unterscheiden sich darin, wie die Pfade in Schritt 5 verkürzt werden.

In den folgenden Abschnitten werden die einzelnen Schritte des in Abbildung 4.14 dargestellten Prinzips detailliert erläutert. Insbesondere wird die Wahl des Zielclusters, die Verkürzung der kritischen Pfade in beiden Stufen und die Bewertung der entstehenden Clusterungen beschrieben.

#### 4.3.3.1 Verkürzen der kritischen Pfade

Der zentrale Punkt des Clusterungsalgorithmus ist die Schleife zwischen den Schritten 5 und 6. Sie dient dem Verkürzen von Pfaden im geclusterten Basisblock, die Kopieroperationen enthalten. Pfade können sich nach dem Verschieben einer Operationen $v$ in den Zielcluster $z$ (Schritt 4 und 5) und dem damit verbundenen Einfügen notwendiger Kopieroperationen verlängern. Um eine Abarbeitung des Basisblocks innerhalb der vorgegebenen Länge $l$ zu gewährleisten, müssen alle Pfade, deren Länge größer als $l$ ist, wieder verkürzt werden. Das kann nur dann erreicht werden, wenn einige oder alle Kopieroperationen auf diesen Pfaden eliminiert werden. Beim Eliminieren der

Kopieroperationen auf diesen Pfaden soll die Zuordnung des ursprünglich in den Zielcluster $z$ verschobenen Knotens $v$ nicht angezweifelt werden. Das bedeutet, dass Kopierknoten auf diesen Pfaden eliminiert werden können, wenn andere Knoten dieser Pfade ebenfalls in den Zielcluster $z$ verschoben werden. Zunächst wird ein Auswahlkriterium angegeben, mit dem der nächste zu verkürzende Pfad $\pi$ in einem Basisblock $b_\chi$ bestimmt wird:

**Auswahlkriterium für einen Pfad $\pi$**

- $\pi$ ist ein kritischer Pfad in $b_\chi$ und
- $\pi$ enthält von allen kritischen Pfaden $\pi_1, \ldots, \pi_n$ in $b_\chi$ die maximale Anzahl Kopieroperationen, deren Vorgänger oder Nachfolger im Zielcluster $z$ liegen. Das heißt,

$$\forall 1 \leq i \leq n : \#copy(\pi, z) \geq \#copy(\pi_i, z)\text{, wobei}$$

$$\#copy(v_1 \ldots v_k, z) = \left|\left\{ v_i \;\middle|\; type(v_i) = copy \wedge \left(v_{i-1} \in z \vee v_{i+1} \in z\right) \wedge 1 < i < k \right\}\right|.$$

□

**Lemma 4.6**
Es sei $v$ die Operation des Basisblocks $b$, die in Schritt 4 (Abbildung 4.14) dem Cluster $z$ zugeordnet wurde und durch die entstehende Clusterung $\chi$ ist $cp(b_\chi) > l$. Dann enthält der mit dem Auswahlkriterium gewählte Pfad mindestens eine der neu eingefügten Kopieroperationen.

**Beweis**
Vor dem Verschieben einer Operation $v$ in den Zielcluster $z$ war kein Pfad im geclusterten Basisblock länger als $l$. Danach hat der ausgewählte kritische Pfad $\pi$ eine Länge größer als $l$. Ein Pfad dieser Länge konnte nur durch eingefügte Kopieroperationen entstehen. Also muss dieser Pfad eine der neu eingefügten Kopieroperationen enthalten.

□

Nach der Wahl des zu verkürzenden Pfades $\pi$ in $b_\chi$ müssen Operationen auf diesem Pfad in den Zielcluster $z$ verschoben werden, um Kopieroperationen zu eliminieren. Dafür kommen nur solche Operationen $u$ in Betracht, die entweder unmittelbarer Vorgänger einer Kopieroperation sind, die das Ergebnis von $u$ in den Zielcluster $z$ kopiert, oder $u$ ist unmittelbarer Nachfolger einer Kopieroperation, die einen Wert aus $z$ in den Cluster $[u]_\chi$ kopiert. Somit markieren solche Kopieroperationen, die Werte in oder aus $z$ kopieren, die signifikanten Stellen im Pfad $\pi$, an denen Kopieroperationen eliminiert werden können. Durch das Ändern der Clusterzuordnung der Vorgänger bzw. Nachfolger solcher Kopieroperationen bewegen sich die Kopieroperationen entlang des Pfades $\pi$. Ein solcher Pfad $\pi$ hat im Allgemeinen die Form $v_1 \ldots v_{n-1} v_n k v_{n+1} v_{n+2} \ldots v_m$, wobei $k$ eine Kopieroperation ist und die übrigen Operationen in $\pi$ einen beliebigen Typ haben können. Für

das Verschieben der Kopieroperation $k$ in $\pi$ ergeben sich somit zwei Bewegungsregeln:

- Ist der Nachfolger $v_{n+1}$ von $k$ dem Zielcluster $z$ zugeordnet, dann wird der Vorgänger $v_n$ von $k$ ebenfalls dem Zielcluster $z$ zugeordnet und $k$ bewegt sich entgegen der Kantenrichtung auf dem Pfad $\pi$ und wird zum Vorgänger von $v_n$. $\pi$ hat dann die Form $v_1 \ldots v_{n-1} k v_n v_{n+1} \ldots v_m$.
- Gehört der Vorgänger $v_n$ von $k$ zum Zielcluster $z$, dann wird der Nachfolger $v_{n+1}$ dem Zielcluster $z$ zugeordnet und $k$ bewegt sich mit der Kantenrichtung auf dem Pfad $\pi$, wodurch $k$ zum Nachfolger von $v_{n+1}$ wird und $\pi$ dann die Form $v_1 \ldots v_n v_{n+1} k v_{n+1} \ldots v_m$ hat.

Durch diese zwei Bewegungsregeln lassen sich die Kopieroperationen im Pfad $\pi$ verschieben. Im Allgemeinen bleibt dadurch aber die Länge des Pfades $\pi$ erhalten. Sie wird nur dann verkürzt, wenn vor dem Verschieben von $k$ eine der vier folgenden Situationen eintritt:

- $k$ wird entgegen der Kantenrichtung bewegt und der Vorgänger von $k$ hat als Vorgänger wieder eine Kopieroperation. Das heißt, $type(v_{n-1}) = copy$ oder
- $k$ wird mit der Kantenrichtung bewegt und der Nachfolger von $k$ hat als Nachfolger eine Kopieroperation. Das heißt, $type(v_{n+2}) = copy$ oder
- $k$ wird entgegen der Kantenrichtung bewegt und der Vorgänger von $k$ hat keinen Vorgänger. Es ist also $n = 1$. Oder
- $k$ wird mit der Kantenrichtung bewegt und der Nachfolger von $k$ hat keinen Nachfolger. Es gilt also $n + 1 = m$.

Liegt die erste oder zweite Situation vor und $v_{n-1}$ bzw. $v_{n+2}$ sind Kopieroperationen, die einen Wert in den Zielcluster oder aus dem Zielcluster kopieren, dann verringert sich die Länge des Pfades $\pi$ um den Wert $2 \cdot lat(copy)$. Ansonsten verringert sich die Länge des Pfades $\pi$ um den Wert $lat(copy)$.

Eine Kopieroperation wird mit den zwei Bewegungsregeln solange verschoben, bis die Länge des Pfades $\pi$ aufgrund der Verkürzungsregeln kleiner oder gleich $l$ ist. Dieser Vorgang wird solange wiederholt, bis alle Pfade im geclusterten Basisblock wieder eine Länge kleiner oder gleich $l$ haben. In Abbildung 4.15 ist ein Beispiel angegeben.

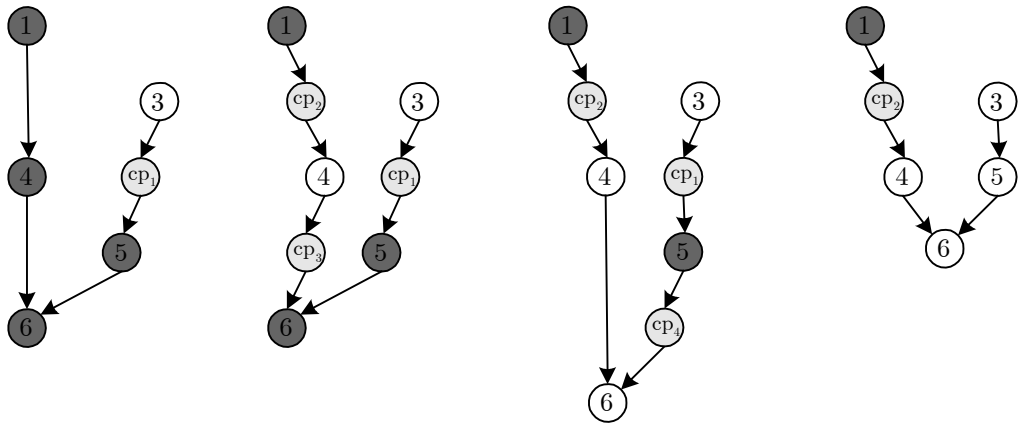



Abbildung 4.15: (a) Basisblock zerlegt in zwei Cluster; (b) Operation 4 aus Basisblock in (a) wurde in den Zielcluster verschoben und benötigte Kopieroperationen eingefügt; (c) Operation 6 wurde in den Zielcluster verschoben; (d) Operation 5 wurde in den Zielcluster verschoben.

Der Basisblock in (a) ist in zwei Cluster zerlegt, dargestellt durch weiße und dunkelgraue Knoten. Die notwendige Kopieroperation ist hellgrau dargestellt. Alle Operationen haben die Latenzzeit eins. Es wird angenommen, dass durch den Clusterungsalgorithmus, wie er in Abbildung 4.14 angegeben ist, in Schritt 4 der Knoten 4 in den weißen Zielcluster verschoben wird. Dann ergibt sich der in (b) dargestellte geclusterte Basisblock. Weiterhin sei nur eine Ablaufplanlänge von vier zugelassen. Durch die eingefügten Kopieroperationen ist die kritische Pfadlänge in (b) aber auf fünf angewachsen und es muss eine Kopieroperation auf dem Pfad $1\ cp_2\ 4\ cp_3\ 6$ eliminiert werden. Es kann dafür als nächstes die Kopieroperation $cp_3$ ausgewählt werden. Da die Zuordnung des Knotens 4 nicht angezweifelt werden soll, führt das Bewegen dieser Kopieroperation zur Zuordnung des Knotens 6 in den weißen Cluster. Außerdem entsteht zwischen Knoten 5 und 6 die neue Kopieroperation $cp_4$, wie in (c) dargestellt. Damit ist zwar die Länge des ursprünglichen Pfades $1\ cp_2\ 4\ cp_3\ 6$ um eins verkürzt worden, der Pfad $3\ cp_1\ 5\ cp_4\ 6$ hat jetzt aber die Länge fünf. In (d) ist der geclusterte Basisblock dargestellt, der nach dem Verschieben der Kopieroperation $cp_4$ entsteht. Da der Vorgänger von $cp_4$ nicht zum weißen Zielcluster gehört, wird die Kopieroperation entgegen der Kantenrichtung verschoben und Knoten 5 wird dem weißen Cluster zugeordnet. Dieselbe Clusterzuordnung wäre entstanden, wenn statt der Operation $cp_4$ die Operation $cp_1$ verschoben worden wäre. Die Länge des Pfades $3\ cp_1\ 5\ cp_4\ 6$ hat sich durch das Verschieben von $cp_4$ um den Wert $2 \cdot lat(copy)$ verringert.

Wie in dem Beispiel dargestellt wurde, können durch das Verschieben von Kopieroperationen auf Pfaden, die gemeinsame Knoten mit dem zu verkürzenden Pfad $\pi$ besitzen, weitere Kopieroperationen entstehen. Das ist immer dann der Fall, wenn der Knoten des Pfades $\pi$, der durch das Verschieben einer Kopieroperation dem Zielcluster zugeordnet wurde, weitere Vorgänger oder Nachfolger in dem Cluster hat, aus dem er verschoben wurde. Diese Kopieroperationen können andere Pfade verlängern. Es ist deshalb nicht zweckmäßig, immer alle Kopieroperationen auf einem Pfad zu eliminieren, weil dadurch zum einen die einzuhaltende Pfadlänge $l$ deutlich unterschritten werden könnte, was nicht dem Optimierungsziel entspricht, und zum anderen zahlreiche weitere Kopieroperationen auf anderen Pfaden entstehen können. Es wird deshalb ein Auswahlkriterium für Kopieroperationen angegeben, mit dem eine einzelne vom Pfad $\pi$ zu eliminierende Kopieroperation bestimmt wird. Dadurch wird es möglich, die Länge eines Pfades schrittweise zu verkürzen. Zunächst wird ein Teilpfad von $\pi$, der zum Zielcluster $z$ gehört, definiert:

**Definition 4.6 (Teilpfad im Cluster *z*)**
Eine Knotenfolge $v_i v_{i+1} \ldots v_k$ des Pfades $\pi = v_1 \ldots v_n$ wird bei einer Clusterung $\chi$ Teilpfad im Cluster $z$ genannt, falls

- $\forall j \in \mathbb{N}$: $i \leq j \leq k \Rightarrow v_j \in z$ und
- $\forall j \in \mathbb{N}$: $i < j < k \Rightarrow type(v_j) \neq copy$ und
- $(i = 1 \vee type(v_{i-1}) = copy)$ und
- $(k = n \vee type(v_{k+1}) = copy)$.

□

Ein solcher Teilpfad ist eine Folge von Knoten des Pfades $\pi$, die alle dem Cluster $z$ zugeordnet sind. Ein Teilpfad im Cluster $z$ wird genutzt, um die nächste zu eliminierende Kopieroperation auszuwählen.

**Auswahlkriterium für eine Kopieroperation**
Es sei $\pi = v_1 \ldots v_n$ ein zu verkürzender Pfad in $b_\chi$. Für einen kürzesten Teilpfad $v_i \ldots v_k$ von $\pi$ im Cluster $z$ mit $1 \leq i \leq k \leq n$ wird als Kopieroperation $v_{k+1}$ ausgewählt, falls $k \neq n$. Ansonsten wird $v_{i-1}$ ausgewählt.

□

Durch dieses Kriterium wird eine Kopieroperation ausgewählt, die besonders kurze Berechnungen im Zielcluster einleitet oder beendet. Durch das Eliminieren dieser Kopieroperation werden ihre Vorgänger- bzw. Nachfolgerknoten in den Zielcluster verschoben und dadurch die Anzahl der Berechnungen im Zielcluster auf diesem kurzen Teilpfad erhöht. Gleichzeitig wird die Länge des Pfades $\pi$ verringert. In der ersten Stufe der Clusterung werden alle Kopieroperationen auf dem Pfad $\pi$ eliminiert, die einen Wert in den oder aus dem Zielcluster kopieren. Auf diese Weise werden alle Pfade im

geclusterten Basisblock solange verkürzt, bis die kritische Pfadlänge die vorgegebene Ablaufplanlänge $l$ nicht mehr überschreitet. In Algorithmus 4.10 ist dieses Verfahren angegeben, das in der ersten Stufe der Clusterung der Schleife zwischen Schritt 5 und 6 in Abbildung 4.14 entspricht.

**Algorithmus 4.10 (Verkürzen der Pfade im Basisblock in Stufe 1)**

```
Eingabe: geclusterter Basisblock b
         Zielcluster z
         Ablaufplanlänge l
Ausgabe: geclusterter Basisblock b', mit cp(b') ≤ l

while (es existiert in b ein Pfad π mit einer Länge gößer l) do
  Wähle Pfad π nach dem Pfadkriterium
  repeat
    Wähle eine Kopieroperation k aus π nach dem Kopierkriterium
    Verschiebe k mittels der Bewegungsregeln, bis k eliminiert ist
  until(π enthält keine Kopieroperation mehr, die in oder aus z
        kopiert)
od
return b
```

In der zweiten Stufe der Clusterung werden Kopieroperationen nur solange auf einem Pfad eliminiert, bis die Länge des Pfades kleiner oder gleich $l$ ist. Das entsprechende Verfahren, das in der zweiten Stufe der Clusterung der Schleife zwischen Schritt 5 und 6 entspricht, ist in Algorithmus 4.11 angegeben.

**Algorithmus 4.11 (Verkürzen der Pfade im Basisblock in Stufe 2)**

```
Eingabe: geclusterter Basisblock b
         Ablaufplanlänge l
Ausgabe: geclusterter Basisblock b', mit cp(b') ≤ l

while (es existiert ein Pfad π mit einer Länge gößer l) do
  Wähle Pfad π nach dem Pfadkriterium
  repeat
    Wähle eine Kopieroperation k aus π nach dem Kopierkriterium
    Verschiebe k mittels der Bewegungsregeln, bis k eliminiert ist
  until(pl(π) ≤ l)
od
return b
```

Um sicherzustellen, dass in jedem durch das Pfadkriterium ausgewählten Pfad auch eine Kopieroperation durch das Auswahlkriterium für Kopieroperationen gefunden werden kann, wird gezeigt:

**Lemma 4.7**
Jeder durch das Auswahlkriterium für Pfade gefundene Pfad enthält mindestens eine Kopieroperation, deren Vorgänger oder Nachfolger zum Zielcluster $z$ gehört.

**Beweis**
Wegen Lemma 4.6 gilt die Behauptung für den durch das Auswahlkriterium für Pfade gewählten Pfad, nachdem ein Knoten $v$ in Schritt 4 (Abbildung 4.14) in den Zielcluster $z$ verschoben wurde. Außerdem erzeugt das Verschieben einer Kopieroperation, die einen Wert in oder aus $z$ kopiert, nur Kopieroperationen im Basisblock, die wiederum Vorgänger oder Nachfolger in $z$ haben. Es können also durch das Verkürzen von Pfaden mit Algorithmus 4.10 und Algorithmus 4.11 keine Kopieroperationen entstehen, die Werte zwischen zwei Clustern kopieren, die beide verschieden von $z$ sind. Angenommen das Auswahlkriterium für Pfade würde einen Pfad mit einer Länge größer $l$ finden, der keine Kopieroperation enthält, die Werte in oder aus $z$ kopiert, dann muss dieser Pfad ausschließlich Kopieroperationen enthalten, die bereits vor dem Verschieben der Operation $v$ in den Zielcluster $z$ vorhanden waren. Damit hätte dieser Pfad bereits vorher eine Länge größer $l$ gehabt, was ein Widerspruch ist, da alle Pfade vor dem Verschieben von $v$ höchstens die Länge $l$ hatten.

□

Durch Lemma 4.7 ist abgesichert, dass jeder durch das Pfadauswahlkriterium gewählte Pfad eine Kopieroperation enthält, die durch das Kopierknotenauswahlkriterium gewählt werden kann. Es verbleibt noch zu zeigen, dass

**Lemma 4.8**
Algorithmus 4.10 und Algorithmus 4.11 terminieren.

**Beweis**
In jeder Iteration der *Repeat*-Schleife von Algorithmus 4.10 bzw. Algorithmus 4.11 wird mindestens ein Operationsknoten in den Cluster $z$ verschoben. Spätestens wenn alle Knoten in den Cluster $z$ verschoben wurden, gibt es keine Kopieroperationen mehr im Basisblock. Damit haben alle Pfade höchstens die Länge des kritischen Pfades im Basisblocks $b$, die höchstens so groß wie $l$ ist.

□

Durch das beschriebene Verkürzen der Pfade wird gewährleistet, dass gezielt eine Clusterung für einen Basisblock gesucht wird, so dass nach dem Verschieben eines Operationsknotens im Schritt 4 in Abbildung 4.14 in einen anderen Cluster die kritische Pfadlänge im Basisblock nicht größer als $l$ ist. Dadurch kann auch für sehr kurze Ablaufplanlängen eine Clusterung gefunden werden.

#### 4.3.3.2 Bewertung und Auswahl einer Clusterung

In diesem Abschnitt wird beschrieben, wie die Bewertung der in Schritt 1 erzeugten Ablaufpläne sowie die Zwischenbewertung der in Schritt 5 und 6 erzeugten Clusterungen durchgeführt wird. Die Bewertung erfolgt unter denselben Kriterien wie bei der Ablaufplanung, d. h. mit denselben Kostenfunktionen $RBLoad(\chi, T, T')$, $TLoad(\chi, T, T')$ und $TDLoad(\chi, T, T')$. Diese Kostenfunktionen berechnen den erwarteten Kostenzuwachs, der aufgrund zusätzlich benötigter Hardware und Ports in der größten Registerbank entsteht, wenn in $T'$ alle noch planbaren Operationen verplant werden. Diese zusätzliche Hardware ist nur erforderlich, wenn die Hardware nicht ausreicht, die zur Ausführung der in $T$ bereits verplanten Operationen erforderlich ist. Wenn in $T$ keine Operationen verplant sind, dann wird auch keine Hardware zur Ausführung dieser Operationen benötigt und der Kostenzuwachs entspricht den erwarteten Hardwarekosten zur Ausführung des Ablaufplans, der nach der Planung aller Operationen in $T'$ entsteht. Sind in $T'$ bereits alle Operationen verplant, wie es für den Ablaufplan $\alpha$ in Schritt 1 in Abbildung 4.14 der Fall ist, dann entspricht dieser Kostenzuwachs der Summe der gewichteten Registerbankkosten im Prozessor und den Datenpfadkosten, wie sie in Formel (1.13) angegeben sind. Der Intervallgraph, der einen Ablaufplan repräsentiert, in dem keine Operationen verplant sind, wird mit $T_\varnothing$ bezeichnet und ist für die Ablaufplanlänge $l$ definiert als

$$T_\varnothing := (\varnothing, \{0, \ldots, l-1\}, \varnothing).$$

Weiterhin sei $\alpha$ der in Schritt 1 unter der Clusterung $\chi$ erzeugte Ablaufplan zum Basisblock $b$ und $T$ ein Intervallgraph mit $\alpha = \sigma(T)$. Dann werden der Clusterung $\chi$ die Kosten

$$cCost(\chi,T) = \alpha \cdot RBLoad(\chi,T_\varnothing,T) + \beta \cdot TLoad(\chi,T_\varnothing,T) + \gamma \cdot TDLoad(\chi,T_\varnothing,T)$$

zugeordnet. In Algorithmus 4.12 sind für beide Stufen der Clusterung die Schritte 1 bis 3 aus Abbildung 4.14 angegeben. Das Verringern der Breite eines Clusters, das in den Schritten 4 bis 7 aus Abbildung 4.14 durchgeführt wird, ist in Algorithmus 4.13 für beide Clusterungsstufen zusammengefasst.

**Algorithmus 4.12 (Clusterung)**

```
Eingabe: Basisblock b
         Startclusterung χ_start
         Clusteranzahl maxC
         Verbesserungsversuche Versuche1 und Versuche2
Ausgabe: Clusterung χ_opt

// Stufe 1
Berechne Ablaufplan α zum Basisblock b_χstart
χ_opt:=χ_start; χ_neu:=χ_start; Versuche:=0;
while(Versuche < Versuche1) do
  Bestimme den Cluster c, für den FUCWidth(α, c) maximal ist
  Berechne die Menge BI für den Ablaufplan α und den Cluster c
  χ_neu := VerkleinereCluster(b, χ_neu, BI, c, maxC, Stufe 1)
  Berechne Ablaufplan α zum Basisblock b_χneu
  Berechne Intervallgraph T mit α = σ(T)
  if(cCost(χ_neu,T) < cCost(χ_opt,T))
    χ_opt = χ_neu; Versuche = 0;
  else Versuche++; fi
od
// Stufe 2
χ_neu := χ_opt; Versuche = 0;
while(Versuche < Versuche2) do
  Bestimme den Cluster c, für den FUCWidth(α, c) maximal ist
  Berechne die Menge BI für den Ablaufplan α und den Cluster c
  χ_neu := VerkleinereCluster(b, χ_neu, BI, c, maxC, Stufe 2)
  Berechne Ablaufplan α zum Basisblock b_χneu
  Berechne Intervallgraph T mit α = σ(T)
  if(cCost(χ_neu,T) < cCost(χ_opt,T))
    χ_opt = χ_neu; Versuche = 0;
  else Versuche++; fi
od
return χ_opt
```

In Algorithmus 4.12 beginnt die Clusterung in Stufe 1 mit einer Startclusterung $\chi_{start}$, die entweder der Clusterung entspricht, die für die Ablaufplanlänge $l-1$ berechnet wurde, oder alle Knoten sind demselben Cluster zugeordnet, wenn zur Ablaufplanlänge $l-1$ keine Clusterung existiert. Die Stufe 2 übernimmt die beste gefundene Clusterung $\chi_{opt}$ aus der Stufe 1 als Startclusterung $\chi_{neu}$. In jeder Stufe wird in den *while*-Schleifen des Clusterungsalgorithmus versucht, die bisher beste gefundene Clusterung zu verbessern. Bei jedem Schleifendurchlauf wird durch Algorithmus 4.13 (*VerkleinereCluster*) aus jeder Instruktion in *BI* mindestens eine Operation,

die dem Cluster $c$ zugeordnet ist, in einen anderen Cluster verschoben. Die daraus resultierende Clusterung $\chi_{neu}$ erlaubt es, einen Ablaufplan $\alpha$ zu erzeugen, der nicht länger als $l$ ist. Dieser Ablaufplan $\alpha$ wird mit dem Planungsalgorithmus, wie in Abschnitt 4.1 beschrieben, erzeugt. Weil in $\alpha$ alle Operationen verplant sind, sind auch in dem Intervallgraphen $T$ mit $\sigma(T) = \alpha$ alle Operationen verplant und $cCost(\chi_{neu}, \alpha)$ entspricht den erwarteten Hardwarekosten des Prozessors. Sind die Kosten der Clusterung $\chi_{neu}$ geringer als die Kosten der bisher besten gefundenen Clusterung $\chi_{opt}$, dann ersetzt $\chi_{neu}$ die beste bisher gefundene Clusterung. Erst wenn nach einer bestimmten Anzahl aufeinander folgender Schleifendurchläufe $\chi_{opt}$ nicht verbessert werden konnte, wird die jeweilige Clusterungsstufe beendet. In Stufe 1 ist die Anzahl dieser Verbesserungsversuche bei der gegebenen Startclusterung von der Differenz zwischen der FU-Anzahl im Cluster mit den meisten und dem Cluster mit den wenigsten FUs abhängig:

$$Versuche1 := \frac{FUWidth(\alpha, \chi_{start}) - minFUWidth(\alpha, \chi_{start})}{2} + 1\,, \qquad (3.6)$$

wobei

$$minFUWidth(\alpha, \chi) = \begin{cases} 0, \text{ falls } \left|V \,/\, \chi - \{[e]_\chi\}\right| < maxC \\ \left\{FUCWidth(\alpha, c)\right|\ c \in (V \,/\, \chi - \{[e]_\chi\}\Big\}, \text{ sonst.} \end{cases}$$

In Stufe 2 ist die Anzahl dieser Versuche auf einen festen Wert gesetzt, der bei den durchgeführten Tests $Versuche2 := 3$ war.

Die in Algorithmus 4.12 verwendete Funktion *VerkleinereCluster* wird in der Stufe 1 und 2 genutzt und ist in Algorithmus 4.13 angegeben. Diese Funktion realisiert die Schritte 4 bis 7 aus Abbildung 4.14. Die in Schritt 4 zu verschiebende Operation $v$ und der Zielcluster $z$ werden in Algorithmus 4.13 ermittelt, indem beginnend mit einer Instruktion $i$ aus der Menge $BI$ jede Operation $v$ aus $i$, die dem Cluster $c$ zugeordnet ist, probeweise in jeden anderen Cluster $z$ verschoben wird. $CL$ ist die Menge aller Cluster, in die $v$ verschoben werden darf und wird um einen leeren Cluster $\varnothing$ erweitert, sofern durch $\chi_{neu}$ nicht bereits die Höchstzahl $maxC$ der erlaubten Cluster benutzt wird. Dadurch ist es möglich, $v$ dem neuen Cluster $\varnothing$ zuzuordnen. Durch die Verschiebung von $v$ in einen anderen Cluster müssen im Allgemeinen neue Kopieroperationen unmittelbar vor und/oder nach $v$ erzeugt werden. Das kann, wie bereits beschrieben, dazu führen, dass im Basisblock Pfade mit einer Länge größer als $l$ entstehen. Zum Verkürzen dieser Pfade wird in der Stufe 1 der Algorithmus 4.10 und in der Stufe 2 der Algorithmus 4.11 genutzt. Zusammen mit der probeweise in den Cluster $z$ verschobenen Operation $v$ werden dadurch weitere Operationen in den Cluster $z$ verschoben. Zu der daraus entstehenden Clusterung $\chi'$ und dem Basisblock $b$ wird ein Intervallgraph $T$ konstruiert. Im Allgemeinen sind in $T$ keine

Operationen verplant[1]. Die notwendigen Kopieroperationen wurden aber in $b_{\chi'}$ eingefügt und sind damit auch in $T$ enthalten. Für $T$ wird eine Zwischenbewertung mit $cCost(\chi', T_\emptyset, T)$ berechnet und aus der aus $BI$ gewählten Instruktion $i$ wird der Knoten $v$ in den Zielcluster $z$ verschoben, für dessen Zuordnung zu einem Zielcluster $z$ $cCost(\chi', T_\emptyset, T)$ minimal ist. Diese Zuordnung liefert eine neue Clusterung $\chi_{neu}$, die weiter bearbeitet wird, bis aus jeder Instruktion in $BI$ mindestens eine Operation in einen anderen Cluster verschoben wurde. Erst dann ist Algorithmus 4.13 beendet.

**Algorithmus 4.13 (*VerkleinereCluster*)**

```
Eingabe: Basisblock b
         Clusterung χneu
         Menge der breitesten Instruktionen BI
         Breitester Cluste c
         Maximale Clusteranzahl maxC
         Stufe der Clusterung stufe
Ausgabe: Clusterung χneu, so dass mindesten ein Knoten aus jeder
         Instruktion in BI verschoben wurde
besteKosten:=∞
repeat
  Wähle i ∈ BI und aus i wurde noch keine Operation verschoben
  foreach v mit v ∈ i und v ∈ c do
    if(|V/χneu| < maxC) then CL:=V/χneu ∪ {∅} else CL:= V/χneu fi
    foreach z mit z ∈ CL und z ≠ [v]χneu und z ≠ [e]χneu do
      χ':=χneu-{(v,w),(w,v)| w ∈ [v]χ} ∪ {(v,w),(w,v)|w ∈ (z∪{v})}
      if(stufe = 1) then
        Verkürze kritische Pfade in bχ' mit Algorithmus 4.10
      else
        Verkürze kritische Pfade in bχ' mit Algorithmus 4.11
      fi
      Konstruiere Intervallgraph T zum Basisblock bχ'
      if(cCost(χ',T) < besteKosten) then
        χneu = χ'
      fi
    od
  od
until aus jedem i ∈ BI wurde eine Operation verschoben
return χneu
```

Bei der von Algorithmus 4.13 berechneten Clusterung $\chi_{neu}$ haben im geclusterten Basisblock $b$ alle Pfade höchstens die Länge $l$. Erst für diese

[1] Es sei denn, es gibt Operationen, die bei der Ablaufplanlänge $l$ eine Mobilität von 0 haben.

Clusterung wird in Algorithmus 4.12 ein Ablaufplan berechnet, der bewertet und für den erneut die Menge $BI$ berechnet wird.

Da durch Algorithmus 4.13 mindestens eine Operation aus jeder Instruktion in $BI$ verschoben wurde, wird sich die Anzahl der parallel auszuführenden Operationen im Cluster $c$ und somit auch die interne Portanzahl in der Registerbank nach der erneuten Ablaufplanung in Algorithmus 4.12 verringern. Damit sinkt auch die Portanzahl in der größten Registerbank des Prozessors. Garantiert werden kann das jedoch nicht, da zum einen die Planung von einer Heuristik vorgenommen wird und zum anderen das Verschieben der Operationen zu neuen Kopieroperationen geführt haben kann, die wiederum die Mobilität anderer Operationen einschränken. Dadurch kann im schlechtesten Fall sogar die Anzahl der parallel auszuführenden Operationen in einem Cluster angestiegen sein. Allerdings akzeptiert der Clusterungsalgorithmus (Algorithmus 4.12) nach dem Verschieben von Operationen aus dem breitesten Cluster in einen anderen Cluster die von Algorithmus 4.13 erzeugte Clusterung auch dann, wenn diese schlechter ist als vor dem Verschieben der Operationen. Dadurch besteht die Möglichkeit, in einer weiteren Iteration erneut Operationen aus einem breiter gewordenen Cluster zu verschieben und dadurch eine bessere Clusterung zu erhalten. Indem die Anzahl der dafür zulässigen Versuche in der Stufe 1 und/oder 2 erhöht wird, lässt sich die Qualität der Clusterung durch zusätzliche Rechenzeit erhöhen. In der ersten Stufe wird die Versuchsanzahl in der gegenwärtigen Implementierung durch die Formel (3.6) berechnet. Durch die so festgelegte Versuchsanzahl ist es möglich, die Anzahl der parallel ausgeführten Operationen zwischen dem Cluster mit der höchsten Parallelität und dem Cluster mit der geringsten Parallelität anzugleichen, bevor sich eine Verbesserung ergeben muss. Außerdem gestattet es der Clusterungsalgorithmus, eine Operation mehrfach zu verschieben und somit einmal getroffene Entscheidungen für eine Clusterzuordnung auch wieder rückgängig zu machen, wenn sich diese im weiteren Verlauf der Clusterung als schlecht erweisen.

Die Verwendung des vorgestellten zweistufigen Verfahrens hat sich in zahlreichen Tests als vorteilhaft gegenüber einem einstufigen Verfahren erwiesen, da das Verschieben vollständiger Pfade in der ersten Stufe dazu führt, dass wenige Kopieroperationen entstehen und diese auf kürzeren Pfaden liegen. Eine solche Clusterung ist aber stark durch die längsten Pfade im Basisblock dominiert, was insbesondere bei den Ablaufplanlängen, die auch Kopieroperationen in kritischen Pfaden erlauben, nachteilig sein kann. Durch die zweite Stufe, in der das Verschieben von Teilpfaden zulässig ist, können auch in Pfade mit kritischer Länge Kopieroperationen eingefügt werden. Diese zusätzlichen Kopieroperationen werden aber nur akzeptiert, wenn dadurch die beste Clusterung aus der ersten Stufe verbessert wird.

Auch die Verwendung der zur Ablaufplanlänge $l-1$ berechneten Clusterung als Startclusterung bei der Berechnung einer Clusterung zur

Ablaufplanlänge $l$ hat sich bei zahlreichen Tests als vorteilhaft erwiesen. Bei gleicher Clusterung kann der zur Länge $l$ errechnete Ablaufplan im Allgemeinen mit einer kostengünstigeren Architektur ausgeführt werden als der zur Länge $l-1$. Eine Veränderung dieser Clusterung wird bei der Länge $l$ nur akzeptiert, wenn dadurch weitere Kosten in der Architektur eingespart werden können. Durch diese Art der Kopplung von Clusterung und Ablaufplanung wird erreicht, dass bei einer festen Clusteranzahl mit steigender Ablaufplanlänge die Kosten der Ablaufpläne kontinuierlich sinken.

Eine weitere Eigenschaft des Clusterungsalgorithmus ist es, dass ein zusätzlicher Cluster nur genutzt wird, wenn dadurch die erwarteten Hardwarekosten sinken. Das ergibt sich aus der Konstruktion der Menge $CL$ in Algorithmus 4.13. Dieser Menge wird ein leerer Cluster $\varnothing$ hinzugefügt, falls die maximale Anzahl erlaubter Cluster noch nicht ausgeschöpft ist. Führt die Zuordnung von Operationen zu diesem zusätzlichen Cluster nicht zu einer Verringerung der Hardwarekosten, dann wird die entsprechende Clusterung in Algorithmus 4.12 auch nicht akzeptiert. Diese Eigenschaft wird genutzt, um mit dem Clusterungsalgorithmus zu berechnen, auf wie viele Cluster die Operationen eines Basisblocks verteilt werden können, um dadurch Hardwarekosten im Prozessor einzusparen.

