Constraintbasierte Codegenerierung für eingebettete Prozessoren

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von
Steven Bashford

Dortmund
2000

Tag der mündlichen Prüfung:

Dekan/Dekanin:

Gutachter:

Vorwort

Die vorliegende Arbeit ist während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik XII der Universität Dortmund unter der Betreuung von Prof. Dr. Peter Marwedel entstanden. Ich möchte mich hier bei allen Personen bedanken, die direkt oder indirekt[1] an der Entstehung und Vollendung dieser Arbeit beigetragen haben.

Ich danke Herrn Prof. Dr. Marwedel für die Möglichkeit zur Umsetzung dieser Arbeit und für seine Ratschläge und Unterstützung, die er mir während der Entwicklung der Arbeit gegeben hat. Herrn Prof. Dr. Padawitz danke ich für die Übernahme des Koreferats und für die konstruktive Kritik an der schriftlichen Konzeption dieser Arbeit.

Ganz besonderer Dank gilt meinen Arbeitskollegen Markus Lorenz, Stefan Steinke und Birger Landwehr für anregsame, interessante Diskussionen und für deren stets offene und aufmerksame Art, meinen Problemen zuzuhören und mir mit guten Ratschlägen weiterzuhelfen; ich danke ihnen sehr für ein äußerst sorgfältiges Korrekturlesen und der konstruktiven Kritik an dieser Arbeit. Ich danke allen weiteren Kollegen am Lehrstuhl XII für die gute Arbeitsathmosphäre und Markus Lorenz, Stefan Steinke und Lars Wehmeyer für den starken Teamgeist, den sie in unsere Gruppe gebracht haben.

Ich danke Erna und Winfried Resch sowie Anette Hölscher für das überaus sorgfältige Korrekturlesen meiner Arbeit. Daniel Kästner und dem Lehrstuhl Compilerbau der Universität des Saarlands sei Dank für anregende Diskussionen und für einen guten Informationsaustausch.

Nicht zuletzt bin ich Gott dankbar für eine sehr liebe und geduldsame Frau und für zwei gesunde Söhne, die während der Entwicklung und Entstehung dieser Arbeit zur Welt kamen. Ich hoffe ihnen nur ein Bruchteil der Liebe wiedergeben zu können, die sie mir bei der Entstehung dieses Werkes entgegengebracht haben, obwohl sie dies von meiner Seite her oftmals haben missen müssen.

[1] Die Arbeit ist von der Deutschen Forschungsgemeinschaft (DFG) gefördert worden.

Für meine Frau Daphne und für meine Kinder Evan und Jonah.

Inhaltsverzeichnis

Abstrakt

Eingebettete Systeme gewinnen zunehmenden Einfluss in vielen Bereichen unseres alltäglichen Lebens, wie z.B. in der Telekommunikation, Fahrzeugelektronik, Medizintechnik und in der Unterhaltungselektronik. Diese Systeme unterliegen strengen Randbedingungen, wie z.B. Realzeitanforderungen und Energieverbrauch. Beim Entwurf eingebetteter Systeme spielt die Realisierung möglichst vieler Systemkomponenten durch sogenannte eingebettete Prozessoren eine große Rolle. Diese können für eine Vielzahl von Systemen wiederverwendet werden, wodurch ein teurer und äußerst zeitaufwändiger Entwurfs-, Test- und Fertigungsprozess von dedizierter Hardware entfallen kann. Die Entwicklung von Software ermöglicht wesentlich schnellere Designprozesse, und man gewinnt weiterhin ein hohes Maß an Flexibilität, da Designfehler noch in einer späten Entwurfsphase behoben werden können.

Bei der Entwicklung von Software besteht natürlich der Wunsch, moderne Hochsprachen einzusetzen. Das Problem, das hier auftritt, ist ein Mangel an guten Compilern, besonders im Bereich der digitalen Signalprozessoren. Traditionelle Compilertechniken sind nicht geeignet, die spezifischen Eigenschaften dieser Prozessoren effektiv auszunutzen. Die erforderliche Qualität des generierten Codes genügt bei weitem nicht den gestellten Randbedingungen der Systeme. Um trotzdem den Einsatz von Prozessoren zu ermöglichen, wird in der Entwicklung von Software häufig auf die Assemblerprogrammierung zurückgegriffen, was zu großen Nachteilen führt: Entwicklungszeiten und Phasen zum Testen und zur Fehlerkorrektur verlängern sich i.d.R. wesentlich, und die Wiederverwendung von Software ist bei einem Prozessorwechsel kaum noch möglich.

Ziel dieser Arbeit ist die Entwicklung neuer Compilertechniken für eingebettet Prozessoren, wobei der Schwerpunkt auf digitalen Signalprozessoren liegt, die hochgradig irreguläre Datenpfade mit eingeschränkter Parallelausführung auf Instruktionsebene besitzen. Ziel ist die Generierung einer sehr hohen Codequalität bzgl. der Ausführungsgeschwindigkeit und der Codegröße. Bei der Entwicklung neuer Techniken wird verstärkt auf die Integration von Teilphasen der Codegenerierung, im Sinne einer Phasenkopplung, hingezielt. Weiterhin spielt die Einbeziehung graphbasierter Techniken zur Instruktionsauswahl eine bedeutende Rolle. Um diesen Anforderungen gerecht zu werden, ist zur handhabbaren Umsetzung entsprechender Compilertechniken der Einsatz neuer Programmierungs- und Optimierungsmethoden unbedingt notwendig. In dieser Arbeit werden Techniken auf der Basis der Constraint-Logikprogrammierung (CLP) entworfen und realisiert. Es wird gezeigt, in welchem Maße der Einsatz von CLP in diesem Problembereich geeignet ist. Ein weiteres Ziel ist der Entwurf von Konzepten, die eine schnelle Adaption von Compilern an neue Prozessoren erlauben.

1. Einführung

Mikroelektronische Systeme gewinnen zunehmenden Einfluss in unserem alltäglichen Leben. Eine stark herausstechende Klasse elektronischer Systeme sind die *eingebetteten Systeme*[1]. Diese finden verstärkten Einsatz in immer mehr Bereichen, wie z.B. der Telekommunikation, Fahrzeugelektronik, Medizintechnik, Unterhaltungselektronik, in Fabriksteuerungen und Industrierobotern. Diese Einsatzbereiche lassen einen hohen Marktanteil erahnen, der stark im Wachstum ist und den PC-Markt voraussichtlich in den kommenden Jahren einholen wird.

Eingebettete Systeme unterliegen strengen Randbedingungen, insbesondere Realzeitanforderungen (bei ABS und Airbagsteuerungen sehr wichtig) und Energieverbrauch (z.B. bei Handys relevant) und die daraus resultiernden Kosten. Um diesen Anforderungen gerecht zu werden, ist streng an die Applikationen angepasste Hardware notwendig, die i.d.R. aus anwendungsspezifischen Hardwarekomponenten (*ASICs = application specific integrated circuits*) und programmierbaren Prozessoren (sogenannte eingebettete Prozessoren) bestehen. Beim Entwurf eingebetteter Systeme spielt die Realisierung möglichst vieler Systemkomponenten durch Prozessoren eine große Rolle. Prozessoren können für eine Vielzahl von Systemen wiederverwendet werden, wodurch ein teurer und äußerst zeitaufwendiger Entwurfs-, Test- und Fertigungsprozess von ASICs entfallen kann. Die Entwicklung von Software ermöglicht wesentlich schnellere Designprozesse und weiterhin gewinnt man ein hohes Maß an Flexibilität, da Designfehler noch in einer späten Entwurfsphase behoben werden können. Upgrades und gegebenenfalls sogar neue Applikationen eines Systems können einfach durch Reprogrammierung integriert werden. Dies sind alles wichtige Aspekte in Hinblick des hohen Konkurrenzdrucks der Hersteller.

GPPs, ASIPs, DSPs und Microcontroler

Je nach Anwendungsbereich und Randbedingen kommen unterschiedliche Klassen eingebetteter Prozessoren zum Einsatz: *allgemeine Prozessoren* (*GPPs = general purpose processor*s), *Microcontroler, anwendungsspezifische Prozessoren* (*ASIPs = application specific instruction set processor*s), und *digitale Signalprozessoren* (*DSPs = digital signal processor*s). Insbesondere DSPs haben einen sehr hohen Anteil, da die digitale Signalverarbeitung eine wichtige Teilaufgabe eingebetteter System darstellt, die sehr effiziente und leistungsfähige Prozessoren erfordert. Um dieser Leistungsfähigkeit gerecht zu werden, besitzen DSPs hochspezialisierte, den Aufgaben der digitalen Signalverarbeitung angepasste Eigenschaften.

[1] Per Definition ist ein eingebettetes System ein "*informationsverarbeitendes System, das physikalische Größen aufnimmt, verarbeitet und beeinflusst*".

Bei der Entwicklung von Software besteht natürlich der Wunsch, moderne Hochsprachen einzusetzen, um schon beim Design von Systemen eine möglichst hohe Produktivität zu erzielen. Das Problem, das hier auftritt, ist ein Mangel an guten Compilern, besonders im Bereich der DSPs und ASIPs. Traditionelle Compilertechniken sind nicht geeignet, die spezifischen Eigenschaften von DSPs effektiv auszunutzen. Die erforderliche Qualität des generierten Codes genügt gerade bei Realzeitanforderungen bei weitem nicht den gestellten Anforderungen. Weiterhin ist die Größe des generierten Codes oft um ein Vielfaches höher als der vergleichbare handgeschriebene Code. Ein Overhead bedeutet hier auch einen unnötigen Mehraufwand an Hardware in Form von größeren Programmspeichern. Bei den hohen Fertigungsstückzahlen der Systeme bedeutet dies einen nicht mehr zu vernachlässigenden Kostenfaktor. Ein weiteres Problem, das der Verfügbarkeit guter Compiler im Wege steht, ist bedingt durch die Tatsache, dass die Entwicklung guter und neuer Compiler zu teuer ist. Compiler für ASIPs und DSPs werden im Vergleich zu Compilern für allgemeine Prozessoren in nur sehr wenigen Applikationen eingesetzt. Daher sind die Entwicklungkosten pro Applikation sehr hoch.

Der Einsatz eingebetteter Prozessoren ist nur möglich, wenn alle Randbedingungen des Systems durch die generierte Software erfüllt werden, was unter dem Einsatz von Compilern oft nicht mehr gewährleistet ist. Um trotzdem den Einsatz von Prozessoren zu ermöglichen, wird in der Entwicklung von Software häufig auf die Assemblerprogrammierung zurückgegriffen, was zu großen Nachteilen bei der Entwicklung der Systeme führt. Entwicklungszeiten und Phasen zum Testen und zur Fehlerkorrektur verlängern sich i.d.R. wesentlich. Weiterhin ist auch die Einarbeitung in die nicht einfache Assemblerprogrammierung der jeweiligen Prozessoren ein Faktor. Die Wiederverwendung von Software ist bei einem Prozessorwechsel kaum noch möglich. Der gesamte Fertigungsablauf vom Design bis hin zum Produkt ist nicht mehr automatisch realisierbar. Bei der zunehmenden Komplexität der Systeme ist der fehleranfällige Einsatz der Assemblerprogrammierung in Zukunft sicher nicht mehr akzeptabel.

Ziele der Arbeit

Ziel dieser Arbeit ist die Entwicklung neuer Compilertechniken für Festkomma-DSPs und ASIPs, wobei der Schwerpunkt auf Prozessoren mit hochgradig irregulären Datenpfaden[2] mit eingeschränkter Parallelausführung auf Instruktionsebene liegt. Dabei liegt der Fokus der Arbeit auf den Optimierungszielen *Codegröße* und *Ausführungsgeschwindigkeit*. Zur Entwicklung neuer Compilertechniken ist der Einsatz neuer Optimierungsmethoden unbedingt notwendig. In dieser Arbeit werden Compilertechniken auf der Basis der *Constraint-Logikprogrammierung* (*CLP*) entwickelt. Es wird gezeigt, in welchem Maße der Einsatz von CLP in diesem Problembereich geeignet ist. Ein weiteres Ziel ist die Entwicklung von Konzepten, die eine schnelle Entwicklung neuer Techniken bzw. eine schnelle Adaption bestehender Techniken an neue Prozessoren erlauben.

Bevor die Ziele im einzelnen genauer spezifiziert werden, wird zunächst eine detailliertere Einführung in den Themenbereich Compiler für eingebettete Prozessoren und

[2] Im Unterschied zu regulären Datenpfaden mit einem homogenen Registerfile und vorwiegend uneingeschränkten Verbindungsstrukturen zwischen Registerfiles und Funktionseinheiten (z.B. eine oder mehrere ALUs) weisen irreguläre Datenpfade verteilte Registerfiles, spezialisierte Funktionseinheiten und eine stark eingeschränkte Verbindungsstruktur zwischen Registerfiles und Funktionseinheiten auf. Siehe auch Kapitel 1.3.

den damit verbundenen Problemstellungen gegeben. Der weitere Inhalt dieses Kapitels gestaltet sich daher folgendermaßen:

- In Kapitel 1.1 wird die *Relevanz von Compilern* im Entwurfsprozess eingebetteter Systeme erläutert.
- In Kapitel 1.2 wird eine kurze Einführung in die *digitale Signalverarbeitung* gegeben.
- Die *Eigenschaften von DSPs*, welche eine effiziente Implementierung der Aufgaben digitaler Signalverarbeitung ermöglichen, werden in Kapitel 1.3 beschrieben.
- Kapitel 1.4 stellt die neuen *Anforderungen* an Compiler heraus, die durch die Eigenschaften von DSPs gegenüber GPPs gegeben sind.
- Eine differenziertere Betrachtung der Ziele dieser Arbeit und ein Überblick über den Rest der Arbeit werden in Kapitel 1.5 gegeben.

1.1. Compiler im Entwurfsprozess eingebetteter Systeme

Im Folgenden wird die Relevanz von Compilern im Entwurfsprozess eingebetteter Systeme verdeutlicht. In Abb 1.1 ist die Struktur eines eingebetteten Systems dargestellt. Neben spezifischen Hardwarekomponenten (ASICs) und Prozessoren (RISCs[3], ASIPs, DSPs) findet man auch Speicher (RAMs und / oder ROMs) sowie Hardware zur Kommunikation mit der Peripherie (E / A).

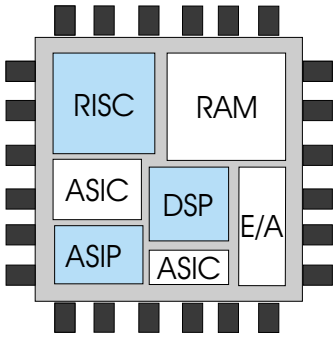


Abbildung 1.1.: Beispiele für eingebettete Systeme.

Hardware-/Softwarecodesign

Der Entwurf eingebetteter Systeme beinhaltet i.d.R. die simultane Entwicklung von Hardware und Software und wird als *Hardware-/Softwarecodesign* bezeichnet (siehe [96]). Der Entwurfsprozess beginnt mit einer Systemspezifikation in Form einer Verhaltensbeschreibung des gesamten Systems (siehe Abb. 1.2). Eines der ersten Probleme, das zu lösen ist, ist die Hardware- / Softwarepartitionierung. Dabei ist zu entscheiden, welche Systemkomponenten durch spezifische Hardware oder Prozessoren umzusetzen sind. Es werden überall dort ASICs eingesetzt, wo die Anforderungen der Systemkomponenten durch eine Realisierung mit Prozessoren nicht möglich ist. Der nächste Schritt besteht in der Synthese der Hardware und der Generierung von Maschinencode für die Zielprozessoren (letzteres idealerweise durch einen Compiler). Da die bisherige Entwurfstechnologie die Einhaltung aller Anforderungen eines Systems noch nicht garantieren kann, wird das generierte System auf der Basis von Simula-

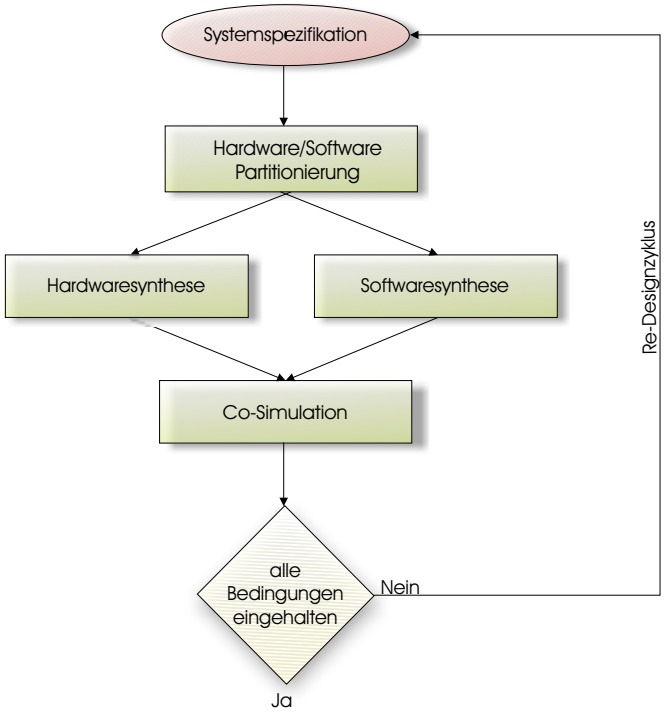


Abbildung 1.2.: Schema des Hardware-/Softwarecodesigns .

tionen getestet. Die Nichteinhaltung der Anforderungen führt zu einem erneuten Designzyklus.

Die Verfügbarkeit von Compilern spielt in folgender Hinsicht eine große Bedeutung innerhalb des Hardware/Software-Codesigns:

- Es ergeben sich wesentlich schnellere Entwicklungszeiten der Systeme durch die bekannten Vorteile des Einsatzes von Hochsprachen im Softwareentwurf:
  - Abstraktere und dadurch wesentlich effizientere Spezifikationsmöglichkeiten,
  - Wiederverwendbarkeit von Software,
  - schnellere Integration von neuen Entwurfsentscheidungen und Änderungen, und
  - wesentlich geringere Fehleranfälligkeit verglichen zur Assemblerprogrammierung.
- Die Automatisierung des Prozesses vom Entwurf hin zum Design ist nur unter Einsatz von Compilern möglich.
- Der Einsatz von Compilern erlaubt es, aus einer Menge alternativer Prozessoren auszuwählen, und den geeignetsten (z.B. kostengünstigsten bzw. energie-

[3] RISC=reduced instruction set computer.

sparendsten) Prozessor zu wählen. Der Einsatz der Assemblerprogrammierung ließe dies nicht mit vertretbarem Aufwand zu.

Die Anwendungsgebiete eingebetteter Systeme stellen höchste qualitative Anforderungen an den generierten Maschinencode und somit auch an die eingesetzten Compiler:

- Echtzeitanforderungen der Applikationen bedingen u.U. extrem hohe Ausführungsgeschwindigkeiten des generierten Codes. Eine unzureichende Performance kann dazu führen, dass entweder höher getaktete Prozessoren eingesetzt werden müssen, was i.d.R. in einer erhöhten Leistungsaufnahme resultiert, oder auf Assemblerprogrammierung zurückgegriffen wird.
- Restriktionen bezüglich des Speicherplatzes erfordern möglichst kompakten Maschinencode. Durch die Einsparung von Speicherplatz werden Fertigungskosten eingespart, da viele Systeme in sehr hohen Stückzahlen gefertigt werden.
- Portable Geräte, wie z.B. Handys, erfordern für eine längere Einsatzdauer eine möglichst geringe Leistungsaufnahme.

Die Forderungen von extrem hohen Ausführungsgeschwindigkeiten rufen sehr spezifische, an die Applikationen angepasste Prozessoren hervor. Die Ausnutzung der spezifischen Eigenschaften dieser Prozessoren ist zwingend, um die gewünschte Codequalität erreichen zu können (siehe auch Kapitel 1.3).

Ein weiterer wichtiger Aspekt im Entwurf eingebetteter Systeme ist die schnelle Verfügbarkeit von Compilern. Da die Entwicklung neuer und guter Compiler sehr zeitaufwendig und teuer ist, besteht ein Bedarf an effizienten Entwicklungsmethoden bzw. einer schnellen Adaption bestehender Compiler an neue Prozessoren. Ein Compiler wird als *retargierbar* bezeichnet, wenn es möglich ist, ihn zur Codegenerierung für verschiedene Prozessoren einzusetzen, ohne dass es erforderlich ist, große Teile des Quellcodes des Compilers zu reimplementieren. Es kann zwischen verschiedenen Stufen von Retargierbarkeit unterschieden werden (siehe auch [88, 69]):

retargierbar

portierbar: Die verwendeten Codegenerierungstechniken sind modular implementiert, so dass der Compiler durch Reimplementierung bestimmter Codegenerierungsprozeduren an andere Zielprozessoren angepasst werden kann.

generativ portierbar: Diese Stufe wird in der Literatur auch als Anwender-retargierbar, oder Entwickler-retargierbar bezeichnet. Der Compiler verwendet eine externe Maschinenbeschreibung, aus der Teile des Quellcodes generiert werden; andere Teile des Quellcodes müssen vom Entwickler selbst reimplementiert werden.

parametrisierbar/skalierbar: Hier ist der Compiler nur für eine bestimmte Klasse von Prozessoren anwendbar, die eine gemeinsame Basisarchitektur aufweisen. Die Retargierung erfolgt nur durch numerische Parameter. Dies umfasst z.B. die Größe der Registerfiles oder die Anzahl an funktionalen Einheiten.

maschinenunabhängig: Der Compiler verwendet eine externe Maschinenbeschreibung, die alle für die Codegenerierung und -optimierung benötigten Informationen über die Zielarchitektur enthält. Die Maschinenunabhängigkeit wird durch eine Verbindung von generischen und generativen Konzepten erreicht. Die Kernfunktionen der Codegenerierung müssen architekturunabhängig, also generisch, programmiert sein.

Maschinenunabhängiges retargieren stellt den Idealfall dar. Leider ist die Umsetzung eines Compilers, der alle speziellen Eigenschaften von Prozessoren integriert und darüber hinaus noch eine hohe Codequalität erzeugt, nicht realisierbar. Somit beinhalten existierende retargierbare Compiler i.d.R. Kompromisslösungen, die nicht alle Prozessoren mit gleich guter Codequalität versorgen können. Daher ist eine Struktur von Compilern erstrebenswert, die ein ausgewogeneres Verhältnis zwischen Retargierbarkeit und Erweiterbarkeit bietet.

1.2. Digitale Signalverarbeitung

Um hohe Ausführungsgeschwindigkeiten zu erreichen, haben DSPs und ASIPs den Applikationen stark angepasste Strukturen. Um darüber hinaus auch Randbedingungen, wie z.B. geringe Leistungsaufnahme und geringe Chipfläche, gerecht zu werden, weisen diese Prozessorklassen im Unterschied zu allgemeinen Prozessoren stark irreguläre Merkmale auf. Es werden zunächst Beispiele aus der digitalen Signalverarbeitung gezeigt und dann die wesentlichen Architekturmerkmale von DSPs erläutert, die auch die wesentlichen Merkmale von ASIPs umfassen.

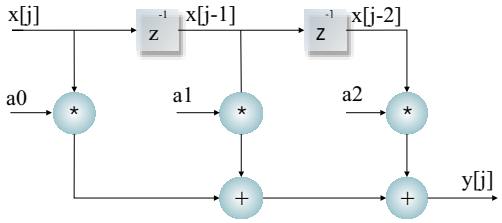


Abbildung 1.3.: FIR-Filter (FIR=*finite impulse response*).

Eine der am häufigsten vorkommenden Operationen in der digitalen Signalverarbeitung ist die Berechnung des inneren Produkts zweier Arrays: $\sum_{i=0}^{n} a[i]*b[i]$. Das Muster dieser Operation findet sich auch in einer der zentralen Komponenten der digitalen Signalverarbeitung, den digitalen Filtern. Diese filtern ein Eingangssignal, so dass nur ein bestimmtes Frequenzspektrum des Eingangssignals ausgegeben wird. In Abb. 1.3 ist der Signalflussgraph eines FIR-Filters dargestellt. Der Filter wird in einer unendlichen Schleife ausgeführt. Dabei wird in jedem Schritt *j* ein neues Eingangssignal *x[j]* eingelesen. Die Ausgabe des Filters ist:

$$y[j] = a_0 * x[j] + a_1 * x[j-1] + a_2 * x[j-2]$$

Die Komponenten z^{-1} bewirken dabei eine Verzögerung des Eingangssignals um einen Schritt, d.h. sie geben das jeweils im vorherigen Schritt in sie eingegangene Signal aus. Die Koeffizienten a_0, a_1 und a_2 spezifizieren das Übertragungsverhalten des Filters. Eine etwas allgemeinere Form von Filtern ist durch

$$y[j] = \sum_{i=0}^{n} x[j-i] * a[i]$$

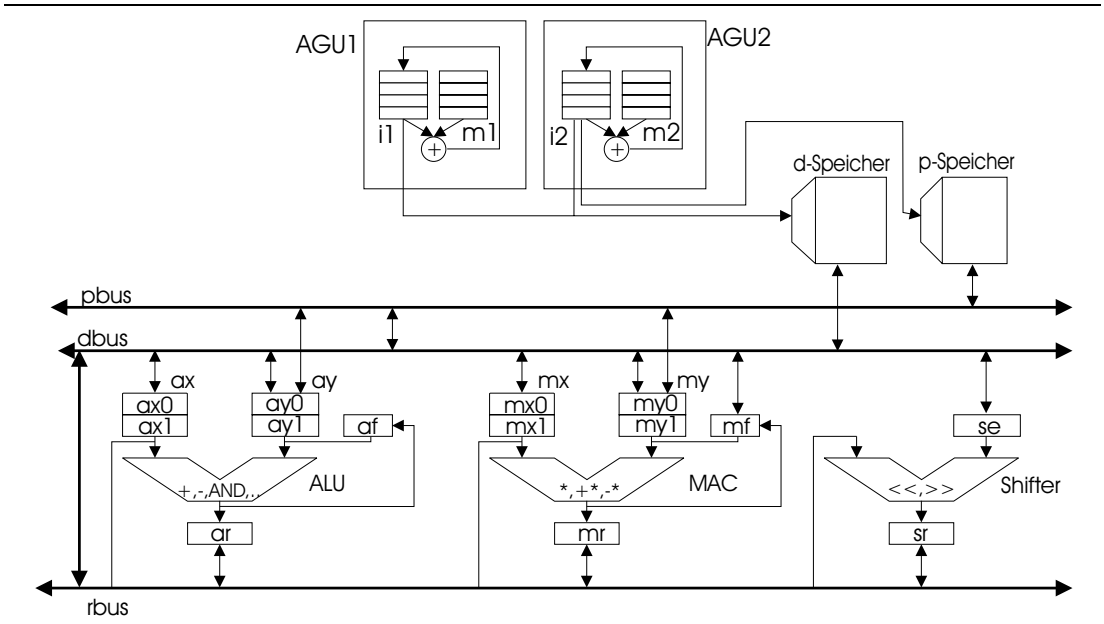


Abbildung 1.4.: Teile des Datenpfades der Analog Devices ADSP2100-Familie.

gegeben, in der nun auch die Koeffizienten durch ein Array referenziert werden. Anhand dieser Beispiele zeigen sich die wesentlichen Operationen von DSP-Algorithmen, die sich dann in der Architektur der Prozessoren widerspiegeln:

- Das Bilden der Summe von Produkten.
- Zugriff auf die letzten n Eingangssignale $x[j], \ldots, x[j-n+1]$.
- Zugriff auf zwei Array-Elemente aus dem Speicher.