# 5 Globale Optimierung mit DESCOMP

Die bisher beschriebenen Phasen der Ablaufplanung, Ressourcenallokation und Clusterung beschäftigten sich mit der Optimierung der Parameter eines VLIW-Prozessors zur Ausführung eines gegebenen Basisblocks bei einer festen Ablaufplanlänge. Alle Basisblöcke einer Anwendung müssen aber vom selben Prozessor ausgeführt werden, weswegen der Prozessor die Anforderungen aller Basisblöcke erfüllen muss. In diesem Abschnitt wird die Optimierung der Parameter eines VLIW-Prozessors beschrieben, der alle gegebenen Basisblöcke unter Einhaltung der vorgegebenen Zeitschranken ausführen kann. Die Zeitschranken müssen nur für die zeitkritischen Basisblöcke des Programms festgelegt werden, die oft nicht mehr als 3% des Programmcodes darstellen [86]. Eine Zeitschranke kann für mehrere Basisblöcke vorgegeben sein. Wie sich dann die Ausführungszeit auf die einzelnen Basisblöcke verteilt, wird durch DESCOMP berechnet, indem für jeden Basisblock eine geeignete Ablaufplanlänge festgelegt wird.

Die Menge aller zeitkritischen Basisblöcke im Programm wird mit $\mathcal{B}$ bezeichnet. Ein **Programmfragment** ist eine Menge von Basisblöcken, die innerhalb einer vorgegebenen Zeit ausgeführt werden müssen, z. B. alle Basisblöcke, die zu einer Schleife gehören. Die Menge der Programmfragmente wird mit $\mathcal{P} \subseteq \wp(\mathcal{B})$ bezeichnet. Die von jedem Programmfragment aus $\mathcal{P}$ einzuhaltende Zeitschranke ist durch die Funktion $t : \mathcal{P} \rightarrow \mathbb{R}$ in Sekunden gegeben. Die Ausführungszeit eines Ablaufplans für einen Basisblock kann berechnet werden, wenn für jeden Basisblock seine Ausführungshäufigkeit und die Taktfrequenz, mit der der Ablaufplan ausgeführt werden kann, bekannt sind. Die Ausführungshäufigkeit kann durch Simulation der Anwendung mit relevanten Eingabedaten für jeden Basisblock ermittelt werden und ist durch die Funktion $exe : \mathcal{B} \rightarrow \mathbb{N}$ gegeben. Die maximal mögliche Taktfrequenz, mit der der Ablaufplan abgearbeitet werden kann, ist von der maximalen Portanzahl $p$ in der größten Registerbank des Prozessors abhängig und durch die bereits mit Formel (1.4) eingeführte Funktion $fm(p)$ gegeben.

Die Parameter des VLIW-Prozessors werden erzeugt, indem für jeden Basisblock und jede seiner zulässigen Ablaufplanlängen Ablaufpläne mit dem in Abschnitt 4 vorgestellten Verfahren zur lokalen Optimierung erzeugt werden. Aus allen so gebildeten Ablaufplänen wird für jeden Basisblock ein Ablaufplan als Kandidat ausgewählt, so dass

- alle ausgewählten Kandidaten vom selben Prozessor ausgeführt werden können und

- bei der Ausführung die vorgegebenen Zeitschranken eingehalten werden und
- entweder die Hardwarekosten des Prozessors minimal sind oder
- sein Stromverbrauch minimiert wurde.

Das dafür angewendete Verfahren minimiert zunächst die Hardwarekosten, indem durch die **Portoptimierung** die mindestens benötigte Portanzahl in der größten Registerbank eines Prozessors berechnet wird, der alle Programmfragmente innerhalb ihrer Zeitschranken ausführen kann. Der Gesamtstromverbrauch des Prozessors kann in einem nachgelagerten Schritt gesenkt werden, indem zusätzliche Hardwarekosten aufgrund zusätzlicher Ports in der größten Registerbank und zusätzlicher funktionaler Einheiten akzeptiert werden. Die dadurch entstehende höhere Parallelität auf Instruktionsebene kann genutzt werden, um die Taktfrequenz und damit den dynamischen Stromverbrauch zu reduzieren. Nachdem eine geeignete Portanzahl für die größte Registerbank des Prozessors bestimmt wurde, wird durch die **Typoptimierung** für jeden Cluster die Anzahl der in ihm durch die FUs bereitgestellten Operatoren minimiert.

## 5.1 Zielprogramme erzeugen

Zunächst soll die Menge der Zielprogramme angegeben werden, aus denen für jeden Basisblock ein Kandidat ausgewählt wird, der zur Ausführung des Basisblocks auf dem Zielprozessor verwendet wird. Um zu verdeutlichen, dass $(\alpha, \chi, \phi)$ ein Zielprogramm für den Basisblock $b$ ist, wird auch $(\alpha, \chi, \phi)_b$ geschrieben. Jedem Basisblock $b \in \mathcal{B}$ wird eine Menge von Zielprogrammen $\mathcal{ZP}_b^{maxC}$ mit $maxC \in \mathbb{N}$ zugeordnet, wobei für jedes Zielprogramm $(\alpha, \chi, \phi)_b \in \mathcal{ZP}_b^{maxC}$ gilt:

$$maxC = |V_b/\chi - \{[e]_\chi\}|.$$

Das heißt, die Clusteranzahl $maxC$ bezieht sich nur auf die Cluster, die eine Registerbank enthalten und nicht auf den externen Cluster. Die Menge $\mathcal{ZP}_b^{maxC}$ wird erzeugt, indem für die feste Clusteranzahl $maxC$ Zielprogramme mit den Ablaufplanlängen $cp(b) \leq l \leq lp(b)$ durch die vorgestellte Planung, Clusterung und Ressourcenallokation erzeugt werden. Begonnen wird mit einem Zielprogramm, dessen Ablaufplan die Länge $l = cp(b)$ hat. Für immer größer werdende $l$ werden die übrigen Zielprogramme erzeugt. Für jede Ablaufplanlänge wird eine Clusterung mit höchstens $maxC$ vielen Clustern durch Algorithmus 4.12 generiert, anschließend die Ablaufplanung und dann die Ressourcenallokation durchgeführt. Als Startclusterung $\chi_{start}$ wird für $l = cp(b)$ $\chi_{start} := V \times V$ verwendet. Die Clusterung $\chi$, die zur Ablaufplanlänge $l$ errechnet wurde, wird als Startclusterung bei der

Erzeugung des Zielprogramms der Länge $l+1$ genutzt. Durch den Clusterungsalgorithmus wird nicht immer eine Clusterung gefunden, die auch genau $maxC$ viele Cluster verwendet. Gründe dafür sind:

- Die Ablaufplanlänge ist so kurz, dass keine Kopieroperationen in die Pfade des Basisblocks eingefügt werden können oder
- der Clusterungsalgorithmus kann die zur Verfügung gestellte Clusteranzahl nicht nutzen, weil die Parallelität im Basisblock nicht hoch genug ist.

In diesen Fällen enthält die Menge $\mathcal{ZP}_b^{maxC}$ kein Zielprogramm der entsprechenden Länge. Wie am Ende des Abschnitts 4.3.3 beschrieben wurde, nutzt der Clusterungsalgorithmus einen zusätzlichen Cluster nur dann, wenn dadurch die Hardwarekosten des zur Ausführung benötigten Prozessors reduziert werden können. Die Menge aller Zielprogramme zu einem Basisblock $b$ wird deshalb erzeugt, indem die Mengen $\mathcal{ZP}_b^{maxC}$ für eine immer größer werdende Clusteranzahl $maxC$, beginnend bei eins, berechnet werden. Die Berechnung endet bei dem kleinsten $maxC$, für das gilt: $\mathcal{ZP}_b^{maxC} = \varnothing$. Es sei $mC_b := \max\{maxC \mid \mathcal{ZP}_b^{maxC} \neq \varnothing\}$, dann ist die Menge aller erzeugten Zielprogramme zu einem Basisblock $b$

$$\mathcal{ZP}_b = \bigcup_{1 \leq maxC \leq mC_b} \mathcal{ZP}_b^{maxC} .$$

Die Menge aller Zielprogramme, die zu den Basisblöcken in $\mathcal{B}$ erzeugt werden, ist

$$\mathcal{ZP} = \bigcup_{b \in \mathcal{B}} \mathcal{ZP}_b .$$

Weiterhin bezeichnet

$$mC = \max\left\{mC_b \mid b \in \mathcal{B}\right\}$$

im Folgenden die größte Clusteranzahl, die bei der Clusterung eines zeitkritischen Basisblocks in $\mathcal{B}$ entstanden ist.

## 5.2 Optimierung der Portkonfiguration

In diesem Abschnitt wird die Berechnung der Portkonfiguration eines VLIW-Prozessors, der geeignet ist, mehrere Basisblöcke auszuführen, beschrieben. Es werden zwei Strategien vorgestellt, von denen eine zur Minimierung der Registerbankkosten und die andere zur Minimierung des Stromverbrauchs genutzt werden kann.

Die **Portkonfiguration eines Zielprogramms** ($\alpha$, $\chi$, $\phi$) zu einem Basisblock $b$ charakterisiert die maximale Anzahl interner Ports in einer Registerbank sowie die Anzahl der externen Lese- und Schreibports, die zur Ausführung

dieses Zielprogramms erforderlich sind. Die Portkonfiguration eines Zielprogramms $(\alpha, \chi, \phi)$ ist ein 3-Tupel $(ip, rp, wp) \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, wobei

- $ip = 3 \cdot FUWidth(\alpha, \chi)$ die maximale Anzahl der internen Lese- und Schreibports in einer Registerbank[1],
- $rp = erPWidth(\alpha, \chi)$ die Anzahl der externen Leseports in jeder Registerbank und
- $wp = ewPWidth(\alpha, \chi)$ die Anzahl der externen Schreibports in jeder Registerbank ist.

Die Portkonfiguration des Zielprogramms $(\alpha, \chi, \phi)$ wird mit $pConf((\alpha, \chi, \phi))$ bezeichnet. Für zwei Portkonfigurationen gilt:

$$(ip, rp, wp) \leq (ip', rp', wp') :\Leftrightarrow ip \leq ip' \text{ und } rp \leq rp' \text{ und } wp \leq wp'.$$

Anhand einer Portkonfiguration $(ip, rp, wp)$ wird für jeden Basisblock aus $\mathcal{B}$ ein Zielprogramm $(\alpha, \chi, \phi) \in \mathcal{ZP}_b$ mit minimaler Ablaufplanlänge ausgewählt, das mit dieser Portkonfiguration ausgeführt werden kann, d. h.

$$pConf((\alpha, \chi, \phi)) \leq (ip, rp, wp).$$

Die **Gesamtportanzahl** $mp$ einer Portkonfiguration $(ip, rp, wp)$ ist $mp := ip + rp + wp$.

### 5.2.1 Optimierung der Registerbankkosten

Bei der Suche nach einer minimalen Portkonfiguration, mit der alle Programmfragmente innerhalb ihrer Zeitschranken ausgeführt werden können, wird die Gesamtportanzahl schrittweise erhöht und für jede Portkonfiguration mit dieser Gesamtportanzahl geprüft, ob sich die Programmfragmente mit dieser Portkonfiguration innerhalb der Zeitschranken ausführen lassen. Die Suche beginnt mit der kleinsten möglichen Portkonfiguration (3, 0, 0), die genau eine FU in jedem Cluster und keine Kopieroperationen zulässt. Um systematisch alle Portkonfigurationen zu erzeugen, werden zu einer Gesamtportanzahl $mp$ beginnend bei $mp = 3$ alle möglichen Portkonfigurationen durch:

$$MP(mp) = \{(ip, rp, wp) \mid ip + rp + wp = mp \text{ und } 3 \text{ teilt } ip\}$$

gebildet. Wenn mit keiner der zu $MP(mp)$ gehörenden Portkonfigurationen alle Programmfragmente innerhalb ihrer Zeitschranken abgearbeitet werden können, dann wird $mp$ um den Wert eins erhöht und erneut die zugehörige Menge $MP(mp)$ bestimmt.

[1] Aus $ip$ ergibt sich somit auch die maximale Anzahl funktionaler Einheiten in einem Cluster.

Um zu prüfen, ob mit einer gegebenen Portkonfiguration ($ip$, $rp$, $wp$) aus $MP(mp)$ alle Programmfragmente innerhalb ihrer Zeitschranken ausgeführt werden können, wird für jeden Basisblock $b$ aus den zur Verfügung stehenden Zielprogrammen $\mathcal{ZP}_b$ durch die Funktion $mL$ die Länge des kürzesten Ablaufplans bestimmt, der mit dieser Portkonfiguration noch ausgeführt werden kann:

$$mL\big(b,(ip,rp,wp)\big) = \min\{|\alpha| \big|\ \exists(\alpha,\chi,\phi) \in \mathcal{ZP}_b \wedge pConf((\alpha,\chi,\phi)) \leq (ip,rp,wp)\},$$

wobei $\min \varnothing = \infty$. Da durch die Portkonfiguration $pConf((\alpha,\ \chi,\ \phi))$ mit $pConf((\alpha,\ \chi,\ \phi)) \leq (ip,\ rp,\ wp)$ die maximale Anforderung eines Zielprogramms $(\alpha,\ \chi,\ \phi)$ an die Registerbänke eines Prozessors ermittelt wird, kann jeder Cluster dieses Zielprogramms von einem Prozessor ausgeführt werden, der in jeder Registerbank $ip$ viele interne Ports, $rp$ viele externe Lese- und $wp$ viele externe Schreibports hat. Aufgrund der Gesamtportanzahl $mp = ip + rp + wp$ ist die maximale Taktfrequenz, mit der dieser Prozessor arbeiten kann, durch $fm(mp)$ festgelegt (vgl. (1.4)). Die minimale Ausführungszeit des Basisblocks $b$ bei der gegebenen Portkonfiguration ($ip$, $rp$, $wp$) ist damit

$$bbExeT(b,(ip,rp,wp)) = \frac{mL(b,(ip,rp,wp)) \cdot exe(b)}{fm(ip+rp+wp)}.$$

Die kürzeste Ausführungszeit $pExeT$ eines Programmfragments $p$ bei einer gegebenen Portkonfiguration ($ip$, $rp$, $wp$) ist dann die Summe der kürzesten Ausführungszeiten seiner Basisblöcke:

$$pExeT\big(p,(ip,rp,wp)\big) = \sum_{b\in p} bbExeT\big(b,(ip,rp,wp)\big). \tag{4.1}$$

Die durch $bbExeT$ und $pExeT$ errechneten Ausführungszeiten für Basisblöcke und Programmfragmente sind bei der gegebenen Portkonfiguration minimal, weil durch $mp$ die maximal mögliche Taktfrequenz vorgegeben ist und es für keinen Basisblock in einem Programmfragment kürzere Ablaufpläne gibt, die mit dieser Portkonfiguration ausgeführt werden können. Damit können alle Programmfragmente mit einer Portkonfiguration ($ip$, $rp$, $wp$) innerhalb ihrer Zeitschranken ausgeführt werden, wenn gilt:

$$\forall p \in \mathcal{P} : pExeT\big(p,(ip,rp,wp)\big) \leq t(p). \tag{4.2}$$

Das kleinste $mp$, so dass es eine Portkonfiguration ($ip$, $rp$, $wp$) $\in MP(mp)$ gibt, die alle Ungleichungen (4.2) erfüllt, ist damit die Gesamtportanzahl, die in mindestens einer Registerbank des Prozessors erforderlich ist, um alle Programmfragmente innerhalb ihrer Zeitschranken auszuführen. Es spielt dabei keine Rolle, wie viele Cluster der Prozessor zur Ausführung von $b$ benötigt, weil jede Registerbank des Prozessors bis zu $mp$ viele Ports haben kann und die Clusteranzahl keinen Einfluss auf die zur Abschätzung der Ausführungszeit benutzte maximale Taktfrequenz hat. Die Suche nach weiteren Portkonfigurationen wird beendet, sobald dieses kleinste $mp$

gefunden wurde. Es ist zwar möglich, weitere Portkonfigurationen zu finden, die es erlauben, alle Programmfragmente innerhalb ihrer Zeitschranken abzuarbeiten, allerdings verursachen diese Portkonfigurationen höhere Kosten in der größten Registerbank. In Abbildung 5.1 ist das Suchprinzip nach einer Portkonfiguration schematisch dargestellt.

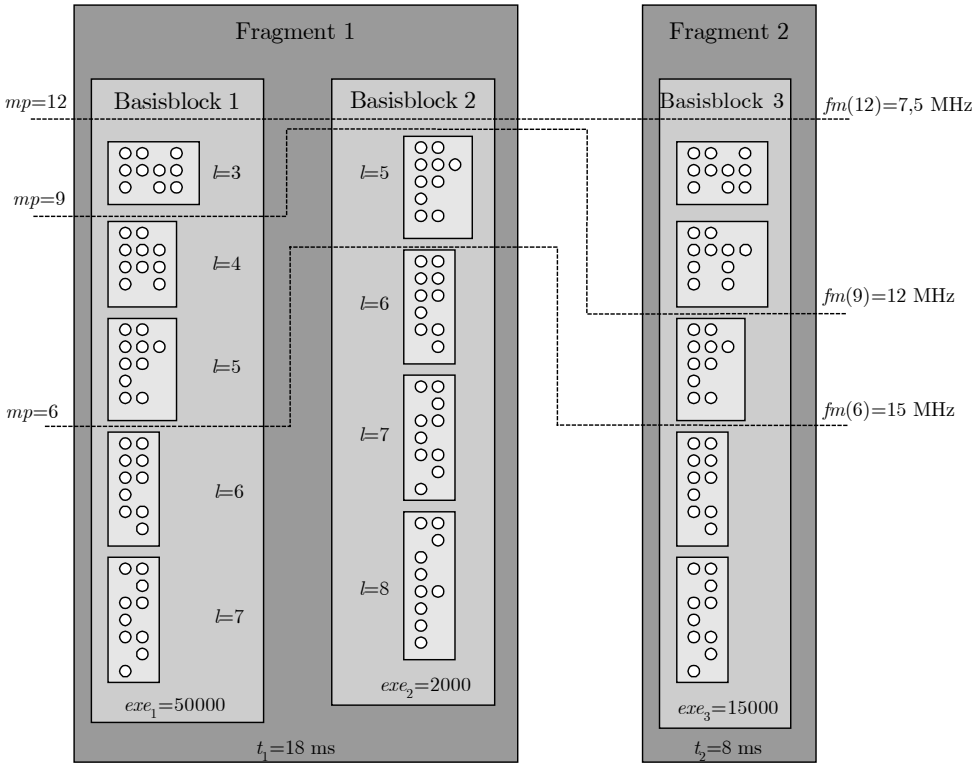



Abbildung 5.1: Prinzip der Portoptimierung.

Es sind zwei Programmfragmente mit den Zeitschranken $t_1 = 18$ Millisekunden und $t_2 = 8$ Millisekunden gegeben, von denen eins aus zwei Basisblöcken und das andere aus einem Basisblock besteht. In der Abbildung enthält jeder Basisblock eine Folge immer länger werdender Ablaufpläne, die durch den Planungsalgorithmus berechnet wurden. Es sind nur ungeclusterte Ablaufpläne dargestellt. Mit zunehmender Länge nimmt die Parallelität in den Ablaufplänen ab. Die kürzesten mit der Portkonfiguration (6, 0, 0) ausführbaren Ablaufpläne haben für jeden Basisblock die Länge sechs. Eine Registerbank mit sechs Ports kann in dem Beispiel mit maximal 15 MHz getaktet werden. Damit beträgt für Programmfragment eins die Ausführungszeit 20,8 Millisekunden und für Programmfragment zwei 6 Millisekunden. Für Programmfragment eins kann also die vorgegebene Zeitschranke nicht eingehalten werden. *mp* wird schrittweise erhöht. Erst für $mp = 9$ existiert wieder eine Portkonfiguration (9, 0, 0) ohne externe Ports. Die Ausführungszeiten für die kürzesten Ablaufpläne, die mit dieser Portkonfiguration ausgeführt werden können, sind 17,5 Millisekunden für

Programmfragment eins und 6,25 Millisekunden für Programmfragment zwei. Die Portkonfiguration (9, 0, 0) ist somit die kostengünstigste Konfiguration, mit der alle Programmfragmente innerhalb ihrer Zeitschranken ausgeführt werden können.

Für die kleinste Gesamtportanzahl $mp$, für die es eine Portkonfiguration ($ip$, $rp$, $wp$) in $MP(mp)$ gibt, die die Ungleichungen (4.2) erfüllt, kann es weitere Portkonfigurationen ($ip'$, $rp'$, $wp'$) in $MP(mp)$ geben, die auch die Ungleichungen (4.2) erfüllen. Für solche Portkonfigurationen gilt $fm(ip + rp + wp) = fm(ip' + rp' + wp')$. Die Zielprogramme, die mit diesen Portkonfigurationen abgearbeitet werden können, können sich aber in der Größe der benötigten Registerbänke und der Clusteranzahl unterscheiden. Um die Portkonfiguration auszuwählen, für die die Summe der Registerbankkosten minimal ist, wird für jede dieser Portkonfigurationen eine Kandidatenmenge $\mathcal{K}_{(ip,\ rp,\ wp)}$ durch

$$\mathcal{K}_{(ip,rp,wp)} = \left\{(\alpha,\chi,\phi)\middle|\ (\alpha,\chi,\phi) \in \mathcal{ZP} \wedge pConf((\alpha,\chi,\phi)) \leq (ip,rp,wp)\right\}$$

gebildet und mit der Kandidatenmenge weitergearbeitet, deren kürzeste Zielprogramme für jeden Basisblock von einer Architektur mit einer minimalen Gesamtanzahl funktionaler Einheiten ausgeführt werden können. Dabei muss beachtet werden, dass die Zielprogramme in $\mathcal{K}_{(ip,\ rp,\ wp)}$ eine unterschiedliche Anzahl von Clustern für ihre Ausführung benötigen können. Weiterhin ist für die Cluster verschiedener Basisblöcke nicht festgelegt, welche dieser Basisblockcluster durch welchen Prozessorcluster ausgeführt werden. Gibt es beispielsweise zwei Basisblöcke $b$ und $b'$, die beide in jeweils zwei Cluster zerlegt wurden mit $c_1$, $c_2 \in b/\chi$ und $c'_1$, $c'_2 \in b'/\chi'$, dann können sowohl $c_1$ und $c'_1$ sowie $c_2$ und $c'_2$ vom selben Prozessorcluster ausgeführt werden oder auch $c_1$ und $c'_2$ sowie $c_2$ und $c'_1$. Es wird deshalb im Folgenden für die kürzesten Zielprogramme einer Kandidatenmenge eine Zuordnung der Basisblockcluster zu Prozessorclustern bestimmt, um daraus die minimale Gesamtanzahl der funktionalen Einheiten im Prozessor zu ermitteln. Für jede Kandidatenmenge wird zu jedem Basisblock $b$ ein **FU-Vektor** $(z_1, \ldots, z_k, \ldots, z_{mC}) \in \mathbb{N}^{mC}$ konstruiert, der die benötigte FU-Anzahl in jedem Cluster des kürzesten Zielprogramms $(\alpha, \chi, \phi)_b \in \mathcal{K}_{(ip,\ rp,\ wp)}$ modelliert. $mC$ ist die größte Clusteranzahl, die ein Zielprogramm in $\mathcal{K}_{(ip,\ rp,\ wp)}$ benötigt. Für die Kandidatenmenge $\mathcal{K}_{(ip,\ rp,\ wp)}$ wird der FU-Vektor $(z_1, \ldots, z_k, \ldots, z_{mC})$ zum Basisblock $b$ gebildet durch:

- Die Auswahl eines kürzesten Ablaufplans $\alpha$ für $b$, für den es ein $\chi$ und ein $\phi$ gibt mit $(\alpha, \chi, \phi) \in \mathcal{K}_{(ip,\ rp,\ wp)}$. Sollte es aufgrund der Clusterung mehrere solcher Zielprogramme geben, dann wird das Zielprogramm $(\alpha, \chi, \phi)$ ausgewählt, für das $FUWidth(\alpha, \chi)$ minimal ist.
- $k := |V_b/\chi - \{[e]_\chi\}|$ ist die benötigte Clusteranzahl zur Ausführung des ausgewählten Ablaufplans $\alpha$ für $b$ und

- für jedes $i$ mit $1 \leq i \leq k$, ist $z_i = FUCWidth(\alpha, c_i)$ die Anzahl der funktionalen Einheiten im Cluster $c_i$, wobei $(V_b/\chi - \{[e]_\chi\}) = \{c_1, \ldots, c_k\}$ und $\forall i$: $1 \leq i < k \Rightarrow z_i \geq z_{i+1}$, und $z_{k+1} = 0, \ldots, z_{mC} = 0$.

Die Menge aller so gebildeten FU-Vektoren zur Kandidatenmenge $\mathcal{K}_{(ip,\, rp,\, wp)}$ wird mit $FUV$ bezeichnet und enthält für jeden Basisblock genau einen FU-Vektor. Aus diesen Vektoren wird ein **maximaler FU-Vektor** $vm = (m_1, \ldots, m_{mC})$ zur Kandidatenmenge $\mathcal{K}_{(ip,\, rp,\, wp)}$ durch

$$m_i = \max\left\{z_i \,\middle|\, (z_1,\ldots,z_{mC}) \in FUV\right\}$$

gebildet. Aus diesem maximalen FU-Vektor wird durch Summieren seiner Komponenten die mindestens benötigte Anzahl funktionaler Einheiten $tF$ im Prozessor berechnet:

$$tF = \sum_{i=1}^{mC} m_i \,.$$

**Satz 5.1**
Für jeden Vektor $(m'_1, \ldots, m'_{mC}) \in \mathbb{N}^{mC}$ mit $m'_i \geq z_i$ für alle $(z_1, \ldots, z_{mC}) \in FUV$ und alle $i \in \{1, \ldots, mC\}$ gilt:

$$tF \leq \sum_{k=1}^{mC} m'_k \,.$$

**Beweis**
Angenommen es gäbe einen Vektor $vm' = (m'_1, \ldots, m'_{mC})$, dessen Summe seiner Komponenten kleiner als $tF$ ist und jede Komponente $m'_i$ ist größer als die Komponente $z_i$ in jedem FU-Vektor $(z_1, \ldots, z_{mC}) \in FUV$. Sei $i$ die kleinste Position an der gilt $m'_i < m_i$ und $\forall k : i < k \leq mC : m'_i > m'_k$, wobei $m_i$ die $i$-te Komponente im maximalen FU-Vektor ist, dann gibt es in $vm'$ höchstens $i - 1$ viele Komponenten, die größer sind als $m'_i$. Außerdem gibt es einen FU-Vektor $z = (z_1, \ldots, z_{mC}) \in FUV$ mit $z_i = m_i$ für den gilt: $\forall k : 1 \leq k < i : z_k \geq z_i > m'_i$. Damit gibt es in $z$ mindestens $i$ viele Komponenten, die größer sind als $m'_i$. Damit gibt es nicht für jede dieser Komponenten in $z$ eine Komponente in $vm'$, die mindestens so groß ist.

□

Es wird mit der Kandidatenmenge $\mathcal{K}_{(ip,rp,wp)}$ weitergearbeitet, deren zugehöriges $tF$ am kleinsten ist. In Algorithmus 5.1 ist die Berechnung der minimalen Portkonfiguration angegeben. Dort ist

$$mPWidth = \max\left\{PWidth(\alpha,\chi)\,\middle|\, (\alpha,\chi,\phi) \in \mathcal{ZP}\right\} \tag{4.3}$$

die Portanzahl, die in der größten Registerbank eines Zielprogramms benötigt wird. Portkonfigurationen, die insgesamt mehr als $mPWidth$ Ports verwenden, müssen nicht betrachtet werden, da es keine Zielprogramme gibt, die solche Portkonfigurationen nutzen.

**Algorithmus 5.1 (Optimierung der Portkonfiguration)**

```
Eingabe: Programmfragment P
         maximale Portanzahl mPWidth eines Zielprogramms
         maximale Clusteranzahl mC eines Zielprogramms
Ausgabe: Bei Erfolg Kandidatenmenge K_minConf und
         minimale Portkonfiguration minConf

mp:=3; PK:=∅;
while(PK = ∅ AND mp ≤ mPWidth) do
  MP:={(i,r,w) | i+r+w = mp und 3 teilt i};
  foreach (i,r,w) ∈ MP do
    if(∀p ∈ P pExeT(p,(i,r,w)) < t(p)) then
      PK:=PK ∪ {(i,r,w)}
    fi
  od
  mp:=mp + 1
od
if(PK = ∅) then
  return Keine zulässige Konfiguration gefunden
fi
minCost:=∞
foreach (i,r,w) ∈ PK do
  Erzeuge die Menge der FU-Vektoren FUV zur Menge K_(i,r,w)
  Berechne maximalen FU Vektor vm:=(v_1,...,v_mC)
  if(v_1 + ... + v_mC < minCost) then
    minCost:=v_1 + ... + v_mC
    minConf:=(i,r,w)
  fi
do
return K_minConf
```

In Algorithmus 5.1 ist *PK* die Menge der Portkonfigurationen, mit denen die Zeitschranken eingehalten werden können. Der Fall, dass durch Algorithmus 5.1 keine Portkonfiguration gefunden wird, tritt dann ein, wenn die Zeitschranken nicht eingehalten werden können, weil sie zu niedrig sind oder die auszuführenden Basisblöcke sehr unterschiedliche Anforderungen an die Architektur stellen. Im letzten Fall existieren Programmfragmente, die viel Parallelität bieten und diese auch genutzt werden muss, um die geforderten Ausführungszeiten einzuhalten. Es gibt aber weitere Programmfragmente, die wenig Parallelität bereitstellen und nur dann innerhalb der geforderten Zeit abgearbeitet werden können, wenn der Prozessor entsprechend hoch getaktet ist. Dieser Prozessortakt kann aber nicht erreicht werden, weil durch die benötigte hohe Parallelität für das erste Programmfragment die Anzahl der Ports den Prozessortakt zu stark reduziert. Im Abschnitt 6.4.1 wird dieser Fall noch genauer diskutiert.

Durch die in Algorithmus 5.1 berechnete Portkonfiguration $minConf = (ip, rp, wp)$ wird garantiert, dass alle Programmfragmente innerhalb der geforderten Ausführungszeiten von einer Architektur ausgeführt werden können, die in keiner Registerbank mehr als $mp = ip + rp + wp$ viele Ports hat. Es ist aber möglich, dass die vorgegebenen Zeitschranken der Programmfragmente unterschritten werden, weil die kürzesten Ablaufpläne, die mit einer kleineren Portkonfiguration ausführbar waren nur knapp die Zeitanforderungen nicht erfüllt haben oder es sich um Programmfragmente handelt, die die Zeitschranken bereits für kleinere Portkonfigurationen eingehalten haben. Dadurch wird es möglich, die real benötigte Taktfrequenz des Prozessors soweit herunterzusetzen, wie es die Zeitschranken erlauben. Aufgrund der bekannten Ausführungshäufigkeiten der Basisblöcke und der Zeitschranken der Programmfragmente ist die **reale Taktfrequenz** $fr$, mit der der Prozessor mindestens getaktet sein muss

$$fr(minConf) = \max\left\{pfr(p, minConf) \middle|\ p \in \mathcal{P}\right\}, \tag{4.4}$$

wobei

$$pfr(p,(ip,rp,wp)) = \frac{\sum_{b \in p} mL(b,(ip,rp,wp)) \cdot exe(b)}{t(p)} \tag{4.5}$$

die niedrigste Taktfrequenz ist, bei der die Zeitschranken von den kürzesten Ablaufplänen der Basisblöcke des Programmfragments $p$, die mit der Portkonfiguration $(ip, rp, wp)$ ausführbar sind, eingehalten werden. Durch die herabgesetzte Taktfrequenz kann die Versorgungsspannung $V_{dd}$ heruntergesetzt werden, wodurch der dynamische Stromverbrauch gesenkt wird.

### 5.2.2 Optimierung des Stromverbrauchs

In diesem Abschnitt wird eine Möglichkeit beschrieben, durch die Auswahl einer Portkonfiguration den Stromverbrauch des zu erzeugenden VLIW-Prozessors zu minimieren. Die durch Algorithmus 5.1 ermittelte minimale Portkonfiguration $minConf = (ip, rp, wp)$ liefert mit $mp = ip + rp + wp$ die mindestens benötigte Portanzahl in der größten Registerbank, damit bei der Ausführung der Programmfragmente die Zeitschranken eingehalten werden können. Wird ein Prozessor, der auch die Programmfragmente innerhalb ihrer Zeitschranken ausführen kann, mit einer größeren Portanzahl in einer Registerbank verwendet, so erhöhen sich bei diesem Prozessor die Hardwarekosten der größten Registerbank. Zur Optimierung der Hardwarekosten wurde die Suche daher nicht weiter fortgesetzt. Soll dagegen der Stromverbrauch minimiert werden, dann kann dies durch eine Architektur, die eine größere Registerbank besitzt, erreicht werden, weil aufgrund der höheren Parallelität kürzere Ablaufpläne zur Ausführung der Basisblöcke verwendet werden können. Dadurch steigt zwar der durch die

Registerbänke und zusätzliche Operatoren verursachte Stromverbrauch an, wegen der kürzeren Ablaufpläne kann aber auch die Taktfrequenz des Prozessors und somit auch die Versorgungsspannung $V_{dd}$ verringert werden, ohne die Ausführungsgeschwindigkeit der Basisblöcke zu verringern. Die Versorgungsspannung geht quadratisch in den dynamischen, linear in den statischen und die Taktfrequenz linear in den dynamischen Stromverbrauch ein (vgl. Formel (1.1) und (1.2)). Das Ziel bei der Optimierung des Stromverbrauchs ist es deshalb, die Taktfrequenz und damit auch die Versorgungsspannung abzusenken.

Der genaue funktionale Zusammenhang zwischen Taktfrequenz und Stromverbrauch ist von verschiedenen Technologiekonstanten bei der Fertigung des Prozessors, der Schaltungsaktivität und den umgeladenen Kapazitäten abhängig. Für alle durch die DSE untersuchten Architekturen sind diese Werte aber konstant. Auch die während der Programmausführung umgeladenen Kapazitäten sind annähernd identisch, weil jede Architektur dieselben Operationen ausführen muss. Aus den in Abschnitt 2.1.3 angegebenen Formeln wird deshalb ein Modell hergeleitet, mit dem der Stromverbrauch in Abhängigkeit von der Portanzahl in den Registerbänken, der Taktfrequenz und den Hardwarekosten (Gatteranzahl) der FUs beschrieben werden kann. Der statische und dynamische Stromverbrauch in den funktionalen Einheiten der Cluster wird durch die Formel

$$P = \alpha_s \cdot V_{dd} \cdot L_g + \alpha_d \cdot f \cdot V_{dd}^2 \tag{4.6}$$

mit den technologieabhängigen Konstanten $\alpha_s$ und $\alpha_d$ beschrieben, wobei $\alpha_s$ die Konstanten aus Formel (1.1) akkumuliert und $\alpha_d$ die Konstanten aus Formel (1.2). Hinzu kommt noch der Stromverbrauch der in jeder Registerbank kubisch mit der Portanzahl wächst [87]. Sei $P_{rbC} : \mathbb{N} \rightarrow \mathbb{R}$ eine Funktion, die den Stromverbrauch einer Registerbank in Abhängigkeit von der Portanzahl angibt, dann ist der Stromverbrauch aller Registerbänke in einem Prozessor mit $mC$ vielen Registerbänken

$$P_{rb}((tp_1,\ldots,tp_{mC})) = \sum_{i=1}^{mC} P_{rbC}(tp_i)\,,$$

wobei $tp_i$ die Anzahl der Ports in der Registerbank des Clusters $i$ ist. Die maximal zulässige Verzögerung *FUDelay* in den funktionalen Einheiten kann aus der realen Taktfrequenz, mit der der Prozessor getaktet werden muss, abgeleitet werden. Aus der maximal erlaubten Verzögerung kann wiederum die mindestens benötigte Versorgungsspannung $V_{dd}$ (vgl. (1.5)) abgeleitet werden. $V_{dd}(fr)$ ist damit abhängig von der realen Taktfrequenz *fr*. Aus der Formel (4.6) kann somit eine Funktion $P$ für den Stromverbrauch eines Prozessors abgeleitet werden, die den statischen und dynamischen Stromverbrauch in Abhängigkeit von der Gatteranzahl $L_g$ in der Schaltung, der realen Taktfrequenz *fr* und der Portanzahl in den Registerbänken des Prozessors $tp_1$, …, $tp_{mC}$ berechnet:

$$P(L_g, fr, (tp_1, \ldots tp_{mC})) = \alpha_s \cdot V_{dd}(fr) \cdot L_g + \alpha_d \cdot fr \cdot \left(V_{dd}(fr)\right)^2 + P_{rb}(tp_1, \ldots tp_{mC}).$$

Zur Optimierung des Stromverbrauchs wird dieselbe Strategie wie zur Optimierung der Registerbankkosten angewendet. Die maximal zulässige Portanzahl *mp* in den Registerbänken wird erhöht, solange

- es eine Portkonfiguration in *MP*(*mp*) und für jeden Basisblock ein Zielprogramm gibt, das mit dieser Portkonfiguration ausgeführt werden kann und
- durch diese Zielprogramme die Zeitschranken bei der maximal möglichen Taktfrequenz *fm*(*mp*) eingehalten werden und
- die Leistungsaufnahme $P(L_g, fr, (tp_1,\ldots,tp_{mC}))$ bei der real benötigten Taktfrequenz *fr* sinkt.

Die real benötigte Taktfrequenz *fr* kann zu der aktuell betrachteten Portkonfiguration aus *MP*(*mp*) nach Formel (4.4) berechnet werden. Weiterhin kann zu den ausgewählten kürzesten Ablaufplänen die Portanzahl in jedem Cluster und eine untere Schranke für die zu implementierenden Operatoren in jedem Cluster bestimmt werden, woraus sich auf die benötigte Gatteranzahl im Prozessor schließen lässt. Die benötigte Portanzahl in jedem Cluster wird zu der aktuell betrachteten Portkonfiguration mittels des maximalen FU-Vektors $vm = (m_1, \ldots, m_{mC})$ berechnet, aus dem die Anzahl der internen Ports für jeden Cluster *i* hervorgeht. Die Anzahl der externen Ports ergibt sich aus der aktuellen Portkonfiguration (*ip*, *rp*, *wp*) für jeden Cluster durch $rp + wp$, so dass $tp_i = 3 \cdot m_i + rp + wp$ ist. Aus der Zuordnung der Basisblockcluster zu Prozessorclustern, die für die Berechnung des maximalen FU-Vektors gewählt wurde, wird für jeden Prozessorcluster die Anzahl der in ihm mindestens zu implementierenden Operatoren berechnet. Deren Anzahl, aufsummiert für alle Cluster, jeden Typ und gewichtet mit *opCost*, liefert eine Abschätzung für die Gatteranzahl $L_g$. Damit kann mittels $P(L_g, fr, (tp_1, \ldots, tp_{mC}))$ für eine Portkonfiguration der Stromverbrauch des Prozessors abgeschätzt werden.

Zur Optimierung des Stromverbrauchs werden alle Portkonfigurationen untersucht, indem für jedes *mp* mit $3 \leq mp \leq mPWidth$ (vgl. Formel (4.3)) die Portkonfigurationen *MP*(*mp*) berechnet werden. Ist die Portkonfiguration (*ip*, *rp*, *wp*) gefunden worden, für die die kürzesten Ablaufpläne den geringsten Stromverbrauch haben, wird als real benötigte Taktfrequenz des Prozessors *fr*((*ip*, *rp*, *wp*)) festgelegt. Niedriger kann die Taktfrequenz nicht gewählt werden, weil dann eine größere Portkonfiguration verwendet werden müsste, in der der zusätzliche Stromverbrauch in den Registerbänken und Operatoren die Einsparungen durch die niedriger gewählte Taktfrequenz übersteigt. Die so gewählte Portkonfiguration (*ip*, *rp*, *wp*) wird, wie die durch Algorithmus 5.1 berechnete minimale Portkonfiguration, mit *minConf* bezeichnet. Die Kandidatenmenge $\mathcal{K}_{minConf}$ enthält dann alle Zielprogramme, die mit dieser Portkonfiguration ausgeführt werden können.

## 5.3 Optimierung der Typkonfiguration

Durch die Portkonfiguration $minConf = (ip, rp, wp)$, die im letzten Abschnitt entweder bezüglich der Registerbankkosten oder des Stromverbrauchs minimiert wurde, ist die maximal verfügbare Anzahl funktionaler Einheiten je Cluster ($ip/3$) und externer Ports festgelegt. Die während der Portoptimierung festgelegte reale Taktfrequenz $fr(minConf)$ wird im Folgenden nur noch als $fr$ bezeichnet. Bei der Berechnung der minimalen Portkonfiguration $minConf$ in den letzten beiden Abschnitten sind immer die kürzesten Ablaufpläne betrachtet worden, die mit $minConf$ ausgeführt werden können und die reale Taktfrequenz wurde soweit wie möglich heruntergesetzt. Trotzdem kann es Programmfragmente geben, die, mit der realen Taktfrequenz ausgeführt, ihre Zeitschranke unterschreiten. Das bedeutet, dass nicht für jeden Basisblock dieser Programmfragmente einer der kürzesten Ablaufpläne aus $\mathcal{K}_{minConf}$ zur Ausführung verwendet werden muss. In längeren Ablaufplänen kann für einzelne Operationstypen die Anzahl der gleichzeitig auszuführenden Operationen dieses Typs kleiner und in einzelnen Clustern die erforderliche Portanzahl in der Registerbank niedriger sein. Das Ziel der Typoptimierung ist es, bei der gegebenen minimalen Portkonfiguration die Anzahl der benötigten Operatoren und FUs in jedem Prozessorcluster zu minimieren, wobei die FU-Anzahl im breitesten Cluster nicht verringert werden kann. Dafür werden zwei Möglichkeiten genutzt:

- Zur Ausführung eines Basisblocks werden nicht die kürzesten Zielprogramme aus $\mathcal{K}_{minConf}$ genutzt, sondern solche Zielprogramme, die möglichst wenig Operatoren in jedem Cluster benötigen, aber trotzdem die Einhaltung der Zeitschranken erlauben.
- Die Zuordnung von Basisblockclustern zu Prozessorclustern wird so gewählt, dass von einem Prozessorcluster solche Basisblockcluster ausgeführt werden, die dieselben Operatoren benötigen.

Die Anforderungen, die ein Zielprogramm $(\alpha, \chi, \phi)_b$ an die verfügbaren Operatoren in jedem Cluster des Prozessors stellt, werden durch die **Typkonfiguration** charakterisiert. Eine Typkonfiguration ist eine Familie $conf$ von Funktionen $conf_c : \mathcal{O} \times \wp(V) \to \mathbb{N}$, die für jeden Operationstypen in einem Cluster $c \in (V_b/\chi - \{[e]\}_\chi)$ und eine Ressourcenallokation $\phi$ angibt, wie viele funktionale Einheiten diesen Operator im Cluster $c$ bereitstellen müssen. $conf$ ist für ein Zielprogramm $(\alpha, \chi, \phi)_b$ definiert als

$$conf_c(t, \phi) = \left| \left\{ [u]_\phi \;\middle|\; type(u) = t \wedge u \in c \right\} \right|$$

für jeden Cluster $c \in (V_b/\chi - \{[e]_\chi\})$. $conf_c$ berechnet somit die Anzahl der FUs, die aufgrund der Bindung $\phi$ den Operator $t$ im Cluster $c$ implementieren müssen und kann größer sein als $TCWidth(\alpha, c, t)$.

Analog zur Typkonfiguration eines Basisblocks wird eine **Clusterkonfiguration** $cConf: \{1, \ldots, mC\} \times \mathcal{O} \rightarrow \mathbb{N}$ definiert, die für jeden Cluster $i \in \{1, \ldots, mC\}$ des zu erzeugenden Prozessors die Anzahl der Operatoren des Typs $t$ in diesem Cluster angibt. $mC$ ist wieder eine obere Schranke für die Anzahl der Cluster, die von einem der Zielprogramme in $\mathcal{K}_{(ip,\ wp,\ rp)}$ genutzt wird. Die Typkonfiguration $conf_c$ des Basisblockclusters $c$ ist kleiner oder gleich der Clusterkonfiguration $cConf$ im Clusters $i$, falls

$$\forall t \in \mathcal{O} : conf_c(t,\phi) \leq cConf(i,t)\,.$$

Es wird eine **maximale Clusterkonfiguration** $mConf: \{1,\ldots, mC\} \times \mathcal{O} \rightarrow \mathbb{N}$ durch

$$mConf(i,t) \coloneqq m_t,\ t \in \mathcal{O}, i \in \{1,\ldots,mC\} \tag{4.7}$$

definiert, wobei

$$m_t = \max\{conf_c(t,\phi) \Big|\ (\alpha,\chi,\phi)_b \in \mathcal{K}_{minConf} \wedge c \in (V_b \,/\, \chi - \{[e]_\chi\})\} \tag{4.8}$$

das häufigste Auftreten des Operationstypen $t$ in einem Basisblockcluster aller Zielprogramme in $\mathcal{K}_{minConf}$ ist. Durch die maximale Clusterkonfiguration $mConf$ wird eine Architektur beschrieben, in der ein Operator des Typs $t$ in jedem Cluster von $m_t$ vielen funktionalen Einheiten implementiert wird. Ein solcher Prozessor erfüllt für jedes Zielprogramm aus $\mathcal{K}_{minConf}$ die Anforderungen, die es an die Operatoranzahl in jedem Cluster stellt. Weiterhin hält der Prozessor die im vorigen Abschnitt festgelegte Portkonfiguration ein, weil alle Ablaufpläne in $\mathcal{K}_{minConf}$ die vorgegebene Portkonfiguration einhalten.

Die Clusterkonfiguration eines Prozessors, der in jedem Cluster möglichst wenig Operatoren und FUs bereitstellt, wird berechnet, indem die maximale Clusterkonfiguration als Startwert verwendet und die Verfügbarkeit der Operatoren in jedem Cluster schrittweise verringert wird. Für jede so entstehende Clusterkonfiguration wird geprüft, ob es noch für jeden Basisblock ein Zielprogramm in $\mathcal{K}_{minConf}$ gibt, das mit dieser Clusterkonfiguration ausgeführt werden kann und ob mit diesen Zielprogrammen die Zeitschranken für die Programmfragmente eingehalten werden können. Um zu prüfen, ob ein Zielprogramm $(\alpha, \chi, \phi)$ mit einer gegebenen Clusterkonfiguration $cConf$ ausgeführt werden kann, muss eine Zuordnung der Cluster des Zielprogramms $(\alpha, \chi, \phi)_b$ zu den Prozessorclustern bekannt sein. Solch eine Zuordnung wird durch eine **Clusterabbildung** beschrieben. Eine Clusterabbildung ist eine injektive Funktion

$$\psi:\ V/\chi - \{[e]_\chi\} \rightarrow \{1, \ldots, mC\},$$

die die Cluster des Basisblocks $b_\chi$ den Prozessorclustern zuordnet. Gegebenenfalls wird durch $\psi_b$ deutlich gemacht, dass $\psi$ die Clusterabbildung des Basisblocks $b$ ist. Das Zielprogramm $(\alpha, \chi, \phi)$ kann mit der Cluster-

konfiguration *cConf* ausgeführt werden, falls eine Clusterabbildung $\psi$ existiert, so dass:

$$\forall c \in (V/\chi - \{[e]_\chi\}) \forall t \in \mathcal{O} : conf_c(t, \phi) \leq cConf(\psi(c), t).$$

Eine solche Clusterabbildung $\psi$ wird als **zulässig** unter *cConf* bezeichnet. Es sei $mtL(b, cConf)$ die Länge des kürzesten Ablaufplans in $\mathcal{K}_{minConf}$ zum Basisblock $b$, für den es eine zulässige Clusterabbildung $\psi$ unter *cConf* gibt. Ein Programmfragment $p$ kann unter der Clusterkonfiguration *cConf* und Einhaltung seiner Zeitschranke ausgeführt werden, wenn es für jeden Basisblock $b \in p$ eine zulässige Clusterabbildung unter *cConf* gibt und

$$\sum_{b \in p} \frac{mtL(b, cConf) \cdot exe(b)}{fr} \leq t(p) \tag{4.9}$$

gilt. Eine Clusterkonfiguration *cConf* wird als **zulässig** bezeichnet, wenn alle Programmfragmente aus $\mathcal{P}$ unter dieser Clusterkonfiguration und Einhaltung ihrer Zeitschranke ausgeführt werden können.

Der VLIW-Prozessor soll von allen zulässigen Clusterkonfigurationen die mit den kleinsten Hardwarekosten besitzen. Um die Hardwarekosten einer Clusterkonfiguration zu bestimmen, werden die Kosten der zu implementierenden Operationstypen herangezogen, die eine untere Schranke der Datenpfadkosten sind (vgl. Formel (1.7)) und die sich direkt aus der Clusterkonfiguration durch

$$mDPCost(cConf) = \sum_{i=1}^{mC} \sum_{t \in \mathcal{O}} (opCost(t) \cdot cConf(i, t))$$

berechnen lassen. Weil für jeden Basisblock alle möglichen Clusterabbildungen der Basisblockcluster auf Prozessorcluster betrachtet werden, kann eine zulässige Clusterabbildung bis zu $ip + rp + wp$ viele Ports in jeder Registerbank des Prozessors erfordern. In Abbildung 5.2 ist dafür ein Beispiel angegeben.

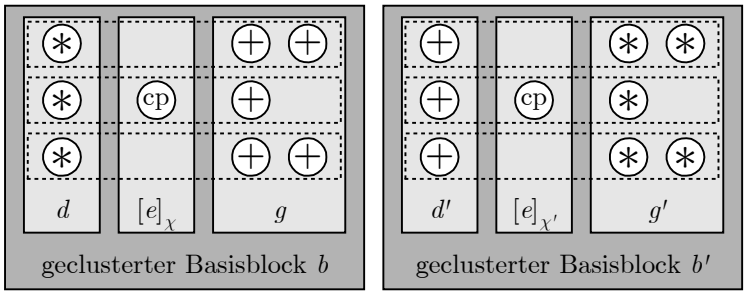



Abbildung 5.2: Beispiel zur Clusterabbildung.

Es sind die Ablaufpläne zweier geclusterter Basisblöcke $b$ und $b'$ dargestellt, deren Operationen auf jeweils zwei Basisblockcluster $d$ und $g$ bzw. $d'$ und $g'$

verteilt sind. $mC$ hat den Wert zwei. Der externe Cluster enthält eine Kopieroperation ($cp$). Für eine Clusterkonfiguration, die in jedem Cluster zwei Addierer und zwei Multiplizierer zulässt, ist auch die Clusterabbildung $\psi_b(d) = \psi_{b'}(g') = 1$ und $\psi_b(e) = \psi_{b'}(g') = 2$ zulässig. Damit werden in jedem Prozessorcluster zwei FUs und damit sechs interne Ports benötigt. Für die ebenfalls zulässige Clusterabbildung $\psi'_b(d) = \psi'_{b'}(d') = 1$ und $\psi'_b(g) = \psi'_{b'}(g') = 2$ werden nur im Prozessorcluster 2 sechs interne Ports benötigt. Im Prozessorcluster 1 genügen drei interne Ports. Der maximale FU-Vektor für die zwei in Abbildung 5.2 dargestellten Ablaufpläne ist $vm = (2, 1)$. Zur Berechnung dieses FU-Vektors wurde eine Clusterabbildung genutzt, die der Clusterabbildung $\psi'$ entspricht. In dem Beispiel ist anhand der Clusterabbildung $\psi$ zu erkennen, dass es zulässige Clusterabbildungen geben kann, unter denen mehr FUs in einigen Prozessorclustern benötigt werden, als durch den maximalen FU-Vektor $vm = (m_1, \ldots, m_{mC})$, der zu der Portkonfiguration ($ip$, $rp$, $wp$) berechnet wurde, abgeschätzt wurde. Die Anzahl dieser zusätzlich benötigten FUs wird genutzt, um für jedes Zielprogramm eine zulässige Clusterabbildung auszuwählen. Für jedes kürzeste Zielprogramm $(\alpha, \chi, \phi)_b$ für das es mindestens eine zulässige Clusterabbildung gibt, wird die zulässige Clusterabbildung $\psi_b$ ausgewählt, für die

$$\sum_{c \in V_b \,/\, \chi} \max(0, FUCWidth(\alpha, c) - m_{\psi_b(c)}) \tag{4.10}$$

minimal ist, wobei $m_{\psi_b(c)}$ die durch den maximalen FU-Vektor abgeschätzte FU-Anzahl im Prozessorcluster $\psi_b(c)$ ist. Es wird also für jeden Prozessorcluster die Anzahl der funktionalen Einheiten aufsummiert, die über den, im maximalen FU-Vektor abgeschätzten Wert hinaus noch benötigt werden. Auf diese Weise wird für jeden Basisblock $b$ ein Zielprogramm $(\alpha, \chi, \phi)_b$ und die zugehörige Clusterabbildung $\psi_b$ ausgewählt, die einen möglichst geringen Zuwachs bei den Registerbankkosten in alle Clustern erfordert. Die Menge dieser ausgewählten Zielprogramme zur Clusterkonfiguration $cConf$ sei $\mathcal{Z}$. Der Prozessor, der diese ausgewählten Zielprogramme unter der jeweiligen Clusterabbildung $\psi_b$ ausführen kann, muss in jedem Cluster $i$

$$F(i) = \max\left\{ FUCWidth(\alpha, c) \,\middle|\, (\alpha, \chi, \phi)_b \in \mathcal{Z} \wedge c \in V_b \,/\, \chi \wedge \psi_b(c) = i \right\}$$

viele FUs bereitstellen. Die interne Portanzahl im Cluster $i$ beträgt damit $3 \cdot F(i)$. Für die Gesamtportanzahl muss noch die Anzahl der externen Ports aus der Portkonfiguration $minConf = (ip, rp, wp)$ beachtet werden. Aus diesen Werten können die Registerbankkosten für den Cluster $i$ berechnet werden. Die einer Clusterkonfiguration $cConf$ anzurechnenden Gesamtkosten $confCost$ ergeben sich somit aus den Registerbankkosten in den Clustern und den minimalen Datenpfadkosten:

$$confCost(cConf) = \left( \sum_{i=1}^{mC} RFCCost(regs_i, (3 \cdot F(i), rp, wp)) \right) + mDPCost(cConf)\,,$$

wobei $regs_i = 3 \cdot F(i) + rp + wp$ die Anzahl der Register in der Registerbank $i$ ist.

In Algorithmus 5.2 ist das Prinzip der Berechnung einer bezüglich *confCost* minimalen Clusterkonfiguration angegeben. Der Aufruf erfolgt mit der maximalen Clusterkonfiguration. Diese ist immer zulässig. Es werden daraus durch rekursive Aufrufe weitere Clusterkonfigurationen erzeugt, indem die Verfügbarkeit der einzelnen Operatoren in den Clustern verringert wird, bis die entstehende Clusterkonfiguration nicht mehr zulässig ist. In einer unzulässigen Clusterkonfiguration wird die Verfügbarkeit von Operatoren nicht mehr verringert, weil die Clusterkonfiguration dadurch nicht zulässig werden kann und die Suche bricht an diesen Stellen ab. Für jede zulässige Clusterkonfiguration werden die Gesamtkosten *confCost* ermittelt.

**Algorithmus 5.2 (*Typoptimierung*)**

```
Eingabe: maximale Clusterkonfiguration cConf
Ausgabe: Minimierte Clusterkonfiguration mincConf

if(cConf ist nicht zulässig) then
    return Clusterkonfiguration c mit confCost(c) = ∞;
else
  mincConf:=cConf;
  for i:=1 to mC do
    foreach t ∈ 𝒪 do
      if(cConf(i,t) > 1) then
        cConf(i,t):=cConf(i,t) - 1;
        mc:=Typoptimierung(cConf)    // rekursiver Aufruf
        cConf(i,t):=cConf(i,t) + 1;
        if(confCost(mc) < confCost(mincConf)) then
          mincConf:=mc
        fi
      fi
    od
  od
  return mincConf
fi
```

Zu der durch Algorithmus 5.2 gefundenen Clusterkonfiguration *mincConf* wird für jeden Basisblock aus $\mathcal{B}$ das kürzeste Zielprogramm $(\alpha, \chi, \phi)_b$ als Kandidat zur Ausführung von $b$ verwendet, für das die Formel (4.10) minimal ist. Die Menge dieser Zielprogramme $\mathcal{Z}$ und die zugehörigen Clusterabbildungen $\psi_b$ wurde bereits während der Ausführung von Algorithmus 5.2 zu der Clusterkonfiguration *mincConf* bestimmt. Die Menge dieser Zielprogramme wird mit $\mathcal{TZ}$ bezeichnet. Im Abschnitt 5.4 wird der Zielprozessor erzeugt, der alle diese Zielprogramme gemeinsam ausführen kann.

Die so errechnete Clusterkonfiguration lässt noch Möglichkeiten zur Optimierung offen, weil zu einer Clusterkonfiguration immer der kürzeste Ablaufplan für jeden Basisblock ermittelt wurde, der mit dieser Konfiguration ausführbar ist. Es ist aber möglich, dass für einzelne Basisblöcke längere Ablaufpläne ausgewählt werden können, ohne dass die Zeitschranke eines Programmfragments überschritten wird. Enthält ein Programmfragment genau einen Basisblock, dann ist dieses Problem für dieses Programmfragment in polynomieller Zeit exakt lösbar, indem für jeden Basisblock das längste Zielprogramm, das mit der ermittelten Clusterkonfiguration *mincConf* ausgeführt werden kann und dabei die Zeitschranke einhält, gesucht wird. Enthält ein Programmfragment dagegen mehrere Basisblöcke, so muss für die festgelegte Port- und Typkonfiguration eine geeignete Kombination der Zielprogramme dieser Basisblöcke gesucht werden. Diese Kombination muss so gewählt sein, dass die Zeitschranke des Programmfragments noch eingehalten wird. Durch diese Optimierung kann die bereits errechnete Clusterkonfiguration nicht verbessert werden. Es können aber Operatorkonflikte, die erst aufgrund von Instruktionen aus verschiedenen Ablaufplänen in der abschließenden Ressourcenallokationsphase entstehen, verringert werden, wodurch zusätzlich bereitzustellende Operatoren in jedem Cluster eingespart werden können.

Weil Algorithmus 5.2 alle zulässigen Clusterkonfigurationen betrachtet, hat die Suche im Allgemeinen einen exponentiellen Aufwand. Allerdings ist die Anzahl der zu untersuchenden Clusterkonfigurationen stark eingeschränkt, da mit sinkender Anzahl verfügbarer Operatoren in den einzelnen Clustern sehr schnell keine Zielprogramme mehr gefunden werden, die die vorgegebenen Zeitschranken einhalten. Die Ursache dafür ist die Einschränkung des Suchraums durch die vorangegangene Portoptimierung und die festgelegte reale Taktfrequenz. Beides hat dazu geführt, dass für mindestens ein Programmfragment $p$ bei der Wahl von längeren Ablaufplänen für die Basisblöcke aus $p$ die Zeitschranken nicht mehr eingehalten werden können.

Wenn für einen Ablaufplan geprüft wird, ob er mit der gegebenen Clusterkonfiguration ausgeführt werden kann, werden in der gegenwärtigen Implementierung alle möglichen Clusterabbildung $\psi$ untersucht. Die Anzahl der zu untersuchenden Clusterabbildungen steigt ebenfalls exponentiell mit der Anzahl der Cluster. Für viele reale Basisblöcke genügen aber bis zu fünf Cluster. Daher ist bei dieser Größenordnung eine vollständige Überprüfung möglich, da höchstens 120 Clusterabbildungen je Zielprogramm überprüft werden müssen. Sollen durch die Typoptimierung auch Architekturen mit mehr Clustern untersucht werden, dann muss der Suchraum stärker eingeschränkt werden. Entsprechende Algorithmen zur Verbesserung der Suchstrategie bei der Typoptimierung können Gegenstand weiterer Untersuchungen sein.

## 5.4 Abschließende Ressourcenallokation

Nachdem durch die Typoptimierung für jeden Basisblock ein Zielprogramm als Kandidat ausgewählt wurde, wird durch die abschließende Ressourcenallokation eine Architektur erzeugt, die alle diese Kandidaten gemeinsam ausführen kann. Die abschließende Ressourcenallokation berechnet für alle Zielprogramme in $\mathcal{TZ}$ die endgültigen FU-Typen in jedem Cluster des Prozessors. Die durch die Typoptimierung berechnete Clusterkonfiguration gibt nur eine untere Schranke für die benötigten Operatoren in jedem Cluster an, da aufgrund der Ablaufpläne von verschiedenen Basisblöcken, die vom selben Prozessorcluster ausgeführt werden müssen, Operatorkonflikte entstehen können, die innerhalb der Instruktionen eines Ablaufplans noch nicht vorhanden waren. Zur Berechnung der Ressourcenallokation werden die Ablaufpläne $\alpha_1$, ..., $\alpha_n$ der Zielprogramme aus

$$\mathcal{TZ} = \{(\alpha_1, \chi_1, \phi_1)_{b_1}, \ldots, (\alpha_n, \chi_n, \phi_n)_{b_n}\}$$

zu einem neuen Ablaufplan $\alpha$ zusammengesetzt, der aus der Aneinanderreihung der Instruktion dieser Ablaufpläne besteht. Für $\alpha$ gilt:

$$\alpha(k) = \alpha_i(m) \Leftrightarrow k = \left(\sum_{j=1}^{i-1} \left|\alpha_j\right|\right) + m\,.$$

Weiterhin wird für $1 \leq k \leq n$ aus den Clusterungen $\chi_k$ der Basisblöcke $b_k$ mittels der Clusterabbildungen $\psi_k$, die während der Typoptimierung ermittelt wurden, eine Clusterung $\chi$ gebildet. Für diese Clusterung $\chi$ gilt $(u, v) \in \chi$ genau dann, wenn

- $\exists i \in \mathbb{N} : 1 \leq i \leq n \wedge (u, v) \in \chi_i$ oder
- $\exists i, j \in \mathbb{N} : u \in V_{b_i}$ und $v \in V_{b_j}$ und $\psi_i([u]_{\chi_i}) = \psi_j([v]_{\chi_j})$.

Für den so definierten Ablaufplan $\alpha$ und die Clusterung $\chi$ wird die Bindung $\phi$, wie in Abschnitt 4.2 beschrieben, berechnet. Diese bildet zusammen mit dem Ablaufplan $\alpha$ und der Clusterung $\chi$ ein Zielprogramm $(\alpha, \chi, \phi)$ für das der Prozessor $(\mathcal{C}, \mathcal{F}, cf, ft, rz, rpc)$, wie am Ende des Abschnitts 2.2 beschrieben, ermittelt wird. Dieser Prozessor kann jeden Basisblock $b$ mit der spezifizierten Ausführungshäufigkeit $exe(b)$ ausführen und hält dabei die für jedes Programmfragment vorgegebene Zeitschranke ein.

# 6 Ergebnisse

Der vorgestellte DESCOMP-Ansatz gestattet eine vollständig automatisierte Design-Space-Exploration von geclusterten VLIW-Prozessoren zur Bestimmung

- der Anzahl der benötigten Cluster,
- der Portkonfiguration der Registerbänke,
- von Anzahl und Typ jeder FU in den Clustern sowie
- der Anzahl der benötigten Kopieroperatoren.

Während der DSE wird außerdem Zielcode für die Basisblöcke der Anwendung generiert, der sich mit dem erzeugten VLIW-Prozessor ausführen lässt. Zur Bewertung der erzielten Ergebnisse nutzt die vorliegende Arbeit die Resultate, die in der Dissertation von Viktor S. Lapinskii [99] veröffentlicht wurden. Die Ergebnisse aus Lapinskii's Arbeit sind 2000 auf der ICCAD (*International Conference on Computer Aided Desing*) und 2004 auf der DAC (*Design Automation Conference*) ausgezeichnet worden[1] [100, 101]. Dass sowohl der mit DESCOMP erzeugte Zielcode als auch die erzeugten Architekturen eine hohe Qualität haben, wird in diesem Abschnitt durch einen Vergleich mit Lapinskii's Ergebnissen belegt, indem gezeigt wird, dass

- die DESCOMP-Architektur weniger Hardware durch kleinere Registerbänke benötigt als die Lapinskii-Architektur, um bei gleicher Ablaufplanlänge einen Basisblock auszuführen, und
- die mit DESCOMP erzeugten Ablaufpläne in vielen Fällen kürzer sind als Ablaufpläne, die zu einer Lapinskii-Architektur generiert wurden, obwohl die zur Ausführung benötigte DESCOMP-Architektur nicht mehr Parallelität auf Instruktionsebene bereitstellt als die Lapinskii-Architektur.

Das bedeutet, dass durch DESCOMP der DSE-Ansatz von Lapinskii verbessert wird, indem

- kostengünstigere Architekturen erzeugt werden, die dieselbe oder eine höhere Ausführungsgeschwindigkeit erlauben, bzw.

[1] http://www.electronicstalk.com/news/aoc/aoc102.html

- bei gleichen Hardwarekosten und gleichem Stromverbrauch eine höhere Ausführungsgeschwindigkeit aufgrund der kürzeren Ablaufpläne möglich ist.

Weil die Qualität der global optimierten Architektur wesentlich von der Qualität der lokal erzeugten Architekturen abhängt, gliedert sich die Darstellung der Ergebnisse in zwei Teile. Der erste Teil bewertet die Qualität der lokal optimierten Architekturen. Zum Vergleich werden die Ergebnisse aus der Arbeit von Lapinskii genutzt. Im zweiten Teil wird beispielhaft eine DSE für eine global optimierte Architektur durchgeführt und es werden Aussagen über die Qualität der globalen Optimierung getroffen. Zuvor wird die für DESCOMP verwendete Konfiguration während der DSE beschrieben.

## 6.1 Implementierung und Konfiguration

Der in dieser Arbeit beschriebene DESCOMP-Ansatz wurde als Prototyp in Java überwiegend durch studentische Hilfskräfte implementiert. Der Prototyp diente der Ermittlung der Messwerte und wurde genutzt, um während der Entwicklung modifizierte Varianten der vorgestellten Heuristiken auszuprobieren und deren Qualität gegeneinander abzuwägen. Die von der DESCOMP-Implementierung benötigten Eingabeinformationen für eine Design-Space-Exploration sind in Tabelle 6.1 mit ihrer Bedeutung angegeben.

| Datei | Beschreibung |
|---|---|
| Operatordatei | Enthält die zulässigen Operatoren mit ihren Bezeichnungen, Hardwarekosten, Latenzzeiten und die Funktion *fm*, die die Abhängigkeit der Taktfrequenz von der maximalen Portanzahl beschreibt. |
| Basisblockdatei(en) | Jede Basisblockdatei enthält einen Steuerflussgraphen mit den zugehörigen Basisblöcken. Die Basisblöcke sind durch einen gerichteten azyklischen Graphen spezifiziert. Die Operationen in den Basisblöcken dürfen nur einen in der Operatordatei definierten Typ besitzen. |
| Profilingdatei | Spezifiziert die Programmfragmente mit maximalen Ausführungszeiten und die zugehörigen Basisblöcke mit Ausführungshäufigkeiten. Die Basisblöcke müssen in einer Basisblockdatei definiert sein. |

Tabelle 6.1: Von DESCOMP für die DSE benötigte Dateien.

Die gesamte DSE wird durch die Profilingdatei gesteuert. Die in den Basisblockdateien definierten Basisblöcke werden durch die Profilingdatei nur referenziert. Die Programmrepräsentation ist somit vollständig von den für

die Design-Space-Exploration benötigten Informationen getrennt. Die in der Operatordatei angegebenen Parameter der Registerbank umfassen die Abhängigkeit der Taktfrequenz von der Portanzahl in der Registerbank und von den Verzögerungen in den Operatoren. Das heißt, für eine niedrige Portanzahl ist die Taktfrequenz konstant und durch die Verzögerung in den Operatoren bestimmt. Erst bei einer höheren Portanzahl wird die Taktfrequenz durch die Registerbank und den Bypass dominiert und sinkt. Die Eigenschaften der Operatortypen, die während der DSE benötigt wurden, können der folgenden Tabelle entnommen werden.

| **Operatortyp $t$** | * | + | – | & | \| | ! | *xor* | *shl* | *copy* |
|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{opCost(t)}$ | 30 | 4 | 4 | 1 | 1 | 1 | 1 | 2 | 1 |
| $\boldsymbol{lat(t)}$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Tabelle 6.2: Bei der DSE angenommene Operatorkosten und Latenzzeiten.

Die Latenzzeiten entsprechen den Latenzzeiten aus der Arbeit von Lapinskii. Zu den Operatorkosten ist in der Arbeit von Lapinskii keine Angabe gemacht worden. Die für die DSE angenommene Abhängigkeit der Taktfrequenz von der Portanzahl einer Registerbank ist in der folgenden Tabelle angegeben.

| **Portanzahl $p$** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{fm(p)}$ **in MHz** | 90 | 90 | 90 | 90 | 88 | 85 | 82 | 79 | 75 | 71 | 67 |
| **Portanzahl $p$** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** |
| $\boldsymbol{fm(p)}$ **in MHz** | 63 | 58 | 53 | 48 | 43 | 37 | 31 | 25 | 19 | 13 | 10 |

Tabelle 6.3: Abhängigkeit der Taktfrequenz von der Portanzahl.

Neben den in Tabelle 6.2 und Tabelle 6.3 angegebenen Parametern haben die Werte der Gewichtungskonstanten in der Zielfunktion $z$ (vgl. Seite 64) während der Planung einen wesentlichen Einfluss auf die Qualität der DSE. Die verwendeten Einstellungen sind in der folgenden Tabelle angegeben.

| **Parameter** | **Wert** | **Beschreibung** |
|---|---|---|
| $\alpha_1$ | 1 | Gewicht von $\Delta f$ in der Kostenfunktion *RBLoad* |
| $\alpha_2$ | 0.01 | Gewicht von $\Delta rf$ der Zielfunktion *RBLoad* |
| $\alpha$ | 1000 | Gewicht von *RBLoad* in der Zielfunktion $z$ |
| $\beta$ | 2/3 | Gewicht von *TLoad* in der Zielfunktion $z$ |
| $\gamma$ | 1/3 | Gewicht von *TDLoad* in der Zielfunktion $z$ |
| $p$ | 1 | Wahrscheinlichkeit in der Zielfunktion *TDLoad* |
| *Versuche2* | 3 | Verbesserungsversuche in der zweiten Clusterungsstufe |

Tabelle 6.4: Werte der Gewichtungskonstanten während der durchgeführten DSE.

Aufgrund dieser Wahl der Gewichtungskonstanten wird während der lokalen Optimierung der Minimierung der maximalen Portanzahl in der größten Registerbank die höchste Priorität gegeben vor der Minimierung der Gesamtportanzahl im Prozessor und der Minimierung der Operatoren in den Clustern.

## 6.2 Benchmarkprogramme und Vergleichbarkeit

Mit dem DESCOMP-Prototypen und den angegebenen Einstellungen wurde eine Design-Space-Exploration für die Basisblöcke der Schleifenkörper von typischen Signalverarbeitungsalgorithmen und von Anwendungen aus dem Bereich des wissenschaftlichen Rechnens durchgeführt. Diese Benchmarkprogramme sind in Tabelle 6.5 mit ihren wesentlichen Eigenschaften angegeben. Die zugehörigen Datenflussgraphen sind im Anhang A aufgeführt.

| **Basisblock** | | | | **Operationstypen** | | |
|---|---|---|---|---|---|---|
| **Name (Abkürzung)** | **Knoten-anzahl** | **Kritische Pfadlänge** | **Kompo-nenten** | + | * | – |
| *Eliptic Wave Filter* (EWF) | 34 | 14 | 1 | 26 | 8 | 0 |
| *Auto Regression Filter* (ARF) | 28 | 8 | 1 | 12 | 16 | 0 |
| *Fast Fourier Transform* (FFT) | 38 | 4 | 3 | 9 | 12 | 17 |
| *Discrete Cosine Transform* (DCT-LEE) | 49 | 9 | 2 | 17 | 21 | 11 |
| *Discrete Cosine Transform* (DCT-DIF) | 41 | 7 | 2 | 17 | 12 | 12 |
| *Discrete Cosine Transform* (DCT-DIT) | 48 | 7 | 1 | 24 | 12 | 12 |
| *Shallow Water Modeling* (SWIM1) | 26 | 4 | 3 | 10 | 8 | 8 |
| *Shallow Water Modeling* (SWIM2) | 15 | 5 | 3 | 6 | 6 | 3 |
| *Wuppertal Wilson Fermion Solver* (WUPWISE) | 14 | 3 | 2 | 4 | 8 | 2 |
| *Seismic Wave Propagation Simulation* (EQUAKE1) | 36 | 4 | 6 | 18 | 18 | 0 |
| *Seismic Wave Propagation Simulation* (EQUAKE2) | 36 | 4 | 6 | 18 | 18 | 0 |
| *Modeling large systems of molecules* (AMMP) | 24 | 3 | 1 | 9 | 9 | 6 |

Tabelle 6.5: Zusammenfassung der Basisblockeigenschaften der verwendeten Benchmarkprogramme aus der Arbeit von Lapinskii.