Darüber hinaus gibt es Operationen zur Unterstützung in bildverarbeitenden Applikationen (z.B. Video), die zu VLIW-artigen Prozessorarchitekturen (*VLIW=very large instruction word*) führen. Diese werden in dieser Arbeit nicht weiter betrachtet.

1.3. Charakteristische Merkmale von DSPs

Die typischen Eigenschaften von DSPs werden am Beispiel der ADSP2100-Familie[4] von Analog Devices [30, 29] erläutert. In Abb. 1.4 ist ein Teil des Datenpfad dargestellt, der bei allen Prozessoren der ADSP2100-Familie gleich ist. Dieser besteht aus den drei Funktionseinheiten ALU, MAC[5], und Shifter. Jede Funktionseinheit besitzt

[4] DSPs dieser Familie besitzten gleiche Datenpfade, unterscheiden sich aber in Merkmalen der Peripherieschnittstellen und des On-Chip-Speichers und weisen weiterhin leichte Unterschiede im Befehlssatz auf. Diese Architektur wurde deswegen ausgewählt, weil sie sehr viele charakteristische Eigenschaften von DSPs umfasst.

[5] Die MAC-Funktionseinheit erlaubt Operationen der Form mr=mr+mx*my und mr=mr-mx*my, die als Multiply-ACcumulate (MAC) Operationen bezeichnet werden.

dedizierte Registerfiles für das Resultat einer Operation und deren Operanden. Alle Funktionseinheiten besitzen direkten Lesezugriff auf alle Akkumulatoren. Der Akkumulator mr ist unterteilt in die 16 Bit breiten Register $mr0$ und $mr1$ und das 8 Bit breite Register $mr2$. Das Resultat wird im gesamten Akkumulator mr abgelegt. $mr2$ dient der Handhabung zwischenzeitlicher Überläufe, und $mr0$ (Bits 0-15) und $mr1$ (Bits 16-32) beinhalten das 32 Bit breite Resultat einer Operation der MAC-Funktionseinheit: $mr0$ enthält Bit 0-15 und $mr1$ Bit 16-32 des Resultats der MAC-Operation. Die MAC-Funktionseinheit kann zum einen im ganzzahligen Modus und zum anderen im Festkommamodus betrieben werden. Bei einer ganzzahligen Multiplikation beinhaltet das $mr0$ Register das eigentliche 16 Bit breite ganzzahlige Resultat (Bit 0-15 des Resultats einer MAC-Operation). Im Falle des Festkommamodus befindet sich das relevante 16 Bit Festkommaresultat im $mr1$ Register (Bit 16-31 des Resultats einer MAC-Operation). Es existiert ein Feedbackregister mf, das nur 16 Bit breit ist und die jeweils höheren 16 Bit einer MAC-Operation aufnimmt.

Adressgenerierungseinheit (AGU)

Weiterhin existieren zwei *Adressgenerierungseinheiten (AGU = address generation unit),* welche eine indirekte Adressierung der beiden Speicherbänke (d-Speicher und p-Speicher) ermöglichen. Jede AGU besitzt ein Registerfile mit jeweils vier Indexregistern zur indirekten Adressierung der Speicherbänke ($i1$ und $i2$) und jeweils ein Registerfile mit jeweils vier Modify-Registern ($m1$ und $m2$). Letztere dienen dem Ablegen von Konstanten, mit denen die Indexregister inkrementiert bzw. dekrementiert werden können. Die Operationen der AGUs können parallel zu den Operationen der anderen Funktionseinheiten durchgeführt werden (unter Einhaltung bestimmter Restriktionen, s.u.). Weiterhin können Daten im p-Speicher alloziert werden, und es besteht die Möglichkeit, Daten parallel aus den beiden Speichern in die Register der Funktionseinheiten zu laden.

In Abb. 1.5 ist eine Teilmenge der Maschinenoperationen der Funktionseinheiten sowie der möglichen Datentransfers zwischen Registerfiles und Speicherbänken schematisch aufgezeigt. Dabei sind zu den jeweiligen Operationen die möglichen alternativen Registerfiles für das Schreiben der Resultate und das Lesen der Operanden dargestellt. Alternativen werden dabei durch den vertikalen Balken '|' getrennt. Die Notation <*rs*> kennzeichnet jeweils gleiche Mengen alternativer Ressourcen.

Die folgenden Architekturmerkmale dienen der direkten Unterstützung der Operationen der digitalen Signalverarbeitung, um eine hohe Ausführungsgeschwindigkeit zu erreichen:

- *Multiplikationseinheiten*: Diese können die Multiplikation innerhalb eines Instruktionszyklus ausführen. Häufig wird nur die Festkomma-Multiplikation unterstützt, da diese wesentlich schnellere und kleinere Funktionseinheiten erfordert. Die MAC-Einheit in Abb. 1.4 enthält solch eine Multiplikationseinheit, die aber auch herkömmliche ganzzahlige Multiplikationen erlaubt. In diesem Fall liegt das Resultat in Register mr0 vor und das Feedbackregister *mf* kann nicht (sinnvoll) genutzt werden.
- *Unterstützung komplexer Ausdrücke:* Am häufigsten in DSPs sind wohl die schon genannten MAC-Einheiten, die das Bilden der Summe von Produkten unmittelbar unterstützen. Im Allgemeinen werden häufig vorkommende arithmetische Ausdrücke der Applikationen unterstützt (gerade bei ASIPs).

Teilmenge der Operationen der ALU-Einheit

ar|af :=<*xop*> + <*yop*>|<*const*>
ar|af :=<*xop*> - <*yop*>|<*const*>
ar|af :=<*yop*>|<*const*> - <*xop*>
ar|af :=<*yop*> + 1
ar|af :=<*yop*> - 1

<*xop*>=ar|ax|mr|sr
<*yop*>=ay|af
<*const*>=wird noch spezifiziert

Teilmenge der Operationen der MAC-Einheit

mr|mf :=<*xop*> * <*yop*>|<*xop*>
mr|mf :=mr + <*xop*> * <*yop*>|<*xop*>
mr|mf :=mr - <*xop*> * <*yop*>|<*xop*>
mr|mf:=0

<*xop*>=mr|mx|ar|sr
<*yop*>=my|mf

Teilmenge der Operationen des Shifters

sr :=ashift (<*xop*>,<*const*>)
sr :=lshift (<*xop*>,<*const*>)

<*xop*>=ar|ax|sr
<*const*>=Konstanten

Teilmenge der Datentransferoperationen

<*dreg*> :=<*dreg*>|<*const*>
<*dreg*> :=load(ir1|ir2|<*addr*>,d)
<*dreg*> :=load(ir2,p)
d :=store(ir1|ir2|<*addr*>,<*dreg*>|<*const*>)
p :=store(ir2,<*dreg*>)

<*dreg*>=ar|ax|ay|sr|mr|mx|my
<*const*>=Konstanten
<*addr*>=direkte Adressierung

Abbildung 1.5.: Teilmenge der Maschinenoperationen der ADSP2100-Familie.

- *Parallelität auf Instruktionssatzebene*: Auf Prozessoren der ADSP2100-Familie können parallel zu einer Operation auf der ALU-, MAC-, oder Shifter-Funktionseinheit ein bis zwei Datentransfers und ein bis zwei Adressoperationen ausgeführt werden (siehe auch Abb. 1.6). Mehrere Speicherbänke ermöglichen dabei das effektive parallele Laden von Operanden.

- *Spezialisierte AGUs:* Diese erlauben die parallele Ausführung von Adressoperationen zu anderen Maschinenoperationen. Dabei beschränken sich die Adressoperationen oft auf indirekte Adressierung und unterstützen besonders die Speicherzugriffsmuster in einer DSP-Applikation, so z.B.

  - automatisches Inkrementieren / Dekrementieren um einen konstanten Offset auf ein Adressregister unmittelbar nach einem indirekten Speicherzugriff, um linearen Zugriff auf im Speicher aufeinanderfolgende Arrayelemente umzusetzen.
  - Modulo-Operationen, die die Umsetzung von Ringpuffern unterstützen.

  Neben speziellen Adressregistern zur indirekten Adressierung besitzen die noch Register, die einen Satz häufig gebrauchter Offset- und Modulowerte aufnehmen. Die ADSP2100-Familie besitzt zwei solcher Addressierungeinheiten. Die Register zum Aufnehmen der Offsets sind die in Abb. 1.4 gezeigten Modify-Register $m1$ und $m2$ (Modulo-Register sind ebenfalls vorhanden, aber in der Abbildung nicht dargestellt). Man beachte, dass zum Adressieren des d-Speichers beide AGUs genutzt werden können, zum Adressieren des p-Speichers aber nur die zweite AGU zur Verfügung steht.

- *Skalierung und Saturierung: Skalierung* durch vor- bzw. nachgeschaltete Shifter und Handhabung von Überläufen (*Saturierung*[6]).

Darüber hinaus gibt es noch Eigenschaften, welche die Reduzierung von Chipfläche, Programmspeichergröße und Leistungsaufnahme des Prozessors bewirken und zu einer hochgradig irregulären Struktur der Prozessoren führen:

- *Irreguläre Registerfiles:* Diese zeichnen sich durch verteilte Registerfiles mit dedizierten Bedeutungen aus, können aber auch schon durch ein einziges Registerfile, in dem einzelnen Registern eine besondere Bedeutung zukommen kann, gegeben sein.

- *Irreguläre Registerfiles und Funktionseinheiten mit stark eingeschränkten Verbindungen:* Dies ist auch bei der ADSP2100-Familie zu beobachten, wo jede Funktionseinheit mit eigenen Registerfiles verbunden ist. Darüber hinaus können auch innerhalb der Einheiten Resultate und Operanden nicht aus beliebigen Registerfiles gelesen / geschrieben werden. Dies ist in der Darstellung der Maschinenoperationen in Abb. 1.5 sehr gut zu erkennen. So muss der erste Operand von ALU-Operationen im ar, mr, sr oder ax Registerfile alloziert sein, der zweite Operand im ay oder af Registerfile und das Resultat kann nur nach ar oder af geschrieben werden.

[6] Überläufe von Zahlen werden auf die größte positive Zahl (bei positiven Überläufen) oder auf die kleinste negative Zahl (bei negativen Überläufen) gesetzt.

<ALU_{ar}>|<MAC_{mr}>|NOP	ax|mx:=load(i1,d)	ay|my:=load(i2,p)	i1+=mo1	i2+=mo2
<*ALU*>|<*MAC*>|NOP	<dreg>:=load(i1|i2,d|p)	i1|i2+=mo1|mo2		
<*ALU*>|<*MAC*>|NOP	d|p:=store(i1|i2,<dreg>)	i1|i2+=mo1|mo2		
<*ALU*>|<*MAC*>|NOP	<dreg>:=<dreg>			

<dreg> = ar|sr|mr|ax|ax|af|mx|mx|mf
<*ALU*> = ALU-Operation
<*MAC*> = MAC-Operation
<ALU_{ar}> = ALU-Operation bei der die Setzung auf das Registerfile ar beschränkt ist
<MAC_{mr}> = MAC-Operation bei der die Setzung auf das Registerfile mr beschränkt ist

In allen parallelisierten Maschinenoperationen dürfen keine Konstanten vorkommen. und LOAD-, STORE-Operationen sind auf indirekte Adressierung beschränkt.

Abbildung 1.6.: Mögliche Parallelausführung von Maschinenoperationen auf der ADSP-2100 Familie.

- *Stark eingeschränkte Parallelität*: Zur parallelen Ansteuerung aller möglichen parallelisierbaren Operationen wären sehr breite Instruktionsworte notwendig, die sehr häufig nicht vollständig genutzt würden. Um den Programmspeicher möglichst klein zu halten und gleichzeitig Energie beim Laden der Instruktionsworte zu sparen, sind nur bestimmte Kombinationen parallel ausführbarer Maschinenoperationen möglich. Diese Kombinationen sind in der Regel auf die Anforderungen der digitalen Signalverarbeitung zugeschnitten. Um die Instruktionswörter weiter zu reduzieren sind diese häufig noch stark kodiert und lassen nur bestimmte Kombinationen von Ressourcen zu. Dies ist im Schema der parallel ausführbaren Operationen der ADSP2100-Familie in Abb. 1.6 zu erkennen. So ist z.B. maximal eine Operation auf der ALU oder MAC-Einheit mit zwei LOAD-Operationen und damit verknüpften Adressoperationen parallelisierbar. Dabei sind nur LOAD-Operationen möglich, die einen Wert aus dem d-Speicher nach ax oder mx laden und den zweiten Wert aus dem p-Speicher nach ay oder my. Die Werte im Speicher werden indirekt über zwei Adressregister adressiert, deren Inhalte nach dem Laden automatisch inkrementiert / dekrementiert werden, so dass sie auf die Nachfolger / Vorgänger der Werte aus d-Speicher und p-Speicher verweisen. Ansonsten besteht die Möglichkeit, eine ALU- / MAC-Operation mit einem Datentransfer und evtl. noch einer Adressoperation zu parallelisieren. Dabei sind die Datentransfers zwischen Registerfiles auf die Resultat- und Operandenregisterfiles der Funktionseinheiten beschränkt.
- *Residual-Control:* Häufig wiederkehrende Kontrollsignale werden in sogenannten Mode-Registern untergebracht und nach Bedarf umgeladen.

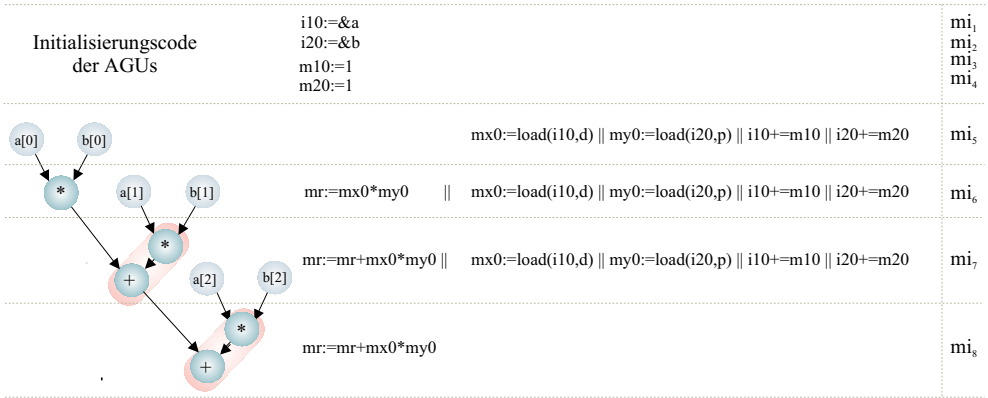


Abbildung 1.7.: Datenflussgraph und ADSP2100 Pseudo-Assemblercode eines Filters.

Beispiel

In Abb. 1.7 ist ein Datenflussgraph und dessen Maschinenprogramm (in Pseudo-Assemblercode) für das innere Produkt zweier Arrays gegeben:

$$\sum_{i=0}^{2} a[i] * b[i].$$

Die parallel ausgeführten Maschinenoperationen einer Maschineninstruktion werden in der Abbildung durch '| |' voneinander getrennt. Das Laden von Werten aus dem Speicher in ein Register wird durch LOAD-Operationen der Form *reg:=load(areg,mem)* dargestellt. Darin ist reg das Register, in das geladen wird, $areg$ das Adressregister, über das indirekt adressiert wird und mem ist die Speicherbank, aus der gelesen wird. Die Autoinkrement- und Autodekrementoperationen der AGU werden in der Form $ireg+ = moreg$ dargestellt, wobei $ireg$ eines der Register der Adressregisterfiles $i1$ oder $i2$ bezeichnet, und $moreg$ eines der Modifyregister aus $mo1$ oder $mo2$ ist (darin steht z.B. $i10$ für Register 0 aus Adressregisterfile $i1$).

Zunächst werden in den Maschineninstruktionen mi_1 bis mi_4 die Adressregister und die Modifyregister geladen. Damit die parallelen LOAD-Operationen ausgenutzt werden können ist es erforderlich, dass Array $a[]$ im d-Speicher und Array $b[]$ im p-Speicher alloziert wird. $i10$ wird dementsprechend zur Adressierung von $a[]$ im d-Speicher und $i20$ zur Adressierung von $b[]$ im p-Speicher benutzt und die Modifyregister $m10$ und $m20$ mit dem zu inkrementierenden Wert (im Beispiel ist das der Wert 1) initialisiert.

In mi_5 werden die ersten beiden parallelen LOAD-Operationen ausgeführt, welche die Register $mx0$ und $my0$ mit den Werten aus $a[0]$ und $b[0]$ laden. Die Wahl der Registerfiles mx und my ist nicht willkürlich, sondern entspricht den Restriktionen der parallelen Ausführung zweier LOAD-Operationen. Die Adressen in den Registern $i10$ und $i20$ werden noch im gleichen Instruktionszyklus (also in mi_5) inkrementiert (Maschinenoperationen $i10+ = mo10$ und $i20+ = mo20$) und zeigen somit auf die Speicheradressen von $a[1]$ und $b[1]$. Es werden in mi_5 also zwei LOAD- und zwei Adressoperationen parallel ausgeführt. Die nächste Maschineninstruktion mi_6 führt die erste Multiplikation über den in mi_5 geladenen Operanden $a[0]$ und $b[0]$ aus. Parallel dazu

werden schon die nächsten Operanden ($a[1]$ und $b[2]$) in die Register $mx0$ und $my0$ geladen und die Adressregister wieder inkrementiert. Die nachfolgende Maschinenintruktion mi_7 führt eine MAC-Operation aus, lädt die nächsten Operanden ($a[2]$ und $b[2]$) und inkrementiert wiederum die Adressregister ($i10$ und $i20$). Die letzte Maschineninstruktion mi_8 führt die letzte notwendige MAC-Operation aus.

Man erkennt deutlich an diesem Beispiel, dass die Prozessorarchitektur und insbesondere die gegebenen Möglichkeiten zur Parallelausführung stark auf diese Applikation hin angepasst wurden.

1.4. Anforderungen an DSP-Compiler

Eine Analyse der notwendigen Anforderungen an Compiler, welche die effektive Nutzung der DSP-Eigenschaften ermöglichen, ist in Kapitel 3 gegeben. Diese Analyse umfasst eine Untersuchung von traditionellen Compilertechniken, ob diese Techniken für den Einsatz im Bereich der eingebetteten Prozessoren geeignet sind. Weiterhin ist der Stand der Technik existierender Compilertechniken, die speziell für eingebettete Prozessoren entwickelt wurden, dargestellt. Ein Abgleich, inwieweit die existierenden Codegenerierungstechniken den Anforderungen eingebetter Prozessoren genügen, stellt die folgenden Kernproblematiken heraus:

- Um Eigenschaften von DSPs effektiv zu unterstützen sind Techniken notwendig, die
  - graphbasierte Mustererkennung von Maschinenoperationen unterstützen,
  - neben der Zuordnung von Programmvariablen zu Registerfiles auch die notwendigen Datentransfers[7] berücksichtigen und
  - die restriktive Parallelität ausnutzen.

  Darüber hinaus müssen die wechselseitigen Abhängigkeiten zwischen diesen Anforderungen berücksichtigt werden. Daraus ergibt sich die Forderung nach phasengekoppelten Techniken, welche die wichtigsten Phasen der Codegenerierung (die Instruktionsauswahl, die Registerallokation und die Instruktionsanordnung) sinnvoll integrieren.
- Die vielen Randbedingungen von irregulären Prozessorarchitekturen sowie die Forderung nach der Phasenkopplung von Techniken machen das Finden guter heuristischer Methoden sehr schwierig. Da es sich um kombinatorische Optimierungsprobleme handelt, besteht ein Bedarf an hocheffizienten suchbasierten Verfahren. Dabei muss ein ausgewogenes Verhältnis zwischen Güte und Effizienz gefunden werden. Es stellt sich auch die Frage nach Methoden, welche die Spezifikation und die Implementierung der Phasenkopplung handhabbar machen, da der Einsatz konventioneller Programmiermethoden hier an seine Grenzen stößt.

[7] Dazu müsen gegebenenfalls Sequenzen von Datentransfer-Operationen generiert werden, um einen Wert über mehrere Registerfiles hinweg zu bewegen, wenn z.B. keine direkte Verbindung zwischen zwei Registerfiles besteht.

- Die Adaption von Techniken an neue Prozessoren erfordert zum einen möglichst retargierbare Techniken, zum anderen aber auch die Erweiterbarkeit eines Compilers. Zur Einhaltung der Codequalität ist es notwendig, klassenspezifische retargierbare Techniken zu entwickeln. Darüber hinaus verlangt die Erweiterbarkeit nach einem transparenten Modell der Zwischenrepräsentation eines Programms während der Codegenerierung, auf das möglichst alle Phasen der Codegenerierung aufsetzen können.

Eine detaillierte Beschreibung der Kernproblematiken wird in Kapitel 3 gegeben.

1.5. Ziele und Überblick

Mit dieser Arbeit werden die folgenden Ziele verfolgt:

- *Hohe Codequalität*: Entwicklung von Techniken zur Generierung hochqualitativen Maschinencodes für Prozessoren mit irregulären Datenpfaden und eingeschränkter Parallelität, was gerade bei Festkomma-DSPs und ASIPs primär vorzufinden ist.
- *Phasenkopplung*: Zur Generierung hochqualitativen Codes sind Techniken zu entwickeln, welche die speziellen Eigenschaften der Zielprozessoren effektiv ausnutzen. Dies erfordert Techniken zur graphbasierten Instruktionsauswahl, Registerallokation mit Handhabung von irregulären Registerfiles sowie der Ausnutzung der stark eingeschränkten Parallelität in der Instruktionsanordnung. Starke wechselseitige Abhängigkeiten zwischen diesen Phasen erfordern den Einsatz phasengekoppelter Techniken.
- *Schnelle Adaption an Prozessoren und Erweiterbarkeit*: Zur schnellen Adaption von Compilern an neue Prozessoren wird ein transparentes Datenmodell entwickelt, das eine einfache Erweiterung um neue Techniken ermöglicht. Auch wenn die Retargierbarkeit ein wichtiges Kriterium bei Compilern für eingebettete Prozessoren ist, hat zunächst die Codequalität zu Lasten der Generalisierung Priorität. Daher werden zunächst Techniken für dedizierte Prozessoren entwickelt (ADSP2100-Familie, Texas Instruments TMS320C1x/C2x/C5x-Familie). Dabei wird allerdings bei der Entwicklung der Techniken schon auf eine Trennung maschinenspezifischer und allgemeinerer Konzepte geachtet.
- *Einsatz neuer Optimierungsmethoden*: Die Entwicklung der Techniken basiert auf dem Einsatz der *Constraint-Logikprogrammierung (CLP)* unter dem Einsatz des Systems ECLiPSe. Die wesentlichen Gründe für diese Entscheidung sind:
  - CLP hat sich im Bereich der Lösung kombinatorischer Probleme bewährt, in deren Klasse auch die hier betrachteten Probleme fallen.
  - In ECLiPSe existieren bereits vordefinierte Optimierungsmethoden, die genutzt werden können. Weiterhin sind flexible und effektive Möglichkeiten zur Anpassung der gegebenen Methoden und zur Entwicklung eigener Optimierungsmethoden gegeben.

  - Formale Spezifikation der Probleme auf der Basis von Constraints erlauben eine handhabbare Spezifikation der Phasenkopplung. Weiterhin können sehr einfach neue Aspekte integriert werden, ohne die bestehenden Spezifikationen modifizieren zu müssen.
  - Eine konzeptionelle Trennung von der Spezifikation der Constraints und den Lösungsstrategien vereinfachen die Implementierung und Adaption neuer Lösungsstrategien. Auf der Basis der gleichen Problemspezifikation können unterschiedliche Zielfunktionen adaptiert werden, so dass neue Problemstellungen sehr einfach umzusetzen sind.

  Ein Ziel der Arbeit ist es, die Eignung dieses Ansatzes im Problembereich zu zeigen.

Die Arbeit gliedert sich wie folgt:

- In Kapitel 2 werden Grundlagen des Compilerbaus, der Codegenerierung, der Constraint-Programmierung sowie der Constraint-Logikprogrammierung eingeführt.
- Kapitel 3 umfasst eine Analyse der zentralen Anforderungen an Compiler für DSPs und ASIPs und gleicht diese mit dem Stand der Technik ab, woraus sich die Ziele der Arbeit ergeben.
- In Kapitel 4 wird die constraintbasierte Zwischenrepräsentation CoLIR definiert. Diese liefert die Grundlage zur Entwicklung phasengekoppelter Codegenerierungstechniken. Weiterhin umfasst sie ein allgemeines und generisches Zwischenformat für Maschinenprogramme, um so eine modulare Entwicklung von Compilertechniken zu ermöglichen und eine einfache Erweiterbarkeit um neue Techniken für neue Zielprozessoren zu erlauben.
- Im daran nachfolgenden Kapitel 5 wird ein generisches Modell zur graphbasierten Instruktionsauswahl entwickelt. Ziel ist es, ein Modell zu liefern, an das schnell neue Kostenfunktionen und Optimierungsstrategien adaptiert werden können. Weiterhin wird hierdurch ein Modell geboten, das zusätzlich noch die Integration der Instruktionsauswahl in andere Codegenerierungsphasen ermöglicht.
- Im Kapitel 6 wird dargestellt, wie das generische Modell zur graphbasierten Instruktionsauswahl zur schnellen Entwicklung exakter Techniken zur Instruktionsauswahl für die ADSP2100-Familie genutzt werden kann. Es wird gezeigt, wie auf einfache Weise Kostenmodelle und Optimierungsstrategien adaptiert werden können.
- In Kapitel 7 wird die Entwicklung phasengekoppelter Techniken am Beispiel der ADSP2100-Familie gezeigt. Dabei werden Techniken beschrieben, welche die zentralen Phasen der Codegenerierung - die Instruktionsauswahl, die Instruktionsanordnung und die Registerallokation - in einer Phase integrieren. Es wird Code generiert, der mit handgeneriertem Assemblercode und compilergeneriertem Code für DSP-Applikationen verglichen wird.

- Kapitel 8 beinhaltet eine Zusammenfassung und gibt einen Ausblick auf zukünftige Arbeiten.
- Im Anhang A werden zwei weitere Ansätze zur Phasenkopplung beschrieben. Eine umfassende Erläuterung dieser Techniken ist im Rahmen dieser Arbeit nicht möglich, da sie diesen sprengen würde. Die Techniken werden daher nur noch in ihren wichtigen konzeptionellen Eigenschaften beschrieben.

2. Grundlagen

In diesem Kapitel werden 1. die grundlegenden Begriffe des Compilerbaus, 2. der Constraint-Programmierung und 3. die Constraint-Logikprogrammierung eingeführt:

1. *Compilerbau*: Zunächst werden Konzepte zur Beschreibung und Repräsentation von Befehlssätzen eingeführt und danach die Struktur von Compilern, deren Phasen und gebräuchliche Zwischenrepräsentationen erläutert.
2. *Constraint-Programmierung*: Es werden zuerst die Grundlagen und Methoden zum Lösen und Optimieren von Constraints bestimmter Constraintsysteme erläutert. Der Schwerpunkt liegt dabei auf Methoden über endlichen Wertebereichen, die eine der zentralen Rollen bei der Lösung kombinatorischer Probleme spielen. Da die in dieser Arbeit betrachteten Probleme der Codegenerierung dieser Problemklasse zuzuordnen sind, bilden Constraints und Lösungsmethoden über endlichen Wertebereichen die Grundlage der hier entwickelten Lösungsstrategien.
3. Der Aufbau und die Syntax von Constraint-Logikprogrammen werden anhand eines einfachen Sprachkerns eingeführt und die Evaluierung dieser Programme anhand einer operationalen Semantik erläutert. Das Kapitel endet mit dem Herausstellen der besonderen Eigenschaften des CLP-Systems ECLiPSe, das zur Implementierung der Techniken in dieser Arbeit verwendet wurde.