Bei EWF, ARF, FFT, DCT-LEE, DCT-DIF und DCT-DIT handelt es sich um Basisblöcke, die häufig in Algorithmen für Bild- und Signalverarbeitung vorkommen. Ursprünglich stammen diese Basisblöcke aus dem MEDIABench [14]. Die übrigen Basisblöcke wurden ursprünglich dem SPEC2000-Benchmark [53] entnommen, das unter anderem zahlreiche Simulations-

anwendungen aus dem Bereich der Chemie, Physik, Geologie und Meteorologie enthält. Die einzigen Operationstypen in diesen Benchmarkprogrammen sind +, – und ∗. In Lapinskii's Dissertation sind die Basisblöcke dieser Benchmarkprogramme bereits als Datenflussgraphen angegeben und konnten unverändert von DESCOMP als Eingabe verwendet werden. Dadurch werden der DESCOMP- und Lapinskii-Ansatz objektiv vergleichbar, weil Einflüsse eines Übersetzungsprozesses von der Quellsprache in die Repräsentation als Datenflussgraph ausgeschlossen werden können.

Aufgrund der automatisierten DSE-Methode in Lapinskii's Arbeit wurden dort zu einem Basisblock immer kürzer werdende Ablaufpläne erzeugt, indem in einem Durchlauf die Clusteranzahl in der Architektur schrittweise bei fester maximaler FU-Anzahl je Cluster erhöht wurde und für jede dieser Architekturen Ablaufpläne erzeugt wurden. Diese Durchläufe wurden mit steigender maximaler FU-Anzahl durchgeführt. Die Bestimmung der Konfiguration des zusätzlichen Clusters ist einfach möglich, weil nur zwei Operationstypen zugelassen waren, die nicht in einer gemeinsamen FU implementiert werden durften (vgl. Abschnitt 3.2.5). Die Operationstypen + und – in den Benchmarkprogrammen werden deshalb wie ein Operationstyp behandelt. Durch dieses Vorgehen bei der DSE in Lapinskii's Arbeit sind zahlreiche Architekturen ermittelt worden, die sich zur Ausführung des Basisblocks eignen und sich in der Clusteranzahl und der maximalen FU-Anzahl in den Clustern unterscheiden. Außerdem wurde zu jeder dieser Architekturen die kürzeste Ablaufplanlänge berechnet, mit der sich der Basisblock auf dieser Architektur ausführen lässt. Bezüglich dieser Ablaufplanlängen werden die Lapinskii- und DESCOMP-Architekturen gegenübergestellt und ihre Anforderungen an die Registerbänke verglichen. Dieses Vorgehen ist gerechtfertigt, weil in beiden Arbeiten dieselbe Architektur bezüglich der Registerbankgröße optimiert wird. In der Arbeit von Lapinskii wird als Maß die FU-Anzahl im größten Cluster genutzt. Aufgrund der gewählten Gewichte in Tabelle 6.4 ist das primäre Optimierungsziel bei DESCOMP die Portanzahl in der größten Registerbank. Diese wird aber wesentlich durch die Anzahl der funktionalen Einheiten im entsprechenden Cluster bestimmt. Einen geringen Einfluss haben die externen Ports für die Kopieroperationen. Zwar erzeugt DESCOMP die Anzahl der benötigten Kopieroperatoren automatisch, bei den Ergebnissen stellt sich aber heraus, dass in fast allen Fällen maximal zwei Kopieroperationen parallel ausgeführt werden müssen, was der Anzahl der Kopieroperatoren in den Lapinskii-Architekturen entspricht. Damit werden, wie bei Lapinskii, maximal vier Ports für die Kopieroperatoren je Registerbank benötigt. Bei DESCOMP ist das sekundäre Optimierungsziel die Gesamtanzahl der Ports in allen Registerbänken. Da deren Anzahl bei gleicher Clusteranzahl und gleicher externer Portanzahl in jeder Registerbank ebenfalls von den funktionalen Einheiten abhängt, können die DESCOMP-Architekturen und Lapinskii-Architekturen auch bezüglich dieser Größe verglichen werden.

## 6.3 Qualität der lokalen Optimierung

In diesem Abschnitt wird die Qualität der mit DESCOMP erzeugten Architekturen beurteilt, die zur Ausführung eines einzelnen Basisblocks mit fester Ablaufplanlänge optimiert wurden. Als Referenz dienen die Ergebnisse aus der Arbeit von Lapinskii. Es wird die Anzahl der funktionaler Einheiten im breitesten Cluster bzw. der Ports in der größten Registerbank zwischen einer DESCOMP- und Lapinskii-Architektur verglichen, die beide dasselbe Benchmarkprogramm mit derselben Ablaufplanlänge ausführen können. Bei geclusterten Architekturen werden zusätzlich die Gesamtanzahl funktionaler Einheiten bzw. Ports in der Architektur gegenübergestellt, sofern in der Arbeit von Lapinskii entsprechende Angaben gemacht wurden. Dadurch wird belegt, dass mit DESCOMP fast immer eine kostengünstigere Architektur als mit Lapinskii's Ansatz gefunden wird, die genauso schnell den Basisblock abarbeiten kann. Dieser Vergleich wird in Abschnitt 6.3.1 für eine ungeclusterte und in Abschnitt 6.3.2 für eine geclusterte Architektur durchgeführt. Beim Vergleich für eine ungeclusterten Architektur entfällt der Einfluss des Clusterungsalgorithmus. So kann die Qualität der Planung und Ressourcenallokation beurteilt werden. In Abschnitt 6.3.3 wird belegt, dass mit DESCOMP kürzere Ablaufpläne als mit Lapinskii's Ansatz gefunden werden, die mit einer DESCOMP-Architektur abgearbeitet werden können, die nicht mehr Parallelität bereitstellt, als die Lapinskii-Architektur. Die für die Vergleiche durch DESCOMP erzeugten Architekturen sind im Anhang B angegeben. Bei deren Erzeugung wurden die Operationstypen + und – in den Benchmarkprogrammen als verschiedene Operationstypen behandelt, um zu demonstrieren, dass der DESCOMP-Ansatz problemlos mehr als zwei Operationstypen verarbeiten kann. Beim Vergleich der DESCOMP- und Lapinskii-Architekturen wurde der Operator – in den DESCOMP-Architekturen wie ein + behandelt.

### 6.3.1 Registerbankkosten ungeclusterter Architektur

Beim Vergleich ungeclusterter Architekturen wird nur die Anzahl der funktionalen Einheiten betrachtet. Bei den vorliegenden Benchmarkprogrammen entspricht deren Anzahl genau einem Drittel der erforderlichen Portanzahl in der Registerbank, da keine Kopieroperationen benötigt werden und in den Basisblöcken keine Lade- oder Speicheroperationen vorkommen. Somit ist die Anzahl der externen Ports in der Registerbank null. Die Ergebnisse des Vergleichs sind in Abbildung 6.1 oben für das MEDIABench und in Abbildung 6.1 unten für das SPEC2000-Benchmark dargestellt. In den Diagrammen ist für mehrere Ablaufplanlängen jeweils durch zwei Säulen die benötigte FU-Anzahl in der Lapinskii-Architektur (linke Säule) und die benötigte FU-Anzahl in der DESCOMP-Architektur (rechte Säule) bei gleicher Ablaufplanlänge gegenübergestellt. Da in der Arbeit von Lapinskii

die Ablaufpläne zu einer Architektur mit einer festen FU-Anzahl erzeugt wurden, existiert nicht für jede Ablaufplanlänge eine Architektur. Teilweise wurden in der Arbeit von Lapinskii auch zu verschiedenen Architekturen Ablaufpläne mit derselben Länge erzeugt. Der Vergleich nutzt dann die kostengünstigere Architektur.

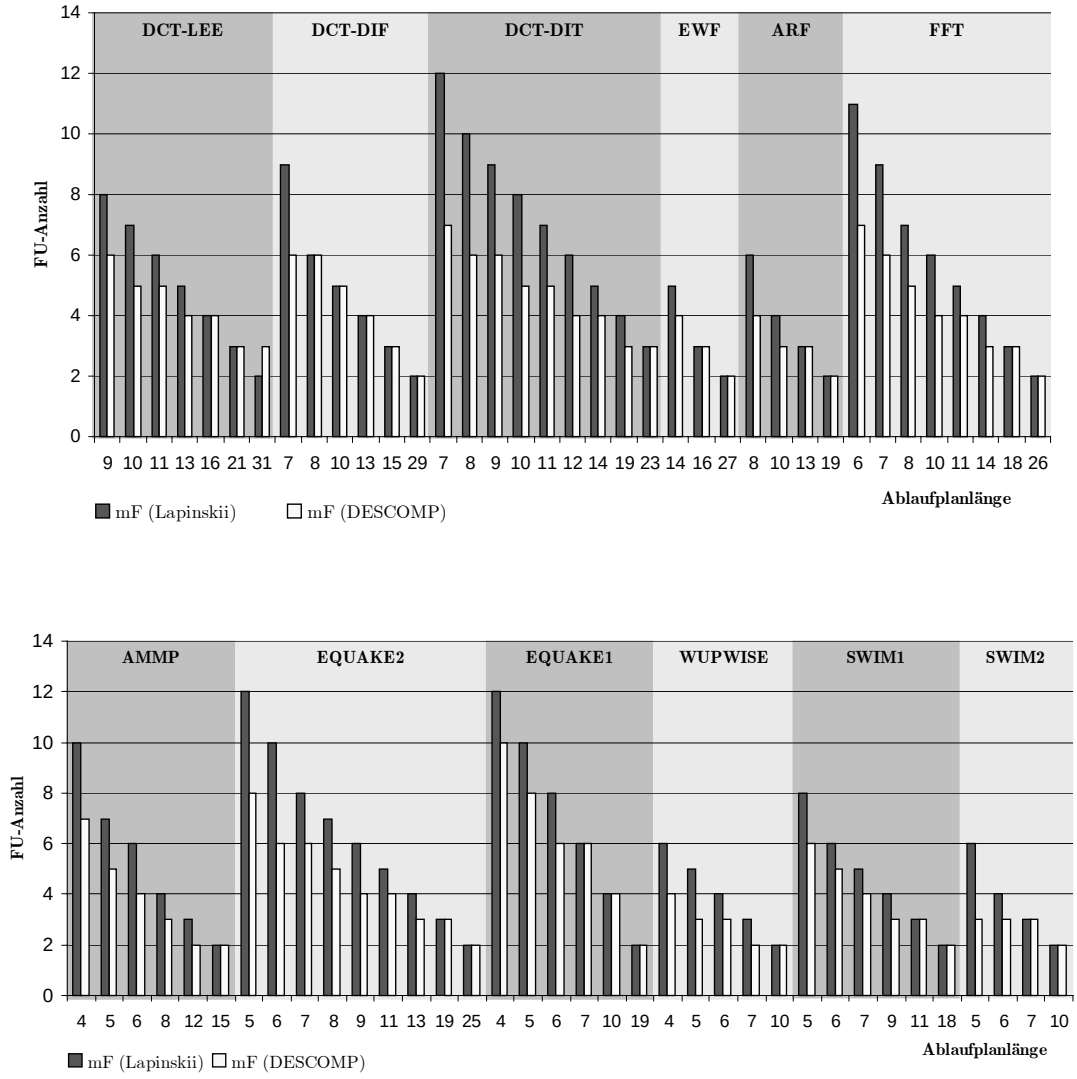



Abbildung 6.1: Gegenüberstellung der FU-Anzahl ($mF$) bei der Lapinskii-Architektur (linke Säule) und DESCOMP-Architektur (rechte Säule) für das MEDIABench (oben) und das SPEC2000 Benchmark (unten) bei jeweils gleicher Ablaufplanlänge und einem Cluster.

In 47 der 73 Fälle (64% der Fälle) benötigt die DESCOMP-Architektur weniger FUs als die Lapinskii-Architektur zur Ausführung eines Ablaufplans derselben Länge. Teilweise konnten 40% bis 50% der funktionalen Einheiten und somit auch der Ports in der Registerbank gespart werden. Im Mittel konnte die Anzahl der benötigten funktionalen Einheiten um 18% verringert

werden. Insbesondere für kleine Ablaufplanlängen war eine deutliche Einsparung der funktionalen Einheiten möglich. So beträgt die durchschnittliche Einsparung funktionaler Einheiten in den Architekturen zur Ausführung des kürzesten Ablaufplans von jedem Benchmarkprogramm 31%. Für Ablaufplanlängen, die etwa dem Doppelten der kritischen Pfadlänge entsprechen, können durch DESCOMP kaum noch Einsparungen erreicht werden. Bei diesen Ablaufplanlängen haben die Lapinskii-Architekturen oft nur noch zwei oder drei FUs, was in vielen Fällen schon das Optimum darstellt. In diesen Fällen werden durch DESCOMP allerdings immer Ablaufpläne gefunden, die kürzer sind, aber mit derselben FU-Anzahl ausgeführt werden können.

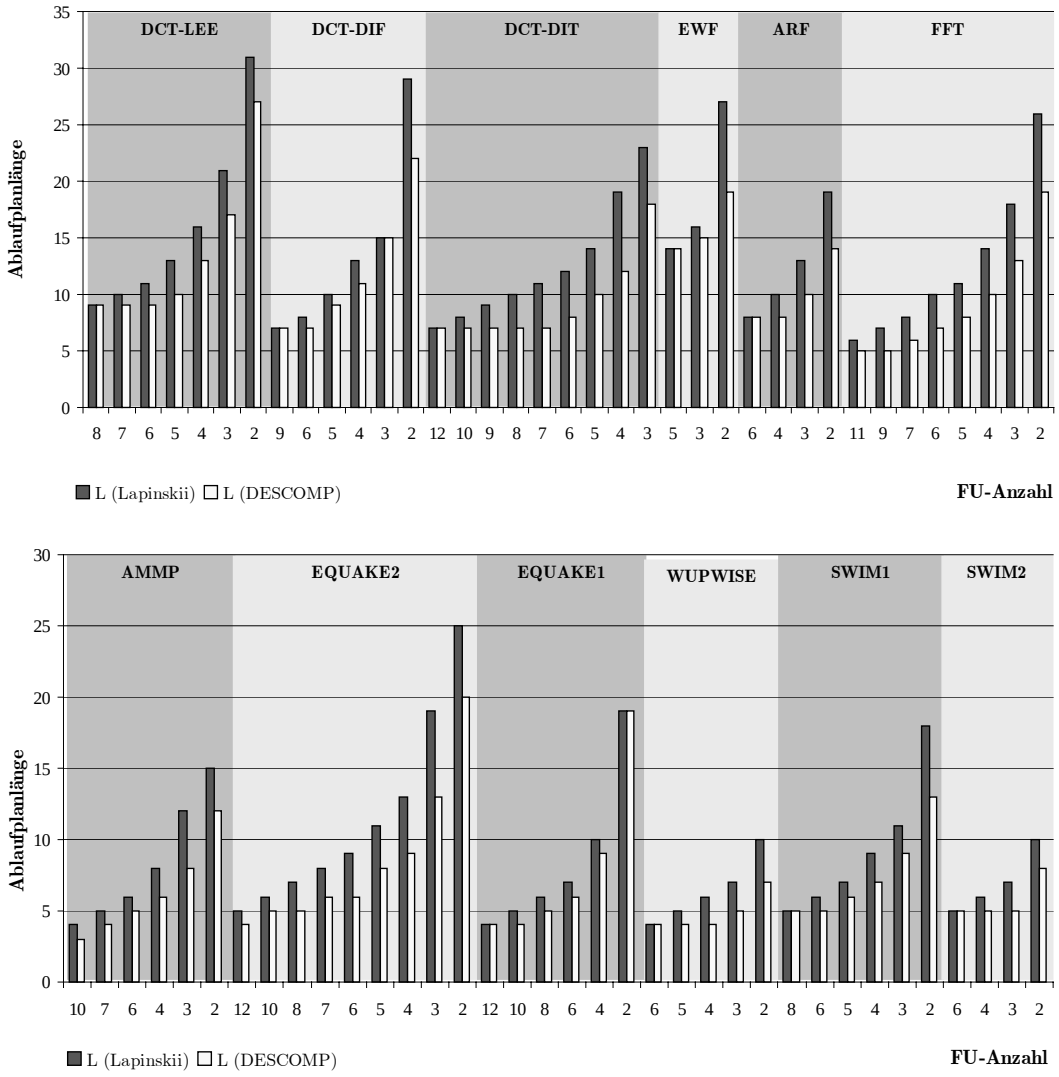



Abbildung 6.2: Gegenüberstellung der Ablaufplanlängen ($L$) von Lapinskii und DESCOMP für das MEDIABench (oben) und das SPEC2000 Benchmark (unten) in Abhängigkeit von der FU-Anzahl bei einer nicht geclusterten Architektur.

In Abbildung 6.2 ist dieser Zusammenhang dargestellt, indem für jedes Benchmarkpogramm die Längen der kürzesten Ablaufpläne gegenübergestellt sind, die zu einer nicht geclusterten Lapinskii-Architektur (linke Säule) bzw. DESCOMP-Architektur (rechte Säule), die beide dieselbe FU-Anzahl haben, durch die jeweiligen Planungsalgorithmen erzeugt wurden.

Insgesamt konnten 63 der 73 Ablaufpläne (86% aller Fälle) verkürzt werden. Im Mittel konnten 19% und im Maximum 37% der Instruktionen eingespart werden, was einer entsprechenden Erhöhung der Ausführungsgeschwindigkeit entspricht. Tendenziell nimmt die Verkürzung der Ablaufpläne mit steigender FU-Anzahl für jedes Benchmarkprogramm ab. Bei drei Benchmarkprogrammen (FFT, AMMP, EQUAKE2) konnte die kürzeste durch Lapinskii gefundene Ablaufplanlänge aber nochmals verkürzt werden, weil der ressourcenbeschränkte Ansatz in diesen Fällen keinen Ablaufplan gefunden hat, der eine kritische Pfadlänge hat.

### 6.3.2 Registerbankkosten geclusterter Architektur

Die Qualität der DSE mit DESCOMP zur lokalen Optimierung von Architekturen mit zwei bis fünf Clustern wird in diesem Abschnitt mit den Ergebnissen aus Lapinskii's Arbeit verglichen. Das Vergleichskriterium ist die Portanzahl in der größten Registerbank. Diese Portanzahl berücksichtigt sowohl die internen als auch die externen Lese- und Schreibports. Jede FU benötigt drei Ports in genau einer Registerbank und für jeden Kopieroperator werden zwei Ports in jeder Registerbank bereitgestellt. Die Lapinskii-Architekturen verfügen immer über zwei Kopieroperatoren. DESCOMP ermittelt die Anzahl der benötigten Kopieroperatoren während der DSE automatisch. In fast alle Fällen beträgt deren Anzahl zwei oder eins. Nur in wenigen Fällen werden drei Kopieroperatoren benötigt. Für einige der Lapinskii-Architekturen sind in [99] auch genauere Angaben über die Datenpfade gemacht worden. In diesen Fällen wird noch die Gesamtanzahl der Ports in den Architekturen gegenübergestellt, die ein Maß für die insgesamt verfügbare Parallelität ist. In den folgenden Abbildungen ist für jedes Benchmarkprogramm und verschiedene Ablaufplanlängen jeweils

- die maximale Portanzahl der Lapinskii-Architektur (linke Säule),
- die maximale Portanzahl der DESCOMP-Architektur (2. Säule v.l.),
- die Gesamtanzahl Ports in der Lapinskii-Architektur (3. Säule v.l.) und
- die Gesamtanzahl Ports in der DESCOMP-Architektur (4. Säule v.l.)

gegenübergestellt. Die jeweils dritte und vierte Säule von links ist im Diagramm nur dargestellt, wenn die entsprechenden Angaben in der Arbeit von Lapinskii vorhanden waren. In der Abbildung 6.3 ist diese Gegenüberstellung für eine Architektur mit zwei Clustern angegeben.

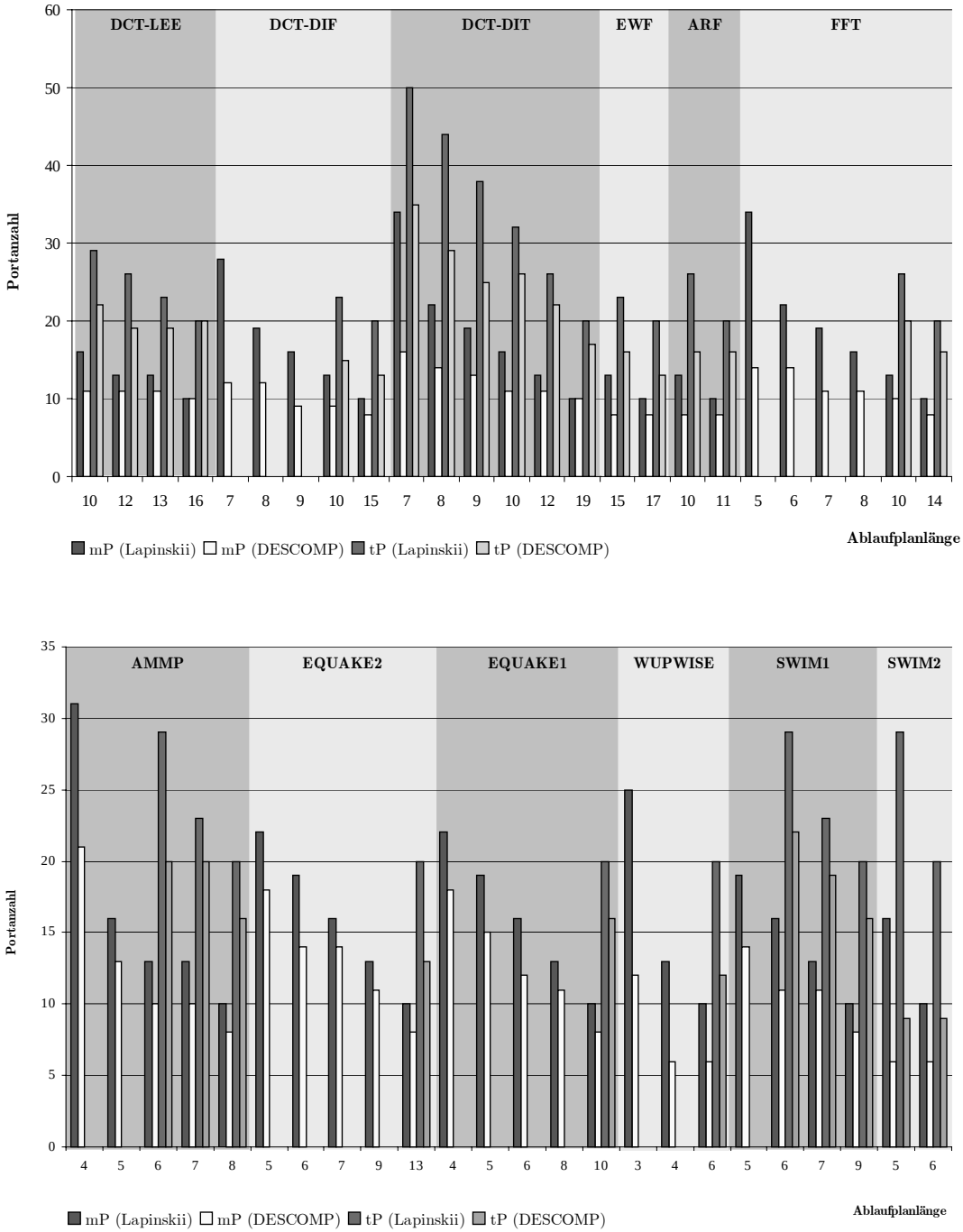



Abbildung 6.3: Vergleich der maximalen Portanzahl ($mP$) und Gesamtanzahl der Ports ($tP$) in den Registerbänken einer Architektur mit zwei Clustern für das MEDIABench (oben) und das SPEC2000 Benchmark (unten).

Im Durchschnitt konnte die Portanzahl in der größten Registerbank um 29%, im Maximum um 63% (SWIM2 mit Länge 5) verringert werden. Keine DESCOMP-Architektur benötigt zur Ausführung eines Ablaufplans eine größere Portanzahl in der größten Registerbank als die Lapinskii-Architektur

zur Ausführung eines Ablaufplans mit derselben Länge. Ebenso ist die Gesamtanzahl der Ports in der Architektur gesunken. Das ist oft auf eine FU weniger pro Cluster und in den Fällen, in denen nur ein Kopieroperator in der DESCOMP-Architektur benötigt wird, auf die dadurch eingesparten Ports zurückzuführen. Die Gesamtanzahl der Ports konnte so im Mittel um 27% reduziert werden und maximal um 69%.

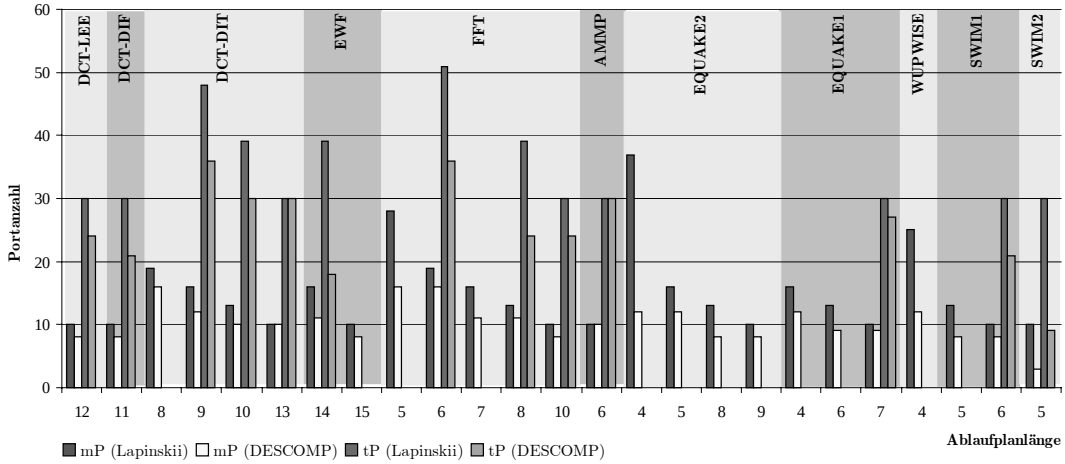



Abbildung 6.4: Vergleich der maximalen Portanzahl ($mP$) und Gesamtanzahl der Ports ($tP$) in den Registerbänken einer Architektur mit drei Clustern.

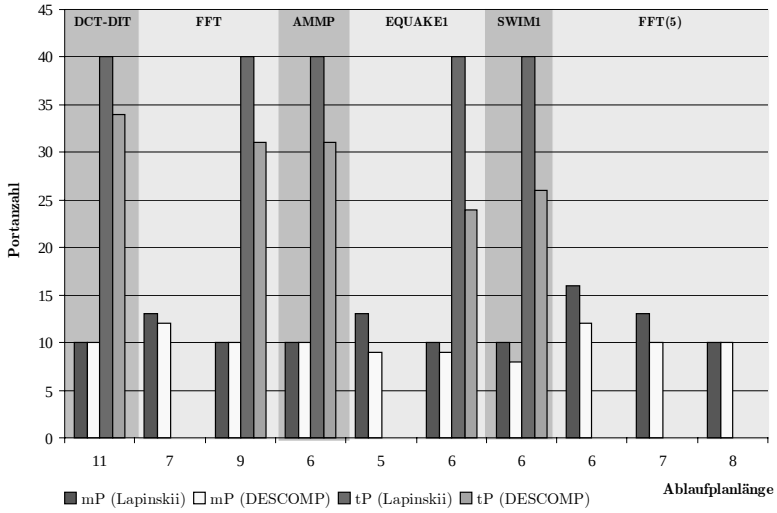



Abbildung 6.5: Vergleich der maximalen Portanzahl ($mP$) und Gesamtanzahl der Ports ($tP$) in den Registerbänken der Lapinskii- ($L$) und DESCOMP-Architekturen ($D$) mit vier und fünf Clustern (FFT (5)).

In Abbildung 6.4 sind die maximale Portanzahl und die Gesamtportanzahl der Lapinskii- und DESCOMP-Architekturen mit drei Clustern gegenüber-

gestellt und in Abbildung 6.5 ist diese Gegenüberstellung für Architekturen mit vier und fünf Clustern angegeben. Für die in Abbildung 6.4 und Abbildung 6.5 nicht angegebenen Benchmarkprogramme wurde in der Arbeit von Lapinskii keine Clusterung mit einer entsprechenden Clusteranzahl angegeben.

Bei Architekturen mit drei Clustern (Abbildung 6.4) ergab sich bei der maximalen Portanzahl im Mittel eine Einsparung von 27%, maximal eine Einsparung von 70% und im schlechtesten Fall keine Veränderung. Bei der Gesamtanzahl der Ports konnten im Mittel 27% der Ports und maximal 70% eingespart werden. Die DESCOMP-Architekturen für die Benchmarkprogramme EQUAKE1 und SWIM2 benötigen keine Kopieroperationen, so dass die Einsparung bei EQUAKE1 ausschließlich auf die fehlenden vier externen Ports in der Registerbank zurückzuführen ist.

Bei einer Architektur mit vier Clustern konnte in der größten Registerbank die Anzahl der Ports um maximal 20% und im Mittel um 10% verringert werden. Die Gesamtportanzahl verringerte sich im Mittel um 27%, maximal um 40 % und im schlechtesten Fall noch um 15%.

In den Abbildungen nicht dargestellt ist, dass in den Fällen, in denen die maximale Portanzahl nicht verringert werden konnte, kürzere Ablaufpläne gefunden wurden, die von einer Architektur mit derselben maximalen Portanzahl in den Registerbänken ausgeführt werden können. Entsprechende Betrachtungen werden im folgenden Abschnitt durchgeführt.

### 6.3.3 Vergleich der Ablaufplanlängen

Im vorangegangenen Abschnitt wurde in Abhängigkeit von der Ablaufplanlänge die Portanzahl geclusterter Lapinskii- und DESCOMP-Architekturen miteinander verglichen. In diesem Abschnitt werden die Ablaufplanlängen in Abhängigkeit von der Portanzahl in der größten Registerbank gegenübergestellt. In der Abbildung 6.6 repräsentiert die linke Säule die Länge des kürzesten Ablaufplans, der zu einer Lapinskii-Architektur mit der entsprechenden maximalen Portanzahl in einem von zwei Clustern erzeugt wurde. Die rechte Säule repräsentiert die kürzeste Ablaufplanlänge, zu der mit DESCOMP ein Prozessor mit einer entsprechenden Portanzahl in der größten Registerbank generiert wurde.

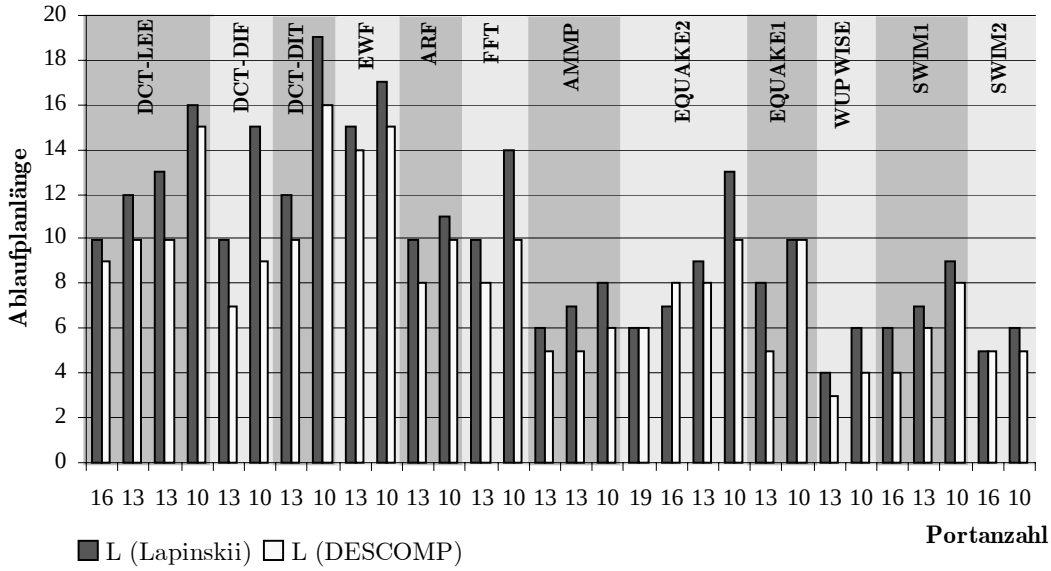



Abbildung 6.6: Gegenüberstellung der kürzesten Ablaufplanlängen $L$ in Abhängigkeit von der Portanzahl in der größten Registerbank einer Architektur mit zwei Clustern.

Die Ablaufplanlänge zur Ausführung des jeweiligen Basisblocks konnte im besten Fall um 40% und im Mittel um 20% verringert werden. In Abbildung 6.7 ist derselbe Vergleich für Architekturen mit drei und vier Clustern angegeben.

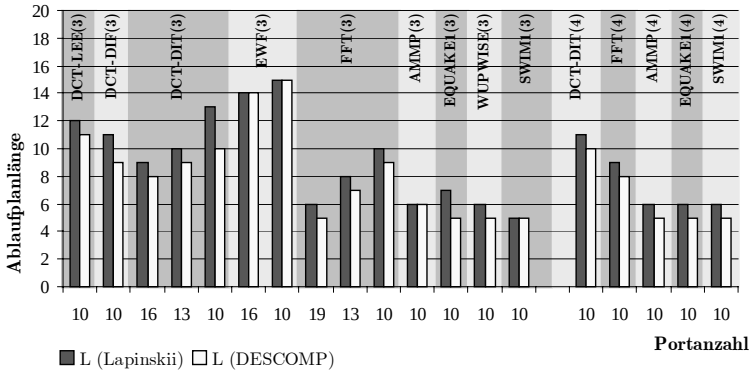



Abbildung 6.7: Gegenüberstellung der kürzesten Ablaufplanlängen $L$ in Abhängigkeit von der Portanzahl in der größten Registerbank einer Architektur mit drei bzw. vier Clustern.

Bei drei Clustern konnte eine maximale Verkürzung der Ablaufplanlängen um 29% und im Mittel um 12% erreicht werden. Bei vier Clustern wurden die Ablaufpläne um maximal 17% und im Mittel um 14% kürzer. Bis auf einen Fall (AMMP, 3 Cluster) konnte in allen Fällen, in denen im vorigen Abschnitt die Portkonfiguration der Lapinskii-Architekturen nicht verbessert werden konnte, ein kürzerer Ablaufplan zu dieser Portkonfiguration durch

DESCOMP gefunden werden. In einigen Fällen konnten beide Größen verkleinert werden. Das heißt, dass ein kürzerer Ablaufplan von einer DESCOMP-Architektur mit weniger Ports in der größten Registerbank ausgeführt werden kann. Die kürzeren Ablaufpläne gestatten eine entsprechend höhere Ausführungsgeschwindigkeit bei gleicher Taktfrequenz des Prozessors oder bei gleichbleibender Ausführungsgeschwindigkeit eine niedrigere Taktfrequenz des Prozessors, wodurch der dynamische Stromverbrauch verringert werden kann.

### 6.3.4 Laufzeit des Planungsalgorithmus

Die Laufzeit der in Abschnitt 4 vorgestellten Planung, Clusterung und Ressourcenallokation ist polynomiell. Dabei hängt die Laufzeit der Planung sowohl von der Knotenanzahl im Basisblock als auch von der Ablaufplanlänge ab, für die der Ablaufplan erzeugt wird. Die von der Java-Implementierung benötigte Zeit für die Planung und Ressourcenallokation der Operationen der ungeclusterten Benchmarkbasisblöcke ist in Abbildung 6.8 dargestellt. Die Laufzeit ist in Abhängigkeit von der Verlängerung der kritischen Pfadlänge des jeweiligen Benchmarkprogramms um die dargestellte Anzahl von Instruktionen angegeben.

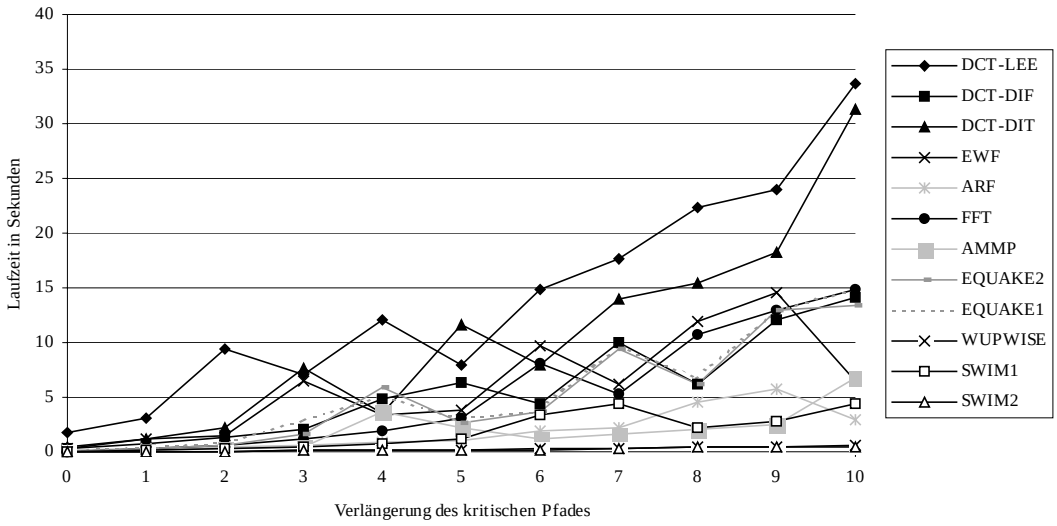



Abbildung 6.8: Laufzeit der Ablaufplanung für die ungeclusterten Benchmarkprogramme.