Bei der Einführung der Begriffe und Definitionen wurde darauf geachtet, nur die Formalismen einzuführen, die im Kontext der Arbeit notwendig sind. Die Grundlagen der Constraint-Programmierung und der Constraint-Logikprogrammierung sind überwiegend aus [86] entnommen und den hier benötigten Bedürfnissen angepasst worden.

2.1. Grundlegende Begriffe zur Beschreibung von Befehlssätzen

Im folgenden werden die für die Codegenerierung relevanten Prozessorkomponenten bzw. Ressourcen beschrieben, die von den Befehlen eines Prozessors verwendet werden. Dabei werden die folgenden beiden Aspekte primär betrachtet:

- Komponenten zur Spezifikation von Befehlen, wie Ressourcen zum Ablegen und Speichern von Werten (z.B. Registerfiles und Hauptspeicher) und Komponenten zum Ausführen von Operationen (wie z.B. die Funktionseinheiten).

- Aufbau und Repräsentation von Maschinenbefehlen.

sequentielle Komponenten

Definition: *Sequenzielle Komponenten* eines Prozessors sind lesbare bzw. schreibbare Komponenten wie Registerbänke, Speicherbänke oder Ein- / Ausgabeports, deren Inhalte mehr als einen Instruktionszyklus eines Prozessors erhalten bleiben.

flüchtige Komponenten

Definition: *Flüchtige Komponenten* eines Prozessors sind lesbare bzw. schreibbare Komponenten, deren Inhalt nur innerhalb eines Instruktionszyklus gültig ist. Dies sind z.B. Signalleitungen zwischen Funktionseinheiten oder Register zum Zwischenspeichern eines Resultats einer Funktionseinheit, das noch im gleichen Instruktionszyklus von einer anderen Funktionseinheit gelesen wird.

Für eine Menge von Registerfiles, gegeben durch $\mathcal{RF} = \{rf_1, \ldots, rf_n\}$, wird durch $regs(rf_i) = \{reg_{i,1}, \ldots, reg_{i,n_i}\}$ jedem Registerfile eine Menge von Registern zugeordnet, wobei n_i der Kapazität des Registerfiles rf_i entspricht. Symbole zur Benennung von Registerfiles und Speicherbänken werden im Folgenden immer klein geschrieben: z.B. ar, ax, ay, af, p und d. Bezeichner der Register werden durch Anhängen des Indizes an den Registerfilenamen gebildet: z.B. $ax0$, $ax1$, $ay0$ und $ay1$.

ST-Komponente

Definition: Eine *ST-Komponente* ist entweder eine sequenzielle oder eine flüchtige Komponente *(S=sequenziell und T=transitory=flüchtig).*

Maschinenoperation

Definition: Eine *Maschinenoperation* ist eine elementare Operation auf einem Prozessor, wobei die Operanden aus sequenziellen Komponenten gelesen werden und das Resultat in eine sequenzielle Komponente geschrieben wird. Eine Maschinenoperation ist an weitere Ressourcen gebunden, wie z.B. an Funktionseinheiten, auf denen die Operation ausgeführt wird. Eine *parzielle Maschinenoperation* benutzt (lesend oder schreibend) mindestens eine flüchtige Komponente. Eine *komplexe Maschinenoperation* ermöglicht die Ausführung komplexer Ausdrücke, wie z.B. die MAC-Operation: a=a+b*c. In dieser Arbeit setzen sich komplexe Maschinenoperationen aus partiellen Maschinenoperationen zusammen.

Registertransferoperation

Eine gängige Repräsentation von Maschinenoperationen ist *die Registertransferoperation*, welche die auszuführende Operation und die Registerfiles bzw. Register zum Schreiben des Resultats und Lesen der Operanden widerspiegelt: $r_0 := op(r_1, \ldots, r_n)$. Hierbei ist jedes r_i aus der Menge $\mathcal{RF}$ oder aus $\bigcup_{rf \in \mathcal{RF}} regs(rf)$.

Definition: Eine *faktorisierte Maschinenoperation* ist eine Repräsentation alternativer Maschinenoperationen zu einem gegebenen Operator. Sie umfasst die Repräsentation alternativer Mengen von Ressourcen, die einer elementaren Operation auf einem Prozessor zur Verfügung stehen. Dies können alternative Registerfiles für Resultate und Operanden sein, sowie alternative Funktionseinheiten, auf denen die Operation ausgeführt werden kann[1].

faktorisierte Maschinenoperation

Es wird im Folgenden die Notation $r_1|r_2|\ldots|r_n$ zur Darstellung alternativer Ressourcen r_1, r_2, ..., r_n verwendet. Eine *faktorisierte Registertransferoperation* hat so die Form

$$r_{0,1}|r_{0,2}|\ldots|r_{0,k} := op(r_{1,1}|\ldots|r_{1,k_1}, \ldots, r_{n,1}|\ldots|r_{n,k_n})$$

mit $r_{i,j} \in \mathcal{RF} \cup (\bigcup_{rf \in \mathcal{RF}} regs(rf))$.

Definition: Eine *Maschineninstruktion* ist eine Menge von parallel ausführbaren Maschinenoperationen auf einem Prozessor. Eine *maximale Maschineninstruktion* ist eine Menge von Maschinenoperationen, die sich mit keiner weiteren Maschinenoperation parallelisieren lässt. Der Instruktionssatz eines Prozessors lässt sich durch eine Menge von maximalen Maschineninstruktionen spezifizieren. Ein *Maschineninstruktionstyp* ist eine Menge maximaler Maschineninstruktionen mit gleichartiger Struktur. Es wird hier auch vereinfacht nur *Instruktionstyp* gesagt.

Maschineninstruktion, Maschineninstruktionstyp bzw. Instruktionstyp

---

Die Menge der Registertransferoperationen {ar:=ar+ay,ax:=load(ir1,d), ay:=load(ir2,p)} stellt eine maximale Maschineninstruktion der ADSP2100-Familie dar (siehe Kapitel 1.6 Seite 11).

Definition: Unter einem *Instruktionswort* wird die binäre Codierung einer Maschineninstruktion verstanden (z.B. '01101'). Ein *partielles Instruktionswort* liegt vor, wenn bestimmte Positionen im Instruktionswort unspezifiziert bleiben. Diese Positionen werden dann durch ein 'X' gekennzeichnet: z.B. '01XX01'. Unspezifizierte Positionen liegen meist dann vor, wenn der Wert des Steuersignals an der Position keinen Einfluss auf die auszuführende Maschineninstruktion hat und spielen gerade bei der Definition von Instruktionswörtern einzelner Maschinenoperationen bzw. nicht maximaler Maschineninstruktionen eine große Rolle. Zwei Maschinenoperationen können parallel ausgeführt werden, wenn ihre partiellen Instruktionswörter I_1 und I_2 *konfliktfrei* bzw. *kompatibel* sind (siehe auch [87]): D.h., dass entweder alle Positionen in I_1 und I_2 gleich sind oder - wenn sie ungleich sind - eine der beiden Positionen unspezifiziert sein muss.

Instruktionswort

---

[1] Im Verlauf der Arbeit wird eine präzise Definition für constraintbasierte faktorisierte Maschinenoperationen gegeben.

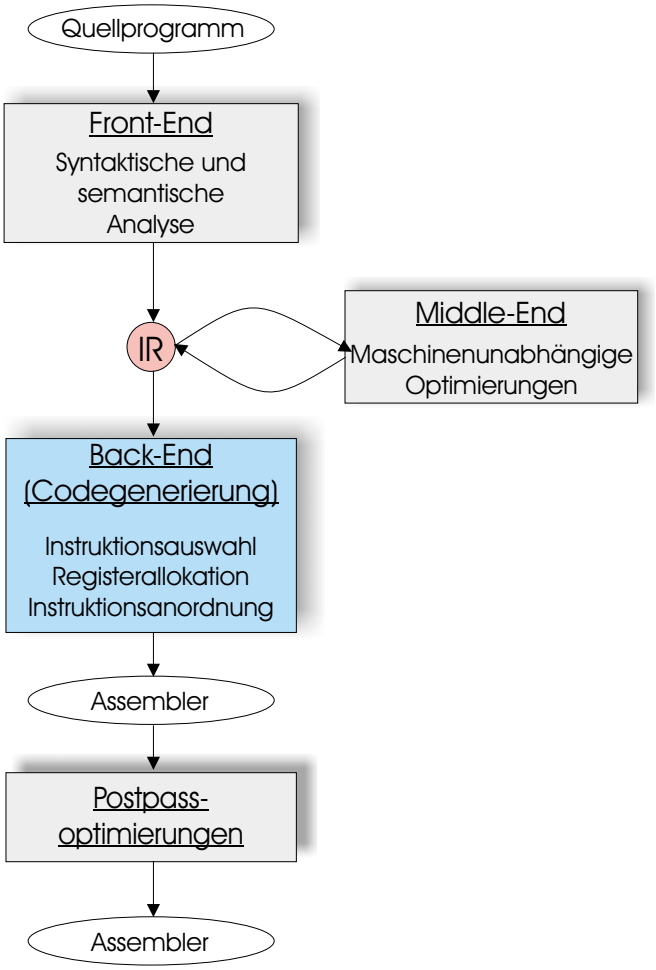


Abbildung 2.1.: Struktur von Compilern.

2.2. Struktur von Compilern

Ein Compiler weist i.d.R. die in Abb. 2.1 gezeigte Struktur auf und kann in folgende Komponenten unterteilt werden:

Front-End: Prüft das Quellprogramm einer Programmiersprache (z.B. *C/C++*) auf syntaktische und semantische Korrektheit und generiert dann eine Zwischenrepräsentation (*IR = intermediate representation*) des Programms. In diesem Stadium enthält die Zwischendarstellung in der Regel noch keine hardwarespezifischen Informationen über den Zielprozessor.

IR

Middle-End: Führt eine Menge maschinenunabhängiger Optimierungen auf der Zwischenrepräsentation aus. Die diesem Bereich zuzuordnenden Optimierungen werden im folgenden als *High-Level-Optimierungen (HLOs)* bezeichnet. Es sei hier bemerkt, dass viele der HLOs gewinnbringend im Back-End erneut durchgeführt werden können und bei einigen Compilern auch in modifizierter Version

HLOs

vorkommen. Dabei werden die Optimierungen i.d.R. auf der gleichen IR ausgeführt, so dass leicht Optimierungen ausgetauscht bzw. ergänzt werden können.

Back-End: Bildet die IR in ein Maschinenprogramm (z.B. Assembler, Objektcode oder Binärcode) des Zielprozessors ab. Dieser Prozess wird auch als Codegenerierung bezeichnet, mit dem Ziel der Generierung von semantisch äquivalenten und möglichst effizientem Maschinencode. Die Optimierungen werden im folgenden als *Low-Level-Optimierungen (LLOs)* bezeichnet. Eine einheitliche IR innerhalb des Back-Ends wäre erstrebenswert, liegt aber in den meisten Fällen nicht vor. Es wird im folgenden der Begriff *LIR (LIR = low-level intermediate representation)* für die Zwischenrepräsentation von Maschinenprogrammen verwendet.

LLOs

LIR

Postpass-Optimierungen: Peephole-Optimierungen und gegebenenfalls Optimierungen aus dem Middle-End, die diesmal maschinenspezifisch angepasst sind. Die hier ausgeführten Optimierungen können auch im Back-End integriert werden.

Das Back-End sowie die Zwischenrepräsentationen IR und LIR spielen eine zentrale Rolle in dieser Arbeit. Daher werden sie im Folgenden noch etwas detaillierter betrachtet.

2.2.1. Codegenerierung (Back-End)

Die *Codegenerierung* ist das Schwerpunkthema dieser Arbeit. Daher werden die einzelnen Phasen der Codegenerierung hier nochmal explizit herausgestellt. Wenn auch die Optimierungen im Middle-End einen hohen Einfluss auf die generierte Codequalität haben, so kommt doch der Codegenerierung die größte Bedeutung bzgl. der Codequalität zu. Die Codegenerierung besteht aus den folgenden primären Teilaufgaben (Phasen):

Instruktionsauswahl: Die Operationen der IR werden auf reale Operationen des Zielprozessors abgebildet, mit dem Ziel, eine möglichst kostengünstige Realisierung des Quellprogramms durch Operationen des Zielprozessors zu erhalten.

Registerallokation: Hier ist das Ziel, möglichst viele Werte des Programms in Registern des Prozessors zu halten, um Datentransfers zwischen dem Hauptspeicher und dem Prozessor zu minimieren.

Instruktionsanordnung: In dieser Phase wird eine möglichst günstige Ausführungsreihenfolge der Maschinenoperationen festgelegt. Bei Prozessoren, die eine parallele Ausführung von Maschinenoperationen erlauben, dient diese Phase der Parallelisierung von Befehlen, i.d.R. mit dem Ziel, möglichst effizienten Code zu generieren. Im Folgenden wird unter dem Begriff Instruktionsanordnung immer das Ziel der Parallelisierung einbezogen. Sollten Techniken die Parallelität nicht berücksichtigen, wird dies explizit durch den Begriff *sequenzielle Instruktionsanordnung* ausgedrückt.

Die Generierung optimalen Maschinencodes zählt zu den NP-harten Problemen, wobei jede Teilaufgabe der Codegenerierung mindestens NP-vollständig ist (siehe [45]). Um die Codegenerierung handhabbar zu machen, werden in traditionellen Compilern die einzelnen Teilaufgaben jeweils in separaten Phasen realisiert und auf der Basis von Heuristiken umgesetzt.

2.2.2. Zwischenrepräsentationen

In der Literatur finden sich viele Arbeiten zur Repräsentation von Programmen[2]. Man kann grundsätzlich zwei Kategorien von Repräsentationen unterscheiden:

abstrakte Maschineninstruktionen

Maschinenunabhängige Zwischenrepräsentationen: Diese sind in Form von *abstrakten Maschineninstruktionen* gegeben. Die Konzepte der jeweiligen Sprachklasse werden unabhängig von einer spezifischen Prozessorarchitektur umgesetzt. Für bestimmte Sprachklassen, wie funktionale und logische Sprachen, dienen sie u.a. dem Zweck, den Compilierungsprozess zu vereinfachen. Für diese Sprachen existieren abstrakte Maschinen in Form von Software, welche die abstrakten Maschineninstruktionen ausführen können (z.B. Graphreduktionsmaschinen für funktionale Programme [38], WAM[3] für logische Programme [115] und JVM[4] für JAVA-Bytecode [83]).
Die wohl verbreitetste Darstellung imperativer Programmiersprachen ist der 3-Adress-Code, der die wesentlichen Konstrukte zur Unterstützung imperativer Sprachen bietet:

1. Zuweisungsinstruktionen.
2. Bedingte und unbedingte Sprünge.
3. Funktionsaufrufe und Parameterübergabe, sowie Rücksprünge aus aufgerufenen Funktionen.
4. Darstellung von Pointern und indizierter Adressierung.

Maschinenabhängige Zwischenrepräsentationen: Hier werden i.d.R. Zwischenstufen von Maschinenoperationen bzw. von Maschineninstruktionen dargestellt. Es gibt z.Z. nur wenige Systeme, die klar definierte Repräsentationen umfassen. Beispiele für solche Systeme sind Trimaran [113], SUIF [49] und SPAM [108]. Weiterhin verwenden neuere Compiler ihren Assembler als Zwischenrepräsentation (z.B. TMS320C6x von Texas Instruments).

abstrakte Operationen der IR

Operationen der LIR

Abstrakte Maschineninstruktionen werden im Folgenden als *abstrakte Operationen* bezeichnet. Auf dem Zielprozessor ausführbare elementare Operationen werden Maschinenoperationen und u.a. auch als *Operationen der LIR* bezeichnet. Es existieren weitere Zwischenrepräsentationen, die in beiden Kategorien Anwendung finden und Formate darstellen, die gewisse Analysen und Optimierungen mehr oder weniger gut unterstützen:

[2] Ein Überblick ist in [8] zu finden.
[3] WAM=Warren Abstract Machine.
[4] JVM=Java Virtual Machine.

Graphbasierte Zwischenrepräsentationen: spielen sowohl in abstrakten Maschinen als auch in der Codegenerierung eine grosse Rolle. Sie dienen der expliziten Darstellung bestimmter Abhängigkeiten in den Programmen, wie z.B. die Darstellung des Kontrollflusses und des Datenflusses zwischen Instruktionen.

Multi-Assignment-Form: Imperative Programmiersprachen erlauben i.d.R. Mehrfachzuweisungen an dieselbe Programmvariable:

```
S1: x=a+b;
S2: y=x+b;
S3: x=y+2;
```

Mehrfachzuweisungen, wie die an Programmvariable x, erzeugen zusätzliche Abhängigkeiten (siehe auch Datenabhängigkeitsgraphen) zwischen Maschinenoperationen, die die Freiheitsgrade bei der Umordnung von Maschinenoperationen stark beeinträchtigen können.

Static-Single-Assignment (SSA): In SSA existieren keine Mehrfachzuweisungen an eine Programmvariable. Neben der Vermeidung unnötiger Abhängigkeiten begünstigt die SSA darüber hinaus auch viele Analysen und Optimierungen. Repräsentationen in Multi-Assignment-Form werden durch eindeutige Umbenennung der Programmvariablen in SSA transformiert. Dabei wird es notwendig, an Punkten zusammenfließenden Kontrollflusses sogenannte ϕ-Knoten einzuführen:

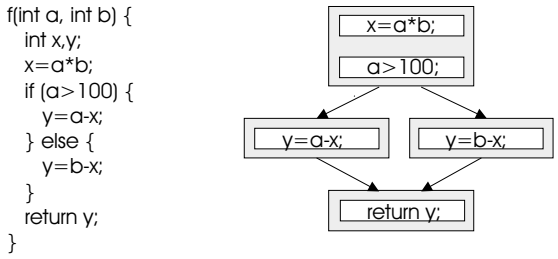
SSA

```
y=a*b;                  y1=a*b;
y=a+y;                  y2=a+y1;
if (y>2){      →        if (y2>2) {
   y=1;                    y3=1;
} else {                } else {
   y=2;                    y4=2;
}                       }
x=y+2;                  y5=φ(y3,y4);
                        x=y5+2;
```

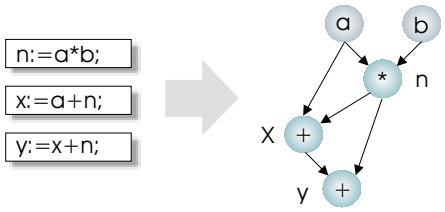
Das Middle-End und das Back-End operieren in der Regel auf graphbasierten Repräsentationen der IR bzw. der LIR. In den letzten Jahren wurde eine Vielzahl unterschiedlicher Graphmodelle entwickelt, die für spezifische Analysen und Optimierungen mehr oder weniger gut geeignet sind. Die wohl geläufigsten graphbasierten Darstellungen von Programmen, die im Back-End Anwendung finden, sind die *Kontrollflussgraphen*, *Datenflussgraphen* und die *Datenabhängigkeitsgraphen*. In Kapitel 4.3 werden die formalen Grundlagen dieser Graphen definiert. Es wird hier zunächst nur eine informelle Einführung gegeben. Der Kontrollflussgraph reflektiert den Kontrollfluss des Programms. Dabei wird das Programm zunächst in *Basisblöcke (engl: basic block)* unterteilt. Diese repräsentieren maximale Sequenzen von Anweisungen, d.h. der Kontrollfluss kann sich nur nach der letzten Anweisung der Sequenz aufteilen und nur bei der ersten Anweisung der Sequenz wieder zusammenfließen (siehe Abb.2.2 (a)). Der

Kontrollflussgraphen, Datenflussgraphen und Datenabhängigkeitsgraphen

Basisblöcke



(a) Kontrollflußgraph



(b) Datenflußgraph

Abbildung 2.2.: Zwischenrepräsentationen von Programmen.

Datenflussgraph reflektiert den Datenfluss innerhalb eines Basisblocks. Dabei spiegeln die Knoten eines Datenflussgraphen die Operationen innerhalb des Basisblocks wider und die Kanten den Fluss von Werten zwischen erzeugenden Operationen und den konsumierenden Operationen (siehe Abb. 2.2 (b)). Knoten, die mehrere ausgehende Kanten besitzen, werden *gemeinsame Teilausdrücke* genannt *(CSE = common subexpression*) wie z.B. die Multiplikation in Abb. 2.2 (b).

CSE

2.3. Constraint-Programmierung

Die Constraint-Programmierung ist ein Gebiet, das zur Lösung von komplexen Optimierungsproblemen sehr an Bedeutung gewinnt und vor allem in der Industrie zunehmend Beachtung findet. So zeigt sich der Einsatz besonders in kommerziellen Anwendungen, die mit Optimierungsproblemen gerade im Bereich kombinatorischer Suche zu tun haben. Das verstärkte Interesse begründet sich vor allem durch einen Nachteil konventioneller Programmiermethoden: eine unzureichende Unterstützung zur Modellierung wechselseitiger Beziehungen (Relationen) zwischen Objekten, also der Darstellung und Handhabung von Constraints. Gerade diese nehmen in komplexen Anwendungen aber immer mehr zu und erschweren somit die Modellierung.

Die wesentliche Idee der Constraint-Programmierung lässt sich am Beispiel des Ohmschen Gesetzes erläutern, das die Beziehung zwischen Spannung, Stromstärke und Widerstand ausdrückt:

$$V = I \times R$$

In traditionellen Programmiersprachen kann diese Beziehung nicht unmittelbar ausgenutzt werden und die Berechnung von V, I und R müssen alle explizit durch drei Zuweisungen

$$I := V/R$$

$$R := V/I$$

$$V := I \times R$$

spezifiziert werden. In der Constraint-Programmierung ist es die Aufgabe des Programmiersystems, die Relationen sicherzustellen. Die oben beschriebene Relation kann durch einfache Angabe der Werte durch die Constraints $V = 100$ und $I = 10$ zur Berechnung von R genutzt werden. Die gleiche Relation kann aber auch durch Angabe von $V = 100$ und $R = 5$ zur Berechnung von I benutzt werden. Weitere Randbedingungen werden einfach durch entsprechende Constraints ergänzt (z.B. $I > 25$ und $V < 200$), die das Programmiersystem bei der Spezifikation neuer Wertebelegungen der Variablen dann auch prüft. Es ergeben sich zwei grundsätzliche Problemstellungen:

1. Der Nachweis, dass überhaupt eine Lösung zu einem gestellten Problem existiert (oft ist hier nur die Existenz einer Lösung, aber nicht deren konkrete Wertebelegung wichtig).
2. Die Aufgabe, unter der Menge der gültigen Lösungen bzgl. einer die beste Lösung zu finden (Optimierungsproblem).

In modernen Constraint-Programmiersystemen existieren für spezifische Wertebereiche und Constraints vordefinierte Lösungsmethoden (Existenznachweis und Finden einer Wertebelegung der Variablen) und Optimierungsmethoden. So zum Beispiel für den Wertebereich der reellen Zahlen mit Gleichungen und Ungleichungen über linearen arithmetischen Ausdrücken. Im Falle existierender und geeigneter Lösungsmethoden besteht die Problemlösung nur noch in der geschickten Formulierung des Problems durch die gegebenen Constraints. Darüber hinaus bieten moderne Systeme auch die Adaptierung eigener Lösungs- und Optimierungsmethoden, um diese effektiv an konkrete Anwendungen anpassen zu können.

2.3.1. Constraints und Variablenbelegungen

Die Aussage $X = Y + 2$ ist ein Beispiel für ein Constraint. Dabei sind X und Y Variablen, also Platzhalter für Werte. Im Verlauf der weiteren Arbeit werden folgende Notationen für Constraints eingehalten: Variablen beginnen mit einem Großbuchstaben oder mit "_"; Elemente symbolischer Wertebereiche werden grundsätzlich klein geschrieben. Ein *Constraintsystem* definiert die folgenden Aspekte[5]: *Constraintsystem*

- die gültigen *Wertebereiche* (im Folgenden auch *Domains* genannt), die Variablen annehmen dürfen (z.B. die reellen Zahlen),
- die Symbole von *Konstanten*, *Operatoren* und *Relationen* (z.B. Konstanten: *1, 2, 3;* Operatoren: $+, *, -, /$; Relationen:$\leq, <, =, \neq$),
- die *Syntax* von Ausdrücken und Constraints (s.u.), und
- die *Semantik, die* die Interpretation von Ausdrücken und Constraints festlegt, d.h. wie Ausdrücke zu Werten des Wertebereichs evaluiert werden und mit welchen Wertebelegungen Prädikate gültig bzw. nicht gültig sind.

Relationen werden im Folgenden immer als *Prädikate* bezeichnet. Beispiele von Constraintsystemen sind lineare arithmetische Constraints über den reellen und ganzen Zahlen, die Boolesche Algebra sowie Constraints über endlichen Wertebereichen.

Die folgenden Definitionen bilden das formale Gerüst für den syntaktischen Aufbau von Ausdrücken und von Constraints, das auf der Basis von Signaturen und Termen definiert wird. Durch *Signaturen* werden die Komponenten zum Aufbau von Termen festgelegt. Die Konstruktionsvorschrift legaler Terme wird durch *Sorten* spezifiziert. Es wird weiterhin definiert, was die Menge der *Terme über einer Signatur* ist.

Signatur

Definition: Eine *Signatur* $\Sigma = (F, S, V, \tau)$ ist gegeben durch eine Menge von Operatorsymbolen F, eine Menge von *Sorten* (Typsymbolen) S und einer abzählbar unendlichen Menge von Variablen V. Dabei wird jedem $f \in F$ durch die Funktion τ ein Typ der Form $s_1, ..., s_n \rightarrow s$ zugeordnet ($s_1, ..., s_n, s \in S$). Typen der Form $\rightarrow s$ werden Konstanten vom Typ s genannt. V ist ebenfalls derart in Sorten unterteilt, dass $V = \bigcup_{s \in S} V_s$. Es wird von einer *überladenen Signatur* gesprochen, wenn die Funktion τ auf die Abbildung von Operationssymbolen auf unterschiedliche Stelligkeiten und Sorten erweitert wird.

Term über Σ

Definition: Die Menge $T_\Sigma = \bigcup_{s \in S} T_\Sigma^s$ der *Terme über* Σ ist induktiv definiert durch:

1. $x \in T_\Sigma^s$ ist ein Term der Sorte s wenn $x \in V_s$.
2. $f \in T_\Sigma$ ist ein Term der Sorte s, wenn $f \in F$ und $\tau(f) = \rightarrow s$. Ein Term dieser Art wird Konstante genannt.
3. $f(t_1, ..., t_n) \in T_\Sigma$ ist ein Term der Sorte s, wenn $\tau(f) = s_1, ..., s_n \rightarrow s$ und $t_1 \in T_\Sigma^{s_1}, \ldots, t_n \in T_\Sigma^{s_n}$.

[5] Auf eine formale Definition von Constraintsystemen wurde hier verzichtet, da sie zum Verständnis der weiteren Ausführung nicht notwendig ist. Es wird sich hier auf die Definition der wichtigen Teilaspekte beschränkt. Eine umfassende formale Definition von Constraintsystemen kann in [44] gefunden werden.

Man bezeichnet T_Σ auch als Baumsprache. T_Σ ist eine *homogene Baumsprache* wenn $S = \{s\}$ und τ beliebig überladen sind.

Ein Constraintsystem definiert eine Signatur Σ_F für Operatorsymbole und eine Signatur Σ_P für Prädikatsymbole. In Σ_P gibt es eine ausgezeichnete Sorte $bool \in S$ und es gibt zwei ausgezeichnete Konstanten $\{true, false\} \in F$ vom Typ $\rightarrow bool$. Alle Prädikatensymbole bilden in den Typ $bool$ ab, besitzen also einen Typen der Form $s_1, ..., s_n \rightarrow bool$.

---

primitive Constraints und Constraints

Definition: Gegeben sei ein Constraintsystem mit Signatur Σ_F und Σ_P und ein n-stelliges Prädikatensymbol p mit $\tau(p) = s_1, \ldots, s_n \rightarrow bool$. Ein *primitives Constraint* ist gegeben durch $p(t_1, ..., t_n)$, mit $\tau(t_1) = s_1, \ldots, \tau(t_n) = s_n$. Ein *Constraint* ist gegeben durch $c_1 \wedge \ldots \wedge c_n$ mit $n > 0$ wobei $c_1, \ldots, c_n$ primitive Constraints sind.

Beispiel: $X \leq Y/Z - 1$ ist ein primitives Constraint mit dem Prädikatensymbol $\leq$ vom Typ $r, r \rightarrow bool$, wobei r das Symbol der reellen Zahlen repräsentiert. $t_1 = X$ und $t_2 = Y/Z - 1$ wobei t_1 und t_2 vom Typ der reellen Zahlen sind.

Die Semantik der Sorten-, Funktions- und Prädikatensymbole wird durch eine Interpretation gegeben, die jedem Symbol ein semantisches Objekt zuordnet.

Interpretation

Definition: Eine *Interpretation* $\mathcal{I}$ ist gegeben durch einen nichtleeren Wertebereich $D = \bigcup_{s \in S} D_s$, wobei D_s die Zuordnung eines semantischen Wertebereichs zur Sorte s ist. Dabei gibt es den ausgezeichneten Wertebereich $Bool_{bool} = \{wahr, falsch\}$. Weiterhin gibt es eine Funktion $I : D^n \longrightarrow D$, die jedem Funktions- und Prädikatensymbol eine semantische Funktion bzw. Relation zuordnet: Für $f \in \Sigma_F$ bzw. $f \in \Sigma_P$ liefert $I(f)$ als Resultat eine Funktion vom Typen $D_{s_1}, ..., D_{s_n} \rightarrow D_s$, wenn $\tau(f) = s_1, ..., s_n \rightarrow s$ ist.

Variablenbelegung und partielle Variablenbelegung eines Constraints

Definition: Für Σ_F ist eine *Variablenbelegung* θ eine Abbildung $V \mapsto D$ mit $D = \bigcup_{s \in S} D_s$, wobei D_s der Wertebereich der Sorte s ist. Es wird davon ausgegangen, dass θ sortenverträglich abbildet. Es wird die Notation $\{X_1 \mapsto d_1, ..., X_n \mapsto d_n\}$ verwendet, wenn für $i \in \{1, \ldots, n\}$ jedem $X_i \in V$ der Wert $d_i \in D$ zugewiesen wurde. Für einen Term bzw. ein Constraint C liefert $vars(C) \subseteq V$ die Menge der in C enthaltenen Variablen.

Eine *partielle Variablenbelegung* von θ ist eine Variablenbelegung θ' über der Variablenmenge $V' \subseteq V$, so dass $\theta'(v) = \theta(v)$ für alle $v \in V'$ gilt, und an Stelle von θ' wird auch $\theta_{V'}$ geschrieben.

Interpretation, Lösung und partielle Lösung

Definition: Für eine gegebene Variablenbelegung θ und Term t ist die *Evaluierung von* t unter einer *Interpretation* $\mathcal{I}$ gegeben durch:

1. $I_\theta(X) = \theta(X)$, für $X_i \in V$
2. $I_\theta(f(t_1, ..., t_n)) = I(f)(I_\theta(t_1), ..., I_\theta(t_n))$

Für ein primitives Constraint C ist die Evaluierung $I_\theta(C)$ definiert durch

$$I_\theta(p(t_1, ..., t_n)) = I(p)(I_\theta(t_1), ..., I_\theta(t_n))$$

Im Folgenden wird für $I_\theta(t)$ (t ist Constraint oder Term) nur kurz $\theta(t)$ geschrieben, sofern die Interpretation aus dem Kontext klar ist. *θ ist eine Lösung von* $c_1 \wedge \ldots \wedge c_n$ wenn jedes $\theta(c_1), \ldots, \theta(c_n)$ unter der Interpretation des Constraintsystems $wahr$ liefert. Ein Constraint ist *erfüllbar*, wenn es eine Lösung besitzt. Eine partielle Variablenbelegung $\theta_{V'}$ heißt *partielle Lösung* eines Constraints C, wenn θ eine Lösung von C ist.

Beispiel: Es werden lineare arithmetische Constraints über reellen Zahlen betrachtet. $X < Y + 3.0 \ \wedge \ X > Y$ ist ein Constraint und $\theta = \{X \mapsto 3.0, Y \mapsto 2.0\}$. Dann ist $\theta(Y + 3.0) = 5.0$ und $\theta(C) = 3.0 < 5.0 \ \wedge \ 3.0 > 2.0$, was unter der gegebenen Interpretation wahr ist.

Substitution

Definition: Für eine Signatur Σ_F ist eine *Substitution* σ eine Abbildung $V \mapsto T_{\Sigma_F}$, wobei σ sortenrein abbildet. Analog zur Variablenbelegung wird $\{X_1 \mapsto t_1, ..., X_n \mapsto t_n\}$ geschrieben, wenn für $i \in \{1, \ldots, n\}$ jedem $X_i \in V$ der Wert $t_i \in T_{\Sigma_F}$ zugewiesen wird. Für eine Substitution σ und einen Term bzw. Constraint C bedeutet $\sigma(C)$ die Ersetzung jedes Vorkommen der Variablen X_i durch t_i in C.

2.3.2. Erfüllbarkeit und Optimierung

Die Erfüllbarkeit eines Constraints konstituiert sich auf der Existenz einer Lösung, wobei die konkrete Wertebelegung unter dieser Fragestellung oft irrelevant ist. Bei der Optimierung besteht die Frage nach einer Lösung, die unter einer gegebenen Kostenfunktion die beste unter allen Lösungen darstellt. In diesem Fall ist die Wertebelegung von zentraler Bedeutung.

Erfüllbarkeit und Constraintlöser

Ein Algorithmus, der die Erfüllbarkeit von Constraints bestimmt, wird *Constraintlöser (constraint solver)* genannt.

Definition: Ein *Constraintlöser solv* für ein Constraintsystem CS nimmt als Eingabe ein Constraint C aus CS und liefert als Resultat $wahr, falsch$ oder $unbekannt$. Wenn $solv(C) = wahr$ ist, so muss C erfüllbar sein.Ist $solv(C) = falsch$, so ist C unerfüllbar. Liefert $solv(C)$ $unbekannt$, kann der Constraintlöser nicht bestimmen, ob C erfüllbar oder nichterfüllbar ist. Ein Solver *solv* heißt *vollständig*, wenn $solv(C)$ für alle Constraints C eines Constraintsystems entweder $wahr$ oder $falsch$ liefert. Ansonsten heißt $solv$ *unvollständig*. *Constraintlöser*

Eine intuitive Vorgehensweise zur Bestimmung der Erfüllbarkeit ist die einfache Aufzählung von Variablenbelegungen. Ein Nachteil dieser Methode ist der potenziell unendliche bzw. exponenziell große Suchraum, der dabei zu durchlaufen ist. Selbst wenn Aufzählungsmethoden existieren, die ein Finden einer Lösung in endlicher bzw. akzeptabler Zeit garantieren, muss spätestens zum Nachweis der Nichterfüllbarkeit der gesamte Suchraum durchlaufen werden. Bei der Aufzählungsmethode werden Constraints nur passiv im Sinne eines Tests genutzt. Bessere Strategien verwenden Constraints aktiv bei der Konstruktion einer gültigen Belegung, was i.d.R. zu wesentlich effizienteren Algorithmen führt.

Viele Constraintlöser basieren auf der Transformation von Constraints. Dabei wird ein Constraint C in ein äquivalentes Constraint C' auf der Basis bestimmter Regeln transformiert, bis das Constraint in sogenannter *gelöster Form (solved form)* vorliegt. Aus dieser kann klar abgeleitet werden, ob das Constraint erfüllbar ist oder nicht. Man spricht hier auch von *normalisierenden Constraintlösern*. In vielen Fällen ist aus der gelösten Form unmittelbar eine Variablenbelegung ablesbar, und man spricht von *deterministisch gelöster Form (deterministic solved form*) des Constraints. Ein Beispiel für einen normalisierenden Solver ist die bekannte Gauss-Jordan Elimination über linearen Gleichungen. Dieser gehört zu den vollständigen Constraintlösern, der ein Constraint in eine deterministisch gelöste Form transformiert.

Ein weiteres Beispiel für einen vollständigen normalisierenden Constraintlöser ist ein Constraintlöser für Tree-Constraints.

Definition: Gegeben sei eine Signatur Σ. Ein *primitives Tree-Constraint* ist von der Form $t = s$ oder $t \neq s$ für $s, t \in T_\Sigma$. Ein Tree-Constraint ist gegeben durch $c_1 \wedge ... \wedge c_n$ mit primitiven Tree-Constraint $c_1, ..., c_n$. *Tree-Constraints*

Beispiel: Tree-Constraints sind demnach Gleichungen und Ungleichungen über Termen. Terme können neben der Darstellung von arithmetischen Ausdrücken auch zur Darstellung komplexer Datenstrukturen verwendet werden. So kann eine Liste über den Operatorsymbolen $cons$ und nil in der Art dargestellt werden, dass die leere Liste [] durch *nil* und die Liste

$$[a_1, .., a_n]$$

durch

$$cons(a_1, cons(a_2, ...cons(a_n, nil)...)$$

repräsentiert wird. Für das Tree-Constraint

$$cons(Y, nil) = cons(X, Z) \wedge Y = cons(a, T)$$

erhält man durch Anwendung des Tree-Constraintlösers aus Abb. 2.3 die normalisierte Form

$$X = cons(a, T) \wedge Y = cons(a, T) \wedge Z = nil$$

aus der eine Lösung sicherlich unmittelbar ablesbar ist, wobei für T ein beliebiger Listenterm eingesetzt werden kann.

Eine allgemeine Variante von Constraintlösern ist der *Local Propagation Solver*, der fast uneingeschränkt für alle Constraintsysteme eingesetzt werden kann, aber generell unvollständig ist. Das Prinzip ist, dass aus primitiven Constraints oft auf die Wertebelegung bestimmter Variablen geschlossen werden kann. Diese Variablen werden dann durch diese Werte substituiert, was zu weiteren Schlussfolgerungen bzgl. der Wertebelegungen führen kann. So wird fortgefahren, bis entweder ein primitives Constraint unlösbar wird oder alle Variablen determiniert sind und C wahr ist oder auf keine weiteren Wertebelegungen von Variablen geschlossen werden kann (und somit $unbekannt$ resultiert).

Optimierungsprobleme

Das Ziel der Optimierung ist unter der Menge der Lösungen die beste Lösung bzgl. einer Kostenfunktion k zu bestimmen. Optimierungsprobleme sind von zentraler Bedeutung in der Industrie, z.B. Optimierung der Ablaufplanung von Prozessen, Ressourcenoptimierung oder Routenplanung von Verkehrsmitteln. In vielen Fällen besteht ein sehr hohes Potenzial zur Kosteneinsparung, daher ist es sinnvoll, Optimierungsmethoden einzusetzen, die möglichst optimale Lösungen liefern.

Optimierungsproblem

Definition: Ein *Optimierungsproblem* (C, k) besteht aus einem Constraint C und einer Kostenfunktion f, die in die reellen Zahlen abbildet. Es wird angenommen, dass k durch einen arithmetischen Term mit $var(k) \subseteq vars(C)$ gegeben ist. Eine Variablenbelegung θ ist eine optimale Lösung für (C, k), wenn für alle θ' mit $\theta' \neq \theta$ gilt: $\theta(k) < \theta'(k)$. Es handelt sich hier um ein Minimierungsproblem. Ein entsprechendes Maximierungsproblem kann einfach durch Negation von k formuliert werden.

Einer der bekanntesten und wohl auch am häufigsten eingesetzten Algorithmen zur Lösung von Optimierungsproblemen im Bereich linearer arithmetischer Constraints ist Dantzig's Simplex Algorithmus. Dieser weist in seiner Worst-Case-Laufzeit immer noch ein exponenzielles Verhalten auf, ist in der Praxis jedoch schneller als andere bekannte Verfahren. So existieren Verfahren, die Probleme der Linearen-Programmierung in polynomieller Laufzeit lösen können [61]. Für ganzzahlige Lösungsbereiche ist das Problem NP-vollständig [45].

```
s,t,s_i,t_j sind Terme
c ist ein primitives Tree-Constraint
C,C_0, und S sind Tree-Constraints
x ist Variable

tree_solve(C)
  if unify(C) then
    return false
  else
    return true
  endif

unify(C)
  S :=true
  while C ist von der Form C_0 ∧ c do
    cases c of
    (1) x = x:
          C := C_0
    (2) f(s_1, ..., s_n) = g(t_1, ..., t_m) mit f ≠ g oder n ≠ m:
          return false
    (3) f(s_1, ..., s_n) = f(t_1, ..., t_n):
          C := s_1 = t_1 ∧ ... ∧ s_n = t_n ∧ C_0
    (4) t = x:
          C := x = t ∧ C_0
    (5) x = t und x kommt in t vor:
          return false
    (6) x = t:
          substituiere alle Vorkommen
              von x in C_0 durch t
          C := C_0
          S := S ∧ c
    endcases
  endwhile
  return S
```

Abbildung 2.3.: Tree-Constraintlöser.

2.3.3. Constraintsysteme über endlichen Wertebereichen

Eine sehr wichtige und weitgenutzte Klasse von Constraintsystemen sind *Constraintsysteme über endlichen Wertebereichen (finite constraint domains).* Darin sind die Werte, die Variablen annehmen können, auf endliche Mengen beschränkt. Diese Constraintsysteme werden gerade in kombinatorischen Problemen eingesetzt, wo eine konkrete und eingeschränkte Auswahl bzgl. der Wertebelegung der Variablen besteht. Daher spielen sie in der vorliegenden Arbeit bei der Entwicklung der Codegenerierungstechniken eine zentrale Rolle.

Erfüllbarkeit über endlichen Wertebereichen und CSPs

Definition: Ein *Erfüllbarkeitsproblem über endlichen Wertebereichen* - im Folgenden *CSP (constraint satisfaction problem)* genannt - besteht aus einem Constraint C mit $vars(C) = \{X_1, ..., X_n\}$ und einem Domain (Wertebereich) D, der jeder Variablen X_i eine endliche Menge von ganzen Zahlen $D(X_i)$ zuordnet. Dabei werden auch symbolische Konstanten zugelassen unter der Annahme, dass eine eindeutige Abbildung dieser Konstanten auf ganze Zahlen existiert (z.B. $\{audi, vw, golf\}$ mit $f(audi) = 1, f(vw) = 2, f(golf) = 3$). Ein CSP ist dann durch das Constraint

$$C \wedge X_1 \in D(X_1) \wedge ... \wedge X_n \in D(X_n)$$

gegeben. $D(X)$ wird im Folgenden auch *Domain der Variablen* X genannt.

Beispiel: Ein typisches Beispiel für ein CSP ist das Graphfärbungsproblem eines ungerichteten Graphen. Dabei ist jedem Knoten des Graphen eine bestimmte endliche Menge von Farben zugeordnet (z.B $D(X) = \{rot, gelb\}$ für alle Variablen X). Das Problem besteht nun darin, allen Knoten derart Farben zuzuordnen, dass benachbarte Knoten (also die durch eine Kante verbundenen Knoten) nie die gleiche Farbe besitzen. Für eine Variablenmenge $V = \{X, Y, Z\}$ mit Nachbarschaftsbeziehung $X \longleftrightarrow Y$ und $Y \longleftrightarrow Z$ ist das entsprechende CSP durch $X \neq Y \wedge Y \neq Z \wedge X, Y, Z \in \{rot, gelb\}$ gegeben. Eine mögliche Lösung ist: $\{X \mapsto rot, Y \mapsto gelb, Z \mapsto rot\}$.

2.3.4. Backtracking-Solver für CSPs

Da die Wertebereiche der Variablen endlich groß sind, ergibt sich ein endlicher Suchraum. Dennoch ist dieser Suchraum exponenziell groß und das Lösen von CSPs stellt generell ein NP-hartes Problem dar [86]. Daher existiert kein vollständiger Constraintlöser, der beliebige CSPs in polynomieller Zeit lösen kann.

In Abb. 2.4 ist ein auf *chronologischem Backtracking* basierender vollständiger Constraintlöser für CSPs mit potenziell exponenzieller Laufzeit dargestellt. Für ein Constraint C und $vars(C) = \{X_1, ..., X_n\}$ wird eine Variable ausgewählt und mit einem Wert aus ihrem Wertebereich belegt. Es wird geprüft, ob C mit der Ersetzung der Variablen X durch d partiell erfüllbar ist, d.h. die Funktion `partiell_erfüllbar`(C) prüft alle primitiven Constraints, die keine Variablen mehr beinhalten, auf Erfüllbarkeit. Ist

```
C und C_1 sind Constraints
c_1, ..., c_n sind primitive Constraints
back_solv(C)
  if vars(C) ≡ ∅ then
    return partiell_erfüllbar(C)
  else
    wähle X ∈ vars(C)
    foreach d ∈ D(X) do
      C_1 := substituiere X durch d in C
       if partiell_erfüllbar(C) then
          if back_solv(C_1) then
             return true
          endif
       endif
    endfor
    return false
  endif


partiell_erfüllbar(C)
  C ist von der Form c_1, ..., c_n
  for i:=1 to n do
    if vars(C) ≡ ∅ then
      if not erfüllbar(c_i) then
        return false
      endif
    endif
  endfor
  return true
```

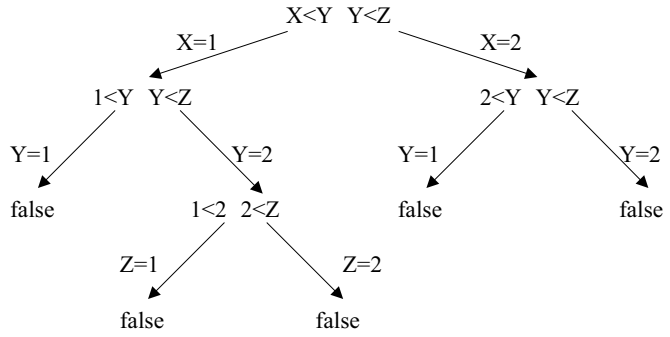


Abbildung 2.4.: Backtracking-Solver für CSPs und Suchbaum für das CSP $X < Y \wedge Y < Z \wedge X, Y, Z \in \{1, 2\}$.

die partielle Erfüllbarkeit gegeben, so wird `back_solv` rekursiv aufgerufen und mit der nächsten Variablen fortgefahren. Ist die partielle Erfüllbarkeit verletzt, wird entweder mit einem noch nicht ausgewählten Wert aus $D(X)$ fortgefahren, oder es wird zur Vorgängervariablen zurückgeschritten (Backtracking), falls alle Werte aus $D(X)$ schon gewählt wurden.

In Abb. 2.4 ist der Suchbaum für das CSP $X < Y \wedge Y < Z \wedge X,Y,Z \in \{1,2\}$ gezeigt. Dabei sind die Kanten mit den Wertebelegungen der Variablen annotiert, wobei der Weg von der Wurzel bis zu einem Blatt die Auswahlstrategie der Variablen- und Wertreihenfolge reflektiert. Die Knoten sind mit dem jeweiligen Zustand des CSPs annotiert. Die Blätter spiegeln das Scheitern (*false*) bzw. das Finden einer Lösung (*true*) unter der Wertebelegung, gegeben durch den Pfad von der Wurzel bis zum Blatt, wider.

2.3.5. Effiziente unvollständige Constraintlöser

Es werden nun Constraintlöser vorgestellt, die polynomielle Laufzeit besitzen aber unvollständig sind. Dazu werden zunächst verfeinerte Begriffe für Domains eingeführt.

widersprüchlicher Domain, determinierter Domain

Definition: Ein Domain D ist ein *widersprüchlicher Domain*, wenn $D(X) = \emptyset$ für eine Variable des Domains gilt. Ein Wertebereich ist ein d*eterminierter Domain*, wenn die Domains aller Variablen $X_i \in \{X_1, ..., X_n\}$ einelementig sind, also $D(X_i) = \{d_i\}$ gilt. Ist $\{X_1, ..., X_n\}$ die Menge der Variablen eines Domains D und ist D determiniert, so ist $\theta = \{X_1 \mapsto d_1, ..., X_n \mapsto d_n\}$ mit $D(X_1) = \{d_1\}, ..., D(X_n) = \{d_n\}$ die zu D korrespondierende Variablenbelegung. Die Notation $D[X \mapsto \{d_1, .., d_n\}]$ definiert eine neue Domain D' mit $D'(X') = D(X')$ für $X' \neq X$ und $D'(X) = \{d_1, ..., d_n\}$. Es wird gesagt, dass $D_1 \subseteq D_2$ gilt, wenn beide Domains die gleiche Menge an Variablen besitzen und für alle Variablen $D_1(X) \subseteq D_2(X)$ gilt. Für eine Variablenbelegung θ und Domain D bedeutet $\theta \preceq D$, dass es eine determinierte Domain D' mit $D' \subseteq D$ gibt und θ die zu D' korrespondierende Variablenbelegung ist. Es werden auch destruktive Zuweisungen an Domains erlaubt, so dass durch $D(X) := M$ der Domain von X auf die Menge M gesetzt wird.

Beispiel: Für die Variablen X,Y ist der Domain D_1 mit $D_1(X) = \{1,2\}$, $D_1(Y) = \emptyset$ ein widersprüchlicher Domain und D_2 mit $D_2(X) = \{1\}$, $D_2(Y) = \{2\}$ ist ein determinierter Domain. Für D_3 mit $D_3(X) = \{1,2,3\}$, $D_3(Y) = \{2\}$ gilt für $D' = D_3[X \mapsto \{2\}]$, dass $D'(Y) = \{1,2,3\}$ und $D'(X) = \{2\}$ ist. Durch die Zuweisung $D(X) := \{1\}$ ist die Domain von X zukünftig gleich $\{1\}$.

Knoten- und Kantenkonsistenz

Die Idee bei den nun betrachteten Constraintlösern für CSPs ist, die Constraints zu benutzen, um die möglichen Domains von Variablen so früh wie möglich einzuschränken, so dass Werte, die zu keiner Lösung eines Constraints gehören können, eliminiert

```
C ist ein Constraint
D ist ein Domain

knoten_konsistent(C,D)
  C ist von der Form c_1 ∧ ... ∧ c_n
  for i:=1 to n do
    D' := primitiv_knoten_konsistent(c_i,D)
  endfor
  return D'

primitiv_knoten_konsistent(c,D)
  if vars(c) = {X} then
    return D[X ↦ {d ∈ D(X)|{X ↦ d} ist Lösung von c}]
  else
    return D
  endif
```

Abbildung 2.5.: Knotenkonsistenz-Algorithmus.

```
C ist ein Constraint ...
D, D' sind Domains

kanten_konsistent(C,D)
  C ist von der Form c_1 ∧ ... ∧ c_n
  for i:=1 to n do
    D' := primitiv_kanten_konsistent(c_i,D)
  endfor
  return D

primitiv_kanten_konsistent(c,D)
  V := vars(c)
  V ist von der Form {X_1, ..., X_n}
  for i:=1 to n do
   M_i := {d ∈ D(X_i)| es gibt eine Lösung θ von C mit θ ≺ D[X_i ↦
d]}
  endfor
  return D[X_1 ↦ M_1] ... [X_n ↦ M_n]
```

Abbildung 2.6.: Kantenkonsistenz-Algorithmus.

werden. So kann z.B. im Constraint $X < Y \wedge X, Y \in \{1,2,3\}$ mit einem Domain D mit $D(X) = \{1,2,3\},\ D(Y) = \{1,2,3\}\}$ der Domain von X auf $\{1,2\}$ und der Domain von Y auf $\{2,3\}$ reduziert werden. Bei der Knoten- bzw. Kantenkonsistenz bzgl. eines Constraints C wird zu einem Domain D gefordert, dass es für $X \in vars(C)$ und alle $d \in D(X)$ eine determinierte Domain D' existiert, so dass $D' \subseteq D[X \mapsto d]$ ist und die zu D' korrespondierende Variablenbelegung eine Lösung von C ist. Dabei stellen die beiden Konsistenzbegriffe unterschiedliche Anforderungen an die Anzahl der vorkommenden Variablen in den primitiven Constraints.

Knotenkonsistenz

Definition: Ein primitives Constraint c heißt unter einer Domain D *knotenkonsistent*, wenn entweder $|vars(c)| \neq 1$ gilt oder $vars(c) = \{X\}$ und für alle $d \in D(X)$ gilt, dass $\{X \mapsto d\}$ eine Lösung von c ist. Ein Constraint $c_1 \wedge ... \wedge c_n$ heißt knotenkonsistent, wenn jedes c_i $(1 \leq i \leq n)$ knotenkonsistent ist.

Kantenkonsistenz

Definition: Ein primitives Constraint c mit $vars(c) = \{X_1, \ldots, X_n\}$ heißt für einen Domain D *hyperkantenkonsistent*, wenn es für jede Variable $X \in vars(c)$ und für alle $d \in D(X_i)$ $(1 \leq i \leq n)$ einen determinierten Domain D' gibt, so dass $D' \subseteq D[X_i \mapsto d]$ gilt und die zu D' korrespondierende Variablenbelegung eine Lösung von c ist. Es wird im Folgenden einfach nur *kantenkonsistent*[6] gesagt. Ein Constraint $c_1 \wedge \ldots \wedge c_m$ heißt kantenkonsistent, wenn jedes c_i $(1 \leq i \leq m)$ kantenkonsistent ist.

Zu einem gegebenem Constraint und einer gegebenen Domain ist die Herleitung konsistenter Domains bzgl. der Knoten- bzw. Kantenkonsistenz in Abb. 2.5 und Abb. 2.6 gezeigt. Knoten- und Kantenkonsistenzalgorithmen nennt man allgemein *Constraintpropagierer*, da sie die lokalen Auswirkungen von Constraints bzgl. der Domains über die Hyperkanten an weitere Constraints propagieren.