Die Laufzeiten wurden auf einem PC mit 1.5 GHz gemessen. Eine bereits begonnene Implementierung des Planungsalgorithmus in C zeigt, dass diese Laufzeiten um den Faktor 10 verringert werden können. Sie bewegen sich damit in einem Bereich, der für eine DSE und auch für optimierende Compiler akzeptabel ist. Die Planung der Operationen eines geclusterten

Basisblocks wirkt sich nicht signifikant auf die in Abbildung 6.8 angegebenen Laufzeiten aus. Die Laufzeit des Clusterungsalgorithmus selbst wird durch die Laufzeit der Planung dominiert, da der Clusterungsalgorithmus die Planung der Operationen mehrfach wiederholt, um eine verbesserte Clusterung zu erzeugen. Die Laufzeiten zur Erzeugung eines geclusterten Ablaufplans bei einer vorgegebenen Länge hängen deshalb wesentlich davon ab, wie oft die Ablaufplanung wiederholt wird. Mit den gewählten Einstellungen während der Clusterung beträgt die Laufzeit zur Erzeugung eines geclusterten Ablaufplans zu einer vorgegebenen Länge das ca. siebenfache der in Abbildung 6.8 angegebenen Zeiten.

### 6.3.5 Zusammenfassung

Sowohl für ungeclusterte als auch für geclusterte Architekturen konnte gezeigt werden, dass zur Abarbeitung von Ablaufplänen mit derselben Länge die DESCOMP-Architekturen weniger Ports benötigen als die Lapinskii-Architekturen. Einsparungen konnten sowohl in der größten Registerbank als auch in den übrigen Registerbänken erreicht werden. Darüber hinaus konnte für DESCOMP- und Lapinskii-Architekturen mit derselben Portanzahl in der größten Registerbank gezeigt werden, dass mit der DESCOMP-Architektur derselbe Basisblock mit weniger Instruktionen abgearbeitet werden kann. Die folgende Tabelle fasst diese Ergebnisse zusammen.

| **Clusteranzahl** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| **Eingesparte Ports in der größten Registerbank** | 18 % | 29 % | 27 % | 10 % |
| **Insgesamt eingesparte Ports im Prozessor** | 18 % | 27 % | 27 % | 27 % |
| **Eingesparte Instruktionen bei gleicher Portanzahl in der größten Registerbank** | 19 % | 20 % | 12 % | 14 % |

Tabelle 6.6: Durchschnittlich eingesparte Ports bzw. Instruktionen verglichen mit den Architekturen bzw. Ablaufplänen aus der Arbeit von Lapinskii.

Aufgrund der in der größten Registerbank eingesparten Ports kann die DESCOMP-Architektur höher getaktet werden, falls eine höhere Ausführungsgeschwindigkeit erforderlich ist. Bei gleicher Taktfrequenz verbrauchen die Registerbänke in der DESCOMP-Architektur weniger Strom als die Registerbänke in der Lapinskii-Architektur. Gleichzeitig sinkt auch der Platzverbrauch. Die Einsparung der insgesamt erforderlichen Ports resultiert aus einer geringeren FU-Anzahl in den einzelnen Clustern und/oder eines Kopieroperators weniger, der in der Architektur benötigt wird. Dies konnte durch einen guten Planungs- und Clusterungsalgorithmus und durch die Implementierung verschiedener Operatoren in derselben FU erreicht werden. Insbesondere bei sehr kurzen Ablaufplanlängen, die eine schnelle Abarbeitung

der Basisblöcke erlauben, lag die Verringerung der Portanzahl in der größten Registerbank deutlich über den in Tabelle 6.6 angegebenen Durchschnittswerten. Die Einsparungen bei den Ports erfordern aber erhöhte Hardwarekosten für die bereitgestellten Operatoren in der DESCOMP-Architektur. Die erhöhte Operatoranzahl ist Folge der geringeren Parallelität in den Fällen, in denen dieselbe Ablaufplanlänge mit weniger FUs als in der Lapinskii-Architektur erreicht wird. Damit können nicht alle in Lapinskii's Ablaufplänen parallel ausgeführten Operationen auch im DESCOMP-Ablaufplan parallel ausgeführt werden. Das führt dazu, dass Operationen in andere Instruktionen verschoben werden müssen, wodurch dort mehr Operatoren desselben Typs als in der Lapinskii-Architektur benötigt werden können. Die Anzahl der Operatoren, die in einer Lapinskii-Architektur benötigt werden, entspricht der FU-Anzahl. Die folgende Tabelle stellt die Zunahme der Operatoren in der DESCOMP-Architektur der Abnahme der Portanzahl in der größten Registerbank für das DCT-DIT-Benchmark gegenüber. Für ein bis drei Cluster ist zu einer Ablaufplanlänge ($L$) jeweils die Anzahl der Addierer/Subtrahierer ($A$) und Multiplizierer ($M$) in den Lapinskii- bzw. DESCOMP-Architekturen angegeben. In der Spalte $t$ ist die prozentuale Zunahme der Operatoren in der DESCOMP-Architektur der prozentualen Abnahme $p$ der Ports in der größten Registerbank gegenübergestellt.

| **1 Cluster** | | | | | | | **2 Cluster** | | | | | | | **3 Cluster** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lapinskii | | Descomp | | | | | Lapinskii | | Descomp | | | | | Lapinskii | | Descomp | | | |
| ***L*** | ***A*** | ***M*** | ***A*** | ***M*** | ***t*** | ***p*** | ***L*** | ***A*** | ***M*** | ***A*** | ***M*** | ***t*** | ***p*** | ***L*** | ***A*** | ***M*** | ***A*** | ***M*** | ***t*** | ***p*** |
| 7 | 7 | 5 | 7 | 6 | 8 | 42 | 7 | 7 | 7 | 9 | 7 | 14 | 53 | 8 | 8 | 5 | 7 | 4 | -20 | 16 |
| 8 | 6 | 4 | 6 | 6 | 20 | 40 | 8 | 7 | 5 | 7 | 4 | -8 | 36 | 9 | 6 | 4 | 6 | 5 | 10 | 25 |
| 9 | 5 | 4 | 6 | 3 | 0 | 33 | 9 | 7 | 3 | 7 | 4 | 10 | 32 | 10 | 6 | 3 | 6 | 5 | 22 | 23 |
| 10 | 4 | 4 | 5 | 4 | 13 | 38 | 10 | 5 | 3 | 6 | 3 | 13 | 31 | 13 | 3 | 3 | 6 | 4 | 67 | 0 |
| 11 | 4 | 3 | 5 | 3 | 14 | 29 | 12 | 4 | 2 | 6 | 3 | 50 | 15 | | | | | | | |
| 12 | 4 | 2 | 4 | 4 | 33 | 33 | 19 | 2 | 2 | 3 | 2 | 25 | 0 | | | | | | | |
| 14 | 3 | 2 | 4 | 2 | 20 | 20 | | | | | | | | | | | | | | |
| 19 | 2 | 2 | 3 | 2 | 25 | 25 | | | | | | | | | | | | | | |
| 23 | 2 | 1 | 3 | 2 | 67 | 0 | | | | | | | | | | | | | | |
| 37 | 1 | 1 | 1 | 1 | 0 | 50 | | | | | | | | | | | | | | |
| Mittelwert in Prozent: | | | | | 20 | 31 | Mittelwert in Prozent: | | | | | 17 | 28 | Mittelwert in Prozent: | | | | | 20 | 16 |

Tabelle 6.7: Gegenüberstellung der Anzahl der Operatoren in allen Clustern und Ports in der größten Registerbank. Legende: $L$ – Ablaufplanlänge, $A$ – Addierer/Subtrahierer, $M$ – Multiplizierer, $t$ – Zuwachs der Operatoren in Prozent, $p$ – Verringerung der Portanzahl in Prozent.

Es ist deutlich zu erkennen, dass insbesondere für kurze Ablaufplanlängen die prozentuale Verringerung der Portanzahl (Spalte $p$) in der DESCOMP-Architektur höher ausfällt als die prozentuale Zunahme der Operatoren (Spalte $t$) bei gleicher Ablaufplanlänge. In zwei Fällen können sogar beide Größen gleichzeitig verringert werden. Dieses $t/p$-Verhältnis kehrt sich für längere Ablaufpläne allerdings um. Für die übrigen Benchmarkprogramme waren die Angaben zu den Operatoren nicht so detailliert wie bei DCT-DIT verfügbar. Eine Unterscheidung nach Additionen und Multiplikationen kann deshalb nicht vorgenommen werden. In den Fällen, in denen die Gesamtanzahl der funktionalen Einheiten in der Lapinskii-Architektur angegeben war[1], liegt die Operatoranzahl in der DESCOMP-Architektur im Mittel um 32% über der Operatoranzahl in der Lapinskii-Architektur. Demgegenüber konnte im Mittel die Anzahl der Ports in der größten Registerbank um 21% gegenüber der Lapinskii-Architektur verringert werden. Dass dieses Verhältnis schlechter ausfällt als beim DCT-DIT Benchmark (vgl. Tabelle 6.7 letzte Zeile) liegt daran, dass für sehr kurze Ablaufplanlängen, bei denen DESCOMP eine besseres $t/p$-Verhältnis erreicht als bei größeren Ablaufplanlängen, die Angaben zu den Lapinskii-Architekturen fehlten und deshalb nicht in den Durchschnittswert aller Benchmarkprogramme eingeflossen sind.

Bezogen auf die Optimierungsziele lässt sich aus diesen Ergebnissen schließen, dass der verwendete Planungsalgorithmus eine DSE für geclusterte VLIW-Prozessoren erlaubt, bei der sehr gut die Portanzahl in den Registerbänken minimiert wird. Durch die verringerte Parallelität steigt aber die Anzahl der Operatoren in der Architektur an. Da die Ausführungsgeschwindigkeit sowie der Platz- und Stromverbrauch aber wesentlich durch die Registerbänke dominiert werden, ist diese Zunahme akzeptabel. Weil außerdem bei der DSE bereits berücksichtigt wird, dass sich die Operatoren in einer FU die Registerbankports teilen, sind die mit DESCOMP erzielten Resultate der DSE aussagekräftiger als die Ergebnisse in der Arbeit von Lapinskii, was die Leistungsfähigkeit des erzeugten Prozessors betrifft. Denn insbesondere bei kurzen Ablaufplanlängen solcher Benchmarkprogramme die viel Parallelität bereitstellen, werden durch DESCOMP Architekturen erzeugt, die deutlich weniger Ports in den Registerbänken benötigen (vgl. zum Beispiel DCT, FFT, AMMP in Abbildung 6.3). Die in der Arbeit von Lapinskii erzeugten Architekturen benötigen teilweise doppelt so viele Ports, wodurch die Registerbänke so hohe Kosten verursachen, dass diese Architekturen praktisch nicht realisierbar sind, obwohl das durch mehrere Operatoren in einer FU vermieden werden kann.

[1] Das sind die Fälle, in denen die Gesamtportanzahl ($tP$) in den Diagrammen in Abschnitt 6.3.2 und die FU-Anzahl ($mF$) in den Diagrammen in Abschnitt 6.3.1 gegenübergestellt wurden.

Dass der Planungsalgorithmus sehr gute Ergebnisse liefert, lässt sich auch durch den in der Studienarbeit [88, 89] durchgeführten Vergleich belegen. In dieser Studienarbeit wurden durch Simulated Annealing ungeclusterte Architekturen zur Ausführung von Ablaufplänen mit einer vorgegebenen Länge und demselben Optimierungsziel wie in DESCOMP gebildet. Die erzielten Resultate konnten die Ergebnisse der heuristischen Planung in 30% der Fälle verbessern. In 33% der Fälle lieferte die heuristische Planung bessere Ergebnisse. Wenn durch das Simulated Annealing kostengünstigere Architekturen erzeugt wurden, dann konnten nur ein bis zwei Operatoren in der Architektur eingespart werden. Die Anzahl der benötigten funktionalen Einheiten konnte in keinem Fall verringert werden [89]. Im Anhang B sind für die ungeclusterten Architekturen die Ablaufplanlängen unterstrichen, für die durch den heuristischen Planungsalgorithmus eine Architektur mit minimaler FU-Anzahl erzeugt wurde. In 79% aller Fälle, in denen dort beim Übergang von der Ablaufplanlänge $l$ zur Ablaufplanlänge $l + 1$ funktionale Einheiten eingespart werden konnten, ist das durch den Planungsalgorithmus erreicht worden. In den übrigen 21% aller Fälle mussten um höchstens zwei Instruktionen längere Ablaufpläne erzeugt werden, um die minimale FU-Anzahl zu erreichen.

## 6.4 Qualität der globalen Optimierung

Die Qualität der globalen Optimierung kann nicht objektiv mit anderen Arbeiten verglichen werden, weil keine Ergebnisse für exakt spezifizierte Eingaben in Form von Datenflussgraphen, wie bei der lokalen Optimierung, vorliegen. Daher wird in Abschnitt 6.4.2 eine globale Datenpfadsynthese beispielhaft für die Kombination einiger der bereits vorgestellten Basisblöcke durchgeführt. Die Zielarchitektur wird dann hinsichtlich der Anforderungen der einzelnen Basisblöcke beurteilt. Außerdem werden im Abschnitt 6.4.1 Aussagen zur Qualität der globalen Optimierung getroffen und Nachteile des vorgestellten Ansatzes diskutiert.

### 6.4.1 Aussagen zur Qualität

Unter der Annahme, dass die lokal erzeugten Zielprogramme bezüglich der internen Portanzahl optimal sind, können Aussagen über die globale Optimierung getroffen werden. Als weitere Annahme soll gelten, dass die Anzahl der externen Lese- und Schreibports durch zwei Konstanten *kewp* und *kerp* festgelegt ist und die Modellannahmen über die Abhängigkeit der Taktfrequenz von der Portanzahl der Realität entsprechen. Ein Zielprogramm $(\alpha, \chi, \phi)$ mit der Portkonfiguration $(ip, kerp, kewp)$ zum Basisblock $b$ ist dann bezüglich der internen Portanzahl optimal, wenn für die Ablaufpläne aller anderen Zielprogramme $(\alpha', \chi', \phi')$ zum Basisblock $b$ mit

$|V_b/\chi| = |V_b/\chi'|$ und $|\alpha'| = |\alpha|$ und $pConf(\alpha') = (ip', kerp, kewp)$ gilt: $ip \leq ip'$.

**Satz 6.1**
Ist jedes Zielprogramm in $\mathcal{ZP}$ optimal bezüglich seiner internen Portanzahl und es existiert ein Prozessor, der jedes Programmfragment $p \in \mathcal{P}$ in der vorgegebenen Zeit $t(p)$ ausführen kann, dann findet Algorithmus 5.1 die Portkonfiguration eines solchen Prozessors.

**Beweis**
Die Optimierung der Portkonfiguration schlägt nur dann fehl, wenn durch Algorithmus 5.1 keine minimale Portkonfiguration $(ip, kerp, kewp)$ gefunden wird. Angenommen das ist der Fall, dann bedeutet das, dass es für jedes Tripel $(ip, kerp, kewp)$ ein Programmfragment $p$ gibt, so dass $pExeT(p, (ip, kerp, kewp)) > t(p)$. Angenommen es gibt doch einen Prozessor $P$, mit dem alle Programmfragmente innerhalb der Zeitschranken ausgeführt werden können. $P$ habe die Portkonfiguration $(ip', kerp, kewp)$ und die Taktfrequenz $f$. Dann gilt für die Ausführungszeit $t'_p$ jedes Programmfragments $p$ auf dem Prozessor $P: t'_p \leq t(p)$. Während der Portoptimierung gab es ein $mp \in \mathbb{N}$, so dass die Portkonfiguration $(ip', kewp, kerp) \in MP(mp)$ betrachtet wurde und für ein Programmfragment $p$ hat gegolten: $t(p) < pExeT(p, (ip', kewp, kerp))$ bei der maximal möglichen Taktfrequenz $fm(ip' + kerp + kewp) \geq f$. Daraus folgt $t'_p < pExeT(p, (ip', kewp, kerp))$. Da $f \leq fm(ip' + kerp + kewp)$ und $exe(b)$ konstant ist, muss es einen Basisblock $b \in p$ geben, für den $P$ einen Ablaufplan $\alpha'$ ausführt, der kürzer ist als der kürzeste Ablaufplan $\alpha$ eines Zielprogramms in $\mathcal{ZP}$ mit $|\alpha| = mL(b, (ip', kewp, kerp))$. Damit waren nicht alle Zielprogramme in $\mathcal{ZP}$ optimal bezüglich ihrer internen Portanzahl, was ein Widerspruch zur Voraussetzung ist.

□

Durch die globale Optimierung mit DESCOMP kann somit ermittelt werden, ob überhaupt eine Architektur existiert, die alle Programmfragmente innerhalb der geforderten Zeitschranken ausführen kann. Die durch DESCOMP lokal optimierten Ablaufpläne haben zwar eine gute Qualität, wie durch die Vergleiche mit der Arbeit von Lapinskii gezeigt wurde, allerdings sind sie nicht immer optimal bezüglich der Portanzahl. Aufgrund der zeitbeschränkten Ablaufplanung wird aber für jede mögliche Ablaufplanlänge eines Basisblocks ein Ablaufplan erzeugt. Dass DESCOMP keine Architektur findet, obwohl eine existiert, kann also nur dann vorkommen, wenn der mit DESCOMP erzeugte Ablaufplan mehr Ports als der bezüglich der internen Portanzahl optimale Ablaufplan dieser Länge benötigt. Dadurch ist die maximal mögliche Taktfrequenz des mit DESCOMP erzeugten Prozessors niedriger als die Taktfrequenz des Prozessors, der den optimalen Ablaufplan ausführen kann.

**Satz 6.2**
Ist jedes Zielprogramm in $\mathcal{ZP}$ optimal bezüglich seiner internen Portkonfiguration und es wird durch Algorithmus 5.1 die Portkonfiguration eines Prozessors $P$ erzeugt, dann gibt es keinen Prozessor $P'$, der alle Programmfragmente innerhalb der Zeitschranken ausführt und in dem jede Registerbank weniger Ports als die größte Registerbank in $P$ hat.

**Beweis**
Der von DESCOMP erzeugte Prozessor $P$ habe die Portkonfiguration ($ip$, $kerp$, $kewp$). Angenommen es gäbe einen Prozessor $P'$, in dem die größte Registerbank die Portkonfiguration ($ip'$, $kerp$, $kewp$) mit $ip' < ip$ hat. Dann ist $ip' + kerp + kewp < ip + kerp + kewp$ und damit wurde während der Portoptimierung von Algorithmus 5.1 die Portkonfiguration ($ip'$, $kerp$, $kewp$) vor der Portkonfiguration ($ip$, $kerp$, $kewp$) betrachtet. Es muss aber ein Programmfragment $p$ gegeben haben, dessen Ausführungszeit $t(p)$ trotz der maximal möglichen Taktfrequenz $fm(ip' + kerp + kewp)$ nicht eingehalten werden konnte. Von $P'$ wird $p$ aber innerhalb der Zeit $t(p)$ mit einer Taktfrequenz $f \leq fm(ip' + kerp + kewp)$ ausgeführt. Damit muss es ein $b \in p$ geben, so dass der zu $b$ gehörende Ablaufplan $\alpha'$, der von $P'$ ausgeführt wird, kürzer ist als der Ablaufplan $\alpha$ eines Zielprogramms aus $\mathcal{ZP}$ mit $|\alpha| = mL(b, (ip', kerp, kewp))$. Damit ist $\alpha$ aber wieder nicht optimal bezüglich der internen Portanzahl, was ein Widerspruch ist.

□

Daraus folgt, dass für einen ungeclusterten Prozessor der kostenminimale Prozessor bezüglich *VLIWCost* gefunden wird, wenn der Zuwachs an Hardwarekosten, der durch eine zusätzliche FU – also drei weitere Ports – entsteht, höher ist, als die Kosten, die durch die Einsparung von Operatoren entfallen. Eine zusätzliche FU für einen Prozessor mit $n$ FUs kann genutzt werden, um Operationen desselben Typs, die von dem Prozessor mit $n$ FUs parallel abgearbeitet werden, in andere Instruktionen umzuverteilen, in denen sie alle denselben Operator nutzen. Dadurch können für jeden Operationstypen bis zu $n - 1$ Operatoren eingespart werden. Die Registerbankkosten für drei zusätzliche Ports müssen also diese Operatorkosten, die für jeden Operationstypen entstehen können, übersteigen. Allerdings sind Ablaufpläne, die solch hohe Einsparungen bei den Operatoren zulassen, in der Praxis sehr unwahrscheinlich. Durch die Ergebnisse für die Benchmarkprogramme ist belegt, dass durch eine größere FU-Anzahl in den Lapinskii-Architekturen bei gleicher Ablaufplanlänge ein bis zwei Operatoren eingespart werden können (vgl. Tabelle 6.7).

Ein Nachteil des DESCOMP-Ansatzes entsteht bei geclusterten Architekturen durch die getrennte Berechnung der Clusterung verschiedener Basisblöcke. Ein entsprechendes Beispiel wurde bereits in Abbildung 5.2 auf Seite 125 angegeben. Der Clusterungsalgorithmus hat die Operationen der Basisblöcke $b$ und $b'$ so zerlegt, dass jeder Basisblockcluster nur Operationen

eines Typs ausführt. Werden die Basisblockcluster $d$ und $g'$ sowie $g$ und $d'$ vom selben Prozessorcluster ausgeführt, dann muss jeder Prozessorcluster sechs interne Ports bereitstellen. Dafür sind nur zwei Addierer und zwei Multiplizierer im Prozessor erforderlich. Werden dagegen die Basisblockcluster $d$ und $d'$ sowie $g$ und $g'$ vom selben Prozessorcluster ausgeführt, dann können in einem Prozessorcluster drei interne Ports eingespart werden. Dafür müssen insgesamt drei Addierer und drei Multiplizierer im Prozessor vorhanden sein. Mit der in Abbildung 6.9 angegebenen Clusterung können die Cluster $d$ und $d'$ sowie $g$ und $g'$ von jeweils einem Prozessorcluster ausgeführt werden und der Prozessor muss nur zwei Addierer, zwei Multiplizierer, eine Registerbank mit sechs und eine Registerbank mit drei Ports bereitstellen. Eine stärkere Kopplung der Clusterung verschiedener Basisblöcke kann somit zur Einsparung von Operatoren im Zielprozessor führen.

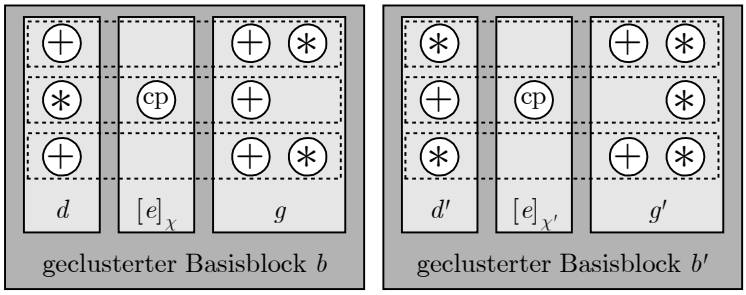



Abbildung 6.9: Clusterung für das Beispiel in Abbildung 5.2, die einen kostengünstigeren Prozessor zulässt.

Gibt es zwischen Operationsknoten, die zu verschiedenen Basisblöcken desselben Programmfragments gehören, Datenabhängigkeiten, dann werden Kopieroperationen zwischen diesen Knoten erforderlich, wenn sie verschiedenen Clustern zugeordnet werden. Diese Kopieroperationen können zwar problemlos in den Ablaufplan für die vollständige Anwendung eingefügt werden, werden aber in den Formeln (4.1) und (4.9) zur Ermittlung der Ausführungszeit eines Programmfragments nicht berücksichtigt. Da aber die Ausführungszeitpunkte und Clusterzuordnungen aller Operationen festgelegt sind, ist es möglich, die zusätzlich benötigten Takte zum Ausführen der Kopieroperationen zu berücksichtigen. Eine Kopplung der Clusterung verschiedener Basisblöcke kann somit auch genutzt werden, um Kopieroperationen zwischen Knoten verschiedener Basisblöcke zu minimieren. Entsprechende Optimierungen können Gegenstand weiterer Arbeiten sein. Für nicht geclusterte Architekturen entstehen diese genannten Problem jedoch nicht, da es nur einen Cluster gibt, in dem alle Operatoren implementiert werden müssen.

### 6.4.2 Beispielhafte Design-Space-Exploration

Eine DSE mit DESCOMP für eine nicht geclusterte Architektur ist in [59] beschrieben. Dort wurde eine Architektur erzeugt, die geeignet ist, drei Basisblöcke auszuführen, die insgesamt acht verschiedene Operationstypen enthalten. Als Basisblöcke wurden ARF, SWIM1 und MD5 verwendet. MD5 ist ein Verschlüsselungsalgorithmus, der der Arbeit [73] entnommen wurde und dessen Basisblock in Anhang B angegeben ist. Die in [59] durchgeführte DSE lief innerhalb weniger Minuten automatisch ab, wobei die Laufzeit durch die lokale Optimierung der Basisblöcke bestimmt wurde. Die Portoptimierung zur Minimierung der Registerbankkosten ermittelte für eine Architektur mit einem Cluster bei den gegebenen Zeitschranken die Portkonfiguration (12, 0, 0), zu der die in Tabelle 6.8 dargestellte maximale Clusterkonfiguration berechnet wurde. Daraus wurde durch die Typoptimierung in weniger als einer Sekunde die ebenfalls in Tabelle 6.8 dargestellte minimierte Clusterkonfiguration berechnet. Trotz der acht verschiedenen Operationstypen war damit die Typoptimierung in sehr kurzer Zeit abgeschlossen.

| **Typ** | + | − | ∗ | **and** | **or** | **not** | **xor** | **shl** |
|---|---|---|---|---|---|---|---|---|
| **maximale Clusterkonfiguration** | 4 | 3 | 4 | 2 | 1 | 1 | 2 | 1 |
| **minimale Clusterkonfiguration** | 2 | 2 | 3 | 1 | 1 | 1 | 1 | 1 |

Tabelle 6.8: Ergebnisse der Typoptimierung der DSE mit DESCOMP in [59].

Eine weitere DSE für eine geclusterte VLIW-Architektur, die die Benchmarkprogramme DCT-DIT, FFT und ARF ausführen kann, wird im Folgenden beschrieben. Jedes Benchmarkprogramm bildet ein eigenes Programmfragment. Über die Ausführungshäufigkeit und Zeitschranken wurden folgende Annahmen getroffen:

| **Name** | **Ausführungshäufigkeit** | **Zeitschranke in Millisekunden** |
|---|---|---|
| FFT | 80000 | 10 |
| DCT-DIT | 100000 | 15 |
| ARF | 100000 | 20 |

Tabelle 6.9: Ausführungshäufigkeiten und Zeitschranken der Programmfragmente für die globale Datenpfadsynthese.

Die Architektur wurde bezüglich der Hardwarekosten optimiert. Um die Ergebnisse besser bewerten zu können, wurde die DSE viermal durchgeführt und bei jeder DSE die zulässige Clusteranzahl um eins erhöht, so dass Architekturen mit bis zu vier Clustern betrachtet wurden. Die DSE ergab,

dass eine Architektur mit einem Cluster nicht in der Lage ist, die Zeitschranken einzuhalten. Damit eine Architektur mit zwei Clustern die Zeitschranken einhält, sind drei FUs je Cluster und zwei Kopieroperatoren erforderlich. Beide Registerbänke haben damit jeweils 13 Ports. Eine Architektur, die mit maximal zwei FUs je Cluster die Zeitschranken einhalten kann, benötigt mindestens drei Cluster. Für diesen Fall wurde durch die Portoptimierung zur Minimierung der Registerbankkosten die Portkonfiguration (6, 1, 2) berechnet. Die Längen der kürzesten mit dieser Portkonfiguration ausführbaren Ablaufpläne sind in der Tabelle 6.10 angegeben. Außerdem ist für jedes Programmfragment die erforderliche reale Taktfrequenz *pfr* (vgl. Seite 120) dargestellt, mit der es abgearbeitet werden muss, damit die Zeitschranke eingehalten wird.

| **Name** | ARF | DCT-DIT | FFT |
|---|---|---|---|
| **Ablaufplanlänge** | 11 | 11 | 9 |
| ***pfr*** | 55 MHz | 74 MHz | 72 MHz |

Tabelle 6.10: Kürzeste Ablaufplanlängen nach der Portoptimierung mit jeweils erforderlichen realen Taktfrequenzen.

Mit der Kandidatenmenge $\mathcal{K}_{(6,\,1,\,2)}$ wurde eine Optimierung der Clusterkonfiguration durchgeführt. Um mehr Möglichkeiten zur Minimierung der Hardwarekosten zu haben, wurde die während der Typoptimierung verwendete reale Taktfrequenz *fr* nicht, wie in Formel (4.4) auf Seite 120 angegeben, auf 74 MHz, sondern auf den maximal möglichen Wert $fm(9) = 82$ MHz gesetzt. Aufgrund dieser höheren Taktfrequenz kann die Typoptimierung längere Ablaufpläne nutzen, weil sich der Suchraum für die Typoptimierung vergrößert und sich somit mehr Möglichkeiten zur Minimierung der Clusterkonfiguration ergeben. Aus den Ablaufplänen, zu den in Tabelle 6.10 angegebenen Ablaufplanlängen, wurde die maximale Clusterkonfiguration

$$\{[2,\,2,\,2],\,[2,\,2,\,2],\,[2,\,2,\,2]\}$$

gebildet, mit der die Typoptimierung startete. Die Konfiguration eines Clusters wird dabei geschrieben als $[s,\,p,\,m]$, wobei $s$ die Anzahl der Subtrahierer, $p$ die Anzahl der Addierer und $m$ die Anzahl der Multiplizierer im Cluster $c$ ist. Von der Typoptimierung wurde die minimierte Clusterkonfiguration

$$\{[2,\,2,\,1],\,[1,\,1,\,1],\,[2,\,2,\,2]\}$$

in weniger als einer Sekunde berechnet. Die Längen der kürzesten Ablaufpläne, die mit dieser Clusterkonfiguration ausführbar sind, sind in Tabelle 6.11 zusammen mit der jeweils erforderlichen minimalen Taktfrequenz angegeben.

| **Name** | ARF | DCT-DIT | FFT |
|---|---|---|---|
| **Ablaufplanlänge** | 11 | 12 | 9 |
| **Taktfrequenz *fr*** | 55 MHz | 80 MHz | 72 MHz |

Tabelle 6.11: Kürzeste Ablaufplanlängen nach der Typoptimierung mit jeweils erforderlichen Taktfrequenzen.

Für Clusterkonfigurationen, die weniger Operatoren in einem Cluster besitzen, können die Zeitschranken nicht mehr eingehalten werden. Es mussten deshalb nur wenige Clusterkonfigurationen untersucht werden und die eigentlich exponentielle Laufzeit der Typoptimierung macht sich nicht bemerkbar. Durch die abschließende Ressourcenallokation ergab sich die in der Tabelle 6.12 angegebene Architektur. Die kürzesten Ablaufpläne, die mit dieser Architektur ausgeführt werden können, sind in Anhang C aufgeführt. Für ARF wurde ein Ablaufplan mit zwei Clustern und der Länge 11 ausgewählt. Zwar unterschreitet dieser Ablaufplan deutlich seine Zeitschranke, allerdings sind die benötigten Operatoren ohnehin in der Architektur zur Ausführung der anderen zwei Basisblöcke notwendig, so dass durch diesen Ablaufplan keine zusätzlichen Hardwarekosten entstehen. Für FFT und DCT-DIT wurden Ablaufpläne mit drei Clustern ausgewählt.

| **externer Cluster** | **Cluster 0** | | **Cluster 1** | **Cluster 2** | |
|---|---|---|---|---|---|
| | **FU 0.0** | **FU 0.1** | **FU 1.0** | **FU 2.0** | **FU 2.1** |
| [*copy*, *copy*] | [–, +, ∗] | [–, +] | [–, +, ∗] | [–, +, ∗] | [–, +, ∗] |

Tabelle 6.12: Konfiguration des VLIW-Prozessors zur Ausführung der Benchmarkprogramme FFT, DCT-DIT und ARF.

Verglichen mit den lokal für die einzelnen Basisblöcke optimierten Architekturen muss diese Architektur einen Subtrahierer mehr bereitstellen als die für DCT-DIT optimierte Architektur. Allerdings erfordert die für FFT optimierte Architektur in jeder FU aller Cluster eine Subtraktion, so dass die in Tabelle 6.12 angegebene Architektur die Mindestanforderungen der einzelnen Ablaufpläne erfüllt und damit für die gegebenen Zielprogramme optimal ist.

Von einer Architektur mit bis zu vier Clustern können die Zeitschranken ebenfalls eingehalten werden. Diese Architektur besitzt die Portkonfiguration (6, 2, 2) und wird nur gefunden, wenn die Portoptimierung die Suche nicht bei der Portkonfiguration (6, 1, 2) beendet, sondern nach einer Architektur sucht, die weniger Strom verbraucht, weil sie niedriger getaktet sein kann. Die Längen der kürzesten Ablaufpläne, die mit der Portkonfiguration (6, 2, 2) ausgeführt werden können, sind in der Tabelle 6.13 angegeben.

| **Name** | ARF | DCT-DIT | FFT |
|---|---|---|---|
| **Ablaufplanlänge** | 11 | 9 | 8 |
| **Taktfrequenz *fr*** | 55 MHz | 60 MHz | 64 MHz |

Tabelle 6.13: Kürzeste Ablaufplanlängen nach der Portoptimierung mit jeweils erforderlichen Taktfrequenzen bei Nutzung von vier Clustern.

Diese Architektur kann mit bis zu 16 MHz weniger getaktet werden als die Architektur mit drei Clustern in Tabelle 6.10, was einer Reduzierung der Taktfrequenz um 20% entspricht. Ob die dadurch erzielte Stromeinsparung den Anstieg des Stromverbrauchs in der für den vierten Cluster zusätzlich benötigten Registerbank übersteigt, hängt von den konkreten Hardwareparametern ab. Entsprechende Untersuchungen werden hier nicht weiter fortgeführt und können Gegenstand weiterer Arbeiten sein.

### 6.4.3 Erweiterungsmöglichkeiten

Die im vorangegangenen Abschnitt dargestellten Zwischenergebnisse der DSE hätten nicht betrachtet werden müssen, da DESCOMP die Parameter eines geeigneten Prozessor vollständig automatisch berechnen kann. Allerdings liefern diese Zwischenergebnisse Ansatzpunkte für Modifikationen an der Strategie zur globalen Optimierung. Das betrifft insbesondere die erschöpfende Suche während der Typoptimierung, deren Suchraum durch die Wahl der realen Taktfrequenz *fr* und der Clusterkonfiguration, mit der Algorithmus 5.2 (*Typoptimierung*) gestartet wird, verkleinert oder auch vergrößert werden kann. Wird die reale Taktfrequenz wie in Formel (4.4) gewählt, so ist der Suchraum eingeschränkt, da es mindestens ein Programmfragment gibt, das mit den kürzesten Ablaufplänen, die durch die Portoptimierung ausgewählt wurden, ausgeführt werden muss. Damit bricht die Suche in Algorithmus 5.2 bei jeder Clusterkonfiguration ab, mit der einer dieser Ablaufpläne nicht mehr ausgeführt werden kann. Der Suchraum kann noch stärker eingeschränkt werden, wenn Algorithmus 5.2 nicht mit der maximalen Clusterkonfiguration sondern einer kleineren Clusterkonfiguration gestartet wird. Diese kann beispielsweise aus der Zuordnung der Basisblockcluster zu Prozessorclustern gebildet werden, die zur Berechnung des maximalen FU-Vektors gebildet wurde. Jeder Prozessorcluster enthält dann nur die Operatoren, die er benötigt, um die Basisblockcluster auszuführen, die ihm durch diese Clusterabbildung zugeordnet wurden. Entsprechend kann der Suchraum für die Typoptimierung vergrößert werden, wenn Algorithmus 5.2 mit der maximalen Clusterkonfiguration gestartet wird, wie es in Abschnitt 5.3 angegeben ist, und die reale Taktfrequenz auf den maximal möglichen Wert für die ermittelte Portkonfiguration gesetzt wird, wie es für das Beispiel mit drei Clustern im letzten Abschnitt gemacht wurde. Eine weitere Möglichkeit den Suchraum zu vergrößern besteht in der

Betrachtung aller Portkonfigurationen durch Algorithmus 5.1, mit denen die Ausführung der Programmfragmente innerhalb ihrer Zeitschranken möglich ist. Dadurch werden Architekturen in den Suchraum aufgenommen, deren größte Registerbank mehr Ports besitzt als die Architektur mit der minimalen Portanzahl in der größten Registerbank. Diese Architekturen besitzen aber möglicherweise weniger Registerbänke, weil aufgrund der höheren Parallelität weniger Cluster erforderlich sind. Die dadurch eingesparten Kosten können insgesamt zu einer kostengünstigeren Architektur führen. Mit diesen dargestellten Alternativen steht ein ganzes Spektrum an Möglichkeiten zur Verfügung, die Laufzeit der Typoptimierung gegen die Größe des Suchraums abzuwägen, um die angegebene Suchstrategie auch auf Architekturen mit einer größeren Clusteranzahl, als der in den oben angegebenen Beispielen anzuwenden.