Ein unvollständiger Constraintlöser auf der Basis von Knoten- und Kantenkonsistenz ist in Abb. 2.7 dargestellt. Dieser Constraintlöser ist deshalb unvollständig, da die Propagierung in einer nichtdeterminierten Domain resultieren kann. Bei einer nichtdeterminierten Variablenbelegung kann nicht generell auf globale Konsistenz aller primitiven Constraints geschlossen werden. Beispiel: Das Constraint $X \neq Y \wedge Y \neq Z \wedge X \neq Z \wedge X, Y, Z \in \{1,2\}$ ist zwar kantenkonsistent, besitzt aber keine Lösung. Ein vollständiger Solver kann aus der Kombination eines Backtracking basierten Constraintlösers und Constraintpropagierern konstruiert werden (siehe Abb. 2.8). Dabei wird jeweils in einem Suchschritt eine Propagierung durchgeführt und dann wieder eine Variable an einen konkreten Wert gebunden. Generell ist dabei der Suchraum aber immer noch exponenziell groß. Dabei dient die Kantenkonsistenz dem Beschneiden

[6] Die eigentliche Definition der Kantenkonsistenz fordert, dass die Menge der Variablen gleich zwei ist bzw. bei einer Variablenmenge ungleich zwei ein Constraint schon als kantenkonsistent gilt. Um die Einführung zu vieler Begriffe zu vermeiden, wird von Kantenkonsistenz hier immer im Sinne von Hyperkantenkonsistenz gesprochen.

```
C ist ein Constraint
D, D' sind Domains
arc_solv(C,D)
  D':=knoten_kanten_konsistent(C, D)
  if D' ist widersprüchlich then
    return false
  elseif D' ist determiniert then
    return erfüllbar(C, D')
  else
    return unbekannt
  endif


knoten_kanten_konsistent(C, D)
  D':= knoten_konsistent(C, D)
  D':= kanten_konsistent(C, D')
  return D'
```

Abbildung 2.7.: Constraintlöser auf der Basis von Knoten- und Kantenkonsistenz.

von sinnlosen Pfaden im Suchbaum und kann die Suche damit drastisch einschränken, so dass auch das Lösen von Problemen mit vielen Variablen noch möglich ist, wo der reine Backtracking basierte Constraintlöser versagen würde. In Abb. 2.8 ist der vollständige Suchbaum gezeigt, den ein Constraintlöser bei vollständiger Suche ohne Constraintpropagierung durchsuchen würde. Darunter sieht man unmittelbar den Effekt der Suchbaumbeschneidung durch die Constraintpropagierung. Die Zuweisung des Wertes 1 an die Variable X führt unmittelbar zur Beschneidung des linken Teilbaums und die Zuweisung von 2 an X zur Beschneidung des rechten Teilbaums.

Erweiterte Constraintpropagierer

Die Effizienz von Constraintpropagierern hängt stark von der Anzahl der Variablen in den primitiven Constraints und deren initialen Domains ab. Für unäre und binäre Constraints mit kleinen Mengen in den Domains kann die Prüfung auf Kanten-/Knotenkonsistenz der primitiven Constraints noch recht effizient ausgeführt werden. Betrachtet man aber ein primitives Constraint wie z.B.

$$X = 3Y - 5Z$$

mit einer initialen Belegung von $X, Y, Z \in \{1, \ldots, 100000\}$, so können die Konsistenzprüfungen sehr aufwendig werden. Generell ist das Problem der Kantenkonsistenz wiederum NP-hart [86]. Daher wurden schwächere Konsistenzbegriffe eingeführt, die auch Werte in den Domains enthalten, die keiner Lösung angehören, aber dafür effizient berechenbar sind. Dazu gehört unter anderem die *Schrankenkonsistenz (bounds consistency)* über arithmetischen Constraints. Diese wurde schon am Beispiel des Constraints $X < Y$ gezeigt: Die Idee ist dabei, die oberen und unteren Schranken der Wertebereiche sinnvoll einzuschränken. So weiß man bei $X < Y \ \wedge \ X, Y \in \{1, 2, 3\}$,

```
C ist ein Constraint
D, D' ist ein Domain
back_arc_solv(C,D)
  D':=knoten_kanten_konsistent(C, D')
  if D' ist widersprüchlich then
    return false
  elseif D' ist determiniert then
    return erfüllbar(C, D')
  endif
  selektiere variable X ∈ vars(C)
  foreach d ∈ D'(X) do
    if back_arc_solv(C ∧ X = d, D')then
      return true
    endif
  endfor
  return false
```

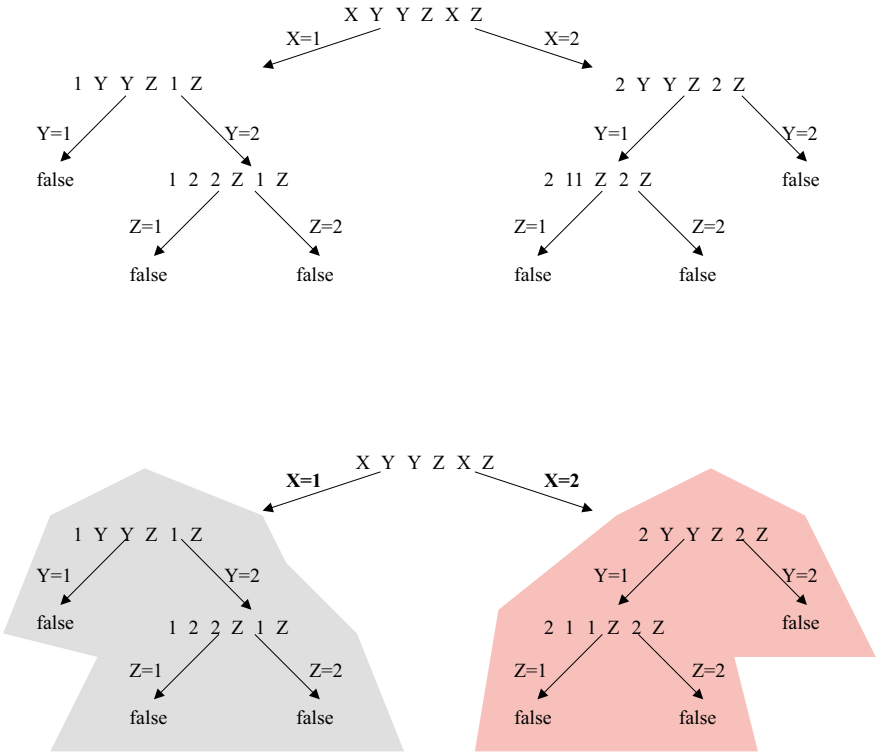


Abbildung 2.8.: Vollständiger Constraintlöser auf der Basis der Kantenkonsistenz und Beispiel für entsprechende Suchbaumbeschneidungen.

dass $min(\theta(Y))$ größer sein muss als $min(\theta(X))$ und dass $max(\theta(Y))$ größer sein muss als $max(\theta(X))$ wodurch die oberen und unteren Schranken der Wertebereiche von X und Y auf $\{X \mapsto \{1,2\}, Y \mapsto \{2,3\}\}$ eingeschränkt werden können. Es können auch wesentlich komplexere Propagierer betrachtet werden, die Zusammenhänge ganzer Mengen von Constraints betrachten und somit den Suchraum noch effektiver beschneiden können. Ein Beispiel ist das Constraint *alldifferent*$([X_1, \ldots, X_n])$, dass die wechselseitige Ungleichheit $X_i \neq X_j$ mit $i \neq j$ zwischen allen X_i, X_j fordert. Dabei werden auch globale Inkonsistenzen eher entdeckt als bei Spezifikation nur mit '$\neq$'.

Der Schwerpunkt bei der Entwicklung von Constraintlösern für CSPs ist die Entwicklung guter Propagierer sowie guter Kombinationen verschiedener Propagierer. So können für dasselbe primitive Constraint je nach Anzahl der ungebundenen Variablen und der verbleibenden Wertebereiche entweder Knoten-, Kanten- oder Schrankenkonsistenzalgorithmen durchgeführt werden. Weiterhin wichtig ist die Frage nach dem geeigneten Zeitpunkt der Aktivierung von Propagierern. Insgesamt muss ein gutes Verhältnis zwischen Suchraumbeschneidung und Overhead für die Ausführung der Propagierungsalgorithmen erlangt werden.

2.3.6. Optimierung von CSPs

Die hier vorgestellten Ansätze machen Gebrauch von den im letzten Unterkapitel vorgestellten Constraintlösern. In Abb. 2.9 ist ein Optimierer `retry_int_opt` gezeigt, der unter Anwendung eines vollständigen Constraintlösers `back_int_solv` sukzessive Lösungen generiert, die bzgl. einer Kostenfunktion k kontinuierlich verbessert werden. `back_int_solv` kann durch den Constraintlöser `back_arc_solv` aus Abb. 2.8 realisiert werden, der so erweitert werden muss, dass er eine gefundene Variablenbelegung zusätzlich als Resultat zurückgibt. Zunächst wird eine erste Lösung generiert und diese - falls sie existiert - als erste beste Lösung für die weitere Optimierung genommen. Alle weiteren Lösungen werden nun sukzessive unter der Hinzunahme des Constraints $k < \theta_{best}(k)$ generiert. So wird sichergestellt, dass die nächste Lösung (falls sie existiert) bzgl. der Kostenfunktion besser ist, als die zuletzt gefundene. Wird keine bessere Lösung mehr gefunden, terminiert das Verfahren.

Nachteil dieser Methode ist, dass gegebenenfalls viele der bereits ausgeführten Berechnungen in `back_int_solv` noch einmal durchgeführt werden, was auf die mehrfache Traversierung des Suchbaums zurückzuführen ist. In Abb. 2.9 ist daher ein zweiter Ansatz `back_int_solv` gezeigt, der dieses vermeidet und die Optimierung direkt in eine auf Backtracking basierenden Lösungssuche integriert (*Branch&Bound-Verfahren*), was "nur" einen Suchbaumdurchlauf erfordert. Dabei wird ein unvollständiger Constraintpropagierer `cp` eingesetzt, der z.B. im Kontext arithmetischer Constraints auf Schrankenkonsistenz (gegebenenfalls in Kombination mit der Kantenkonsistenz) basieren sollte. Der Einsatz von Propagierern bewirkt hier das vorzeitige Beschneiden vieler ungültiger Bereiche des Suchbaums, was zu einer erheblichen Reduktion des Suchraums führen kann.

```
C ist ein Constraint
D, D_det sind Domains
θ_best ist die beste Variablenbelegung
retry_int_opt(C,k,D)
  D_det :=back_int_solv(C,D)
  if D_det ist widersprüchlich then
    return false
  else
    θ ist die zu D_val korrespondierende Variablenbelegung
    retry_int_opt'(C,k,D,θ)
  endif

retry_int_opt'(C,D,k,θ_best)
  D_det:=back_int_solv(C∧k < θ_best(k),θ)
  if D_det ist widersprüchlich then
    return θ_best
  else
     θ ist die zu D_det korrespondierende Variablenbelegung
    return retry_int_opt'(C,D,k,θ')
  endif


back_int_opt(C,k,D, θ_best)
  D':=cp(C,D)
  if D' ist widersprüchlich then
    return θ_best
  elseif D' ist determiniert then
    θ ist die zu D' korrespondierende
    return θ
  endif
  selektiere X ∈ vars(C)
  foreach d ∈ D'(X) do
    c := k < θ_best(k) ∧ X = d
    θ_best :=back_int_opt(C ∧ c,k,D',θ_best)
  endfor
  return θ_best
```

Abbildung 2.9.: Ganzzahliger Optimierer auf der Basis der Reevaluierung (retry_int_op) und eine Branch&Bound-Methode mit integrierter Constraintpropagierung (back_int_op).

2.4. Constraint-Logikprogrammierung

Die Familie der Constraint-Logikprogrammiersprachen entstand Mitte der achtziger Jahre als eine Synthese von Constraintlösen und Logikprogrammieren [86]. In der Logikprogrammierung wird das zum Problem gehörende Wissen durch Regeln und Fakten spezifiziert. Die wohl bekannteste Umsetzung einer Logikprogrammiersprache ist Prolog [109, 26]. Die Einbettung von Constraintlösern in die Logikprogrammierung erlaubt wesentlich effizientere Suchmethoden, als sie durch die reine Logikprogrammierung gegeben sind. Im Gegenzug liefert die Logikprogrammierung einen natürlichen und ausdrucksstarken Sprachrahmen zur Formulierung von Constraints (wie z.B. CSPs) sowie zur Formulierung von Suchstrategien bei unvollständigen Constraintlösern.

2.4.1. Benutzerdefinierte Constraints

Die Logikprogrammierung erlaubt eine relativ natürliche Erweiterung um Constraints. Der Grund dafür liegt darin, dass Fakten und Regeln schon auf der Basis von Prädikaten formuliert werden. Weiterhin existiert auch hier schon das Konzept der bidirektionalen Anwendung von Prädikaten auf Variablen, so dass Argumente und Resultate beliebig wechseln können. Regeln ermöglichen eine elegante Definition neuer komplexer Constraints auf der Basis primitiver Constraints, so dass intuitivere und kompaktere Modelle von Problemen formuliert werden können.

Beispiel: Es wird ein hypothetischer DSP betrachtet, der addieren und subtrahieren kann und dessen Maschinenoperationen durch die folgenden Registertransferbefehle gegeben ist:

```
a1:=a1+c
a1:=b+c
a2:=b+a2
a2:=b+c
a1:=a1-c
a1:=b-c
```

Das Resultat einer Operation wird in eines der Registerfiles `a1` oder `a2` geschrieben. Der erste Operand wird in Registerfile `a1` oder `b` erwartet, der zweite Operand in Registerfile `a2` oder `c`. Bei der Addition muss sich beim Schreiben des Resultats in Registerfile `a1` der zweite Operand im Registerfile `c` befinden. Beim Schreiben des Resultats in Registerfile `a2` muss der erste Operand im Registerfile `b` sein. Bei der Subtraktion ist nur Schreiben in Registerfile `a1` erlaubt.

In der logischen Programmiersprache Prolog kann dieser Befehlssatz durch das Prädikat `mo(Def,Op,Arg1,Arg2)` mit den folgenden Fakten spezifiziert werden:

```
mo(a1,+,a1,c).
mo(a1,+,b,c).
```

```
mo(a2,+,b,c).
mo(a2,+,b,a2).
mo(a1,-,a1,c).
mo(a1,-,b,c).
```

Die Möglichkeit der symbolischen Notation für Registerfiles und Operationen erlaubt eine relativ intuitive und lesbare Spezifikation der Maschinenoperationen. Man kann nun im Prologsystem Anfragen der Form

```
?- mo(Def,+,a1,c).
```

stellen, worauf das System mit `Def=a1` antworten würde, wobei `Def` eine logische Variable ist. Die Notation '`?-`' macht dabei dem System bekannt, dass es sich um eine Anfrage handelt. Die Anfrage

```
?- mo(Def,Op,A1,a2).
```

würde das System mit `Def=a2,Op='+',A2=b` beantworten. Das System sucht selbst nach einem Fakt, das sich mit dem Fakt der Anfrage vereinen lässt. Eine elegantere Formulierung der MOs ergibt sich durch die Einbeziehung von Constraints und Regeln:

```
mo(Def,Op,A1,A2):-                          (R1)
   Def=a1, Op::[+,-], A1::[a1,b], A2=c.
mo(Def,Op,A1,A2):-                          (R2)
   Def=a2, Op='+', A1=b, A2::[a2,c].
```

Eine Regel hat die Form `A:-B` in der auch `A` der Kopf der Regel und `B` der Rumpf der Regel genannt wird. Die Notation `A1::[a1,b]` ist ein primitives Constraint und bedeutet $A1 \in \{a1, b\}$. Wie schon Fakten und Anfragen werden auch Regeln durch einen Punkt '.' abgeschlossen[7].

Die Regeln besagen, dass θ eine Lösung von $mo(X, O, Y, Z)$ ist, wenn θ entweder eine Lösung von Regel `R1` oder eine Lösung von Regel `R2` ist. Für Regel `R1` müsste θ also eine Lösung von

$$X = a1 \ \wedge \ O \in \{+, -\} \ \wedge \ Y \in \{a1, b\} \ \wedge \ Z = c$$

sein. Die Regeln können somit auch als Makros für bestimmte wiederkehrende Constraintmuster verstanden werden, indem im Rumpf die Vorkommen der Variablen des Regelkopfes durch die aktuellen Argumente substituiert werden. Diese Sichtweise wird im Folgenden noch modifiziert, da sie generell nicht gültig ist (s.u.).

Es wird nun eine Anfrage gestellt, ob die beiden Instruktionen eines C-Programmes

[7] Programme sowie Programmfragmente in Erläuterungen werden im Folgenden immer im Typewriter-Font notiert. Wird auf der Ebene der mathematischen Bedeutung von Constraints argumentiert, wird die kursive Notation gewählt.

```
i=x+y;
j=x+i;
```

in Maschinenoperationen des gegebenen Datenpfades umgesetzt werden können, ohne dabei Datentransfers einfügen zu müssen. Dieses Problem lässt sich durch die folgende Anfrage formulieren:

```
?- [I,J,X,Y]::[a1,a2,b,c],
   mo(I,+,X,Y),
   mo(J,+,X,I)
```

Diese Anfrage kann durchaus als CSP verstanden werden, wobei das Komma ',' als Konjunktion zu interpretieren ist. `[I,J,X,Y]::[a1,a2,b,c]` spezifiziert den initialen Domain und ist im Sinne von $I, J, X, Y \in \{a1, a2, b, c\}$ zu verstehen. Es gibt mehrere Möglichkeiten (genau vier), durch Anwendung der Regeln `R1` und `R2` die Anfrage in ein Constraint zu transformieren. Eine Möglichkeit ist die jeweilige Anwendung der Regel `R1` auf beide Vorkommen von `mo`, wodurch die Anfrage in das Constraint

$$\begin{aligned} &I = a1 \ \wedge \ + \in \{+, -\} \ \wedge \ X \in \{a1, b\} \ \wedge \ Y = c \quad \wedge \\ &J = a1 \ \wedge \ + \in \{+, -\} \ \wedge \ X \in \{a1, b\} \ \wedge \ I = c \end{aligned}$$

transformiert wird, so dass es jetzt nur noch aus primitiven Constraints besteht. Dieses Constraint ist nicht erfüllbar, da sich die primitiven Constraints $I = c$ und $I = a1$ widersprechen. Eine weitere Möglichkeit ist, beide Vorkommen von `mo` durch Anwendung der Regel `R2` zu ersetzen:

$$\begin{aligned} &I = a2 \ \wedge \ + \in \{+, -\} \ \wedge \ X = b \ \wedge \ Y \in \{a2, c\} \quad \wedge \\ &J = a2 \ \wedge \ + \in \{+, -\} \ \wedge \ X = b \ \wedge \ I \in \{a2, c\} \end{aligned}$$

Für dieses Constraint existieren die beiden gültigen Lösungen

$$\{I \mapsto a2, J \mapsto a2, X \mapsto b, Y \mapsto a2\}$$

und

$$\{I \mapsto a2, J \mapsto a2, X \mapsto b, Y \mapsto c\}$$

Diese unterscheiden sich nur in der Belegung der Variablen Y. Man sieht, dass auf der Basis von Regeln alternative Möglichkeiten eines Constraints formuliert werden können, die dann in alternativen Versionen des gesamten Constraints resultieren. Die Evaluierung von Constraint-Logikprogrammen muss daher über Mechanismen zur Suche über die möglichen Alternativen verfügen.

2.4.2. Programme auf der Basis von Regeln

Es wird nachfolgend die Syntax eines Constraint-Logikprogramms über einem gegebenen Constraintsystem spezifiziert.

Constraint-Logikprogramm, CLP-Programm

Definition: Gegeben ist ein Constraintsystem CS mit den Signaturen Σ_F und Σ_P. Weiterhin gegeben ist eine abzählbar unendliche Menge P_{bd} mit frischen Prädikatsymbolen. Ein *Constraint-Logikprogramm* über CS ist wie folgt definiert:

- Ein *benutzerdefiniertes Constraint* ist ein Ausdruck $p(t_1, ..., t_n)$, wobei $p \in P_{bd}$ und $t_1, ..., t_n \in T_{\Sigma_F}$. p wird *n-stelliges benutzerdefiniertes Prädikat* genannt und im Folgenden auch mit p/n notiert.
- Ein *Literal* ist ein primitives Constraint aus CS oder ein benutzerdefiniertes Constraint.
- Ein *Goal G* ist von der Form $L_1, ..., L_n$, wobei jedes L_i ein Literal ist. Für $n = 0$ sagen wir, dass *G* leer ist und notieren dies durch das Symbol $\Diamond$.
- Eine *Regel* ist von der Form $L :- G$, wobei L ein Literal ist und G ein Goal. L wird auch der Kopf der Regel und G der Rumpf der Regel genannt.
- Ein *Fakt* ist eine Regel mit leerem Rumpf, also $L :- \Diamond$. Dieses wird auch nur durch Angabe von L notiert.
- Ein *Constraint-Logikprogramm* ist eine Sequenz $R_1, \ldots, R_n$ von Regeln.

Die *Definition eines n-stelligen benutzerdefinierten Prädikates* p in einem Constraint-Logikprogramm ist die Menge der Regeln mit Kopf $p(t_1, ..., t_n)$. Es wird im Folgenden für ein Constraint-Logikprogramm auch nur kurz *CLP-Programm* gesagt.

Beispiel: `mo` ist ein 4-stelliges benutzerdefiniertes Prädikat $mo/4$, dessen Definition aus den zwei Regeln `R1` und `R2` besteht. Der Kopf von `R1` ist `mo(Def,Op,A1,A2),` und der Rumpf besteht aus den Literalen `Op::[+,-],Def=a1,A1::[a1,b],A2=c.` Die Regeln `R1` und `R2` definieren schon ein vollständiges Programm: `R1,R2`.

Anstelle des Begriffs "*benutzerdefiniertes Prädikat* p/n" wird im Folgenden auch einfach nur *Prädikat* p/n gesagt.

2.4.3. Evaluierung von Constraint-Logikprogrammen

Es wird nun eine operationale Semantik von CLP-Programmen vorgestellt. Die Berechnung einer Lösung eines CLP-Programms kann man sich als schrittweise Anwendung von Termersetzungsregeln auf die Literale eines Goals vorstellen.

Definition: Ein *Reduktionsschritt von Goal* G ist definiert durch eine Termersetzung eines Literals L_i in G derart, dass G von der Form

$$L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n$$

Reduktionsschritt

in ein Goal G' der Form

$$L_1, \ldots, L_{i-1}, t_1 = \rho(u_1), \ldots, t_n = \rho(u_n), \rho(L'_1), \ldots, \rho(L'_m), L_{i+1}, \ldots, Ln$$

transformiert wird, wobei L_i ein benutzerdefiniertes Constraint $p(t_1, ..., t_n)$ ist und $L'_1, ..., L'_m$ der Rumpf einer Regel R der Form $p(u_1, ..., u_n) :- L'_1, ..., L'_m$. Dabei ist ρ eine eindeutige Variablenumbenennung der Variablen aus R, die die Variablen in R durch *frische Variablen* ersetzt mit $vars(\rho(R)) \bigcap vars(G) = \emptyset$.

Die jeweilige Beziehung zwischen den aktuellen Argumenten - gegeben durch die t_i - und den Parametern aus R werden durch die Gleichungen $t_1 = \rho(s_1), ..., t_n = \rho(s_n)$ hergestellt. Eine reine Substituierung der Terme $t_1, .., t_n$ im Sinne einer Substitution ist im Allgemeinen nicht hinreichend, da die Rümpfe von Regeln Variablen besitzen können, die nicht in den Argumenten des Regelkopfes vorkommen. Darüber hinaus können die Argumente des Regelkopfes selbst auch wieder Terme sein, was ein Pattern-Matching mit Bindung an die entsprechenden Unterterme mit den aktuell übergebenen Argumenten erfordert.

Beispiel: Gegeben sei die Anfrage:

```
?- mo(I,+,X,Y), mo(J,+,X,I).
```

Durch Anwendung eines Reduktionsschrittes mit der Regel `R1` des Prädikates `mo` ergibt sich das folgende Goal:

$$\begin{gathered} I = Def', + = Op', X = A1', Y = A2', \\ Op' \in \{+, -\}, Def' = a1, A1' \in \{a1, b\}, A2' = c \\ mo(J, +, X, I) \end{gathered}$$

Im obigen Beispiel zeigen sich viele Redundanzen. Daher ist der Einsatz von Constraintvereinfachern sinnvoll, die ein Constraint in ein einfacheres, äquivalentes Constraint transformieren. So kann das obige Beispiel stark vereinfacht werden:

$$X \in \{a1, b\}, mo(J, +, X, a1)$$

Weiterhin ist es wichtig, die Nichterfüllbarkeit bei der Anwendung der Regeln so früh wie möglich zu entdecken. Man kann sich vorstellen, dass eine Anfrage aus einigen

hundert mo-Constraints besteht. Treten nun schon am Anfang erkennbare Inkonsistenzen in den bereits generierten primitiven Constraints auf, würde die Anwendung der restlichen Regeln einen unnötigen Overhead bedeuten. Inkonsistenzen sollten also so früh wie möglich erkannt werden, um alternative Regeln anwenden zu können. Daher wird die Evaluierung eines CLP-Programms durch sukzessive Anwendung von Reduktionsschritten und Constraintlösen definiert.

Zustände, Zustandsübergänge und Ableitungen für ein Goal G, Constraintstore

Definition: Ein *Zustand* in der Berechnung eines CLP-Programms ist gegeben durch $\langle G|C\rangle$ wobei G ein Goal ist und C ein Constraint. C wird hier auch der *Constraintstore* des Zustandes genannt. Ein *Zustandsübergang* $\langle G_1|C_1\rangle \rightarrow \langle G_2|C_2\rangle$ mit $G_1 = L_1, ..., L_n$ ist wie folgt definiert:

1. Wenn L_1 ein primitives Constraint ist, so ist $C_2 = C_1 \wedge L_1$.
   Ist $solv(C_2) = false$, so ist G_2 das leere Goal, sonst ist $G_2 = L_2, ..., L_n$.
2. Wenn L_1 ein benutzerdefiniertes Constraint ist, so ist $C_2 = C_1$ und G_2 ergibt sich durch die Anwendung eines Reduktionsschrittes auf G_1.

Eine *Ableitung eines Zustandes* $\langle G_0|C_0\rangle$ ist eine (möglicherweise unendliche) Sequenz von Zuständen

$$\langle G_0|C_0\rangle \rightarrow \langle G_1|C_1\rangle \rightarrow \langle G_2|C_2\rangle \rightarrow \ldots$$

Eine *Ableitung für ein Goal* G ist eine Ableitung für den Zustand $\langle G|true\rangle$. Eine *Ableitung* von G

$$\langle G|true\rangle \rightarrow ... \rightarrow \langle \diamondsuit|C_n\rangle$$

wird *erfolgreich* genannt, wenn $solv(C_n) = true$, die Ableitung wird *gescheitert* genannt, wenn $solv(C_n) = false$ ist. $\langle \diamondsuit|C_n\rangle$ heißt dann entsprechend *erfolgreicher* bzw. *gescheiterter Zustand*.

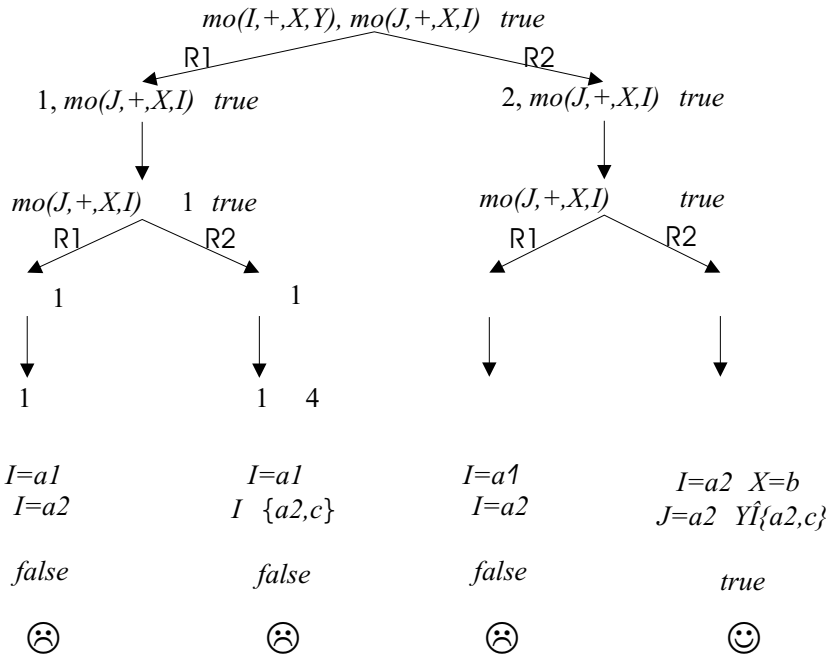
Ableitungsbaum

Auswahlpunkt (choice point), Backtrackingpunkt

Definition: Ein *Ableitungsbaum für ein Goal* G ist ein Baum, dessen Knoten Zustände sind. Die Wurzel des Baumes ist $\langle G|true\rangle$. Die Söhne jedes Knotens $\langle G_i|C_i\rangle$ sind alle Zustände, die mit einem Zustandsübergang von $\langle G_i|C_i\rangle$ erreicht werden können. Ein Knoten mit mehr als einem Sohn wird auch *Auswahlpunkt (choice point)* oder *Backtrackingpunkt* genannt. Knoten, deren erstes Literal in G ein primitives Constraint ist, besitzen nur einen Nachfolger. Söhne von Knoten, deren erstes Literal ein benutzerdefiniertes Constraint besitzen resultieren durch Anwendung einer Regel R aus der Definition des Prädikates. Es wird angenommen, dass die ausgehenden Kanten eines solchen Knotens gemäß der definierenden Regeln des Prädikates geordnet sind.

Beispiel: In Abb. 2.10 ist ein Ableitungsbaum für die Herleitung gültiger MOs für das Goal

$$mo(I, +, X, Y),\ mo(J, +, X, I)$$



1:= *I=a1* '+' {+,-} *X* {*a1,b*} *Y=a2*

2:= *I=a2* '+' {+,-} *X=b* *Y* {*a2,c*}

3:= *J=a1* '+' {+,-} *X* {*a1,b*} *I=a2*

5:= *J=a1* '+' {+,-} *X* {*a1,b*} *I=a1*

4:= *J=a2* '+' {+,-} *X=b* *I* {*a2,c*}

6:= *J=a2* '+' {+,-} *X=b* *I* {*a2,c*}

Abbildung 2.10.: Ableitungsbaum.

gegeben. Darin wurden die Transformationsschritte, die sukzessive Constraints zum Constraintstore hinzufügen, zu einem Schritt zusammengefasst. Weiterhin wurden die Argumente der benutzerdefinierten Constraints direkt in den Rümpfen der Regeln substituiert, was in diesem Beispiel legitim ist und den Ableitungsbaum erheblich verständlicher macht. Die Constraints, die durch jeweils einen Reduktionsschritt in das Goal eingefügt werden, sind jeweils durch ein Symbol $\bigtriangledown_i$ dargestellt, die unten in der Abbildung spezifiziert sind. Die Kanten des Baumes sind (im Falle von Auswahlknoten) mit der entsprechenden Regel, die für einen Reduktionsschritt angewandt wurde, markiert. Die Blätter des Ableitungsbaums sind mit den primitiven Constraints annotiert, die zu scheiternden bzw. erfolgreichen Zuständen führen. So führte z.B. nur die Anwendung der ersten Regel (R1) zum Scheitern des Goals, da sich hier der Widerspruch $I = a1 \wedge I = a2$ ergibt (erster Pfad).

Das Finden der Lösung eines Goals basiert auf der Traversierung des Ableitungsbaums. Dieser wird in einem *preorder Durchlauf* traversiert. Wird ein erfolgreicher Zustand erreicht, wird dieser dem Benutzer gemeldet. Dieser hat die Option, die Suche nach einer weiteren Lösung fortzusetzen. Der Ableitungsbaum wird nicht vorab konstruiert (was generell nicht realisierbar wäre), sondern simultan zur Traversierung erstellt. Beim Erreichen eines scheiternden Zustandes wird zu einem Auswahlknoten im Ableitungsbaum zurückgeschritten, an dem noch nicht alle Regeln angewandt wurden. Dieser Vorgang wird auch als *Backtracking (Rücksetzung)* bezeichnet.

2.4.4. CLP-Systeme

Die Evaluierung von Goals auf der Basis der Ableitungsbäume ist generisch bzgl. des zu Grunde liegenden Constraintsystems. Unterschiedliche CLP-Systeme können durch Austausch der unterliegenden Constraintlöser realisiert werden. Ein wesentliches Ziel bei der Entwicklung von CLP-Systemen ist die effiziente Implementierung der Constraintlöser und der Traversierung durch den Ableitungsbaum. Weiterhin relevant ist die Kombination verschiedener unterliegender Constraintsysteme innerhalb eines CLP-Systems.

die logische Programmiersprache Prolog

In dieser Arbeit liegt der Schwerpunkt der Betrachtungen auf CLP-Systemen über Tree-Constraints und Constraints über endlichen Wertebereichen. Einer der wesentlichen Gründe ist deren Eignung zur Modellierung des betrachteten Problembereichs. Das CLP-System über Tree-Constraints spielt eine besondere Rolle, da es das grundlegende Berechnungsmodell der logischen Programmiersprache *Prolog* widerspiegelt und somit einen allgemeinen Sprachrahmen zur Formulierung beliebiger Datenstrukturen und Algorithmen bietet. Es existieren effiziente Implementierungen für Prolog, wobei die Umsetzung der Tree-Constraintlöser auf der Basis intelligenter Unifikationsmethoden in den Reduktionsschritten integriert ist. Viele der existierenden CLP-Systeme sind Erweiterungen der Sprache Prolog, wobei die Unifikation um Constraintlösen ergänzt wurde. Ein guter Überblick über existierende CLP-Systeme ist in [86] gegeben.

Das ECLiPSe-System

Die in dieser Arbeit entwickelten Techniken wurden unter Verwendung des ECLiPSe-Systems [114, 103] implementiert. Dieses System ist ebenfalls eine Erweiterung der Sprache Prolog um Constraintlösen. Es existieren vordefinierte Bibliotheken, die Constraints, Constraintlöser und Optimierer über bestimmten Constraintsystemen zur Verfügung stellen. So bietet die FD-Bibliothek (*FD = finite domain*) vordefinierte Constraintlöser über endlichen Wertebereichen auf der Basis von Schranken- und Kantenkonsistenz und weiteren komplexen Propagierern. ECLiPSe bietet darüber hinaus eine äußerst flexible Umgebung zur Entwicklung eigener Constraintsysteme und Constraintlöser.

FD-Bibliothek von ECLiPSe

Zur Implementierung neuer Constraintlöser über endlichen Wertebereichen bietet die FD-Bibliothek von ECLiPSe Prozeduren (oder besser Prädikate), die expliziten Zugriff auf die aktuellen Wertebelegungen der Variablen ermöglichen und erlauben, diese zu modifizieren. Dadurch ist die Spezifikation von Constraintlösern und Optimierern innerhalb des Sprachrahmens der Constraint-Logikprogrammierung möglich. Neben dem Zugriff und der Modifikation der Domains von Variablen spielt das Konzept der *Delayed-Goals* eine große Rolle. Dies ermöglicht, Prädikate als Agenten auszuzeichnen, die unter bestimmten Bedingungen während der Programmausführung aktiviert und ausgeführt werden. Als Bedingung zur Aktivierung kann u.a. die Modifikation bestimmter Variablen spezifiziert werden. Delayed-Goals werden automatisch vom ECLiPSe-System aktiviert. Die FD-Bibliothek von ECLiPSe beinhaltet einen vordefinierten unvollständigen Constraintlöser auf der Basis der Constraintpropagierung, der das Konzept der Delayed-Goals ausnutzt. So kann man sich den Constraintstore als Menge wartender Agenten vorstellen, die bei der Modifikation bestimmter Variablen automatisch aktiviert werden und z.B. die Prüfung auf Kantenkonsistenz sowie die Reduktionen der Variablenbelegungen selbst vornehmen. Die automatische Aktivierung der Agenten entspricht dabei der Aktivierung des Constraintlösers bei einem Zustandsübergang im Ableitungsbaum. Ein großer Vorteil dieses Konzepts ist, dass sehr einfach eigene Constraints ergänzt werden können, ohne den Propagierer dadurch ändern zu müssen. Dies wird dadurch erreicht, dass benutzerdefinierte Prädikate und Constraints als Delayed-Goals deklariert werden können.

Delayed-Goals

Modellierungsbeispiel

Bei der Entwicklung eigener und vollständiger Constraintlöser über endlichen Bereichen kann man sich jetzt auf die Entwicklung von Strategien zur Wertebelegung der Variablen konzentrieren. Die Strategie legt dabei fest, in welcher Reihenfolge Variablen mit welchen Werten belegt werden und wird als *Labeling* bezeichnet. Eine Lösungsstrategie auf der Basis des Labelings wird im Folgenden auch *Labelingstrategie* genannt.

Lösungssuche durch Labeling

Labelingstrategie

Das ECLiPSe-System stellt das Prädikat `dom(X,D)` zur Verfügung, das zu einer Variablen `X` den Wertebereich `D` in Form einer Liste $[d_1, ..., d_n]$ zurückgibt, wobei die $d_1, ..., d_n$ aufsteigend sortiert sind. Der nachfolgend angegebene Constraintlöser, definiert durch das Prädikat $labeling/1$, traversiert eine vorgegebene Liste von Variablen und weist den Variablen Werte gemäß ihres Wertebereiches zu:

```
labeling([]).                              (L1)
labeling([X|Xs]):-                         (L2)
   dom(X,D),
   indomain(X,D),
   labeling(Xs).
indomain(X,[W|Ws]):-X#=W.                  (I1)
indomain(X,[_|Ws]):-indomain(X,Ws).        (I2)
```

Man beachte, dass es sich hier schon um einen vollständigen Constraintlöser handelt, der sich das Konzept von auf Agenten basierenden Propagierern zunutze macht. Die Propagierung wird durch das Constraint `X#=W` in Regel `I1` aktiviert, da dieses Constraint die Modifikation der Wertebelegung der Variablen `X` bewirkt. Das benutzerdefinierte Constraint $labeling/1$ muss selbst nur noch die Strategie zur Variablenbelegung definieren:

1. Regel `L1` besagt, dass bei leerer Variablenliste (notiert durch `[]`- Prolog-Notation für die leere Liste) nichts zu tun ist.
2. Sonst hat die Liste die Form `[X|Xs]`, wobei `X` das erste Element der Liste ist und `Xs` den Rest der Liste bezeichnet (Prolognotation für nichtleere Listen). In diesem Fall wird der aktuelle Wertebereich von `X` bestimmt und im Prädikat `indomain X` ein Wert zugeordnet.

Dabei besteht die Definition von $indomain/2$ wiederum aus zwei Regeln (I1,I2). Die erste Regel I1 bewirkt die Zuweisung des ersten Wertes aus der Liste der möglichen Werte. Die Regel bewirkt, dass X ein Wert aus dem Rest der Liste zugewiesen wird. Die Tatsache, dass kein Wert zur Belegung der Variablen zur Verfügung steht, muss nicht explizit spezifiziert werden. Ist die Liste der möglichen Wertebelegungen leer, so trifft keine Regel zu, was im Suchbaum zum Backtracking führt. Das Prädikat `indomain` ist zentral für die Suche über die Variablenbelegungen. Sollte eines der Constraints $X = W$ zu einem gescheiterten Zustand führen, so wird in einem Backtrackingschritt Regel I2 ausprobiert. Sollte kein Wert mehr zur Verfügung stehen, so wird zum nächstmöglichen Auswahlknoten zurückgeschritten, an dem Regel I2 noch nicht ausprobiert wurde. Das Prädikat $labeling/1$ entspricht dem vollständigen Constraintlöser aus Abb.2.8.

Es wird das folgende CSP betrachtet:

$$X < 2X + Z \ \wedge \ X = 3Z - Y \ \wedge \ Y < Z \ \wedge \ X, Y, Z \in \{1, \ldots, 10\}$$

Eine Anfrage zur Lösung dieses CSPs kann wie folgt aussehen:

```
?- [X,Y,Z]::1..10,
   X#<2*X+Z, X#=3*Z-Y, Y#<Z,
   labeling([X,Y,Z]).
```

Dabei liefert das System auf Wunsch sukzessive alle fünf Lösungen (durch Bestätigung der Anfrage `More` mit ';'):

```
X = 5  Y = 1 Z = 2 More? (;)
X = 7  Y = 2 Z = 3 More? (;)
X = 8  Y = 1 Z = 3 More? (;)
X = 9  Y = 3 Z = 4 More? (;)
X = 10 Y = 2 Z = 4
```

Zur Implementierung von Optimierern kann auf vorhandene Minimierer der FD-Bibliothek zugegriffen werden. Dabei handelt es sich um generische Optimierungsprozeduren, die neben einer Kostenfunktion über einem CSP auch Labelingstrategien als Parameter akzeptieren. Angenommen, das obige CSP soll bzgl. der Kostenfunktion $k(X,Y,Z) = K = -3X + Y + Z$ minimiert werden. Eine entsprechende Anfrage kann wie folgt aussehen:

```
?- [X,Y,Z]::1..10,
   X#<2*X+Z, X#=3*Z-Y,Y#<Z,
   K#=-3X+Y+Z
   minimize(labeling([X,Y,Z]),K).
```

Worauf das System die Lösung

```
Found a solution with cost -12
Found a solution with cost -16
Found a solution with cost -20
Found a solution with cost -24
X = 10
Y = 2
Z = 4
K= -24
```

ausgibt. Dabei wird auch das Finden einer besseren Lösung während der Suche in Form der Kosten protokolliert.

In der bisherigen Vorgehensweise spiegelt sich ein allgemeineres Konzept zur Entwicklung von Problemlösungen in der Constraint-Logikprogrammierung wider. Zu Beginn steht die Entwicklung eines CSP-Modells, welches dann die Objekte und deren Wechselbeziehungen eines Problems auf der Basis von Variablen und Constraints widerspiegelt. Darauf folgt die Entwicklung eines Constraintlösers und gegebenenfalls eines Optimierers, wobei gerade gute Labelingstrategien eine elementare Rolle spielen, da sie sich drastisch auf die Suchraumbeschneidungen auswirken können. Bei der Entwicklung kann auf vordefinierte Constraints, Constraintlöser und Optimierer zurückgegriffen werden. Diese können aber auch durch problemangepasste Lösungen ersetzt werden. Dabei ist es i.d.R. möglich, Labelingstrategien ohne irgendeine Modifikation des CSP-Modells auszutauschen. Die einfache Umsetzung der Labelingstrategie ermöglicht es, eine große Bandbreite an Strategien auszuprobieren. Es

wird im Verlauf der Arbeit noch detailliert auf die effizienzsteigernden Möglichkeiten eingegangen.

Im weiteren Verlauf der Arbeit werden die Variablen in Constraints über endlichen Wertebereichen *Domainvariablen* genannt, um Begriffsverwechslungen mit Variablen in zu übersetzenden Programmen zu vermeiden .

Domainvariablen

2.5. Zusammenfassung

Es wurden zunächst Begriffe grundlegender Komponenten und Begriffe zur Spezifikation und Darstellung von Befehlssätzen eingeführt. Von zentraler Bedeutung in dieser Arbeit sind die faktorisierten Maschinenoperationen, die eine Repräsentation alternativer Maschinenoperationen bieten. Alternativen werden durch die für eine Operation alternativ zur Verfügung stehenden Ressourcen dargestellt, wie z.B. alternative Registerfiles für den lesenden Zugriff auf Operanden.

Die Struktur von Compilern wurde erläutert und die einzelnen für diese Arbeit zentralen Phasen der Codegenerierung dargelegt: Instruktionsauswahl, Registerallokation und Instruktionsanordnung. Die unterschiedlichen Kategorien der Zwischenrepräsentationen wurden gezeigt. Dabei wurde grob zwischen den maschinenunabhängigen Zwischenrepräsentationen (IR genannt) und den maschinenabhängigen Zwischenrepräsentationen (mit LIR bezeichnet) unterschieden. Zur Abgrenzung von Operationen der IR und der LIR wurden erstere mit dem Begriff abstrakte Operationen und letztere durch die Begriffe (faktorisierte) Maschinenoperationen oder Operationen der LIR bezeichnet. Gerade die LIR spielt eine grosse Rolle bei der Adaption neuer Techniken, da sie die Schnittstelle zum Austausch der Zwischenstufen des Maschinenprogramms zwischen den einzelnen Phasen darstellt. Sie sollte daher ein einheitliches Format aufweisen, das Informationen zur Unterstützung aller Codegenerierungsphasen umfasst.

Constraints bieten eine elegante und vielseitige Methode zur Formulierung komplexer Probleme mit vielen wechselseitigen Beziehungen. Für bestimmte Constraintsysteme existieren effiziente Constraintlöser und Optimierer mit polynomieller Laufzeit. Generell handelt es sich aber schon beim Lösen von Constraints um ein mindestens NP-vollständiges Problem [86]. Viele Verfahren zum Finden von Lösungen basieren daher auf einer Suche mit Backtracking. Eine Klasse vollständiger Constraintlöser, die unter die Klasse der NP-harten Probleme fallen, sind die über den endlichen Wertebereichen [86]. Es existieren aber effiziente unvollständige Verfahren auf der Basis von der Constraintpropagierung, die aber nicht immer eine Lösung liefern. Das Konzept dieser Propagierer ist, sukzessiv Werte aus den Variablenbelegungen primitiver Constraints zu eliminieren, die keiner Lösung angehören können. Die Kombination von Constraintpropagierern und auf Backtracking basierter Suche führt zu Verfahren, die zwar potenziell immer noch exponenzielle Laufzeit haben, aber durch die Propagierer eine drastische Beschneidung des Suchraums bewirken, so dass immer noch sehr grosse Probleme in akzeptabler Zeit handhabbar sein können.

Die Constraint-Logikprogrammierung ist eine Synthese aus Constraint- und Logikprogrammierung und bietet einen noch eleganteren und mächtigeren Sprachrahmen

zur Formulierung komplexer Constraintmengen. Es besteht die Möglichkeit der Spezifikation von benutzerdefinierten Constraints sowie der Möglichkeit, explizit Nichtdeterminismus auszudrücken, was nur auf der Basis von Constraints sehr schwierig ist und zu schwer zu durchschaubaren Constraints führt. Es wurde ein generisches Modell zur Evaluierung von Constraint-Logikprogrammen definiert, das die operationale Semantik von Constraint-Logikprogrammen über beliebigen Constraintsystemen widerspiegelt. Als Besonderheit ist herauszustellen, dass die Entwicklung eigener Constraintlöser und Optimierer selbst wieder im Sprachrahmen der Constraint-Logikprogrammierung möglich ist, wobei gerade die Formulierung von suchbasierten Verfahren innerhalb dieses Sprachrahmens sehr einfach und elegant ist.

Das CLP-System ECLiPSe wird zur Implementierung der Techniken in dieser Arbeit verwendet. Dieses System ist eine Erweiterung der Sprache Prolog um Constraintlösen. Es existieren vordefinierte Bibliotheken, die Constraints, Constraintlöser und Optimierer über bestimmten Constraintsystemen zur Verfügung stellen. Zentral für diese Arbeit ist die FD-Bibliothek (FD=finite domain) mit vordefinierten Constraintlösern über endlichen Wertebereichen auf der Basis von Schranken- und Kantenkonsistenz und weiteren komplexen Propagierern. Es bestehen weiterhin flexible und mächtige Konzepte zur Entwicklung und Implementierung eigener Constraints, Constraintlöser und Optimierer über endlichen Wertebereichen. ECLiPSe ist generell zur Entwicklung beliebiger Constraintsysteme und Constraintlöser sowie zur Integration verschiedener Lösungsmethoden zu hybriden Methoden konzipiert.

Die in diesem Kapitel eingeführten Begriffe und Konzepte sollten ausreichen, um auch ohne Vorkenntnisse des Compilerbaus, der Constraintprogrammierung bzw. Logikprogrammierung, die hier entwickelten Techniken zu verstehen. Weitere Hintergründe können den folgenden weiterführenden Literaturhinweisen entnommen werden. Allgemeine Informationen zum Aufbau von Rechnern, Prozessoren und Befehlssätzen können aus [58, 85, 56] entnommen werden. Das wohl bekannteste Werk im Bereich des Compilerbaus ist das “*Drachenbuch*” [3]. Mittlerweile sind aber auch Bücher herausgegeben worden, die den doch mittlerweile veralteten Stand dieses Buches aktualisieren. Hier sind die folgenden Werke besonders zu empfehlen: [118, 40, 93, 6, 92]. In [8] ist ein Überblick weiterführender Techniken des traditionellen Compilerbaus gegeben, der auch eine Analyse der Nachteile traditioneller Arbeiten im Kontext irregulärer Prozessorarchitekturen umfasst. Eine sehr gute Einführung in die Constraintprogrammierung und die Constraintlogikprogrammierung ist in [86] gegeben. Neben den formalen Grundlagen finden sich hier auch viele Beispiele zur Modellierung in CLP sowie zur Effizienzsteigerung von Constraintlösern und Optimierern. Weiterhin ist ein umfassender Überblick über Systeme zur Constraintprogrammierung gegeben, d.h. zum einen CLP-Systeme aber auch Constrainterweiterungen anderer Programmiersprachen. Die formalen Grundlagen des gesamten Themenbereichs der Logikprogrammierung und der Constraintprogrammierung können in [44] und [59] gefunden werden. Das ECLiPSe-System und weitere Informationen zu ECLiPSe sind unter [103] zu finden. Hier sind auch weitere Referenzen auf Literatur und andere CLP-Seiten im Internet gegeben. Eine sehr gute Einführung in die Eigenschaften und die Konzepte des ECLiPSe-Systems zur Entwicklung hybrider Constraintlöser und Optimierer ist in [114] gegeben.

3. Problemanalyse

Dieses Kapitel beginnt mit einer Analyse der Anforderungen an Codegenerierungstechniken, die notwendig sind, um die Eigenschaften von DSPs effektiv unterstützen zu können. Danach werden die Techniken des traditionellen Compilerbaus dahingehend untersucht, inwieweit hier noch Eigenschaften von DSPs unterstützt werden und wo die wesentlichen Mängel bestehen. Im Anschluss daran wird der aktuelle Stand der Forschung in der Entwicklung von Codegenerierungstechniken und Compilersystemen für eingebettete Prozessoren reflektiert. Aus den sich daraus ergebenden Defiziten bzgl. der analysierten Anforderungen ergeben sich die primären Zielsetzungen dieser Arbeit.

3.1. Problemstellungen für DSP-Compiler

Es werden die Eigenschaften von DSPs erläutert, die die Codegenerierung für DSPs so schwierig machen. Dies wird zunächst an einem Beispiel illustriert. In Abb.3.1 a) ist ein Datenflussgraph und eine der möglichen optimalen Umsetzungen in Assemblercode für die ADSP2100-Familie gezeigt, worin die wesentlichen Eigenschaften der Prozessorfamilie ausgenutzt werden.

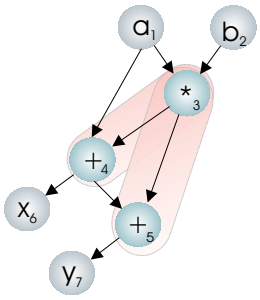
Die Knotenmarkierungen des Datenflussgraphen (im Bsp. Programmvariablen oder Operatoren) sind zur besseren Referenzierung mit eindeutigen Indizes markiert (z.B. a_1 oder $'+'_4$). Die mit den Knoten assoziierten Maschinenoperationen sind mit den korrespondierenden Indizes markiert (mo_i). Bei komplexen Maschinenoperationen ist die Menge der Knoten, die die Maschinenoperation umfasst, gegeben: So fasst z.B. $mo_{3,4}$ die Knoten $*_3$ und $+_4$ zu einer komplexen MAC-Operation[1] zusammen. Die Parallelausführung von Maschinenoperationen wird, wie schon in Abb. 1.7, durch $mo_1||...||mo_n$ notiert. Eine Datentransfer-Operation zwischen Knoten mit Indizes i und j wird durch $mo_{i\rightarrow j}$ notiert. Variablenknoten sind zum einen Blätter in den Graphen aber zum anderen auch Wurzeln, die zunächst[2] das Speichern der Programmvariablen denotieren (im Beispiel x_6 und y_7).

In Abb. 3.1 b) ist zunächst der zum Datenflussgraphen korrespondierende Maschinencode ohne Initialisierungscode für Adressregister gezeigt. Außerdem sind darin nur Registerfiles, aber noch keine konkreten Register zugeordnet worden. In dem unterlegten Kästchen (auf der rechten Seite) sind zu den LOAD- und STORE-Operationen

[1] MAC=multiply accumulate (siehe auch Kapitel 1.3)

[2] Später werden sie zur Annotation der möglichen ST-Komponenten verwendet, in denen sich Werte bei Austritt aus einem Basisblock befinden dürfen.

a)



b)

mo_1:mx:=load(i1,d) || mo_2:my:=load(i2,p)	mx:=D[&a]	my:=P[&b]
$mo_{1\ 4}$:mr:=mx		
$mo_{4,3}$:mr:=mr+mx*my		
$mo_{5,3}$:mr:=mr+mx*my || mo_6:d:=store(i1,mr)	D[x]:=mr	
mo_7:d:=store(i1,mr)	D[y]:=mr	

c)

	d-Speicher	p-Speicher
0	a	b
1	x	
2	y	

d)

m0:=1	Initialisierung der Adressregister	
m1:=1		
i1:=&a		
i2:=&b		
mo_1:mx:=load(i1,d) || mo_2:my:=load(i2,p)	i1+=mo1	i2+=mo2
$mo_{1\ 4}$:mr:=mx		
$mo_{4,3}$:mr:=mr+mx*my		
$mo_{5,3}$:mr:=mr+mx*my || mo_6:d:=store(i1,mr)	i1+=mo1	
mo_7:d:=store(i1,mr)		

Abbildung 3.1.: Codegenerierung für Datenflussgraphen.