# 7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein neues methodisches Vorgehen für die Design-Space-Exploration von geclusterten VLIW-Prozessoren vorgestellt. Aus den zeitkritischen Basisblöcken einer Anwendung lassen sich damit bei bekannten Zeitschranken automatisch die Parameter eines solchen Prozessors berechnen, der für die Ausführung dieser Basisblöcke optimiert ist. Dabei wird der typische zeitintensive DSE-Zyklus, der auf dem wiederholten Übersetzen der Anwendung für verschiedene Prozessorvarianten beruht, vermieden. Stattdessen wird der Prozessor konstruktiv aus den Basisblöcken der Anwendung erzeugt, wodurch es möglich wird, die für die Anwendung erforderliche Parallelität im Prozessor sowie die FU-Typen in jedem Cluster zu bestimmen. Insbesondere deren Bestimmung wurde in anderen DSE-Ansätzen vermieden oder verursachte hohe Laufzeiten. Das wesentliche Optimierungsziel von DESCOMP bei der Erzeugung der Architektur ist die Minimierung der Portanzahl in den Registerbänken. Von der Portanzahl hängt in jedem Cluster die Komplexität der Registerbank und des Bypasses ab. Deren Komplexität führt bei VLIW-Prozessoren zu einem hohen Platz- sowie Stromverbrauch und beschränkt die Taktfrequenz. Die Abhängigkeit der maximalen Taktfrequenz von der Komplexität der Registerbänke und des Bypasses wird während der DSE von DESCOMP bei der Berechnung der Ausführungszeiten der Basisblöcke berücksichtigt.

Die Berechnung der Parameter des Prozessors wurde in eine lokale und eine globale Phase unterteilt. Dadurch wird es möglich, in der lokalen Optimierungsphase für jeden Basisblock mehrere Zielprogramme unterschiedlicher Länge zu erzeugen, deren Ressourcenbedarf bei der jeweiligen Ablaufplanlänge optimiert ist. Entsprechende Planungs- und Ressourcenallokationstechniken sind bereits für die High-Level-Synthese entwickelt worden und konnten an die hier gestellten Optimierungsziele für geclusterte VLIW-Prozessoren angepasst werden. Die Ablaufplanung und Ressourcenallokation sind mit einem neu entwickelten zeitbeschränkten Clusterungsalgorithmus gekoppelt worden. Durch Vergleiche mit den Ergebnissen aus der Arbeit von Lapinskii wurde nachgewiesen, dass mit DESCOMP die Qualität der DSE erhöht werden konnte, weil mehr Parametern des Prozessors in die DSE einbezogen wurden und gleichzeitig die Komplexität der größten Registerbank in den erzeugten VLIW-Prozessoren um bis zu 29% reduziert werden konnte. Aufgrund der kleineren Registerbänke weisen die DESCOMP-Architekturen einen geringeren Hardware- und Stromverbrauch auf und können höher getaktet werden. Das erlaubt es, die zeitkritischen Basisblöcke

schneller abzuarbeiten, wenn dies für die Einhaltung der Zeitschranken erforderlich ist.

Die lokal optimierten Zielprogramme werden in der globalen Optimierungsphase genutzt, um einen Prozessor zu erzeugen, der alle zeitkritischen Basisblöcke ausführen kann und die vorgegebenen Zeitschranken einhält. Die angegebenen Zeitschranken können sich auf Programmfragmente beziehen, wodurch es möglich ist, eine Zeitschranke für mehrere Basisblöcke anzugeben. Die Ausführungszeit, die für jeden Basisblock erforderlich ist, wird während der globalen Optimierung zusammen mit der Taktfrequenz des Prozessors auf der Grundlage der zuvor lokal optimierten Ablaufpläne ermittelt. Die Strategie zur Optimierung der Portkonfiguration der Registerbänke wurde in zwei Varianten vorgestellt, die es erlauben, einmal die Hardwarekosten der größten Registerbank zu minimieren und einmal den Stromverbrauch des Prozessors unter Berücksichtigung der Taktfrequenz und Versorgungsspannung. Die Bestimmung einer minimalen Portanzahl in der größten Registerbank ist, bezogen auf die vorhandenen lokal optimierten Zielprogramme, in polynomieller Zeit exakt möglich. Die Anzahl der Operatoren in den Clustern des Prozessors wird nach der Portoptimierung durch ein Suchverfahren mit exponentieller Laufzeit minimiert. Da die vorangegangene Portoptimierung den Suchraum so eingeschränkt hat, dass nur Prozessoren mit bestimmten Portkonfigurationen und einer bestimmten Taktfrequenz betrachtet werden, macht sich die exponentielle Laufzeit bei den betrachteten Benchmarkprogrammen und bis zu vier Clustern kaum bemerkbar.

Möglichkeiten für weiterführende Arbeiten ergeben sich aus den in dieser Arbeit nicht behandelten Bereichen. So kann nach einer DSE noch der Aufbau des Caches und Arbeitsspeichers optimiert werden, damit deren Verhalten gut auf die festgelegte Zugriffsreihenfolge in den Zielprogrammen abgestimmt ist. Die in einer Studienarbeit bereits begonnenen Arbeit zur Nutzung des vorgestellten Planungsalgorithmus bei gegebenen Ressourcenbeschränkungen sollte fortgesetzt werden, so dass mit DESCOMP unter Beachtung von Ressourcenbeschränkungen Ablaufpläne für geclusterte Architekturen und Operationen mit einer Latenzzeit größer als eins erzeugt werden können.

Die in Kapitel 5 vorgestellten Strategien zur globalen Optimierung bieten zahlreiche Möglichkeiten, um die Auswahl der lokal optimierten Basisblöcke an andere Kriterien anzupassen. Solche Kriterien können beispielsweise das Heruntertakten oder Abschalten einzelner Cluster und funktionaler Einheiten zur Verringerung des Stromverbrauchs sein. Das kann insbesondere dann Vorteile bringen, wenn die auszuführenden Basisblöcke sehr unterschiedliche Anforderungen an die Cluster- und FU-Anzahl im Prozessor stellen und einige Basisblöcke deshalb einzelne FUs oder Cluster nicht benötigen. Eine modifizierte Suchstrategie zur Typoptimierung kann, zusätzlich zu den in 6.4.3 bereits beschriebenen Möglichkeiten zur Eingrenzung des Suchraums, genutzt werden, um die exponentielle Laufzeit während der Typoptimierung

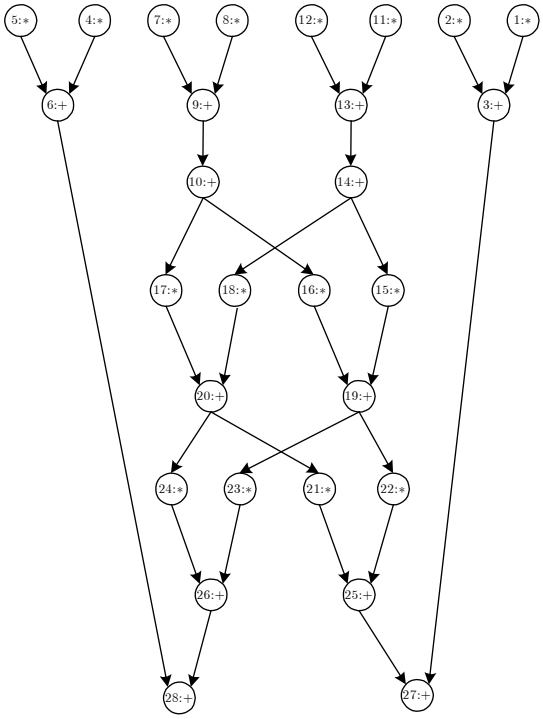
niedrig zu halten, damit die Typoptimierung auch für Architekturen mit mehr als vier Clustern und vielen Operationstypen durchgeführt werden kann. Eine solche modifizierte Suchstrategie kann dafür beispielsweise die vorhandenen Informationen zu den lokal optimierten Zielprogrammen nutzen und die Optimierung auf die Programmfragmente konzentrieren, für die sich während der globalen Optimierung herausstellt, dass sie die höchsten Anforderungen an den Prozessor stellen.

Einige Arbeiten, wie beispielsweise [23], beschäftigen sich mit der automatischen Erzeugung von Operatoren, um komplexe Operationsfolgen zu einer Operation zusammenzufassen und so die Ausführungszeit einzelner Basisblöcke zu verringern. In den Fällen, in denen der Algorithmus 5.1 zur Optimierung der Portkonfiguration mit einem Fehler abbricht, können die Informationen über die Programmfragmente, die dazu geführt haben, dass keine geeignete Portkonfiguration gefunden wurde, genutzt werden, um automatisiert Ansatzpunkte zur Bildung solcher komplexer Operationen zu finden oder um alternative algorithmische Varianten für einzelne zeitkritische Teile der Anwendung auszuprobieren. Darüber hinaus bietet sich auch eine interessante Nutzung der vorgestellten DSE-Methode im Bereich des Hardware/Software-Codesigns an. Wenn für mehrere Algorithmen jeweils verschiedene Implementierungsvarianten existieren und diese Algorithmen auf demselben Prozessor ausgeführt werden sollen, so kann mit DESCOMP eine geeignete Kombination der Implementierungsvarianten ermittelt werden, die sich gemeinsam auf einem kostengünstigen Prozessor ausführen lassen. Das ist möglich, indem jede Implementierungsvariante ein Programmfragment bildet und die Überprüfung der Zeitschranken während der globalen Optimierungsphase so angepasst wird, dass es zu jedem Algorithmus ein Programmfragment geben muss, das die Zeitschranken einhält. Eine vollständige DSE für jede Kombination von Implementierungsvarianten durchzuführen, ist somit nicht notwendig.

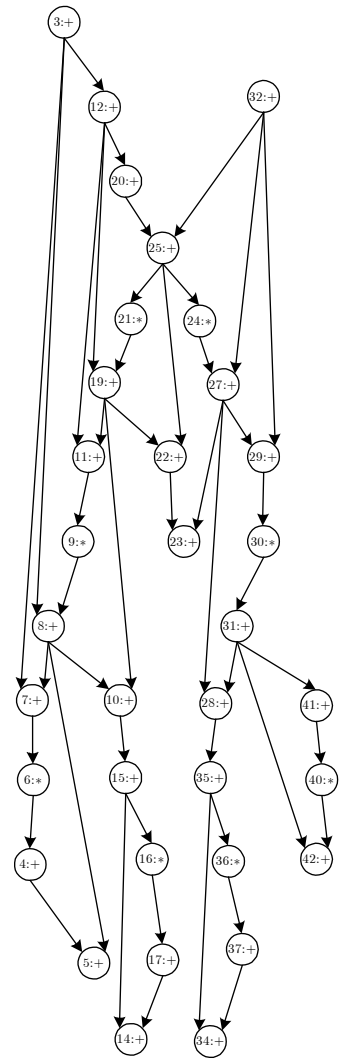
Für eine praktische Nutzung des DESCOMP-Werkzeugs ist die Anbindung eines Frontends und eines Backends erforderlich. Das Frontend übersetzt eine Hochsprache in die graphenbasierte Eingabe von DESCOMP und führt zielcodeunabhängige Optimierungen durch. Das Backend erzeugt aus den mit DESCOMP berechneten Prozessorparametern und einer Bibliothek mit parametrisierten Hardwarekomponenten Code zum synthetisieren des Prozessors. Damit ließe sich der erzeugte VLIW-Prozessor für weitere Analysen simulieren. Es ist dann auch eine Integration von DESCOMP in ein Werkzeug für das Hardware/Software-Codesign möglich, um beispielsweise während des Entwurfsprozesses eines eingebetteten Systems die Anforderungen an einen VLIW-Prozessor, der verschiedene Aufgaben in dem System übernimmt, schnell und automatisiert zu berechnen.

# Anhang A

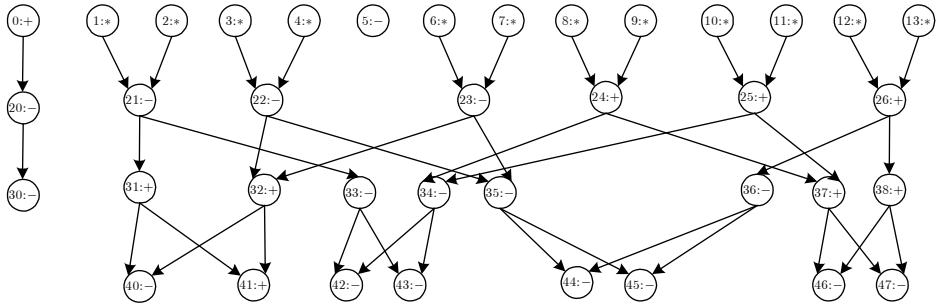
## Basisblöcke der Benchmarkprogramme



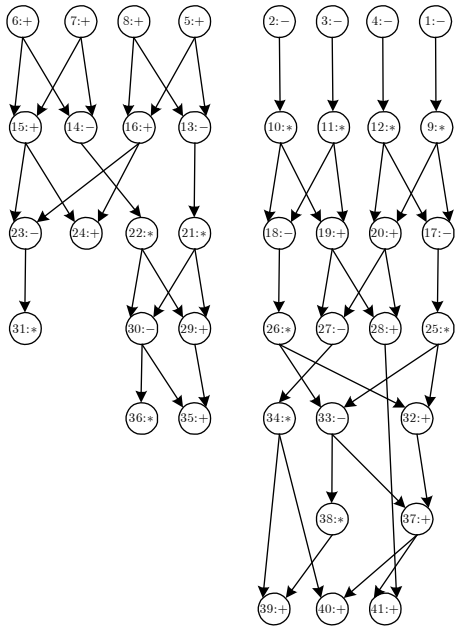

**ARF**


3:+
12:+
32:+
20:+
25:+
21:∗
24:∗
19:+
27:+
11:+
22:+
29:+
9:∗
23:+
30:∗
8:+
31:+
7:+
10:+
28:+
41:+
6:∗
15:+
35:+
40:∗
4:+
16:∗
36:∗
42:+
5:+
17:+
37:+
14:+
34:+

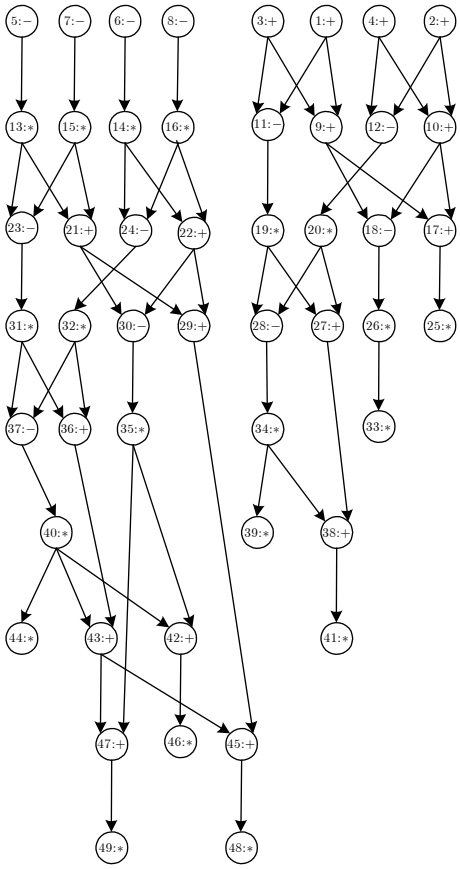
**EWF**


0:+
1:∗
2:∗
3:∗
4:∗
5:−
6:∗
7:∗
8:∗
9:∗
10:∗
11:∗
12:∗
13:∗
20:−
21:−
22:−
23:−
24:+
25:+
26:+
30:−
31:+
32:+
33:−
34:−
35:−
36:−
37:+
38:+
40:−
41:+
42:−
43:−
44:−
45:−
46:−
47:−

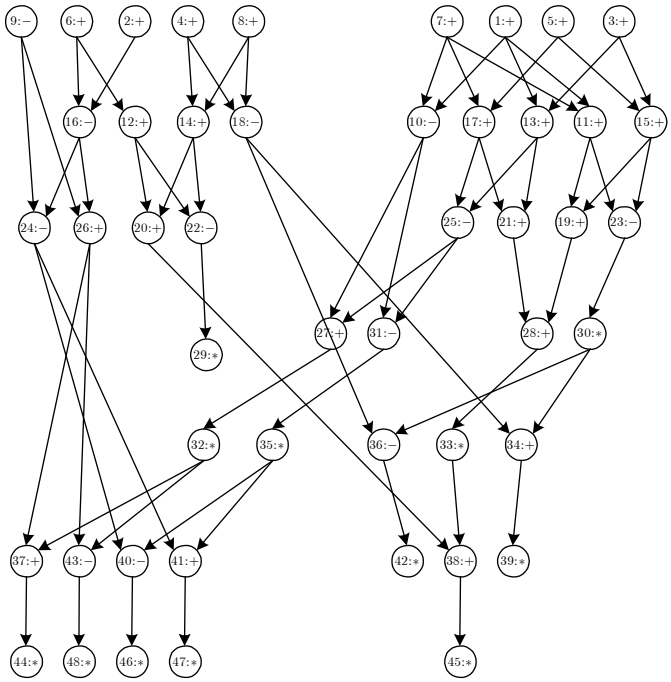
**FFT**



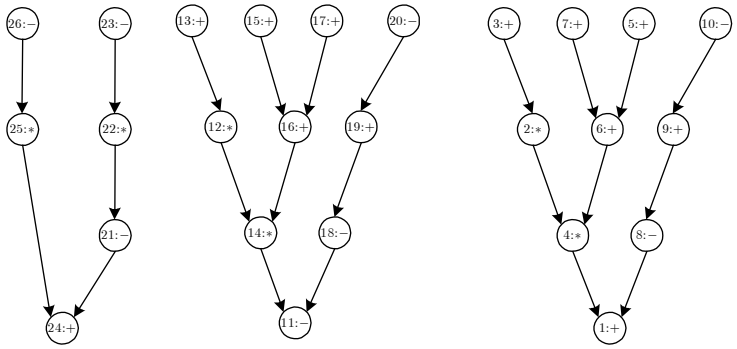

**DCT-DIF**



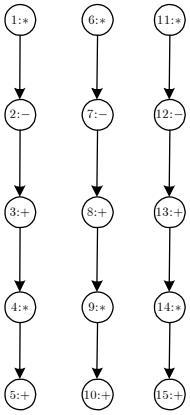

**DCT-LEE**



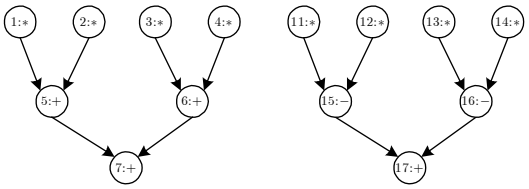

**DCT-DIT**



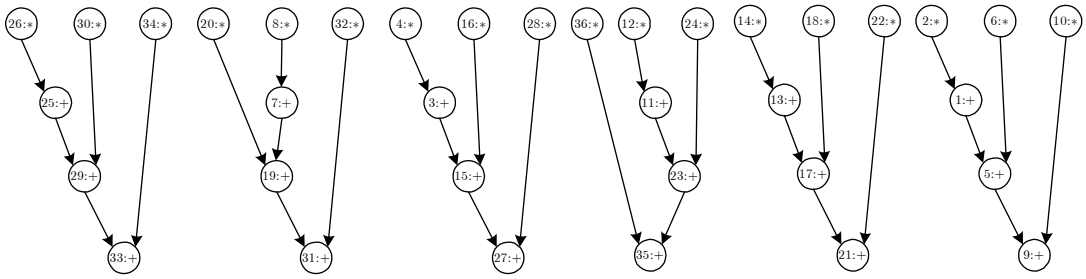

**SWIM1**



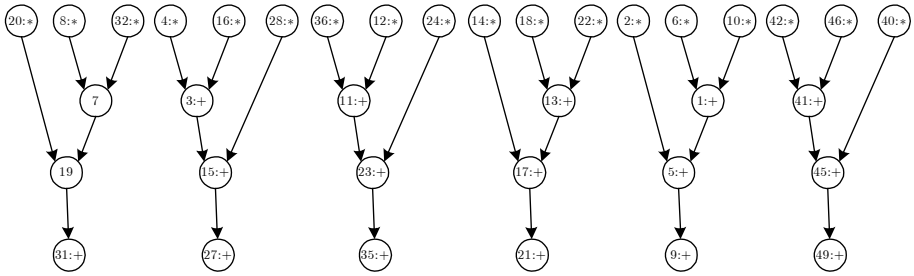

**SWIM2**



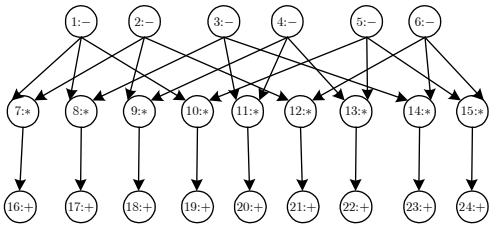

**WUPWISE**



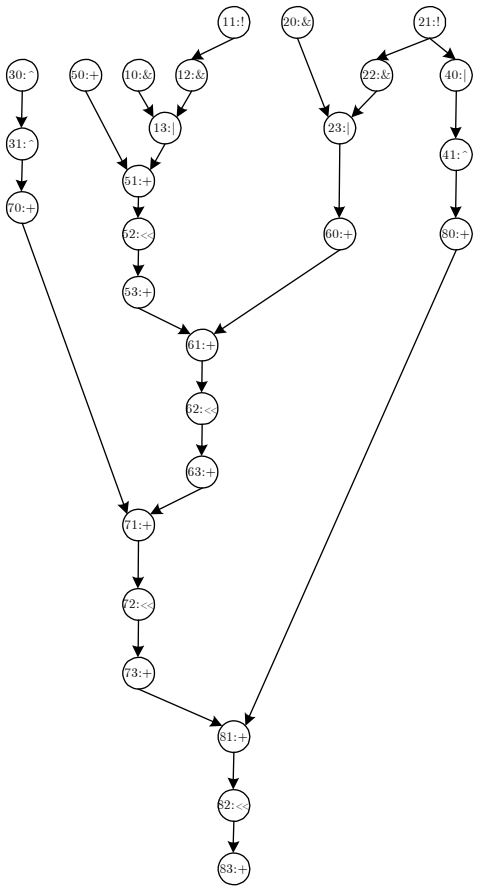

**EQUAKE1**




**EQUAKE2**




**AMMP**


11:!
20:&
21:!
30:¯
50:+
10:&
12:&
22:&
40:|
13:|
23:|
31:¯
41:¯
51:+
70:+
52:<<
60:+
80:+
53:+
61:+
62:<<
63:+
71:+
72:<<
73:+
81:+
82:<<
83:+

**MD5**

# Anhang B

## Ergebnisse der DSE mit DESCOMP für einen Cluster

Jede Tabelle enthält für ein Benchmarkprogramm die mit DESCOMP erzeugten Zielarchitekturen mit einem Cluster, d. h. $\mathcal{ZP}_b^1$ für $b$ aus der Mengen der Benchmarkprogramme. Zu jeder Ablaufplanlänge $L$ ist in eckigen Klammern für jede benötigte FU die Menge der in ihr zu implementierenden Operatoren angegeben. Für die unterstrichenen Ablaufplanlängen ist die Anzahl der FUs im Prozessor minimal. Die Architekturen sind jeweils bis zu der größten Ablaufplanlänge angegeben, die für den Vergleich mit den Lapinskii-Architekturen benötigt wurde.

DCT-LEE

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 |
|---|---|---|---|---|---|---|
| 9 | [-, *] | [-, +, *] | [-, +] | [-, *] | [+] | [-, *] |
| 10 | [-, +, *] | [-, +, *] | [+, *] | [-, *] | [-, *] | |
| 11 | [-, +] | [-, *] | [-, +] | [-, *] | [-, +, *] | |
| 12 | [-, *] | [-, +, *] | [+] | [-, *] | [-, +, *] | |
| 13 | [-, *] | [-, *] | [+, *] | [-, +, *] | | |
| 14 | [+, *] | [-, +, *] | [+] | [-, *] | | |
| 15 | [-, +, *] | [-] | [+] | [-, +, *] | | |
| 16 | [-, +] | [+, *] | [-, +] | [+, *] | | |
| 17 | [-, *] | [-, +, *] | [-, +, *] | | | |
| 18 | [-, *] | [-, +, *] | [-, +, *] | | | |
| 19 | [-, +, *] | [-, *] | [-, +] | | | |
| 20 | [-, +, *] | [-, *] | [+, *] | | | |
| 21 | [-, +, *] | [-] | [-, +, *] | | | |
| 22 | [-, +, *] | [-, *] | [-, +] | | | |
| 23 | [-, +, *] | [-, *] | [+] | | | |
| 24 | [-, +, *] | [-, *] | [+] | | | |
| 25 | [-] | [-, +, *] | [+] | | | |
| 26 | [-, *] | [-] | [+] | | | |
| 27 | [-, *] | [-, +] | | | | |
| 28 | [-, *] | [-, +] | | | | |
| 29 | [-, *] | [-, +] | | | | |
| 30 | [-, +, *] | [-, *] | [-, +] | | | |
| 31 | [-, +, *] | [-, *] | [-, +] | | | |

DCT-DIF

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 |
|---|---|---|---|---|---|---|
| 7 | [-, +, *] | [-, *] | [-, +] | [-, +, *] | [+] | [-, *] |
| 8 | [-, +, *] | [+] | [+] | [-, +] | [-] | [-, *] |
| 9 | [-, +, *] | [-, +] | [+] | [-, *] | [-, +] | |
| 10 | [-, *] | [+, *] | [+] | [-] | [-, +] | |
| 11 | [-, +, *] | [-, +] | [+] | [-, *] | | |
| 12 | [-, +, *] | [-, +] | [+] | [-, +, *] | | |
| 13 | [-, +] | [+, *] | [-, +] | [+, *] | | |
| 14 | [+] | [-, +] | [-, +] | [+, *] | | |
| 15 | [-, +, *] | [+, *] | [-, +] | | | |
| 16 | [+, *] | [-, +] | [-, +] | | | |
| 17 | [-, +] | [-, +, *] | [+] | | | |
| 18 | [-, +] | [-, +, *] | [+] | | | |
| 19 | [-, +, *] | [-, *] | [+] | | | |
| 20 | [+, *] | [-] | [+] | | | |
| 21 | [+, *] | [+] | [-] | | | |
| 22 | [+, *] | [-, +] | | | | |
| 23 | [+, *] | [-, +] | | | | |
| 24 | [+, *] | [-, +] | | | | |
| 25 | [+, *] | [-, +] | | | | |
| 26 | [+, *] | [-, +] | | | | |
| 27 | [-, *] | [-, +] | | | | |
| 28 | [+, *] | [-, +] | | | | |
| 29 | [-, *] | [-, +] | | | | |

DCT-DIT

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 |
|---|---|---|---|---|---|---|---|
| 7 | [-, +, *] | [+, *] | [+] | [+, *] | [-, +, *] | [-, *] | [+, *] |
| 8 | [-, +, *] | [-, +, *] | [+, *] | [-, +, *] | [+, *] | [+, *] | |
| 9 | [-, +, *] | [-, +] | [+] | [+, *] | [-] | [+, *] | |
| 10 | [-, +, *] | [-, +, *] | [+] | [+, *] | [-, *] | | |
| 11 | [-, +] | [+, *] | [-, *] | [+, *] | [+] | | |
| 12 | [-, +, *] | [-, +, *] | [+, *] | [+, *] | | | |
| 13 | [+] | [+, *] | [-, +] | [-, +, *] | | | |
| 14 | [-, +] | [+, *] | [+] | [-, +, *] | | | |
| 15 | [-, +] | [+, *] | [-, +] | [+, *] | | | |
| 16 | [-, +, *] | [+, *] | [-, +] | [+] | | | |
| 17 | [-, +, *] | [-, +, *] | [-] | [+, *] | | | |
| 18 | [-, +, *] | [-, +, *] | [+] | | | | |
| 19 | [-, +, *] | [-, +, *] | [+] | | | | |
| 20 | [-, +] | [+, *] | [-, +] | | | | |
| 21 | [-, +, *] | [+, *] | [-, +] | | | | |
| 22 | [-, +, *] | [+, *] | [-, +] | | | | |
| 23 | [-, +, *] | [-, *] | [+] | | | | |

EWF

| L | FU 0 | FU 1 | FU 2 | FU 3 |
|---|---|---|---|---|
| 14 | [+] | [+, *] | [+] | [+, *] |
| 15 | [+, *] | [+, *] | [+] | |
| 16 | [+] | [+, *] | [+] | |
| 17 | [+] | [+, *] | [+] | |
| 18 | [+] | [+, *] | [+] | |
| 19 | [+, *] | [+] | | |
| 20 | [+, *] | [+, *] | | |
| 21 | [+, *] | [+] | | |
| 22 | [+, *] | [+] | | |
| 23 | [+, *] | [+] | | |
| 24 | [+, *] | [+] | | |
| 25 | [+, *] | [+] | | |
| 26 | [+, *] | [+] | | |
| 27 | [+, *] | [+] | | |
| 28 | [+, *] | [+] | | |
| 29 | [+, *] | [+] | | |

ARF

| L | FU 0 | FU 1 | FU 2 | FU 3 |
|---|---|---|---|---|
| 8 | [+, *] | [+, *] | [+, *] | [+, *] |
| 9 | [+, *] | [+, *] | [+, *] | [*] |
| 10 | [+, *] | [+, *] | [+, *] | |
| 11 | [+, *] | [+, *] | [+, *] | |
| 12 | [+, *] | [+, *] | [+] | |
| 13 | [+, *] | [+, *] | [+] | |
| 14 | [+, *] | [+, *] | | |
| 15 | [+, *] | [+, *] | | |
| 16 | [+, *] | [+, *] | | |
| 17 | [+, *] | [+, *] | | |
| 18 | [+, *] | [+, *] | | |
| 19 | [+, *] | [+, *] | | |
| 20 | [+, *] | [*] | | |

FFT

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 | FU 7 | FU 8 | FU 9 | FU 10 | FU 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | [*] | [-, +, *] | [+, *] | [-, *] | [+, *] | [*] | [-, *] | [-, *] | [-, *] | [-, *] | [-, +, *] | [-, *] |
| 5 | [-, +, *] | [-, *] | [-] | [-, *] | [-, +, *] | [+] | [-, *] | [-, +, *] | | | | |
| 6 | [-, +, *] | [-] | [+] | [-, +, *] | [-, *] | [-, *] | [-, *] | | | | | |
| 7 | [-, +] | [-, *] | [+] | [-, *] | [-, *] | [-, *] | | | | | | |
| 8 | [-, +, *] | [-, +, *] | [-] | [+] | [-, *] | | | | | | | |
| 9 | [-, +, *] | [-, *] | [-] | [+] | [-, *] | | | | | | | |
| 10 | [-, +, *] | [-, *] | [-, *] | [+, *] | | | | | | | | |
| 11 | [-, *] | [-, +] | [-, +] | [-, *] | | | | | | | | |
| 12 | [-] | [-, +, *] | [-, *] | [+] | | | | | | | | |
| 13 | [-, +, *] | [-, *] | [-, +] | | | | | | | | | |
| 14 | [-, *] | [+, *] | [-, *] | | | | | | | | | |
| 15 | [-, +, *] | [+, *] | [-] | | | | | | | | | |
| 16 | [-, +, *] | [+, *] | [-] | | | | | | | | | |
| 17 | [-, +, *] | [+] | [-, *] | | | | | | | | | |
| 18 | [-, *] | [+] | [-, *] | | | | | | | | | |
| 19 | [-, +, *] | [-, +, *] | | | | | | | | | | |
| 20 | [-, *] | [-, +, *] | | | | | | | | | | |
| 21 | [-, *] | [-, +, *] | | | | | | | | | | |
| 22 | [-, *] | [-, +, *] | | | | | | | | | | |
| 23 | [-, *] | [-, +, *] | | | | | | | | | | |
| 24 | [-, +, *] | [-, *] | | | | | | | | | | |
| 25 | [-, +, *] | [-, *] | | | | | | | | | | |
| 26 | [-, +, *] | [-, *] | | | | | | | | | | |

AMMP

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 | FU 7 | FU 8 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | [+, *] | [-, +, *] | [+, *] | [-, +, *] | [-, +, *] | [+, *] | [-, +, *] | [-, +, *] | [-, +, *] |
| 4 | [-, +] | [-, +, *] | [+] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | | |
| 5 | [-, +] | [-, +, *] | [-, +] | [-, +, *] | [-, +, *] | | | | |
| 6 | [-, +, *] | [-, +, *] | [-, +] | [-, +, *] | | | | | |
| 7 | [-, +] | [-, +, *] | [-, +] | [-, +, *] | | | | | |
| 8 | [-, +, *] | [-, +, *] | [-, +] | | | | | | |
| 9 | [-, +, *] | [-, +, *] | [-, +] | | | | | | |
| 10 | [-, +, *] | [-, +, *] | [-, +] | | | | | | |
| 11 | [-, +] | [-, +] | [+, *] | | | | | | |
| 12 | [-, +, *] | [-, +] | | | | | | | |
| 13 | [-, +, *] | [-, +] | | | | | | | |
| 14 | [-, *] | [-, +] | | | | | | | |
| 15 | [*] | [-, +] | | | | | | | |

EQUAKE2

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 | FU 7 | FU 8 | FU 9 | FU 10 | FU 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | [*] | [*] | [+, *] | [*] | [+, *] | [*] | [+, *] | [*] | [+, *] | [+, *] | [*] | [+, *] |
| 5 | [*] | [+, *] | [+, *] | [+, *] | [+, *] | [*] | [+, *] | [+, *] | | | | |
| 6 | [+, *] | [+, *] | [+, *] | [+, *] | [+, *] | [+, *] | | | | | | |
| 7 | [+, *] | [+, *] | [+] | [+, *] | [+, *] | [+, *] | | | | | | |
| 8 | [+, *] | [+, *] | [+, *] | [+, *] | | | | | | | | |
| 9 | [+, *] | [+, *] | [+, *] | [+, *] | | | | | | | | |
| 10 | [+, *] | [+, *] | [+] | [+, *] | | | | | | | | |
| 11 | [+, *] | [+, *] | [+] | [+, *] | | | | | | | | |
| 12 | [+] | [+, *] | [+] | [+, *] | | | | | | | | |
| 13 | [+, *] | [+, *] | [+, *] | | | | | | | | | |
| 14 | [+, *] | [+, *] | [+] | | | | | | | | | |
| 15 | [+, *] | [+, *] | [+] | | | | | | | | | |
| 16 | [+, *] | [+, *] | [+] | | | | | | | | | |
| 17 | [+, *] | [+, *] | [+] | | | | | | | | | |
| 18 | [+, *] | [+, *] | [+] | | | | | | | | | |
| 19 | [+, *] | [+, *] | [+] | | | | | | | | | |
| 20 | [+, *] | [+, *] | | | | | | | | | | |
| 21 | [+, *] | [+, *] | | | | | | | | | | |
| 22 | [+, *] | [+, *] | | | | | | | | | | |
| 23 | [+, *] | [+, *] | | | | | | | | | | |
| 24 | [+, *] | [+, *] | | | | | | | | | | |
| 25 | [+, *] | [+] | | | | | | | | | | |

EQUAKE1

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 | FU 7 | FU 8 | FU 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 4 | [*] | [+, *] | [+, *] | [*] | [+, *] | [*] | [+, *] | [*] | [+, *] | [+, *] |
| 5 | [+, *] | [+, *] | [+] | [*] | [+, *] | [+, *] | [+, *] | [*] | | |
| 6 | [+, *] | [+, *] | [+, *] | [+, *] | [+, *] | [+, *] | | | | |
| 7 | [+] | [+, *] | [+] | [+, *] | [+, *] | [+, *] | | | | |
| 8 | [+, *] | [+, *] | [+] | [+, *] | [+, *] | | | | | |
| 9 | [+, *] | [+, *] | [+, *] | [+, *] | | | | | | |
| 10 | [+, *] | [+, *] | [+] | [+, *] | | | | | | |
| 11 | [+, *] | [+, *] | [+] | [+, *] | | | | | | |
| 12 | [+, *] | [+, *] | [+, *] | | | | | | | |
| 13 | [+, *] | [+, *] | [+] | | | | | | | |
| 14 | [+, *] | [+, *] | [+] | | | | | | | |
| 15 | [+, *] | [+, *] | [+] | | | | | | | |
| 16 | [+, *] | [+, *] | [+] | | | | | | | |
| 17 | [*] | [+, *] | [+] | | | | | | | |
| 18 | [+, *] | [+, *] | [+] | | | | | | | |
| 19 | [+, *] | [+, *] | | | | | | | | |

WUPWISE

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 | FU 7 |
|---|---|---|---|---|---|---|---|---|
| 3 | [*] | [+, *] | [*] | [+, *] | [+, *] | [*] | [+, *] | [*] |
| 4 | [+, *] | [+, *] | [+, *] | [+, *] | | | | |
| 5 | [+, *] | [+, *] | [+, *] | | | | | |
| 6 | [+, *] | [+] | [*] | | | | | |
| 7 | [+, *] | [+, *] | | | | | | |
| 8 | [+, *] | [+, *] | | | | | | |
| 9 | [+, *] | [+, *] | | | | | | |
| 10 | [+, *] | [+] | | | | | | |
| 11 | [+, *] | [+] | | | | | | |
| 12 | [+, *] | [+] | | | | | | |
| 13 | [+] | [*] | | | | | | |

SWIM1

| L | FU 0 | FU 1 | FU 2 | FU 3 | FU 4 | FU 5 | FU 6 | FU 7 | FU 8 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | [+, *] | [+] | [-] | [+, *] | [-] | [+, *] | [+] | [+, *] | [-, *] |
| 5 | [-, *] | [-, +] | [+] | [-, *] | [-, +] | [-, *] | | | |
| 6 | [-, +] | [+, *] | [-] | [-, +] | [+, *] | | | | |
| 7 | [-, +] | [-, *] | [-, +] | [-, +, *] | | | | | |
| 8 | [-, *] | [-, +] | [+] | [-, *] | | | | | |
| 9 | [+, *] | [-, +] | [-, *] | | | | | | |
| 10 | [-, +, *] | [+, *] | [-, +] | | | | | | |
| 11 | [-] | [+] | [+, *] | | | | | | |
| 12 | [-, +] | [+, *] | [-] | | | | | | |
| 13 | [+, *] | [-, +] | | | | | | | |
| 14 | [+, *] | [-, *] | | | | | | | |
| 15 | [+, *] | [-, *] | | | | | | | |
| 16 | [+, *] | [-, +] | | | | | | | |
| 17 | [+, *] | [-, +] | | | | | | | |
| 18 | [+, *] | [-, +] | | | | | | | |

SWIM2

| L | FU 0 | FU 1 | FU 2 |
|---|---|---|---|
| 5 | [-, +, *] | [-, +, *] | [-, +, *] |
| 6 | [+, *] | [-, +, *] | [-, +] |
| 7 | [+, *] | [-, +] | [*] |
| 8 | [-, *] | [-, +, *] | |
| 9 | [-, *] | [+, *] | |
| 10 | [+, *] | [-, *] | |
| 11 | [+, *] | [-, *] | |
| 12 | [-, *] | [+, *] | |
| 13 | [+, *] | [-] | |
| 14 | [+, *] | [-] | |
| 15 | [-, +, *] | | |

# Ergebnisse der DSE mit DESCOMP für zwei Cluster

Jede der folgenden Tabellen enthält für ein Benchmarkprogramm die mit DESCOMP erzeugten Zielarchitekturen mit zwei, drei oder vier Clustern, d. h. $\mathcal{ZP}_b^2$, $\mathcal{ZP}_b^3$, $\mathcal{ZP}_b^4$ oder $\mathcal{ZP}_b^5$ für $b$ aus der Menge der Benchmarkprogramme. Zu jeder Ablaufplanlänge $L$ ist in eckigen Klammern für jede benötigte FU die Menge der in ihr zu implementierenden Operatoren angegeben und in der Spalte $C$ die Anzahl der benötigten Kopieroperatoren. Für jeden Basisblock $b$ und jede Clusteranzahl wurden Architekturen zu den Ablaufplanlängen $cp(b) \leq l \leq cp(b) + 10$ erzeugt, was für den Vergleich mit den Lapinskii-Architekturen ausreichend war. Leere Zeilen in einer Tabelle bedeuten, dass zu der entsprechenden Ablaufplanlänge und Clusteranzahl keine Clusterung erzeugt wurde.