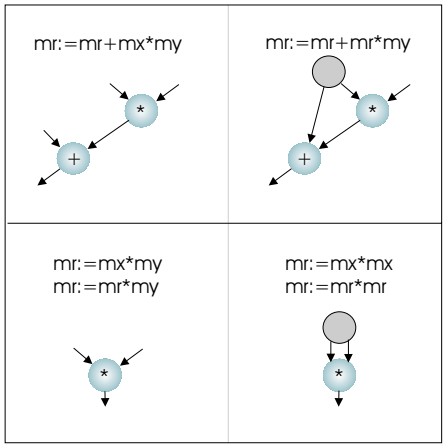


Abbildung 3.2.: Maschinenoperationsmuster der ADSP2100-Familie.

korrespondierende Speicherzugriffe in abstrakterer Form wiedergegeben (um die Lesbarkeit des Codes zu erhöhen). Man erkennt hier deutlich, dass sowohl Parallelausführung und die komplexe MAC-Operation der ADSP2100-Familie ausgenutzt werden. In Abb. 3.1 c) wird aus der Zugriffsreihenfolge der Programmvariablen und deren Zuordnung zu Speicherbänken ein Adresslayout bestimmt, so dass die Autoinkrementoperationen der AGU-Einheiten ausgenutzt werden können. Der notwendige Initialisierungscode und die erzeugten Autoinkrementoperationen sind in Abb. 3.1 d) ergänzt worden. Im Folgenden werden die zentralen Anforderungen an Compiler für DSPs herausgestellt und am gezeigten Beispiel im Detail verdeutlicht.

Graphbasierte Instruktionsauswahl

Das Ausnutzen komplexer Maschinenoperationen, wie z.B. $mo_{4,3}$ und $mo_{5,3}$ in Abb. 3.1, erfordert spezielle Techniken zum Erkennen von Graphmustern, die über die Möglichkeiten von baumbasierten Verfahren hinausgehen. Es lassen sich folgende wichtige Ausprägungen bzgl. der Maschinenoperationsmuster unterscheiden:

- Generelle Handhabung von komplexen, aber noch baumartigen Maschinenoperationsmustern in der Graphstruktur. Im Beispiel muss erkannt werden, dass der Knoten $*_3$ mit Knoten $+_4$ und $+_5$ jeweils auf eine MAC-Operation abgebildet werden kann.
- Die Erkennung von echten Graphmustern in den Datenflussgraphen, wie z.B. x^2 bzw. die SQUARE-Operation oder die MAC-Operation, deren erster und zweiter Operand identisch sein dürfen. In Abb. 3.2 sind die Pendants von baumbasierter und graphbasierter MAC- und SQUARE-Operation gezeigt.
- Die Erkennung von Maschineninstruktionen, also meist unzusammenhängender, aber parallelausführbarer Graphmuster. Hier wird schon partiell eine Instruktionsanordnung durchgeführt, da die Zuordnung von Maschinenoperationen zu Maschineninstruktionen eher der Instruktionsanordnung zugeordnet wird.

Somit kann dieser Aspekt auch als integrierte Instruktionsauswahl und Instruktionsanordnung bezeichnet werden. Im Beispiel könnte man die Erkennung der parallel auszuführenden LOAD-Operationen mo_1 und mo_2 entsprechend umsetzen. Die Erkennung von SIMD[3]-Instruktionen fällt ebenfalls in diese Kategorie (siehe auch [71]).

Handhabung irregulärer Registerfiles

Dies erfordert eine sorgfältige Zuordnung von Programmvariablen zu Registerfiles. Dabei sind folgende Aspekte zu berücksichtigen:

- Eine gute Verteilung von Programmvariablen auf Registerfiles vermeidet, dass Werte gespillt[4] werden müssen. Es besteht eine starke Wechselbeziehung zur Instruktionsauswahl, da die Auswahl der Maschinenoperationen die Verwendung bestimmter Registerfiles impliziert. So muss ein Kompromiss zwischen der Auswahl guter Maschinenoperationen und Spillcode getroffen werden.

Datentransferpfade

- Die Transportwege - d.h. die *Datentransferpfade* - zwischen verschiedenen Registerfiles sind zu berücksichtigen, wenn z.B. derselbe Operand in unterschiedlichen Funktionseinheiten konsumiert wird (nur bei gemeinsamen Teilausdrücken - CSEs[5]). Wenn diese Funktionseinheiten keine gemeinsamen Eingangsregister besitzen, muss der Wert durch Datentransfers in die erforderlichen Registerfiles bewegt werden. In Abb. 3.1 wird der Wert der Variablen a sowohl in Registerfile mr als auch in mx verlangt, um beide MAC-Operationen korrekt ausführen zu können. Dabei ist ein Register-Register-Transfer ($mo_{1\rightarrow 4}$) günstiger als ein erneuter Zugriff auf den Speicher. Auch hier kommt wieder die Wechselbeziehung zur Instruktionsauswahl zum Tragen. Würde die Instruktionsauswahl die Kosten für Datentransfers ignorieren, könnte sie in Abb. 3.1 den Wert für Programmvariable b in mo_2 z.B. auch willkürlich in Registerfile ay laden und es wäre ein weiterer Datentransfer für die Benutzung von b in Registerfile my in $mo_{4,3}$ notwendig. Bei der Allokation der Registerfiles muss zudem darauf geachtet werden, dass überhaupt Datentransferpfade existieren.

Ausnutzung der Parallelität

Eine der zentralen Aufgaben ist das Finden guter Kombinationen von Maschinenoperationen zu Maschineninstruktionen, die zu möglichst kompaktem Maschinencode führen und allen Randbedingungen genügen müssen. Dies stellt wiederum sehr hohe Anforderungen an die Instruktionsanordnung:

[3] SIMD=single instruktion multiple data.

[4] Unter Spilling versteht man das Auslagern von Registerinhalten in den Speicher, falls die Registerkapazitäten nicht alle lebendigen Werte aufnehmen können. An den Stellen im Programm, an dem der ausgelagerte Wert benutzt wird, muss entsprechender Code eingefügt werden, der den Wert zunächst wieder in das Register einliest.

[5] CSE=common subexpression (siehe Kapitel 2.2.2).

- Ein sorgfältiges Binden von Ressourcen ist notwendig, da ansonsten die starken Restriktionen der parallelen Ausführung von Maschinenoperationen verletzt werden können.

- Gerade die starken Restriktionen bzgl. der Kombination von ausschließlich spezifischen Maschinenoperationen mit spezifischen Ressourcenbindungen stellen die Instruktionsanordnung in starke Wechselbeziehung mit der Instruktionsauswahl und der Registerallokation. Ist hier schon eine ungünstige Auswahl getroffen worden, kann dies den Verlust an potenziell vorhandener Parallelität bedeuten. So ist die Bindung der parallelen Datentransfers von mo_1 und mo_2 an die Zielregister mx und my unbedingt notwendig. Die mögliche Parallelausführung von Datentransfers stellt daher noch zusätzliche Anforderungen an die Berechnung von Spillcode. Die Parallelausführung von Datentransfer-Operationen macht die Vermeidung von Spillcode nicht einmal zwingend notwendig. Somit ist die Integration von Instruktionsanordnung, Registerallokation und Instruktionsanordnung von sehr großer Bedeutung.

- Die speziellen Adressoperationen der AGU-Einheiten müssen ausgenutzt werden. Die Auswahl dieser Operationen ist nun abhängig von der Verteilung von Variablen auf die Speicherbänke. In Abb. 3.1 hängt die Ausnutzung des parallelen Ladens (mo_1 und mo_2) von der Allokation der Programmvariablen a im d-Speicher und der Variablen b im p-Speicher ab. Weiterhin hat sich herausgestellt, dass ein gutes Adresslayout der Variablen im Speicher die Verwendung von Autoinkrement- und Autodekrementoperationen begünstigt. In Abb. 3.1 c+d) wird gezeigt, dass durch die Organisation der Variablen a,x und y gemäß der Zugriffsreihenfolge (a,y,x) die Ausnutzung von Autoinkrementoperationen ideal genutzt werden kann. Die Zuordnung zu Speicherbänken und das Adresslayout hängen von der Instruktionsanordnung ab, so dass diese zuvor ausgeführt werden sollte.

Ausnutzung spezifischer Kontrollstrukturen

Das Erkennen von neuen Kontrollflussoperationen stellt wiederum eine Menge neuer Anforderungen. Zum einen spielen auch hier wechselseitige Abhängigkeiten zwischen den Phasen eine Rolle. Zum anderen müssen sehr spezifische Schleifenstrukturen mit sehr spezifischem Code im Schleifenrumpf vorliegen, um bestimmte Kontrollflussoperationen ausnutzen zu können. Letzteres stellt vollkommen neue Anforderungen an das Erkennen solcher Strukturen. In dieser Arbeit werden Techniken zur Ausnutzung solch spezifischer Kontrollflussoperationen nur am Rande behandelt.

Möglichst optimale Resultate

Auch geringe prozentuale Verbesserungen können hier zum einen das Einhalten von Echtzeitanforderungen bewirken oder auf der anderen Seite Ressourcen einsparen (z.B. durch Verringerung der Programmspeichergröße). Das Problem ist, dass kombinatorische Suchprobleme vorliegen, die mindestens NP-vollständig sind, und die

vielen Randbedingungen das Finden guter Heuristiken kaum noch ermöglichen. Bei irregulären Prozessoren besteht sogar grundsätzlich die Frage nach der Existenz von Maschinencode, der das Programm realisieren kann.

Schnelle Adaption

Zur schnellen Verfügbarkeit neuer Compiler wären automatisch retargierbare Compiler wünschenswert. Da diese aber viele Kompromisse eingehen müssen, ist der resultierende Maschinencode nicht für alle Prozessoren von gleich guter Qualität. Es scheint realistischer zu sein, retargierbare Techniken für Klassen von Prozessoren zu entwickeln. Selbst hier lassen sich weitere Untergliederungen vornehmen, so dass nicht der ganze Compiler für eine Klasse von Prozessoren ausgelegt wird, sondern Codegenerierungsphasen auf bestimmte Klassen von Prozessoreigenschaften ausgerichtet werden. So können bei Überschneidungen bestimmter Eigenschaften unterschiedlicher Prozessorklassen die Techniken, die für die gemeinsamen Eigenschaften entwickelt wurden, wiederverwendet werden. Und selbst hier sollte die Möglichkeit gegeben sein, neue Techniken einbinden zu können, um neuen Entwicklungen gerecht zu werden. Neben retargierbaren Techniken, die auf spezifische Eigenschaften von Prozessoren ausgerichtet werden, spielt daher die Erweiterbarkeit und eine einfache Neukonfiguration der Phasen eine sehr große Rolle bei der Entwicklung von Compilern.

3.2. Kernproblematiken

Aus den Anforderungen lassen sich die folgenden Kernprobleme ableiten, deren Lösung im weiteren Verlauf des Kapitels zur Bewertung von Lösungsansätzen herangezogen werden:

- Um Eigenschaften von DSPs effektiv zu unterstützen, sind Techniken notwendig, die
  - graphbasierte Mustererkennung von Maschinenoperationen unterstützen,
  - neben der Verteilung von Werten auf Registerfiles auch die notwendigen Datentransfers berücksichtigen und
  - die Parallelität ausnutzen.

  Darüber hinaus müssen die wechselseitigen Abhängigkeiten zwischen diesen Anforderungen berücksichtigt werden. Daraus ergibt sich die Forderung nach phasengekoppelten Techniken, die die Instruktionsauswahl, Registerallokation und Instruktionsanordnung sinnvoll integrieren.
- Die vielen Randbedingungen von irregulären Prozessorarchitekturen sowie die Forderung nach der Phasenkopplung von Techniken machen das Finden guter heuristischer Methoden sehr schwierig. Da es sich um kombinatorische Suchprobleme handelt, besteht ein Bedarf an hocheffizienten suchbasierten Verfahren. Dabei muss ein guter Kompromiss zwischen Güte und Effizienz gefunden

werden. Es stellt sich die Frage nach Methoden, die die Spezifikation und die Implementierung der Phasenkopplung handhabbar machen, da der Einsatz konventioneller Programmiermethoden hier an seine Grenzen stößt.

- Die Adaption von Techniken an neue Prozessoren erfordern zum einen möglichst retargierbare Techniken, zum anderen aber auch die Erweiterbarkeit eines Compilers. Zur Einhaltung der Codequalität ist es notwendig, klassenspezifische retargierbare Techniken zu entwickeln. Darüber hinaus verlangt die Erweiterbarkeit nach einem transparenten Modell einer IR im Bereich der Low-Level-Optimierungen. Von zentraler Bedeutung ist ein Modell der Maschinenoperationen und Maschineninstruktionen, auf das möglichst alle Phasen der Codegenerierung aufsetzen können, was auch in Hinblick auf die Entwicklung phasengekoppelter Techniken von zentraler Bedeutung ist.

3.3. Compiler für allgemeine Prozessoren (GPPs)

Die Probleme, die sich ergeben, Techniken des traditionellen Compilerbaus zur Codegenerierung für DSPs mit irregulären Datenpfaden einzusetzen, wurden in einer Studie erarbeitet [8]. Im Folgenden wird ein kurzer Überblick der wesentlichen Forschungsschwerpunkte des traditionellen Compilerbaus gegeben. Es werden die wesentlichen Problematiken gezeigt, die bzgl. der Codegenerierung für DSPs gelöst bzw. nicht gelöst werden. Weitere Details können der oben genannten Studie aus [8] entnommen werden.

3.3.1. Instruktionsauswahl

Der Schwerpunkt in der Entwicklung von Techniken zur Instruktionsauswahl liegt in der Entwicklung retargierbarer baumbasierter Techniken auf der Basis des *Tree-Parsings* ([36, 118]). Dazu werden Datenflussgraphen von Basisblöcken zunächst in eine Menge von Datenflussbäumen zerlegt. Dies wird erreicht, indem der Datenflussgraph an den CSEs aufgebrochen wird (siehe Abb. 3.3). Dabei wird für jede Benutzung einer CSE ein korrespondierender Variablenknoten eingefügt. In Abb. 3.3 führt dies zur Duplizierung des Knotens a_1 zu a_1 und a_{12} [6]. Für Knoten $*_3$ ist die Neueinführung der zu speichernden Programmvariablen n (Knoten n_8) und der beiden lesenden Zugriffe durch Knoten n_9 und n_{11} notwendig. Weiterhin wird die Duplizierung des Knotens x_6 zu x_6 und x_{10} erforderlich.

Der Vorteil dieser Verfahren liegt darin, dass für Datenflussbäume lokal optimaler Code in linearer Zeit generiert werden kann. Dabei basieren die Techniken auf *Tree-Pattern-Matching* und *dynamischer Programmierung* (siehe [2, 1, 118]). Ein weiterer Vorteil der Verfahren ist, dass sie auf der Basis von formalen Beschreibungssprachen (den regulären Baumgrammatiken, die eine Teilmenge der kontextfreien Grammatiken sind) durch Instruktionsauswahlgeneratoren automatisch erzeugt werden kön-

Tree-Pattern-Matching, dynamische Programmierung

[6] Es sei noch einmal bemerkt, dass die Indizes zur Bezeichnung der einzelnen Knoten dienen und nicht Bestandteil der Variablenbezeichner sind.

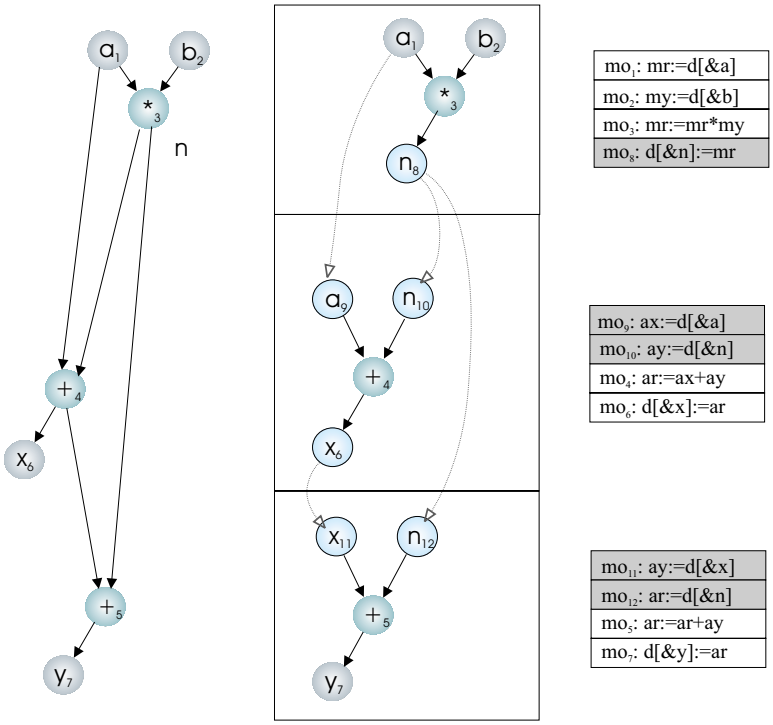


Abbildung 3.3.: Probleme baumbasierter Instruktionsauswahl.

nen. Beispiele für solche Werkzeuge sind *BEG [32]*, *Twig [1]*, *burg [41]*, *iburg* [42] und OLIVE [7].

Vorteile dieser Methode bzgl. der Anforderungen von DSPs:

- Komplexe Baummuster werden unterstützt.
- Es werden korrekte, kostengünstigste Datentransferpfade innerhalb der Bäume ausgewählt.
- Eine schnelle Adaption an Zielarchitekturen ist gegeben, da die Spezifikation von Befehlssätzen auf der Basis regulärer Baumgrammatiken sehr einfach und intuitiv ist.
- Baumgrammatiken lassen sich aus hardwarenahen Beschreibungen extrahieren [69].

Viele der Nachteile und Probleme des Verfahrens resultieren im Wesentlichen durch die Zerlegung von Datenflussgraphen in Bäume:

- Grundsätzlich können keine wirklichen Graphmuster - wie z.B. x^2- mehr erkannt werden. Vielmehr wird ein Muster x*y erkannt, wobei der Operand dann in zwei unterschiedliche Register untergebracht wird (wodurch ein zusätzlicher Datentransfer notwendig ist).

- CSEs können nicht mehr als Bestandteil komplexer Maschinenoperationen identifiziert werden. So werden in Abb. 3.3 die beiden potenziellen MAC-Operationen aus Abb. 3.2 nicht mehr erkannt und eine Multiplikation (mo_3) und zwei Additionen (mo_4 und mo_5) generiert.

- Datentransferpfade zwischen CSEs und deren Benutzung werden unzulänglich gehandhabt. Wesentlich prekärer ist, dass den Wurzeln der Bäume eindeutige sequenzielle Komponenten zugeordnet werden müssen. In DSP-Compilern wird hier i.d.R. der Speicher gewählt, was dann zusätzlich zu einem unnötigen Overhead an LOADs und STOREs führt. Für Bäume, die CSEs benutzen, werden die Datentransferpfade somit nicht mehr hinsichtlich der ursprünglichen Registerfiles der CSEs optimiert. In Abb. 3.3 werden somit zusätzliche LOAD- und STORE-Operationen generiert ($mo_8, mo_9, mo_{10}, mo_{11}, mo_{12}$). Es wird hier häufig das Argument angebracht, dass diese zusätzlichen LOAD- und STORE-Operationen ja durch Standardoptimierungen, die Redundanzen beim Laden und Speichern von Werten erkennen, eliminiert werden können. Das setzt aber voraus, dass auch jeweils die gleichen Registerfiles zum Speichern und Laden verwendet wurden. Das Beispiel zeigt, dass dies durch die baumbasierte Instruktionsauswahl nicht gegeben sein muss.

- Das zu Grunde liegende Kostenmodell der dynamischen Programmierung erlaubt es nicht, Parallelität zwischen Teilbäumen eines Baumes zu modellieren. Das Kostenmodell erlaubt nur eine sequenzielle Sichtweise exakt zu modellieren. Somit kann eine falsche Auswahl an Maschinenoperationen bzw. eine ungünstige Bindung an Ressourcen eine Parallelausführung verhindern. Im Beispiel wurden LOAD-Operationen generiert, die nicht mehr den Restriktionen von parallelen LOADs genügen. Im Beispiel kann nur noch mo_9 mit mo_3 parallelisiert werden.

- Die Auswahl von Autoinkrement- und Autodekrementoperationen kann durch Baummuster nicht mehr spezifiziert werden. Hier sind also schon Graphmuster erforderlich.

Der Einsatz von Tree-Parsing liefert für GPPs sehr gute Resultate. Im Kontext stark irregulärer Prozessoren treten, wie gezeigt, allerdings große Probleme auf, die den Einsatz in der bestehenden Form der Verfahren fraglich machen. Im Beispiel aus Abb. 3.3 zeigt sich, dass der eigentlich notwendige Code von 9 auf 15 Maschineninstruktionen wächst, wenn man die Initialisierungscodes aus Abb. 3.2 zugrunde legt.

3.3.2. Registerallokation

Die Schwerpunkte der Entwicklungen im Bereich der Registerallokation konzentrieren sich auf die Unterstützung von Prozessoren mit homogenen Registerfiles. Dabei stehen globale Techniken auf der Basis von Graphfärbung im Vordergrund, die das Ziel haben, Spillcode zu minimieren [23, 24, 51, 25, 20, 21, 5, 22]. Zwar können diese Verfahren mit verteilten Registerfiles umgehen, wobei die Berechnung guter Datentransferpfade aber nicht gehandhabt wird. Weitere Entwicklungen befassen sich mit

der Phasenkopplung (siehe Kapitel 3.3.4). Die Retargierbarkeit ist hier kein zentrales Thema.

Interessant werden diese Techniken bei VLIW-artigen DSPs mit der Tendenz zu großen homogenen Registerfiles, wie der Texas Instruments TMS320C6x oder Philips Trimedia. Solche Architekturen werden aber in dieser Arbeit nicht betrachtet. Weiterhin ist der Einsatz der Techniken interessant, um Spillcode für größere Adressregisterfiles zu generieren. Es sollte auf jeden Fall geprüft werden, ob der Einsatz globaler Registerallokationstechniken möglich und sinnvoll ist. Weiterhin sind in diesem Bereich gute Heuristiken entwickelt worden, die gute Suchreihenfolgen für suchbasierte Verfahren vorgeben können.

3.3.3. Instruktionsanordnung

Die Schwerpunkte in der Entwicklung von Techniken zur Instruktionsanordnung konzentrieren sich auf die Entwicklung globaler Techniken, um Pipelinestufen und Parallelität in RISC-Prozessoren und / oder VLIW-Prozessoren besser nutzen zu können [28, 39, 47, 50, 14, 4]. Dabei gibt es i.d.R. nur sehr wenige Restriktionen bzgl. der Parallelisierung von Maschinenoperationen. Die grundlegenden Probleme werden hier nicht gelöst, allerdings wird der Einsatz globaler Techniken wiederum bei VLIW-artigen DSPs interessant. Weitere Arbeiten befassen sich mit der Entwicklung von Phasenkopplungstechniken, die im Folgenden Kapitel (3.3.4) beschrieben werden. Retargierbarkeit ist hier kein zentrales Thema.

Auch hier sollte jeweils geprüft werden, inwieweit man Nutzen aus den Verfahren ziehen kann. So sind auch hier viele Heuristiken entwickelt worden, die gute Suchstrategien vorgeben könnten.

3.3.4. Phasenkopplung

Phasenkopplungsprobleme existieren nicht nur im Kontext irregulärer Prozessorarchitekturen, sondern auch schon unter der Verwendung von GPPs. Die negativen Auswirkungen auf die Codequalität sind jedoch nicht so verheerend wie bei DSPs, auf der anderen Seite aber auch nicht zu vernachlässigen. Es besteht die Möglichkeit, baumbasierte Techniken mit sequenzieller Instruktionsanordnung und der Registerallokation zu koppeln. Dabei beschränkt sich die Phasenkopplung aber lokal auf den jeweiligen Baum. Das Optimierungsziel erweitert sich insofern, als dass durch Einbeziehung der Instruktionsanordnung und der Registerallokation versucht wird, den Registerdruck[7] zu minimieren.

Der Schwerpunkt der Arbeiten ist, die wechselseitigen Abhängigkeiten zwischen Registerallokation und Instruktionsanordnung bei Prozessoren mit großen homogenen Registerfiles und Parallelität in den Griff zu bekommen. In [48] wird der Datenabhängigkeitsgraph so manipuliert, dass er an keiner Stelle breiter ist als die zur Ver-

[7] Der Registerdruck ist ein Maß für die zu einem bestimmten Zeitpunkt in einem Programm lebendigen Werte in Registern. Ist die Anzahl dieser Werte zu einem Zeitpunkt größer als die eigentliche Registerkapazität, so spricht man von Registerdruck.

fügung stehende Registerkapazität. Die Manipulation basiert auf dem Einfügen zusätzlicher Sequenzialisierungskanten. In der gleichen Arbeit wird ein zweiter Ansatz *IPS* (IPS=integrated prepass scheduling) vorgestellt. Darin versucht ein Scheduler zunächst unter einer bestimmten Beschränkung der Registeranzahl eine Instruktionsanordnung zu generieren. Die eigentliche Registerallokation ist nachgeschaltet, die Spillcode einfügt, was eine erneute Instruktionsanordnungsphase erfordert. In [17, 16, 18] werden ebenfalls zwei Varianten untersucht, wobei die erste Variante eine Erweiterung von IPS ist. Die zweite Variante führt zunächst eine initiale Instruktionsanordnung durch, die einmal unter sehr beschränkter Registerkapazität durchgeführt wird und ein zweites mal unter der maximal verfügbaren Anzahl von Registern. Hieraus werden Kosten für die jeweils generierten Instruktionsanordnungen ermittelt, die in einer auf Graphfärbung basierten Registerallokation zur Ermittlung der besten Spillkandidaten genutzt werden. Dieser Phase ist ein lokaler List-Scheduler nachgeschaltet. Eine Technik, die Trace-Scheduling mit einer Registerallokation on-the-fly verbindet, wird in [43] gezeigt. Eine weitere Arbeit in [15] sequentialisiert wiederum den Datenabhängigkeitsgraphen auf globaler Ebene, mit dem Ziel, den Registerdruck sowie die Benutzung von Funktionseinheiten zu beschränken. Weitere Ansätze beginnen mit einer Registerallokation, erlauben aber während der Instruktionsanordnung, Register zu realloziieren, um so Antiabhängigkeiten, die einer Parallelisierung im Wege stehen, zu eliminieren [91, 95, 100].

Es gibt eine Reihe von Arbeiten, die graphfärbungsbasierte Registerallokation um die Betrachtung von Parallelität erweitern [97, 105, 5, 98, 19, 99]. Ansätze, die die Handhabung irregulärer Datentransferpfade auf der Basis des Data-Routings in VLIW-Prozessoren behandeln, sind in [31, 57] zu finden.

3.4. Compiler für eingebettete Prozessoren

Die letzten Kapitel haben die wesentlichen Problematiken herausgestellt, die bei der Generierung hochqualitativen Codes für DSPs im Wege stehen. Weiterhin wurden die Defizite aufgeführt, die beim Einsatz von Techniken für eigentlich homogene GPPs zu schlechter Codequalität führen.

In diesem Kapitel wird eine Übersicht über den Kenntnissstand in der Entwicklung von Compilern speziell für eingebettete Prozessoren gegeben. Es wird zuerst ein Überblick über bestehende Compilersysteme und darin integrierte Techniken gegeben. Darauf folgt die Beschreibung von Techniken, die unabhängig von irgendwelchen Systemen entwickelt wurden. Geschlossen wird mit einer Konklusion, die die gelösten und verbleibenden Probleme herausstellt. Bei der Beschreibung der Compilersysteme werden zuerst jeweils das Einsatzumfeld und die primären Zielarchitekturen und dann die Art der Adaption an neue Prozessoren erläutert. Der Schwerpunkt der Beschreibung zielt auf die Beschreibung der integrierten Techniken ab. Zur Bewertung werden die Kriterien aus Kapitel 3.2 Seite 60 herangezogen.