DCT-LEE

| | | Cluster 0 | | | | Cluster 1 | | |
|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 |
| 9 | 1 | [-, *] | [-, +, *] | [-, +, *] | [-, *] | [-, +, *] | [-, +] | [+, *] |
| 10 | 1 | [-, +, *] | [-, *] | [-, +] | | [+] | [-, +, *] | [-, *] |
| 11 | 1 | [-, +, *] | [-, *] | [-, +] | | [-, +] | [-, +, *] | [+, *] |
| 12 | 1 | [-, +, *] | [-, +, *] | | | [-, +, *] | [-, +] | [-, *] |
| 13 | 1 | [-, *] | [-, +, *] | | | [-, +, *] | [+] | [-, *] |
| 14 | 1 | [-, *] | [-, +, *] | | | [-, +, *] | [-, +, *] | |
| 15 | 2 | [-] | [-, +, *] | | | [+, *] | [-, +, *] | |
| 16 | 2 | [-, *] | [-, +] | | | [-, +] | [-, +, *] | |
| 17 | 2 | [-, *] | [-, +] | | | [-, +] | [+, *] | |
| 18 | 2 | [-, *] | [-, +] | | | [-, +, *] | [+, *] | |
| 19 | 2 | [-, +, *] | [-, +] | | | [-, +, *] | | |

DCT-DIF

| | | Cluster 0 | | | | Cluster 1 | | |
|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 |
| 7 | 0 | [-, +, *] | [-, +, *] | [-, *] | [-, +, *] | [-, +] | [+] | [+, *] |
| 8 | 0 | [-, +] | [-, +, *] | [-] | [-, +, *] | [-, +] | [+] | [+, *] |
| 9 | 0 | [-, +, *] | [-, +, *] | [-, +] | | [+, *] | [-, +] | |
| 10 | 0 | [-, +, *] | [-, +, *] | [+] | | [-, +] | [+, *] | |
| 11 | 0 | [+, *] | [-, +] | [+] | | [-, +, *] | [+] | |
| 12 | 0 | [+, *] | [-, +] | [+] | | [+, *] | [-, +] | |
| 13 | 1 | [-, +, *] | [-, +, *] | | | [+, *] | [-, +] | |
| 14 | 1 | [+, *] | [-, +] | | | [+, *] | [-, +] | |
| 15 | 1 | [-, +, *] | | | | [-, +] | [-, +, *] | |
| 16 | 1 | [-, +, *] | | | | [-, +] | [-, *] | |
| 17 | 1 | [-, +, *] | | | | [-, +] | [+, *] | |

DCT-DIT

| | | Cluster 0 | | | | | Cluster 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 0.4 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 |
| 7 | 2 | [+, *] | [-, *] | [+, *] | [-, +, *] | [+, *] | [-, +] | [+, *] | [+] | [-, +, *] |
| 8 | 2 | [-, +, *] | [-, +, *] | [+] | [-, +, *] | | [+, *] | [+] | [-] | |
| 9 | 1 | [+, *] | [-, +, *] | [+] | [-, +, *] | | [-, +] | [-, +] | [+, *] | |
| 10 | 2 | [-, +, *] | [-, +, *] | [+] | | | [-, +, *] | [+] | [-] | |
| 11 | 1 | [+, *] | [-, +, *] | [-, +] | | | [+] | [+, *] | [-] | |
| 12 | 1 | [-, +, *] | [-] | [+, *] | | | [-] | [+] | [+, *] | |
| 13 | 1 | [+, *] | [-, +, *] | | | | [+, *] | [-, +] | | |
| 14 | 1 | [-, +, *] | [-, +] | | | | [+, *] | [-, +] | | |
| 15 | 1 | [+, *] | [-, +, *] | | | | [+, *] | [-, +] | | |
| 16 | 1 | [+, *] | [-, +] | | | | [-, +, *] | [-, +, *] | | |
| 17 | 1 | [-, +, *] | | | | | [-, +, *] | [-, +, *] | | |

EWF

| | | Cluster 0 | | | Cluster 1 | |
|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 1.0 | FU 1.1 |
| 14 | 1 | [+, *] | [+, *] | [+] | [+, *] | |
| 15 | 1 | [+, *] | [+] | | [+, *] | [+] |
| 16 | 1 | [+, *] | | | [+] | [+, *] |
| 17 | 1 | [+, *] | | | [+] | [+, *] |
| 18 | 1 | [+, *] | | | [+] | [+, *] |
| 19 | 1 | [+, *] | | | [+] | [+, *] |
| 20 | 0 | [+, *] | [+, *] | | | |
| 21 | 0 | [+, *] | [+] | | | |
| 22 | 1 | [+, *] | | | [+, *] | |
| 23 | 1 | [+, *] | | | [+, *] | |
| 24 | 1 | [+, *] | | | [+, *] | |

ARF

| | | Cluster 0 | | | | Cluster 1 | |
|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 |
| 8 | | [+, *] | [+, *] | [+, *] | [+, *] | | |
| 9 | | [+, *] | [+, *] | [+, *] | [*] | | |
| 10 | 1 | [+, *] | [+, *] | | | [+, *] | [+, *] |
| 11 | 1 | [+, *] | [+] | | | [+, *] | [+, *] |
| 12 | 1 | [+, *] | | | | [+, *] | [+] |
| 13 | 1 | [+, *] | | | | [+, *] | [+] |
| 14 | 1 | [+, *] | | | | [*] | [+] |
| 15 | 1 | [+, *] | | | | [+, *] | |
| 16 | 1 | [+, *] | | | | [+, *] | |
| 17 | 1 | [+, *] | | | | [+, *] | |
| 18 | 1 | [+, *] | | | | [+, *] | |

FFT

| | | Cluster 0 | | | | Cluster 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 |
| 4 | | | | | | | | | |
| 5 | 2 | [-, *] | [-, *] | [-, *] | [+, *] | [-, +, *] | [-, *] | [-, +] | [-, *] |
| 6 | 1 | [-, +, *] | [-, *] | [-] | [-, +] | [-, +, *] | [-, +, *] | [-, +, *] | |
| 7 | 1 | [-, +, *] | [-, +] | [-, *] | | [-] | [-, *] | [-, +] | [-, +, *] |
| 8 | 1 | [-, +, *] | [-] | [-, +, *] | | [-, *] | [-, +, *] | [-, +] | |
| 9 | 1 | [-, *] | [+, *] | | | [-, +, *] | [-, +] | [-, *] | |
| 10 | 2 | [-, +, *] | [-] | | | [-, +, *] | [-, +, *] | | |
| 11 | 2 | [-, +, *] | [-] | | | [-, +, *] | [-, +, *] | | |
| 12 | 2 | [-, +] | [-, *] | | | [-, +, *] | [-, +, *] | | |
| 13 | 1 | [-, *] | [-, +, *] | | | [-, +, *] | [-, *] | | |
| 14 | 1 | [-, +, *] | [-, *] | | | [-, +, *] | [-, *] | | |

AMMP

| | | Cluster 0 | | | Cluster 1 | | |
|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 1.0 | FU 1.1 | FU 1.2 |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | 2 | [-, +, *] | [-, +, *] | [-, +] | [-, +, *] | [+] | [-, +, *] |
| 6 | 2 | [-, +, *] | [-, +] | | [-, +, *] | [-, +, *] | |
| 7 | 2 | [-, +, *] | [-, +] | | [-, +] | [-, +, *] | |
| 8 | 1 | [-, +, *] | [-, +] | | [-, +, *] | [-, +] | |
| 9 | 1 | [-, +] | [+, *] | | [-, +, *] | [-, +] | |
| 10 | 1 | [-, +, *] | | | [-, +] | [-, +, *] | |
| 11 | 1 | [-, +, *] | | | [+] | [-, +, *] | |
| 12 | 1 | [-, +, *] | | | [+] | [-, +, *] | |
| 13 | 1 | [-, +, *] | | | [-, +, *] | | |

EQUAKE2

| | | Cluster 0 | | | | | | Cluster 1 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 0.4 | FU 0.5 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 | FU 1.4 | FU 1.5 |
| 4 | 0 | [*] | [+, *] | [+, *] | [*] | [+, *] | [*] | [*] | [*] | [+, *] | [+, *] | [*] | [+, *] |
| 5 | 0 | [+] | [*] | [+] | [*] | [*] | [+] | [*] | [+] | [*] | [+] | [+] | [*] |
| 6 | 1 | [+, *] | [+, *] | [+] | | | | [*] | [+, *] | [+] | [+, *] | | |
| 7 | 1 | [+, *] | [+, *] | [+] | | | | [+] | [+] | [*] | [+, *] | | |
| 8 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+] | [+, *] | | | |
| 9 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+] | [+, *] | | | |
| 10 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+, *] | | | | |
| 11 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+] | | | | |
| 12 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+] | | | | |
| 13 | 1 | [+, *] | | | | | | [+, *] | [+, *] | | | | |
| 14 | 1 | [+, *] | | | | | | [+, *] | [+, *] | | | | |

EQUAKE1

| | | Cluster 0 | | | | | | Cluster 1 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 0.4 | FU 0.5 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 | FU 1.4 | FU 1.5 |
| 4 | 0 | [*] | [+] | [+] | [*] | [+] | [*] | [*] | [*] | [+] | [+] | [*] | [+] |
| 5 | 0 | [*] | [+] | [+] | [*] | [+, *] | | [+] | [*] | [*] | [+, *] | [+] | |
| 6 | 0 | [+] | [*] | [+] | [+, *] | | | [+] | [+] | [*] | [+, *] | | |
| 7 | 1 | [+, *] | [+, *] | [+] | | | | [+] | [+, *] | [+, *] | | | |
| 8 | 1 | [+, *] | [+, *] | [+] | | | | [+] | [*] | [+] | | | |
| 9 | 1 | [+, *] | [+, *] | [+] | | | | [+] | [+, *] | | | | |
| 10 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+, *] | | | | |
| 11 | 1 | [+, *] | [+, *] | | | | | [+, *] | [+] | | | | |
| 12 | 1 | [*] | [+] | | | | | [+, *] | [+] | | | | |
| 13 | 1 | [+, *] | | | | | | [+, *] | [+, *] | | | | |
| 14 | 1 | [+, *] | [+] | | | | | [+, *] | | | | | |

WUPWISE

| | | Cluster 0 | | | | Cluster 1 | | | |
|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 |
| 3 | 0 | [*] | [+, *] | [*] | [+, *] | [*] | [*] | [+, *] | [+, *] |
| 4 | 0 | [+, *] | [+, *] | | | [+, *] | [+, *] | | |
| 5 | 0 | [+, *] | [+, *] | | | [+, *] | [+, *] | | |
| 6 | 0 | [*] | [+] | | | [+] | [*] | | |
| 7 | 0 | [+, *] | | | | [+, *] | | | |
| 8 | 0 | [+, *] | | | | [+, *] | | | |
| 9 | 0 | [+, *] | | | | [+, *] | | | |
| 10 | 0 | [+, *] | | | | [+, *] | | | |
| 11 | 0 | [+, *] | | | | [+, *] | | | |
| 12 | 0 | [+, *] | | | | [+, *] | | | |
| 13 | 0 | [+, *] | | | | [+, *] | | | |

SWIM1

| | | Cluster 0 | | | | Cluster 1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 | FU 1.4 |
| 4 | 0 | [+, *] | [+] | [-] | [+, *] | [-] | [+] | [+, *] | [+, *] | [-, *] |
| 5 | 1 | [+] | [-, *] | [-, +] | [+, *] | [+, *] | [-] | | | |
| 6 | 1 | [+, *] | [+] | [-] | | [+] | [-, *] | [-] | | |
| 7 | 1 | [+, *] | [+] | [-] | | [-] | [+, *] | | | |
| 8 | 1 | [+, *] | [-, +] | | | [+, *] | [-] | | | |
| 9 | 1 | [+] | [-, *] | | | [+, *] | [-] | | | |
| 10 | 1 | [-, *] | [-, +] | | | [-, +, *] | | | | |
| 11 | 1 | [+] | [-, *] | | | [-, +, *] | | | | |
| 12 | 1 | [+] | [-, *] | | | [-, +, *] | | | | |
| 13 | 1 | [+] | [-, *] | | | [-, +, *] | | | | |
| 14 | 1 | [+] | [-, *] | | | [-, +, *] | | | | |

SWIM2

| | | Cluster 0 | Cluster 1 | |
|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 1.1 |
| 5 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 6 | 0 | [-, +, *] | [+, *] | [-, +] |
| 7 | 0 | [-, +, *] | [-, *] | [+, *] |
| 8 | 1 | [-, +, *] | [-, +, *] | |
| 9 | 1 | [-, +, *] | [-, +, *] | |
| 10 | 1 | [-, +, *] | [-, +, *] | |
| 11 | 1 | [-, +, *] | [-, +, *] | |
| 12 | 1 | [-, +, *] | [-, +, *] | |
| 13 | 1 | [-, +, *] | [-, +, *] | |
| 14 | 1 | [-, +, *] | [-, +, *] | |
| 15 | 0 | [-, +, *] | | |

# Ergebnisse der DSE mit DESCOMP für drei Cluster

DCT-LEE

| | | Cluster 0 | | | | Cluster 1 | | Cluster 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 |
| 9 | 1 | [-, *] | [-, +, *] | [-, +, *] | [-, *] | [*] | | [+, *] | [-, +, *] | [-, +] |
| 10 | 1 | [-, *] | [-, +, *] | [-, +] | | [-, *] | | [-, +, *] | [-, +] | [+, *] |
| 11 | 1 | [-, *] | [-, +, *] | | | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | |
| 12 | 1 | [-] | [+, *] | | | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | |
| 13 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [-, +, *] | [-, +, *] | |
| 14 | 1 | [-, +, *] | | | | [-] | [-, +, *] | [-, +] | [-, +, *] | |
| 15 | 1 | [-, +, *] | | | | [-, +, *] | [-, *] | [-, +] | [-, +, *] | |
| 16 | 1 | [-, +, *] | | | | [-, +, *] | | [+, *] | [-, +] | |
| 17 | 1 | [-, +, *] | | | | [-, +, *] | | [-, +, *] | [-, +] | |
| 18 | 1 | [-, +, *] | | | | [-, +, *] | | [+, *] | [-, +] | |
| 19 | 1 | [-, +, *] | | | | [-, +, *] | | [+, *] | [-, +] | |

DCT-DIF

| | | Cluster 0 | | | | Cluster 1 | | Cluster 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 |
| 7 | 0 | | | | | | | | | |
| 8 | 1 | [-, +] | [-, +, *] | [+] | | [-, +, *] | [-, *] | [+, *] | [+] | [-, +] |
| 9 | 1 | [-, +, *] | [-, +, *] | | | [-, +, *] | [-, *] | [+, *] | [-, +] | |
| 10 | 1 | [-, +] | [+, *] | | | [-, +] | [-, *] | [+, *] | [-, +] | |
| 11 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [-, +] | [+, *] | |
| 12 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [+] | [-, +, *] | |
| 13 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [+] | [-, +, *] | |
| 14 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [+, *] | [-, +] | |
| 15 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [+, *] | [-, +] | |
| 16 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [+, *] | [-, +] | |
| 17 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [+, *] | [-, +] | |

DCT-DIT

| | | Cluster 0 | | Cluster 1 | | Cluster 2 | |
|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | 3 | [+, *] | [-, +, *] | [+, *] | [-, *] | [+, *] | [-, +] |
| 10 | 2 | [-, +, *] | [+, *] | [-, *] | [-, +, *] | [-, +] | [+, *] |
| 11 | 2 | [+, *] | [-, +] | [-, *] | [-, +, *] | [-, +] | [+, *] |
| 12 | 2 | [-, +, *] | | [-, +, *] | [-, +, *] | [+, *] | [-, +] |
| 13 | 2 | [+, *] | [-, +] | [-, +] | [-, *] | [-, +] | [+, *] |
| 14 | 1 | [-, +, *] | | [-, +, *] | [-, +] | [-, +] | [+, *] |
| 15 | 1 | [-, *] | [+] | [-, +] | [+, *] | [+, *] | [-, +] |
| 16 | 1 | [+, *] | [-, +] | [-, +, *] | | [-, +] | [-, +, *] |
| 17 | 1 | [+, *] | [-, +] | [-, +, *] | | [-] | [+, *] |

EWF

| | | Cluster 0 | Cluster 1 | Cluster 2 |
|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 2.0 |
| 14 | | | | |
| 15 | | | | |
| 16 | 1 | [+, *] | [+, *] | [+, *] |
| 17 | 1 | [+, *] | [+, *] | [+, *] |
| 18 | 1 | [+, *] | [+, *] | [+, *] |
| 19 | 1 | [+, *] | [+, *] | [+, *] |
| 20 | 1 | [+, *] | [+, *] | [+, *] |
| 21 | | | | |
| 22 | | | | |
| 23 | | | | |

ARF

| | | Cluster 0 | Cluster 1 | Cluster 2 |
|---|---|---|---|---|
| | | FU 0.0 | FU 1.0 | FU 2.0 |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | 2 | [+, *] | [+, *] | [+, *] |
| 13 | 2 | [+, *] | [+, *] | [+, *] |
| 14 | 2 | [+, *] | [+, *] | [+, *] |
| 15 | 2 | [+, *] | [+, *] | [+, *] |
| 16 | | | | |
| 17 | | | | |

FFT

| | | Cluster 0 | | | | Cluster 1 | | Cluster 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 |
| 4 | | | | | | | | | | | |
| 5 | 2 | [-, +, *] | [-, *] | [-, +] | [-, +, *] | [-] | [+, *] | [-, *] | [+] | [-, *] | [-, *] |
| 6 | 2 | [-, +] | [-, *] | [-, +] | [-, +, *] | [-, +, *] | | [-, *] | [+, *] | [-, *] | |
| 7 | 1 | [-, +, *] | [+, *] | [-, +] | | [-, +, *] | | [+] | [-, *] | [-, *] | |
| 8 | 1 | [-, +, *] | [-, +, *] | | | [-, +, *] | | [+] | [-, *] | [-, *] | |
| 9 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [-, *] | [-, +, *] | | |
| 10 | 1 | [-, +, *] | [-] | | | [+] | [-, *] | [-, +, *] | [-] | | |
| 11 | 1 | [-, +, *] | | | | [-, *] | [-, +] | [-, +] | [-, *] | | |
| 12 | 1 | [-, +, *] | | | | [+, *] | [-] | [-] | [+, *] | | |
| 13 | 1 | [-, +, *] | | | | [-, +, *] | | [-, *] | [-, +, *] | | |
| 14 | | [-, +, *] | | | | [-, +, *] | | [-, *] | [-, +] | | |

AMMP

| | | Cluster 0 | | | Cluster 1 | | | Cluster 2 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 1.0 | FU 1.1 | FU 1.2 | FU 2.0 | FU 2.1 | FU 2.2 |
| 3 | | | | | | | | | | |
| 4 | 3 | [-, +, *] | [-, +, *] | [-, +] | [-, +, *] | [+, *] | [+, *] | [+] | [-, +, *] | [-, +, *] |
| 5 | 3 | [-, +, *] | [-, +] | | [-, +] | [-, +, *] | | [-, +, *] | [-, +, *] | |
| 6 | 2 | [-, +] | [+, *] | | [+] | [-, +, *] | | [+, *] | [-, +, *] | |
| 7 | 2 | [-, +, *] | | | [-, +, *] | | | [-, +] | [-, +, *] | |
| 8 | 2 | [-, +, *] | | | [-, +, *] | | | [-, +, *] | | |
| 9 | 2 | [-, +, *] | | | [-, +, *] | | | [-, +, *] | | |
| 10 | 2 | [-, +, *] | | | [-, +, *] | | | [-, +, *] | | |
| 11 | 2 | [-, +, *] | | | [-, +, *] | | | [-, +, *] | | |
| 12 | | | | | | | | | | |
| 13 | | | | | | | | | | |

EQUAKE2

| | | Cluster 0 | | | | Cluster 1 | | | | Cluster 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 |
| 4 | 0 | [*] | [+, *] | [+, *] | [*] | [+, *] | [*] | [+, *] | [*] | [+, *] | [+, *] | [*] | [*] |
| 5 | 0 | [+] | [*] | [+] | [*] | [+] | [+] | [*] | [*] | [*] | [+] | [+] | [*] |
| 6 | 0 | [+, *] | [+, *] | | | [+] | [+, *] | [*] | | [+, *] | [+, *] | | |
| 7 | 0 | [+, *] | [*] | [+] | | [*] | [+] | [+, *] | | [*] | [+, *] | [+] | |
| 8 | 1 | [+, *] | | | | [+, *] | [+, *] | | | [+] | [+, *] | | |
| 9 | 1 | [+, *] | | | | [+, *] | [+, *] | | | [+] | [+, *] | | |
| 10 | 1 | [+, *] | | | | [+, *] | | | | [+, *] | [+, *] | | |
| 11 | 1 | [+, *] | | | | [+, *] | | | | [+, *] | [+] | | |
| 12 | 1 | [+, *] | | | | [+, *] | | | | [+, *] | [+] | | |
| 13 | 1 | [+, *] | | | | [+, *] | | | | [+, *] | | | |
| 14 | 1 | [+, *] | | | | [+, *] | | | | [+, *] | | | |

EQUAKE1

| | | Cluster 0 | | | | Cluster 1 | | | | Cluster 2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 |
| 4 | 0 | [*] | [+] | [+] | [*] | [+] | [*] | [+] | [*] | [+] | [+] | [*] | [*] |
| 5 | 0 | [*] | [+, *] | [+] | | [*] | [+] | [+, *] | | [+, *] | [*] | [+] | |
| 6 | 0 | [+, *] | [*] | [+] | | [*] | [+] | [+, *] | | [*] | [+, *] | [+] | |
| 7 | 0 | [+] | [*] | [+] | | [*] | [+] | [+] | | [*] | [+] | [+] | |
| 8 | 0 | [+, *] | [+] | | | [+] | [+, *] | | | [+] | [+, *] | | |
| 9 | 0 | [+] | [*] | | | [+] | [+, *] | | | [+] | [+, *] | | |
| 10 | 0 | [+] | [*] | | | [*] | [+] | | | [+] | [+, *] | | |
| 11 | 0 | [+] | [*] | | | [*] | [+] | | | [+] | [+, *] | | |
| 12 | 0 | [+, *] | | | | [+, *] | | | | [+, *] | | | |
| 13 | 0 | [+, *] | | | | [+, *] | | | | [+, *] | | | |
| 14 | 0 | [+, *] | | | | [+, *] | | | | [+, *] | | | |

SWIM1

| | | Cluster 0 | | | | Cluster 1 | | | | Cluster 2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 0.3 | FU 1.0 | FU 1.1 | FU 1.2 | FU 1.3 | FU 2.0 | FU 2.1 |
| 4 | 0 | [+, *] | [+] | [-] | [+, *] | [-] | [+, *] | [+] | [+, *] | [-, +, *] | [-] |
| 5 | 1 | [+, *] | [-, +] | | | [-, +] | [+, *] | | | [+, *] | [-] |
| 6 | 1 | [-, +, *] | | | | [+, *] | [-, +] | | | [-] | [+, *] |
| 7 | 1 | [-, +, *] | | | | [-, *] | [-, +] | | | [-] | [+, *] |
| 8 | 1 | [-, +, *] | | | | [-, +, *] | | | | [+, *] | [-] |
| 9 | 1 | [-, +, *] | | | | [-, +, *] | | | | [+, *] | [-] |
| 10 | 1 | [-, +, *] | | | | [-, +, *] | | | | [+, *] | [-] |
| 11 | 1 | [-, +, *] | | | | [-, +, *] | | | | [-, +, *] | |
| 12 | 1 | [-, +, *] | | | | [-, +, *] | | | | [-, +, *] | |
| 13 | 1 | [-, +, *] | | | | [-, +, *] | | | | [-, +, *] | |
| 14 | 1 | [-, +, *] | | | | [-, +, *] | | | | [-, +, *] | |

SWIM2

| | | Cluster 0 | Cluster 1 | Cluster 2 |
|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 2.0 |
| 5 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 6 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 7 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 8 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 9 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 10 | 0 | [-, +, *] | [-, +, *] | [-, +, *] |
| 11 | | | | |
| 12 | | | | |
| 13 | | | | |
| 14 | | | | |
| 15 | | | | |

# Ergebnisse der DSE mit DESCOMP für vier Cluster

DCT-LEE

| | | Cluster 0 | Cluster 1 | | Cluster 2 | | Cluster 3 | |
|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 3.0 | FU 3.1 |
| 9 | | | | | | | | |
| 10 | | | | | | | | |
| 11 | 2 | [-, +, *] | [-, +, *] | [-, *] | [-, +, *] | [-, +] | [-, +, *] | [+, *] |
| 12 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | [-, +, *] | [-] | [+, *] |
| 13 | 2 | [-, +, *] | [-, +, *] | | [-, +] | [-, *] | [+] | [-, *] |
| 14 | 2 | [-, +, *] | [-, +, *] | | [-, +] | [-, *] | [-, +] | [+, *] |
| 15 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | [-, +] |
| 16 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 17 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 18 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 19 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |

DCT-DIF

| | | Cluster 0 | Cluster 1 | | Cluster 2 | | Cluster 3 | |
|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 3.0 | FU 3.1 |
| 7 | | | | | | | | |
| 8 | | | | | | | | |
| 9 | 2 | [+, *] | [-, *] | [-, +, *] | [-, +, *] | [-, +, *] | [+, *] | [-, +] |
| 10 | 2 | [-, +, *] | [-, *] | | [-, +] | [-, +, *] | [-, +] | [-, +, *] |
| 11 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | | [+, *] | [-, +] |
| 12 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 13 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 14 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 15 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 16 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 17 | | | | | | | | |

DCT-DIT

| | | Cluster 0 | | Cluster 1 | | Cluster 2 | | Cluster 3 | |
|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 3.0 | FU 3.1 |
| 7 | | | | | | | | | |
| 8 | | | | | | | | | |
| 9 | 3 | [+, *] | [-, *] | [-, +, *] | [+, *] | [+, *] | | [+, *] | [-, +, *] |
| 10 | 2 | [-, +, *] | | [+, *] | | [-, +, *] | [+, *] | [-, +] | [+, *] |
| 11 | 2 | [-, +, *] | | [+, *] | | [-, +] | [+, *] | [-] | [+, *] |
| 12 | 2 | [+, *] | [-] | [-, +, *] | | [+, *] | | [-, +] | [+, *] |
| 13 | 2 | [-, +, *] | | [-, +, *] | | [+, *] | | [+, *] | [-, +] |
| 14 | 2 | [-, +, *] | | [-, +, *] | | [+, *] | | [-, *] | [+] |
| 15 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | [-, *] | [+] |
| 16 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 17 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 18 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 19 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | [-, +, *] | |

EWF

| | | Cluster 0 | Cluster 1 | Cluster 2 | Cluster 3 |
|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 2.0 | FU 3.0 |
| 14 | | | | | |
| 15 | 1 | [+, *] | [+, *] | [+] | [+, *] |
| 16 | 1 | [+, *] | [+, *] | [+] | [+, *] |
| 17 | 1 | [+, *] | [+, *] | [+] | [+, *] |
| 18 | | | | | |
| 19 | | | | | |
| 20 | | | | | |
| 21 | | | | | |
| 22 | | | | | |
| 23 | | | | | |
| 24 | | | | | |

FFT

| | | Cluster 0 | | Cluster 1 | | Cluster 2 | | | | Cluster 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 | FU 3.0 | FU 3.1 | FU 3.2 | FU 3.3 |
| 4 | | | | | | | | | | | | | |
| 5 | 2 | [-, +, *] | | [*] | | [-, *] | [-] | [-, *] | [+] | [-, *] | [-, +] | [-, +, *] | [-, +, *] |
| 6 | 2 | [-, +, *] | | [+, *] | [-] | [-, *] | [-, +, *] | | | [-, +, *] | [-, +, *] | [-, *] | |
| 7 | 3 | [-, +, *] | | [+, *] | [-] | [-, *] | [-, +, *] | | | [-, +, *] | [-, +, *] | | |
| 8 | 2 | [-, *] | [+] | [-, +, *] | | [-, +, *] | [-, *] | | | [-, *] | [-, +] | | |
| 9 | 2 | [-, +, *] | | [-, +] | [-, *] | [-, +, *] | | | | [-, +, *] | | | |
| 10 | 2 | [-, +, *] | | [-, *] | [-, +] | [-, +, *] | | | | [*] | [-, +, *] | | |
| 11 | 2 | [-, +, *] | | [-, +] | [-, *] | [-, +, *] | | | | [-, +, *] | | | |
| 12 | 1 | [-, +, *] | | [-, +, *] | | [-, +, *] | | | | [-, +] | [-, *] | | |
| 13 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | | | [-, +, *] | | | |
| 14 | 2 | [-, +, *] | | [-, +, *] | | [-, +, *] | | | | [-, +, *] | | | |

AMMP

| | | Cluster 0 | | | Cluster 1 | Cluster 2 | | | Cluster 3 | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 0.2 | FU 1.0 | FU 2.0 | FU 2.1 | FU 2.0 | FU 3.0 | FU 3.1 |
| 3 | | | | | | | | | | |
| 4 | 3 | [-, +, *] | [-, +, *] | [+] | [+] | [-, +, *] | [-, +, *] | [+] | [-, +, *] | [-, +, *] |
| 5 | 2 | [-, +, *] | [-, +] | | [+] | [-, +, *] | [-, +] | | [-, +, *] | [-, +, *] |
| 6 | 2 | [-, +, *] | | | [+, *] | [-, +, *] | | | [-, +, *] | [-, +, *] |
| 7 | 2 | [-, +, *] | | | [-, +, *] | [-, +, *] | | | [-, +, *] | |
| 8 | 2 | [-, +, *] | | | [-, +, *] | [-, +, *] | | | [-, +, *] | |
| 9 | 2 | [-, +, *] | | | [-, +, *] | [-, +, *] | | | [-, +, *] | |
| 10 | | | | | | | | | | |
| 11 | | | | | | | | | | |
| 12 | | | | | | | | | | |
| 13 | | | | | | | | | | |

EQUAKE2

| | | Cluster 0 | | Cluster 1 | | Cluster 2 | | | | Cluster 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 | FU 3.0 | FU 3.1 | FU 3.2 | FU 3.3 |
| 4 | 0 | [*] | [+, *] | [+, *] | [*] | [+, *] | [*] | [+, *] | [*] | [+, *] | [+, *] | [*] | [*] |
| 5 | 0 | [*] | [+] | [+] | [*] | [+] | [*] | [*] | [+] | [*] | [+] | [+] | [*] |
| 6 | 0 | [+, *] | | [+, *] | [+, *] | [*] | [+] | [+, *] | | [+, *] | | | |
| 7 | 0 | [+, *] | | [+, *] | | [*] | [+] | [+, *] | | [+, *] | [+] | [*] | |
| 8 | 1 | [*] | [+] | [+, *] | | [+, *] | | | | [+, *] | [+] | | |
| 9 | 1 | [+, *] | | [+, *] | | [+, *] | | | | [+] | [+, *] | | |
| 10 | 2 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |
| 11 | 2 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |
| 12 | 2 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |
| 13 | 2 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |
| 14 | 2 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |

EQUAKE1

| | | Cluster 0 | | Cluster 1 | | Cluster 2 | | | | Cluster 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 | FU 3.0 | FU 3.1 | FU 3.2 | FU 3.3 |
| 4 | 0 | [*] | [+] | [+] | [*] | [+] | [*] | [+] | [*] | [+] | [+] | [*] | [*] |
| 5 | 0 | [+] | [*] | [+] | [*] | [+] | [+, *] | [*] | | [*] | [+, *] | [+] | |
| 6 | 0 | [+, *] | | [+, *] | | [*] | [+] | [+, *] | | [+, *] | [+] | [*] | |
| 7 | 1 | [+, *] | | [+, *] | [+] | [+, *] | | | | [+, *] | [+] | | |
| 8 | 1 | [+, *] | | [+, *] | [+] | [+, *] | | | | [+, *] | [+] | | |
| 9 | 1 | [+, *] | | [+, *] | | [+, *] | | | | [+] | [+, *] | | |
| 10 | 1 | [+, *] | | [+, *] | | [+, *] | | | | [+] | [+, *] | | |
| 11 | 1 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |
| 12 | 1 | [+, *] | | [+, *] | | [+, *] | | | | [+, *] | | | |
| 13 | | | | | | | | | | | | | |
| 14 | | | | | | | | | | | | | |

SWIM1

| | | Cluster 0 | Cluster 1 | | Cluster 2 | Cluster 3 | |
|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 1.1 | FU 2.0 | FU 3.0 | FU 3.1 |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | 1 | [+, *] | [+, *] | [-, +] | [-, *] | [+, *] | [-] |
| 7 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | [-, +, *] | |
| 8 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | [-, +, *] | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | | | | | | | |
| 14 | | | | | | | |

# Ergebnisse der DSE mit DESCOMP für fünf Cluster

DCT-LEE

| | | Cluster 0 | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 2.0 | FU 3.0 | FU 3.0 |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |
| 16 | | | | | | |
| 17 | 2 | [-, +, *] | [+] | [-, +, *] | [-, +, *] | [-, +, *] |
| 18 | | | | | | |
| 19 | | | | | | |

DCT-DIF

| | | Cluster 0 | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
|---|---|---|---|---|---|---|
| L | C | FU 0.1 | FU 3.1 | FU 2.0 | FU 3.0 | FU 3.0 |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | 3 | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] |
| 13 | 3 | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] |
| 14 | 3 | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] |
| 15 | 3 | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] |
| 16 | | | | | | |
| 17 | | | | | | |

DCT-DIT

| | | Cluster 0 | Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 |
|---|---|---|---|---|---|---|
| L | C | FU 0.1 | FU 3.1 | FU 2.0 | FU 3.0 | FU 3.0 |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | | | | | | |
| 15 | | | | | | |
| 16 | | | | | | |
| 17 | | | | | | |
| 18 | 2 | [-, +, *] | [*] | [-, +, *] | [-, +, *] | [-, +, *] |

FFT

| | | Cluster 0 | Cluster 1 | | Cluster 2 | | Cluster 3 | | Cluster 4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 | FU 3.0 | FU 3.1 |
| 4 | | | | | | | | | | |
| 5 | | | | | | | | | | |
| 6 | 3 | [-, +, *] | [+, *] | [-, *] | [-, +, *] | [-] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] |
| 7 | 2 | [-, +, *] | [+, *] | [-] | [-, +, *] | [-] | [-, *] | [-, +] | [-, *] | [-, +, *] |
| 8 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, *] | [-, +] | [-, +, *] | [+, *] |
| 9 | 2 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | [-] | [-, *] | [+, *] |
| 10 | | | | | | | | | | |
| 11 | | | | | | | | | | |
| 12 | | | | | | | | | | |
| 13 | | | | | | | | | | |
| 14 | | | | | | | | | | |

AMMP

| | | Cluster 0 | Cluster 1 | | Cluster 2 | | Cluster 3 | | Cluster 4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 | FU 3.0 | FU 3.1 |
| 3 | | | | | | | | | | |
| 4 | 3 | [+] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [-, +, *] | [+] | [+] |
| 5 | 2 | [+] | [-, *] | [-, +, *] | [-, *] | [-, +] | [-, +, *] | [-, +, *] | [+] | [+] |
| 6 | 3 | [-, +, *] | [-, +, *] | | [-, +, *] | | [-, +, *] | | [-, +, *] | |
| 7 | | | | | | | | | | |
| 8 | | | | | | | | | | |
| 9 | | | | | | | | | | |
| 10 | | | | | | | | | | |
| 11 | | | | | | | | | | |
| 12 | | | | | | | | | | |
| 13 | | | | | | | | | | |

EQUAKE2

| | | Cluster 0 | | Cluster 1 | | Cluster 2 | | Cluster 3 | | Cluster 4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 0.1 | FU 1.0 | FU 1.1 | FU 2.0 | FU 2.1 | FU 2.2 | FU 2.3 | FU 3.0 | FU 3.1 | FU 3.2 | FU 3.3 |
| 4 | 0 | [*] | [+, *] | [+, *] | [*] | [+, *] | [*] | [+, *] | [*] | [+, *] | [+, *] | [*] | [*] |
| 5 | 1 | [*] | [+] | [+, *] | [+, *] | [+] | [*] | [*] | [+] | [+, *] | [*] | | |
| 6 | 1 | [+, *] | | [+, *] | | [+] | [*] | [+, *] | | [+, *] | [+, *] | | |
| 7 | 1 | [+, *] | | [+, *] | | [+] | [+, *] | [+, *] | | [+, *] | | | |
| 8 | 1 | [+, *] | | [+, *] | | [+, *] | | [+, *] | | [+, *] | | | |
| 9 | 1 | [+, *] | | [+, *] | | [+, *] | | [+, *] | | [+, *] | | | |
| 10 | | | | | | | | | | | | | |
| 11 | | | | | | | | | | | | | |
| 12 | | | | | | | | | | | | | |
| 13 | | | | | | | | | | | | | |
| 14 | | | | | | | | | | | | | |

EQUAKE1

| | | Cluster 0 | Cluster 1 | | Cluster 2 | Cluster 3 | Cluster 4 |
|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 1.1 | FU 2.0 | FU 3.0 | FU 4.0 |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | 1 | [+, *] | [+, *] | [+] | [+, *] | [+, *] | [+, *] |
| 9 | 1 | [+, *] | [+, *] | | [+, *] | [+, *] | [+, *] |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | | | | | | | |
| 14 | | | | | | | |

SWIM1

| | | Cluster 0 | Cluster 1 | | Cluster 2 | Cluster 3 | Cluster 4 |
|---|---|---|---|---|---|---|---|
| L | C | FU 0.0 | FU 1.0 | FU 1.1 | FU 2.0 | FU 3.0 | FU 4.0 |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | [+, *] | [+, *] | [+, *] | [+, *] | [+, *] | [+, *] |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | | | | | | | |
| 14 | | | | | | | |

# Anhang C

## Ergebnisse der globalen DSE

| Cluster 1 | Cluster 2 | | Cluster 3 | | externer Cluster | |
|---|---|---|---|---|---|---|
| **DCT-DIT, Länge 12** | | | | | | |
| 5: + | 2: + | 1: + | | | | |
| 4: + | 9: - | 6: + | 3: + | | copy | copy |
| | 16: - | 8: + | 13: + | 7: + | copy | copy |
| 15: + | 24: - | 12: + | 17: + | 10: - | | copy |
| 11: + | 18: - | 14: + | | 25: - | | copy |
| 19: + | 26: + | 20: + | 27: + | 21: + | | copy |
| 23: - | 22: - | | 32: * | 31: - | copy | copy |
| 30: * | | | 28: + | 35: * | copy | copy |
| 36: - | 37: + | 29: * | 43: - | 40: - | copy | copy |
| 33: * | | 44: * | 34: + | 46: * | copy | copy |
| 38: + | | 48: * | 41: + | 39: * | | |
| 45: * | | | 42: * | 37: * | | |
| **ARF, Länge 11** | | | | | | |
| 8: * | | | 7: * | 12: * | | |
| 1: * | | | 4: * | 11: * | copy | |
| 2: * | | | 13: + | 9: + | copy | |
| 5: * | | | 14: + | 10: + | copy | |
| | | | 16: * | 15: * | | |
| | | | 19: + | 18: * | copy | |
| | | | 17: * | 22: * | copy | |
| 23: * | | | 20: + | 3: + | | |
| 6: + | | | 24: * | 21: * | copy | |
| | | | 25: + | 26: + | copy | |
| | | | 27: + | 28: + | | |
| | | | | | | |
| **FFT, Länge 9** | | | | | | |
| 8: * | | 10: * | 4: * | 9: * | | |
| 11: * | | 2: * | | 3: * | copy | |
| 24: + | | 6: * | 7: * | 13: * | copy | |
| 25: + | 5: - | 1: * | | 12: * | copy | |
| 34: - | 21: - | 0: + | 26: + | 22: - | | |
| 37: + | 33: - | 20: - | 38: + | 23: - | copy | |
| | 43: - | 30: - | 32: + | 36: - | copy | |
| 47: - | 42: - | 31: + | | 35: - | copy | |
| 46: - | 40: - | 41: + | 44: - | 45: - | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| ALAP | As Late As Possible |
| ALU | Arithmetic Logic Unit |
| ASAP | As Soon As Possible |
| ASIC | Application Specific Integrated Circuit |
| CDFG | Control Dataflow Graph |
| DESCOMP | Design By Compilation |
| DFG | Dataflow Graph |
| DSE | Design-Space-Exploration |
| DSP | Digitaler Signalprozessor |
| EDGE | Explicit Dataflow Graph Execution |
| FPGA | Field Programmable Gate Array |
| FU | Functional Unit |
| HLS | High-Level-Systhese |
| MAC | Multiply And Accumulate |
| PICO | Program In Computer Out |
| SoC | System on Chip |
| TTA | Transport Triggered Architecture |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |
| VLIW | Very Long Instruction Word |

# Symbolverzeichnis

| Symbol | Beschreibung |
|---|---|
| $A$ | Reihenfolgebedingung im Basisblock ... 21 |
| $\alpha$ | Ablaufplan ... 22 |
| $availCol$ | verfügbare Farben für einen Interferenzgraphknoten ... 91 |
| $\mathcal{B}$ | Menge der zeitkritischen Basisblöcke ... 111 |
| $bbExeT$ | Ausführungszeit eines Basisblocks in Abhängigkeit von einer Portkonfiguration ... 115 |
| $BI$ | Menge der breitesten Instruktionen ... 97 |
| $\mathcal{C}$ | Menge der Prozessorcluster ... 19 |
| $\chi$ | Clusterung ... 23 |
| $cConf$ | Clusterkonfiguration ... 124 |
| $cCost$ | Kostenfunktion der Clusterung ... 105 |
| $cf$ | Clusterallokation ... 19 |
| $ClusterCost$ | Hardwarekosten eines Prozessorclusters ... 20 |
| $col$ | Knotenfärbung des entfalteten Interferenzgraphen ... 86 |
| $conf_c$ | Typkonfiguration in einem Cluster $c$ ... 123 |
| $deg$ | Kantengrad eines Knotens ... 55 |
| $D$ | Datenabhängigkeiten im Basisblock ... 21 |
| $DPCost$ | Datenpfadkosten ... 19 |
| $E$ | Abhängigkeiten im Basisblock ... 21 |
| $eet$ | früheste Ausführungszeit eines Knotens ... 54 |
| $ePWidth$ | Anzahl externer Lese- und Schreibports ... 27 |
| $erPWidth$ | Anzahl externer Leseports ... 26 |
| $ewPWidth$ | Anzahl externer Schreibports ... 27 |
| $exe$ | Ausführungshäufigkeit eines Basisblocks ... 111 |
| $\mathcal{F}$ | Menge der funktionalen Einheiten im Prozessor ... 19 |
| $\phi$ | Bindung ... 25 |
| $fm$ | maximal mögliche Taktfrequenz ... 16 |
| $fr$ | reale Taktfrequenz ... 120 |
| $ft$ | Ressourcenallokation ... 19 |

| | | |
|---|---|---|
| $FUCIWidth$ | Instruktionsbreite in einem Cluster | 26 |
| $FUCost$ | Hardwarekosten einer FU | 20 |
| $FUCWidth$ | Clusterbreite | 26 |
| $FUDelay$ | maximale Verzögerung in einer FU | 17 |
| $FUV$ | Menge der FU-Vektoren | 118 |
| $FUWidth$ | Ablaufplanbreite | 26 |
| $iPCWidth$ | interne Registerbankgröße | 26 |
| $\mathcal{K}_{(ip,\ rp,\ wp)}$ | Menge der Zielprogramme, die sich mit der Portkonfiguration ($ip$, $rp$, $wp$) ausführen lassen | 117 |
| $lat$ | Latenzzeit eines Operationstyps | 13 |
| $let$ | spätester Ausführungszeitpunkt | 54 |
| $L_g$ | Gatteranzahl | 15 |
| $lp$ | Längster Pfad in einem Basisblock | 23 |
| $maxDelay$ | maximale Verzögerung im Prozessor | 16 |
| $mC$ | maximale Clusteranzahl der Zielprogramme | 113 |
| $mC_b$ | maximale Clusteranzahl der Zielprogramme zum Basisblock $b$ | 113 |
| $mConf$ | maximale Clusterkonfiguration | 124 |
| $mL$ | Länge des kürzesten Ablaufplans zum Basisblock $b$, der mit einer gegebenen Portkonfiguration abgearbeitet werden kann | 115 |
| $mob$ | Mobilität einer Operation | 54 |
| $MP$ | Menge von Portkonfigurationen | 114 |
| $mPWidth$ | maximale Portanzahl eines Zielprogramms | 118 |
| $mtL$ | Länge des kürzesten Ablaufplans zum Basisblock $b$, der mit einer gegebenen Clusterkonfiguration abgearbeitet werden kann | 125 |
| $\mathcal{O}$ | Menge der Operationstypen | 8 |
| $\mathcal{O}_E$ | Menge der externen Operationstypen | 18 |
| $\mathcal{O}_I$ | Menge der internen Operationstypen | 18 |
| $opCost$ | Hardwarekosten eines Operators | 8 |
| $P$ | statischer und dynamischer Stromverbrauch | 121 |
| $\mathcal{P}$ | Menge der Programmfragmente | 19 |

| | |
|---|---|
| $TLoad$ | Kostenfunktion für Datenpfadkosten bei der Planung..... 68 |
| $type$ | Typ einer Operation..... 8 |
| $\mathcal{TZ}$ | Zielprogramme zur minimalen Clusterkonfiguration ..... 127 |
| $usedCol$ | für einen Operationstyp bereits verwendete Farben ..... 91 |
| $V_{dd}$ | Versorgungsspannung..... 15 |
| $VLIWCost$ | Hardwarekosten eines Prozessors..... 19 |
| $wPC$ | Anzahl der benötigten externer Schreibports in einer Registerbank ..... 27 |
| $\psi$ | Clusterabbildung ..... 124 |
| $z$ | Zielfunktion für die Planung ..... 64 |
| $\mathcal{Z}$ | Zielprogramme zu einer Clusterkonfiguration..... 126 |
| $\mathcal{ZP}$ | Menge aller Zielprogramme ..... 113 |
| $\mathcal{ZP}_b$ | Menge der Zielprogramme zum Basisblock $b$..... 113 |
| $\mathcal{ZP}_b^C$ | Menge der Zielprogramme zum Basisblock $b$ mit $C$ vielen Clustern ..... 112 |

# Literaturverzeichnis


[1] *TMS320C6000: A High Performance DSP Platform*. Texas Instruments Inc., 1998. Online verfügbar unter: http://www.ti.com/. Letzter Zugriff: 27.3.2006.

[2] *TriMedia TM1100 Data Book*. Philips Electronics North America Corporation, 1999. Online verfügbar unter: http://www.semiconductors.philips.com/acrobat/other/tm1100.pdf. Letzter Zugriff: 27.3.2006.

[3] *ADSP-21061 EZ-KIT Lite Reference Manual*. Analog Devices, 2001. Online verfügbar unter: http://www.analog.com/. Letzter Zugriff: 27.3.2006.

[4] A.Capitanio, N.Dutt und Alexandru Nicolau: *Partitioned register files for VLIWs: A preliminary analysis of tradeoffs*. Proc. of the 25th Annual Int. Symp. on Microarchitechture (MICRO'92), S. 292-300, 1992.

[5] A.Hashimoto J.Stevens: *Wire routing by optimizing channel assignment within large apertures*. Proc. of the 8th Design Automation Workshop, S. 155-163, 1971.

[6] Achim Nohl, Gunnar Braun, Weihua Sheng, Jian Jiang Ceng, Manuel Hohenhauer, Hanno Scharwächter, Rainer Leupers und Heinrich Meyr: *A Novell Approach for Flexible and Consistent ADL-driven ASIP Design*. Proc. of the 41st Design Automation Conference (DAC'04), S. 717-722, 2004.

[7] Alexandru Andrei, Marcus Schmitz, Petru Eles, Zebo Peng und Bashir M.Al Hashimi: *Simultaneous Communication and Processor Voltage Scaling for Dynamic and Leakage Energy Reduction in Time-Constrained Systems*. Proc. of the International Conference on Computer Aided Design (ICCAD'04), S. 362-369, 2004.

[8] Alfred V.Aho, Sethi Ravi and Jeffrey D.Ullman: *Compilers - principles, techniques, and tools*. Addison-Wesley, 1996.

[9] Alice C.Parker, Jorge Pizarro und Mitch Mlinar: *MAHA: A Program for Datapath Synthesis*. Proc. of the 23rd Design Automation Conference (DAC'86), S. 461-466, 1986.

[10] Andreas Hoffmann, Tim Kogel, Achim Nohl, Gunnar Braun, Oliver Schliebusch, Oliver Wahlen, Andreas Wieferink und Heinrich Meyr: *A Novel Methodology for the Design of Application-Specific Instruction Set Processors (ASIPs) Using a Machine Description Language*. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 20(11), S. 1338-1354, 2001.

[11] Andrew Mihal, Chidamber Kulkarni, Matthew Moskewicz, Mel Tsai, Niraj Shah, Scott Weber, Yujia Jin und Kurt Keutzer: *Developing Architectural Platforms: A Disciplined Approach*. IEEE Design & Test of Computers, 19(6), S. 6-15, 2002.

[12] Andy Heinig: *Entwicklung einer Clusterungstechnik für Datenflussgraphen zur VLIW-Prozessor Synthese*. Diplomarbeit, BTU Cottbus, Institut für Informatik, 2005.

[13] Andy Heinig und Mario Schölzel: *Zeitbeschränkte Clusterung zur Design-Space-Exploration geclusterter VLIW-Prozessoren*. 9. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, S. 319-328, 2006.

[14] C.Lee, M.Potkonjak und W.H.Mangione-Smith: *MediaBench: A tool for evaluating and synthesizing multimedia and communications systems*. Proc. of the 30th Annual International Symposium on Microarchitecture (MICRO'97), S. 330-335, 1997.

[15] Cheng-Tsung Hwang, Yu-Chin Hsu und Youn-Long Lin: *Optimum and Heuristic Data Path Scheduling Under Resource Constraints*. Proc. of the 27th Design Automation Conference (DAC'90), S. 65-70, 1990.

[16] Chi-Min Chu, M.Potkonjak, M.Thaler und J.Rabaey: *HYPER : An Interactive Synthesis Environment for High Performance Real Time Applications*. Proc. of the International Conference on Computer Design (ICCD'89), S. 432-435, 1989.

[17] Chia-Jeng Tseng und Daniel P.Siewiorek: *Facet: A Procedure for the Automated Synthesis of Digital Systems*. Proc. of the 20th Design Automation Conference (DAC'83), S. 490-496, 1983.

[18] Chris H.Kim und Kaushik Roy: *Dynamic Vth Scaling Scheme for Active Leakage Power Reduction*. Proc. of the Design, Automation And Test in Europe (DATE'02), S. 163-167, 2002.

[19] Christian Köhler: *Erweiterung des Tools DESCOMP zur Datenpfadsynthese für VLIW-Architekturen um die Möglichkeit der Generierung einer geclusterten Architektur (LAR)*. Studienarbeit, BTU Cottbus, 2005.

[20] Clifford Liem, Francois Breant, Sarveta Jadhav, Ray O'Farrell, Ray Ryan und Oz Levia: *Embedded Tools for a Configurable and Customizable DSP Architecture*. IEEE Design & Test of Computers, 19(6), S. 27-35, 2002.

[21] David C.Ku und Giovanni De Micheli: *High Level Synthesis and Optimization Strategies in Hercules and Hebe*. Proc. of the European ASIC Conference, S. 124-129, 1990.

[22] David C.Ku und Giovanni De Micheli: *Relative Scheduling Under Timing Constraints*. Proc. of the 27th Design Automaion Conference (DAC'90), S. 59-64, 1990.

[23] David Goodwin und Darin Petkov: *Automatic Generation of Application Specific Processors*. Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'03), S. 137-147, 2003.

[24] David I.August, Daniel A.Connors, Scott A.Mahlke, John W.Sias, Kevin M.Crozier, Ben-Chung Cheng, Patrick R.Eaton, Qudus B.Olaniran und Wen-mei W.Hwu: *Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture*. Proc. of the 25th Annual International Symposium on Computer Architecture (ISCA'98), S. 227-237, 1998.

[25] Doug C.Burger, Stephen W.Keckler, Kathrin S.McKinley, Mike Dahlin, Lizy K.John, Calvin Lin, Charles R.Moore, James Burrill, Robert G.McDonald und William Yoder: *Scaling to the End of Silicon with EDGE Architectures*. IEEE Innovative Technology for Computing Professionals, 37(7), S. 44-55, 2004.

[26] E.J.D.Pol, B.J.M.Aarts, J.T.J.van Eijndhoven, P.Struik und F.W.Sijstermans: *TriMedia CPU64 Application Development Environment*. Proc. of the International Conference on Computer Design (ICCD'99), S. 593-598, 1999.

[27] E.Ofner, R.Forsyth, und A.Gierlinger: *GEPARD, ein parametrisierbarer DSP Kern für ASICs*. Technical Report, Austria Mikro Systeme International AG, 1998.

[28] Emre Özer, Sanjeev Banerjia und Thomas M.Conte: *Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures*. Proc. of 31th Annual Int. Symp. on Microarchitecture (MICRO'98), S. 308-315, 1998.

[29] Erik Nystrom und Alexandre E.Eichenberger: *Effective Cluster Assignment for Modulo Scheduling*. Proc. of 31th Annual Int. Symp. on Microarchitecture (MICRO'98), S. 103-114, 1998.

[30] G.J.Hekstra, G.D.La Hei, P.Bingley und F.W.Sijstermans: *TriMedia CPU64 design space exploration*. Proc. of the International Conference on Computer Design (ICCD'99), S. 599-606, 1999.

[31] Gert Slavenburg, S.Rathnam und H.Dijkstra: *The TriMedia TM-1 PCI VLIW Media Processor*. Hot Chips 8, 1996.

[32] Giovanni De Micheli: *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[33] Giovanni De Micheli, David C.Ku, Frédéric Mailhot und Thomas Truong: *The Olympus synthesis system*. IEEE Design and Test of Computers, 7(5), S. 37-53, 1990.

[34] Giuseppe Desoli: *Instruction assignment for clustered VLIW DSP compilers: A new approach*. Technical Report, HP Laboratories Cambridge, HPL-98-13, 1998.

[35] Greg Snider: *Spacewalker: Automated Design Space Exploration for Embedded Computer Systems*. Technical Report, HP Laboratories Palo Alto, HPL-2001-220, 2001.

[36] Gregory J.Chaitin: *Register allocation and spilling via graph coloring*. SIGPLAN Notices, S. 98-105, 1982.

[37] H.De Man, J.Rabaey, J.Vanhoof, Gert Goossens, P.Six und L.Claesen: *CATHEDRAL-II - A Computer-Aided Synthesis System for Digital Signal Processing VLSI Systems*. Computer-Aided Engineering Journal, S. 55-66, 1988.

[38] Heiko Hinkelmann, Thilo Pionteck, Oliver Kleine und Manfred Glesner: *Prozessorintegration und Speicheranbindung dynamisch rekonfigurierbarer Funktionseinheiten*. Proc. of the Workshop Dynamically Reconfigurable Systems - Self-Organization and Emergence, S. 45-51, 2005.

[39] Heinrich Theodor Vierhaus, Paul G.Plöger, und Jörg Wilberg: *Hardware-Software Codesign for Embedded Systems: A New Technology or just a New Name?* Technical Report, Brandenburgische Technische Universität Cottbus, Informatik I-03/1997, 1997.

[40] Henk Corporaal: *Transport Triggered Architectures: Design and Evaluation*. Dissertation, Technische Universiteit Delft, 1995.

[41] Henk Corporaal und Hans J.M.Mulder: *MOVE: a framework for high-performance processor design*. Proc. of the 1991 ACM/IEEE conference on Supercomputing (SC'91), S. 692-701, 1991.

[42] J.Bhasker und Huan-Chih Lee: *An Optimizer For Hardware Synthesis*. IEEE Design and Test of Computers, 7(5), S. 20-36, 1990.

[43] J.Hoogerbrugge und Henk Corporaal: *Automatic Synthesis of Transport Triggered Processors*. First Annual Conference of the Advanced School for Computing and Imaging (ASCI'95), S. 79-88, 1995.

[44] J.Rabaey und M.Potkonjak: *Resource Driven Synthesis in the HYPER System*. Proc. of the Int. Symp. on Circuits and Systems (ISCAS'90), S. 2592-2595, 1990.

[45] J.T.J.van Eijndhoven, F.W.Sijstermans, K.A.Vissers, E.J.D.Pol und M.J.A.Tromp: *TriMedia CPU64 Architecture*. Proc. of the International Conference on Computer Design (ICCD'99), S. 586-592, 1999.

[46] Jesús Sánchez und Antonio Gonzáles: *Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture*. Procs. of the 33th Annual Int. Symp. on Microarchitecture (MICRO'00), S. 124-133, 2000.

[47] Johan Janssen und Henk Corporaal: *Partitioned Register Files for TTAs*. Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO'95), S. 303-312, 1995.

[48] John L.Hennessy and David A.Patterson: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[49] John R.Ellis: *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, Massachusettes, 1985.

[50] Jörg Wilberg: *Codesign for Real-Time Video Applications*. Kluwer Academic Publishers, 1997.

[51] Joseph A.Fisher: *Trace Scheduling: A Technique for global microcode compaction*. IEEE Transactions on Computers, S. 478-490, 1981.

[52] Jürgen Teich: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer, 1997.

[53] Kaivalya Dixit: *Performance SPECulations-Benchmark, Friend or Foe*. 7th International Symposium on High Performance Computer Architecture (HPCA'01), Invited Talk, 2001.

[54] Katja Winder: *Algorithmus zur Zerlegung eines Datenflussgraphen für eine geclusterte VLIW-Architektur nach dem Scheduling*. Studienarbeit, BTU Cottbus, 2005.

[55] M.Auguin, F.Boeri und C.Carriere: *Automatic exploration of VLIW processors architectures from a designer's experience based specification*. Proc. of the Int. Workshop on Hardware-Software Co-Design (CODES'94), S. 108-115, 1994.

[56] M.J.Bass und C.M.Christensen: *The Future of the Microprocessor Business*. IEEE Spectrum, 39(4), S. 34-39, 2002.

[57] M.Potkonjak und J.Rabaey: *A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs*. Proc. of the Design Automation Conference (DAC'89), S. 7-12, 1989.

[58] M.Sami, D.Sciuto, C.Silvano, V.Zaccaria und R.Zafalon: *Exploiting Data Forwarding to Reduce the Power Budget of VLIW Embedded Processors*. Proc. of the Design, Automation and Test in Europe (DATE'01), S. 252-257, 2001.

[59] Mario Schölzel und Peter Bachmann: *DESCOMP: A New Design Space Exploration Approach*. Proc. of the 18th Int. Conference on Architecture of Computing Systems (ARCS'05), S. 178-192, 2005.

[60] Mario Schölzel, Peter Bachmann und Heinrich Theodor Vierhaus: *Application Specific Processor Design for Digital Signal Processing*. Proc. of the Int. Workshop Signal Processing'2004, S. 7-15, 2004.

[61] Mark R.Hartoog, James A.Rowson, Prakash D.Reddy, Soumya Desai, Douglas D.Dunlop, Edwin A.Harcourt und Neeti Khullar: *Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign*. Proc. of the 34th Design Automation Conference (DAC'97), S. 303-306, 1997.

[62] Matthias Gries, Scott Weber, und Christopher Brooks: *The Mescal Architecture Development System (Tipi) Tutorial*. Technical Report, University of California at Berkley, 2002.

[63] Matthias H.Weiss, Dirk Fimmel, Renate Merker und Gerhard P.Fettweis: *Designing Performance Enhanced Digital Signal Processing Using Loop Transformation*. PACT'98, S. 90-94, 1998.

[64] Matthias H.Weiss, U.Walther und Gerhard P.Fettweis: *A Structural Approach for Designing Performance Enhanced Signal Processors: A 1-MIPS GSM Fullrate Vocoder Case Study*. Proc. of the Int. Conference on Acoustics, Speech, and Signal Processing (ICASSP'97), S. 4085-4088, 1997.

[65] Michael L.Chu, Kevin C.Fan und Scott A.Mahlke: *Region-based Hierarchical Operation Partitioning for Multicluster Processors*. Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03), S. 300-311, 2003.

[66] Michael R.Garey and David S.Johnson: *Computers and Intractability*. Bell Telephone Laboratories, 1979.

[67] Michael Vogel: *Ressourcenbeschränktes Scheduling unter Verwendung der Architektursynthesealgorithmen von DESCOMP*. Studienarbeit, BTU Cottbus, 2005.

[68] Mika Kuulusa, Jari Nurmi, Janne Takala, Pasi Ojala und Henrik Herranen: *A Flexible DSP Core for Embedded Systems*. IEEE Design & Test of Computers, S. 60-68, 1997.

[69] Nikhil Bansal, Sumit Gupta, Nikil Dutt und Alexandru Nicolau: *Analysis of the Performance of Coarse-Grain Reconfigurable Architectures with Different Processing Element Configurations*. Workshop on Application Specific Processors (WASP'03), 2003.

[70] P.G.Paulin und J.P.Knight: *Force-directed scheduling in automatic data path synthesis*. Proc. of the 24th Design Automation Conference (DAC'87), S. 195-202, 1987.

[71] P.G.Paulin, J.P.Knight und E.F.Girczyc: *HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis*. Proc. of the 23rd Design Automation Conference (DAC'86), S. 587-594, 1986.

[72] P.Geoffrey Lowney, M. F. Stefan, Thomas J.Karzes, W.D.Lichtenstein, Robert P.Nix, John S.O'Donnel und John C.Ruttenberg: *The Multiflow Trace Scheduling Compiler*. The Journal of Supercomputing, 7(1-2), S. 51-142, 1993.

[73] Paolo Faraboschi, Geoffrey Brown, Joseph A.Fisher, Giuseppe Desoli und Fred Homewood: *Lx: a technology platform for customizable VLIW embedded processing*. Proc. of the 27th Annual International Symposium on Computer Architecture (ISCA'00), S. 203-213, 2000.

[74] Paul G.Plöger und Jörg Wilberg: *HW/SW co-synthesis using CASTLE*. Technical Report, German National Research Center for Computer Science (GMD), 1999.

[75] Pedro Trancoso: *Design Space Navigation for Neighboring Power-Performance Efficient Microprocessor Configurations*. Proc. of the 18th Int. Conference on Architecture of Computing Systems (ARCS'05), S. 193-206, 2005.

[76] Peter Marwedel: *The Mimola Design System: Tools for the Design of Digital Processors*. Proc. of the 21st Design Automation Conference (DAC'84), S. 587-593, 1984.

[77] Peter Marwedel and Gert Goossens: *Code Generation for Embedded Processors*. Cluwer Academic Publishers, Boston, 1995.

[78] Pierre G.Paulin, Clifford Liem, Trevor C.May, and Shailesh Sutarwala: *FlexWare: A Flexible Firmware Development Environment For Embedded Systems*. In: *Code Generation for Embedded Processors*. Kluwer Academic Publishers, S. 67-84, 1995.

[79] Pierre G.Paulin und John P.Knight: *Algorithms for High-Level Synthesis*. IEEE Design & Test of Computers, 6(6), S. 18-31, 1989.

[80] Preston Briggs: *Register Allocation via Graph Coloring*. Dissertation, Department of Computer Science, Rice University, 1992.

[81] R.Camposano: *From Behavior to Structure: High-Level Synthesis*. IEEE Design & Test of Computers, 7(5), S. 8-19, 1990.

[82] R.Ernst, J.Henkel und T.Benner: *HW / SW Cosynthesis for microcontrollers*. IEEE Design & Test of Computers, 10(4), S. 64-75, 1993.

[83] Rainer Leupers: *Instruction Scheduling for Clustered VLIW DSPs*. Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT'00), S. 291-300, 2000.

[84] Ricardo E.Gonzales: *XTensa: A Configurable and Extensible Processor*. IEEE Micro, 20(2), S. 60-70, 2000.

[85] S.Dutta, R.Jensen und A.Rieckmann: *VIPER: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems*. IEEE Design & Test of Computers, 18(5), S. 21-31, 2001.

[86] Satish Pillai und Margarida F.Jacome: *Compiler-Directed ILP Extraction for Clustered VLIW/EPIC Machines: Predication, Speculation and Modulo Scheduling*. Proc. of the Design, Automation and Test in Europe (DATE'03), S. 10422-10427, 2003.

[87] Scott Rixner, William J.Dally, Brucek Khailany, Peter Mattson, Ujval J.Kapasi und John D.Owens: *Register Organization for Media Processing*. Proc. of the 6th Int. Symp. on High-Performance Computer Architecture (HPCA'00), S. 375-386, 2000.

[88] Sebastian Scholz: *Datenpfadsynthese mit Simulated Annealing*. Studienarbeit, BTU Cottbus, 2005.

[89] Sebastian Scholz und Mario Schölzel: *Design-Space-Exploration Using Simulated Annealing*. Proc. of the Int. Workshop Signal Processing'2005, S. 165-170, 2005.

[90] Shail Aditya, B.Ramakrishna Rau und Vinod Kathail: *Automatic Architectural Synthesis of VLIW and EPIC Processors*. Proc. of the 12th International Symposium on System Synthesis (ISSS'99), S. 107-113, 1999.

[91] Shekhar Borkar: *Design Challanges of Technology Scaling*. IEEE Micro, 19(4), S. 23-29, 1999.

[92] Srinivas Devadas und Richard Newton: *Algorithms for Hardware Allocation in Data Path Synthesis*. IEEE Transactions on Computer-Aided Design, 8(7), S. 768-781, 1989.

[93] Srinivas Devadas und Silviana Hanno: *Instruction Selection, resource allocation and scheduling in the AVIV retargetable code generator*. Proc. of the Design Automation Conference (DAC'98), S. 510-515, 1998.

[94] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic und Heinrich Meyr: *LISA-Machine Description Language for Cycle-Accurate Models of Programmable DSP-Architectures*. Proc. of the Design Automation Conference (DAC'99), S. 933-938, 1999.

[95] Steffen Rülke: *EASY - Ein Werkzeug zur Unterstützung der automatischen High-Level-Synthese des Datenteils digitaler Systeme*. Dissertation, Akademie der Wissenschaften der DDR, Institut für Kybernetik und Informationsprozesse, 1990.

[96] Subbarao Palacharla, Norman P.Jouppi und J.E.Smith: *Complexity-Effective Superscalar Processors*. Proc. of the 24th Annual International Symposium on Computer Architecture (ISCA'97), S. 206-218, 1997.

[97] Thomas D.Burd und Robert W.Broderson: *Energy Efficient CMOS Microprocessor Design*. Proc. of the 28th Hawaii International Conference on System Sciences (HICSS'95), S. 288-297, 1995.

[98] Vikas Agarwal, H.S.Murukkathampoondi, Stephen W.Keckler und Doug C.Burger: *Clock rate versus IPC: The end of the road for conventional microarchitectures*. Proc. of the 27th Annual International Symposium on Computer Architecture (ISCA'00), S. 248-259, 2000.

[99] Viktor S.Lapinskii: *Algorithms for Compiler-Assisted Design-Space-Exploration of Clustered VLIW ASIP Datapaths*. Dissertation, University of Texas at Austin, 2001.

[100] Viktor S.Lapinskii, Margarida F.Jacome und Gustavo A.de Veciana: *Exploring performance tradeoffs for clustered VLIW ASIPs*. Proc. of the International Conference on Computer Aided Design (ICCAD'00), S. 504-510, 2000.

[101] Viktor S.Lapinskii, Margarida F.Jacome und Gustavo A.de Veciana: *Cluster Assignment for High-Performance Embedded VLIW Processors*. ACM Transactions on Design Automation of Electronic Systems, 7(3), S. 430-454, 2002.

[102] Youn-Long Lin: *Recent Developments in High-Level Synthesis*. ACM Transactions on Design Automation of Electronic Systems, 2(1), S. 2-21, 1997.