In den vorgestellten Techniken kommen unterschiedliche LIRs zur Anwendung, die ebenfalls die Darstellung alternativer Ressourcen erlauben. Faktorisierte Maschinenoperationen werden als einheitliches Bewertungsmodell zur Charakterisierung der je-

weilig erlaubten Freiheitsgrade in den Ressourcen verwendet. Es wird auch auf die jeweilige Art der Repräsentation eingegangen.

3.4.1. Compilersysteme

Bei den vorgestellten Compilern handelt es sich bei CHESS und Cosy um kommerzielle Systeme. Die restlichen Compiler sind Forschungscompiler.

CHESS

Das CHESS-System [67] wurde als Synthesewerkzeug und retargierbare Codegenerierungsumgebung für Festkomma-DSPs entworfen. Hardware und zugehöriger Codegenerator werden gleichzeitig entwickelt (Hardware/Software-Codesign). Nach derzeitigem Kenntnisstand stellt CHESS das einzige kommerziell verfügbare Compilersystem dar, das sowohl retargierbar ist, als auch den Aspekt der Phasenkopplung berücksichtigt. CHESS kann nur für Prozessoren eingesetzt werden, bei denen alle Instruktionen in einem Maschinenzyklus ausgeführt werden können. Quellsprachen von CHESS sind C und DFL[8]. Der Compiler ist automatisch retargierbar. Alle Phasen von CHESS werden automatisch auf der Basis eines Instruktionssatzgraphen retargiert, der aus einer externen Maschinenbeschreibung des Prozessors gewonnen wird; die verwendete Hardwarebeschreibungssprache ist nML [33]. Es besteht keine explizite Möglichkeit zur Integration neuer Techniken.

Die Instruktionsauswahl ist graphbasiert: zunächst werden die Operationen der IR auf partielle Maschinenoperationen durch Anwendung von Graphtransformationen abgebildet, die durch partielle Instruktionswörter dargestellt werden. Diese werden dann zu komplexen Maschinenoperationen zusammengefügt. Die verwendete Technik (*Bundling*) [106] basiert auf der heuristischen Vereinigung partieller, kompatibler Instruktionswörter. Dabei werden keinerlei Bindungen an spezifische Lokationen vorgenommen. Optimierungsziel ist die Minimierung der Gesamtzahl an Maschinenoperationen. Vorteil gegenüber baumbasierter Instruktionsauswahl ist, dass auch basisblockübergreifende komplexe Maschinenoperationen erkannt werden. Nachteilig ist, dass Kosten für Datentransfers nicht berücksichtigt werden. Weiterhin werden in dieser Phase sowohl Funktionseinheiten als auch partielle Instruktionswörter fest gebunden, was zu starken Parallelitätsbeschränkungen führen kann. Nach der Instruktionsauswahl wird versucht, die Auslastung der funktionalen Einheiten und Register durch Verschiebung von Maschinenoperationen über Basisblockgrenzen hinweg zu balancieren. Die Entscheidungen der Registerallokation werden durch Schätzungen der Auswirkungen auf die Parallelität gesteuert [68]. Für jeden Knoten eines Kontrolldatenflussgraphen (hybride Repräsentation von Kontrollfluss- und Datenflussgraphen) werden die besten Registerzuordnungen und die Auswahl guter Datentransfers auf der Basis einer Branch-and-Bound-Suche durchgeführt. Als Optimierungsmaß werden dabei Registerdruck und Parallelität verwendet. Dabei wird ein

[8] Die DFL-Entwicklung wurde mittlerweile aufgegeben, da hier die Akzeptanz bei den Programmierern fehlt, obwohl DFL wesentlich bessere Formulierungsmöglichkeiten für DSP-Applikationen ermöglicht. So können z.B. beliebige Fixpunkt-Datentypen sowie Delay-Lines direkt spezifiziert werden.

probabilistisches Modell für die Parallelität eingesetzt, das allerdings die Modellierung von Parallelitätseinschränkungen nicht ermöglicht. Es werden somit zwar Aspekte zwischen den Phasen berücksichtigt, aber es findet keine echte Phasenkopplung statt. Die eigentliche Parallelisierung wird durch eine nachgeschaltete globale Instruktionsanordnung durchgeführt, die u.a. auch Softwarepipelining sowie spezifische Kontrollflussmechanismen unterstützt [62].

Bezüglich der Codequalität für existierende Applikationen liegen leider keine aussagekräftigen Informationen vor, was die Bewertung der eingesetzten Heuristiken schwierig macht.

Cosy

Cosy [111] ist ein Compilersystem, das auf flexible Erweiterung von Optimierungen ausgerichtet ist. Leider beschränkt sich dies auf das Middle-End, das modular strukturiert ist, und hier die Adaption neuer Techniken ermöglicht. Das Back-End ist konzeptionell davon getrennt und weist diese Modularität nach unseren Erkenntnissen leider nicht auf. Es ist nicht speziell auf eingebettete Prozessoren zugeschnitten und basiert auf Standardtechniken für GPPs.

CBC

Der retargierbare Codegenerator CBC [33] wurde zusammen mit einem retargierbaren Instruktionssatzsimulator entwickelt. Als Zielarchitekturen werden DSPs und ASIPs betrachtet. Es existiert ein Satz an Techniken, die alle automatisch retargierbar sind. Die Retargierung basiert auf nML [35]. Es werden keine Konzepte zur Erweiterung neuer Techniken angeboten.

Die Codegenerierung umfasst die Instruktionsauswahl und eine phasengekoppelte heuristische Registerallokation und Instruktionsanordnung, die die Berechnung guter Datentransferpfade berücksichtigt. Die Instruktionsauswahl wird in zwei Phasen durchgeführt. Die erste bildet Operationen der IR auf Maschinenoperationen ab und basiert auf der Anwendung baumbasierter Methoden. In der zweiten, graphbasierten Phase werden Maschinenoperationen global über Basisblockgrenzen hinweg heuristisch zu komplexen Maschinenoperationen kombiniert [34]. Die Berechnung guter Datentransfers findet dabei keine Berücksichtigung. Die Phasenkopplung von Registerallokation und der Instruktionsanordnung basiert auf einem erweiterten List-Scheduling Ansatz [55]. Auch hier ist eine relativ frühe Bindung von Funktionseinheiten nachteilig.

Resultate bzgl. der Verbesserungen der Codequalität existieren nicht. Es werden lediglich Verbesserungen bzgl. der Reduktion von Maschinenoperationen gezeigt, die aber nicht unbedingt auf die resultierende Güte des letztendlichen Codes schließen lassen. Bezüglich der Codequalität wurden bislang keine relevanten Resultate publiziert, so dass hier keine Bewertungskriterien für die Güte der Heuristiken vorliegen.

CodeSyn

CodeSyn [104] ist ein retargierbarer Codegenerator, der integriert ist in Flexware, einer Toolkette zur Entwicklung eingebetteter Systeme, die neben CodeSyn noch einen retargierbaren Instruktionssatzsimulator und Assembler umfasst. Als Zielarchitekturen sind ASIPs angegeben. CodeSyn bildet C-Programme auf Maschinencode ab. Zur Hardwarebeschreibung wird ein gemischt strukturelles und verhaltensorientiertes Modell angegeben. Alle Techniken sind automatisch retargierbar. Der Codegenerator sieht keine Konzepte zur Erweiterungen neuer Techniken vor.

Zuerst werden die Operationen der IR auf faktorisierte Maschinenoperationen auf der Basis von Graphtransformationen abgebildet [81]. Dabei werden keinerlei Ressourcen gebunden. Es folgt eine baumbasierte Instruktionsauswahl, die die Bildung komplexer faktorisierter Maschinenoperationen umfasst, und in der resultierenden Überdeckung werden alternative Registerfiles beibehalten. Es werden aber keine Datentransfers bestimmt und die alternativen Mengen von Registerfiles dürfen keine wechselseitigen Abhängigkeiten untereinander haben. In der Berechnung einer optimalen Überdeckung der Datenflussbäume gehen weder die Beachtung legaler Datentransferpfade, noch die Kosten evtl. notwendiger Datentransfers ein. Der Instruktionsauswahl folgt eine globale, sequenzielle Instruktionsanordnungsphase, mit dem Ziel der Registerdruckminimierung. Die nächste Phase umfasst eine Registerallokation, basierend auf einem heuristischen, erweiterten Left-Edge-Algorithmus [82]. Dieser führt die Berechnung notwendiger Datentransferpfade aus und nutzt dabei die nach der Instruktionsauswahl verbliebenen Freiheitsgrade bzgl. der alternativen Registerfiles. Dabei wird eine feste sequenzielle Anordnung der faktorisierten Maschinenoperationen vorausgesetzt, die zur Bindung der Registerfiles und zur Generierung notwendiger Datentransfers sequenziell durchlaufen wird. Die Bindung der Registerfiles beinhaltet allerdings keinerlei Vorausschau. Auch notwendige Datentransfers werden mehr oder weniger ad-hoc ausgewählt. Es werden keinerlei Aspekte bzgl. der Parallelität in Betracht gezogen.

In [104] sind Resultate angegeben, die handgeschriebenen Assemblercode mit CodeSyn generiertem Code vergleicht (20% Overhead bei CodeSyn). Allerdings ist hier nicht klar zu erkennen, aus welcher Phase die Codegüte bei CodeSyn resultiert. Bis auf das verzögerte Binden der Registerfiles wird keine Phasenkopplung durchgeführt.

EXPRESS

EXPRESS [52] ist Teil einer Toolkette zur schnellen Entwurfsraumerkundung, bei der - ähnlich wie in CHESS - die Hardware und die zugehörige Toolkette gleichzeitig entwickelt werden (Hardware / Software-Codesign). EXPRESS ist eigentlich auf die Entwicklung von VLIW-Architekturen zugeschnitten, wird aber auch für DSPs genutzt. Die Informationen über die Zielarchitektur werden automatisch aus der Maschinenbeschreibungssprache *Expression* extrahiert. Der Compiler ist automatisch retargierbar und sieht bislang keine Erweiterung um neue Techniken vor.

EXPRESS beinhaltet eine heuristische integrierte Instruktionsauswahl, Registerallokation und Instruktionsanordnung, genannt Mutation Scheduling [101]. Dabei wird die

Integration der Instruktionsauswahl durch die Generierung alternativer Mengen von Maschinenoperationen umgesetzt. Diese Generierung basiert auf der Anwendung algebraischer Transformationen. Es wird also keine faktorisierte Maschinenoperation sondern explizit eine Menge von Maschinenoperationen gebildet. Die Auswahl spezifischer Maschinenoperationen aus dieser Menge - während des Mutation Schedulings - wird auf der Basis von Heuristiken durchgeführt.

Die Techniken sind primär auf VLIW-artige Architekturen ausgelegt, die nicht allzu große Irregularität und Restriktionen bzgl. der Parallelausführung besitzen. Für den Einsatz für Architekturen mit hoher Irregularität existieren bislang keine aussagekräftigen Resultate um den Einsatz in diesem Bereich bewerten zu können.

MSSQ

MSSQ [102] ist ein retargierbarer Codegenerator und Bestandteil des MIMOLA-Systems, das zur Synthese und Simulation von digitalen mikroprogrammierbaren Prozessoren sowie der retargierbaren Codegenerierung der Mikroprogramme für diese Prozessoren entwickelt wurde [102, 89, 76]. Hierbei handelt es sich um eines der ersten Systeme, welches die simultane Entwicklung der Prozessorhardware und der dazugehörigen Toolkette inklusive der Codegenerierung umsetzt. Alle Komponenten des Systems werden automatisch basierend auf MIMOLA[9] retargiert.

MSSQ führt zuerst eine Instruktionsauswahl sowie eine Registerallokation (für temporäre Variablen) auf Statement-Ebene durch. Resultate von Statements werden a priori im Speicher abgelegt (es wird also nicht versucht, diese in Registern zu halten). In der anschließenden lokalen Instruktionsanordnung wird dann Binärcode generiert. Die sehr lokale Sichtweise der Instruktionsauswahl und Registerallokation sowie das grundsätzliche Speichern der Variablen (bis auf temporäre Variablen) führt leider zu keiner guten Codequalität.

PROPAN

PROPAN[63] ist ein System, das die Adaption von maschinenabhängigen Postpass-Optimierungen ermöglicht. Dabei können die Techniken sowohl an neue Prozessoren als auch an die Assemblersprache eines Prozessors angepasst werden. Letzteres ermöglicht es, PROPAN hinter die Assemblerausgaben von Compilern zu schalten. Die Retargierung von PROPAN erfolgt auf der Basis von TDL [66]. PROPAN ist modular organisiert und besitzt eine klar definierte Schnittstelle zur Repräsentation von Maschinenprogrammen auf denen Optimierungen und Analysen aufsetzen.

PROPAN umfasst einen phasengekoppelten Ansatz zur integrierten globalen Instruktionsanordnung und Registerallokation, der auf einem ILP-Ansatz basiert [64, 65, 63]. Allerdings wird nur das Registerassignment betrachtet und keine Betrachtung von Spillcode bzw. das Einfügen weiterer Datentransfers durchgeführt. Der Ansatz kann mit akzeptablen Laufzeiten aufwarten und liefert sehr gute Resultate für Standard DSPs.

RECORD

RECORD [69] ist ein automatisch retargierbarer Compiler für Festkomma-DSPs. Dabei sind die Prozessoren auf Architekturen beschränkt, die MIs innerhalb eines Instruktionszyklus bearbeiten können. RECORD erwartet DFL als Eingabe und generiert Binärcode als Ausgabe. Aus einer MIMOLA-Beschreibung des Prozessors wird eine Extraktion des Befehlssatzes durchgeführt [74]. Auf der Basis dieses Befehlssatzes werden alle Phasen automatisch retargiert. RECORD ist modular organisiert und hat eine textuelle RT[9]-basierte IR. Diese Schnittstelle dient aber offensichtlich nur der internen Erweiterbarkeit, da sie nicht explizit spezifiziert wird.

Es erfolgt zuerst eine traditionelle baumbasierte Instruktionsauswahl[10], gefolgt von einer lokalen Registerallokation. Da hier traditionelle Standardverfahren zur Instruktionsauswahl zum Einsatz kommen, entstehen hier auch die in Kapitel 3.3 beschriebenen Probleme (z.B. überflüssige LOAD- und STORE-Operationen und zu starke Bindung an Ressourcen).

Die nachfolgenden Phasen umfassen allerdings spezifisch für DSPs entwickelte Techniken. Es wurden generische Adressgenerierungstechniken entwickelt. Zum automatischen Retargieren der Techniken wurde ein generisches Modell von AGUs von DSPs entwickelt. Es wurden existierende Techniken zur Speicherzuordnung [79] skalarer Variablen verallgemeinert und die bestehenden Resultate stark verbessert [75]. Weiterhin wurden Techniken entwickelt, die die AGUs bei Array-Zugriffen effektiv ausnutzen [13]. Die in [77] beschriebene lokale Instruktionsanordnung basiert auf der Anwendung von ILP. Hierbei handelt es sich um einen exakten Ansatz, der auch die Restriktionen bzgl. der Parallelität in DSPs berücksichtigt. Es können noch Basisblöcke von normaler Größe in akzeptabler Zeit (einige Minuten) gehandhabt werden. Der Ansatz kann bei der Parallelisierung Nutzen aus den Freiheitsgraden in den Funktionseinheiten ziehen. Ansonsten existieren aber keine Konzepte zur Phasenkopplung.

Der Einsatz der Adressgenerierungstechniken und der ILP-basierten Instruktionsanordnung führen zu sehr guten Resultaten (siehe u.a. [69]). Für den TI TMS320C2x konnten gegenüber dem kommerziellen TI-Compiler, für eine Reihe DSP-typischer Applikationen sehr gute Verbesserungen erzielt werden.

SPAM

SPAM [108, 110] ist im wesentlichen eine Bibliothek, die die schnelle Entwicklung von eingebetteten Compilern als Intention hat. SPAM verwendet die SUIF-Bibliothek[11] [49] als optimierendes C-Frontend (und nutzt auch dessen Middle-End). Das Back-End, genannt TWIF, beinhaltet dabei die retargierbare Bibliothek von maschinenunabhängigen Optimierungen und Codegenerierungstechniken. Die einzelnen Techniken liegen in generischer bzw. parametrisierbarer Form vor und erwarten jeweils eigene, prozessorbezogene Parameter, die vom Compiler-Entwickler spezifiziert werden

[9] RT=Register-Transfer Befehl.

[10] Zum Retargieren der Instruktionsauswahl wird aus dem extrahierten Befehlssatz automatisch eine Baumgrammatik für iburg erzeugt, aus der dann ein Codeselektor generiert wird.

[11] SUIF ist eine an der Universität von Stanford entwickelte IR.

müssen. Die eigentliche Intention des Systems ist die schnelle Entwicklung neuer, an die Prozessoren angepasster Techniken, wobei zunächst möglichst viele der bestehenden Techniken verwendet werden sollten.

Die Bibliothek enthält - neben Standardoptimierungen - Techniken, die speziell für bestimmte DSPs entwickelt wurden, z.B. Erweiterungen zur baumbasierten Instruktionsauswahl, Phasenkopplung von Instruktionsauswahl, Registerallokation und sequenzieller Instruktionsanordnung und weiterhin Techniken zur Ausnutzung verteilter Speicher und spezifischer AGUs. Diese können vom Entwickler zu einem Codegenerator kombiniert werden. SPAM ist modular strukturiert auf der Basis eines generischen Assemblerformats, das allerdings sehr beschränkt in der Darstellung von Ressourcen ist. Die einzelnen Techniken liegen in generischer bzw. parametrisierbarer Form vor und erwarten jeweils eigene, prozessorbezogene Parameter, die vom Compiler-Entwickler spezifiziert werden müssen. Eine weitere Intention bei der Entwicklung des Systems war es, die schnelle Entwicklung neuer an die Prozessoren angepasster Techniken zu ermöglichen, wobei möglichst viele der bestehenden Techniken wiederverwendet werden sollten. Im Folgenden wird ein Überblick über die in SPAM integrierten Techniken gegeben.

In [78] wird eine exakte Technik zur gekoppelten Registerallokation und Instruktionsanordnung für den Texas Instruments TMS320C2x auf der Basis von *Binate-Covering* beschrieben. In einer graphbasierten Instruktionsauswahlphase wird zuerst eine optimale Überdeckung eines Datenflussgraphen mit komplexen Maschinenoperationen generiert. Dabei werden allerdings Kosten für notwendige Datentransfers nicht berücksichtigt. Das exakte Verfahren zur gekoppelten Registerallokation und Instruktionsanordnung beinhaltet u.a. die Berechnung von Datentransfers und Spillcode. Ziel ist die Generierung einer möglichst kurzen sequenziellen Sequenz von Maschinenoperationen. Nachteil der Technik ist, dass die mögliche Parallelität des Prozessors nicht berücksichtigt wird. Gerade bei akkumulatorbasierten Prozessoren - wie der Texas Instruments TMS320C2x - kann die Generierung zunächst sequenzieller Anordnungen von Maschinenoperationen die potenziell vorhandene Parallelität durch unnötige Abhängigkeiten verhindern. Leider wurden keine Resultate angegeben.

In [80] wird ebenfalls eine Technik zur gekoppelten Registerallokation und Scheduling vorgestellt, die auf einer Branch&Bound-Methode basiert. Das Optimierungsziel ist die Spillcodeminimierung und die Minimierung notwendiger Operationen zum Residual-Control (siehe Kapitel 1.3). Es wird wiederum keine Parallelität berücksichtigt. Weitere Techniken aus dem Bereich der Ausnutzung der Adresseinheiten werden in [79] beschrieben. Diese wurden in [75] (s.o.) aber stark verbessert.

Der retargierbare Codegenerator AVIV [54, 53] wird derzeit am MIT entwickelt. Als Ausgangspunkt wird die Optimierungsbibliothek SPAM verwendet. Der Schwerpunkt liegt auf Prozessoren, die Parallelität auf Instruktionsebene ermöglichen. AVIV beinhaltet einen retargierbaren Assembler, erwartet als Eingabe SUIF[49] und gibt Binärcode aus. AVIV wird auf der Basis von ISDL[54] automatisch retargiert. AVIV integriert die Phasenkopplung von Instruktionsauswahl, Registerallokation und Instruktionsanordnung. Zunächst werden alle möglichen Überdeckungen eines Datenflussgraphen mit Maschinenoperationen durch einen sogenannten Splitt-Node-DAG dargestellt. Zur Verminderung der Komplexität werden vorab kostenungünstige Überdeckungen eliminiert. Dieser Schritt ist generell sinnvoll, allerdings geht aus dem

Kostenmodell nicht klar hervor, ob nicht potenziell gute Lösungen ausgeschlossen werden können. Der nächste Schritt besteht aus der Generierung maximaler Cliquen, auf der Basis einer Branch&Bound-Methode, wobei jede Clique MIs maximaler Größe darstellt[12]. Dabei kann eine Maschinenoperation in mehreren Cliquen vorkommen. Die eigentliche Instruktionsanordnung besteht in der Auswahl einer Reihenfolge für die Cliquen, wobei in jedem Schritt die jeweils größte Clique ausgewählt wird, deren Operanden bereits gescheduled wurden und die kein Spilling erfordert. Alle Maschinenoperationen in dieser Clique werden aus den noch verbleibenden Cliquen gelöscht. Sollte Spilling notwendig sein, wird ein Spillkandidat ausgewählt, der entsprechende Code in den Splitt-Node-DAG eingefügt, und die verbleibenden maximalen Cliquen werden re-evaluiert. Die Cliquen sind bzgl. der beinhaltenden Maschinenoperationen nicht disjunkt, so dass das eigentliche Problem der Instruktionsanordnung durch diese Phase noch nicht gelöst wird. Dies wird erst durch die Auswahl von Cliquen umgesetzt, was aber in AVIV wiederum auf Heuristiken basiert. Eine vollständige globale Register-Allokation wird getrennt ausgeführt. Derzeit ist AVIV noch im Entwicklungsstadium. Es sind noch keine aussagekräftigen Resultate verfügbar, die gerade bei den eingesetzten Heuristiken sehr wichtig für die Bewertung des Ansatzes sind.

In [110] sind Techniken zur Zuordnung von Variablen und Arrays auf verteilte Speicher gegeben. Das Ziel ist, die Parallelität bzgl. der Nutzung von Adressoperationen besser auszunutzen.

Trimaran

Trimaran [113] ist speziell für Optimierungen VLIW-artiger Architekturen konzeptioniert worden. Die verwendeten Techniken werden für VLIW-Prozessoren automatisch retargiert. Es beinhaltet bislang aber keine Unterstützung für stark irreguläre Architekturen. Die integrierten Techniken beschäftigen sich primär mit Analysen und Techniken zur Ausnutzung hoher Parallelität, wobei auch globale Instruktionsanordnungstechniken zum Einsatz kommen. Dabei werden eher Architekturen homogener Natur betrachtet. Aber auch hier spielt die Phasenkopplung von Registerallokation und Instruktionsanordnung eine große Rolle. Das Back-End ist modular strukturiert. Es existiert eine auf VLIW-Architekturen ausgerichtete IR, die sich an neue Prozessoren adaptieren lässt und eine modulare Entwicklung von Techniken ermöglicht.

3.4.2. Techniken

Im Folgenden werden Techniken vorgestellt, die in keinem Compilersystem integriert sind.

[12] Im Unterschied zu maximalen MIs bezieht sich eine MI maximaler Größe auf eine feste Menge von MOs innerhalb eines Programms. Hier wird versucht, maximale Gruppen der zur Verfügung stehenden MOs zu parallelisieren, was aber nicht notwendigerweise zu einer maximalen MI führen muß. Bei der Bildung müssen auch Daten- und Kontrollflußrelationen berücksichtigt werden.

Gebotys In [46] ist eine exakte Technik zur Phasenkopplung von Codeselektion, Registerallokation und Instruktionsanordnung beschrieben. Dieses basiert auf der Anwendung der Linearen-Programmierung. Dabei werden irreguläre Datentransferpfade und Spilling in der Technik berücksichtigt. Diese Technik ist allerdings auf Architekturen mit einzelligen RFs beschränkt (Bsp TI TMS320C2x), wodurch auch die Betrachtung des Spilling erst realistisch wird. Es wird von einer festen sequenziellen Reihenfolge der abstrakten Operationen ausgegangen, die auch bei den generierten Maschinenoperationen eingehalten werden muss. Lediglich die Parallelisierung von aufeinanderfolgenden Maschinenoperationen ist hier erlaubt. Dies sind Bedingungen, die auch durch den eingesetzten Formalismus bedingt sind: Hier werden nur lineare Gleichungen erlaubt, die aus Horn-Klauseln ableitbar sind. Dies garantiert beim Einsatz einer LP-Lösung immer noch die Gültigkeit der Optimalität der durch Rundung entstandenen ILP-Lösung. Hierdurch wird zwar das Verfahren sehr effizient, es können aber auch potenziell gute Lösungen ausgeschlossen werden. Somit sind die gezeigten Resultate mit Bezug auf das Modell zwar optimal, nicht aber in Hinblick auf den bestmöglichen generierbaren Code, wenn bzgl. der Anordnungen und der Parallelisierung der Maschinenoperationen mehr Freiheitsgrade bestehen würden.

Philips Research Die Arbeiten der Gruppe des Philips Research Laboratories (Eindhoven, Niederlande) befassen sich mit Constraint-Analysen und darauf aufbauenden Techniken. Dabei geht es um die Entwicklung von Constraints, die im Vorfeld den möglichen Lösungsraum für optimierende Techniken drastisch einschränken, ohne dabei gute Lösungen auszuschließen. Bislang wurden Techniken [112, 90] im Bereich des Register-Assignments, sequenziellen Schedulings und der Instruktionsanordnung vorgestellt, die diese Constraint-Analysen ausnutzen. Diese beinhalten aber bislang keine Phasenkopplung.

Leupers In [73] wird eine gekoppelte Instruktionsauswahl und Registerallokation durchgeführt. Der Ansatz beruht darauf, für die Lokationen der CSEs Registerfiles auszuwählen, so dass die Codegröße und die Zugriffe auf den Hauptspeicher, bedingt durch die Allokation von CSEs im Hauptspeicher reduziert werden. Hier wird eine baumbasierte Instruktionsauswahl in Kombination mit Simulated Annealing eingesetzt. Es wird keine Parallelität berücksichtigt.

In einer Reihe von Arbeiten werden Techniken zur Unterstützung von neuen Eigenschaften VLIW-artiger DSPs beschrieben [72], die allerdings architekturspezifisch und nicht retargierbar sind. Dies umfasst Techniken zur Ausnutzung von bedingten Instruktionen [70] und SIMD-Instruktionen [71], wie sie z.B. im TI TMS320C6x und im Philips TriMedia vorkommen. Diese Techniken sind die ersten veröffentlichten Ansätze zur effektiven Ausnutzung dieser Eigenschaften. Die Resultate der Techniken wurden mit den Resultaten des kommerziellen TI Compilers verglichen und führten zu guten Verbesserungen.

Lorenz In [116] wird ein Ansatz zur Phasenkopplung von Instruktionsauswahl, Registerallokation und sequenzieller Instruktionsanordnung auf der Basis genetischer Algorithmen vorgestellt, in dem alle drei Phasen simultan ausgeführt werden. Diese

Technik ist speziell auf den M3 - ein DSP, der an Universität Dresden entwickelt wurde [37] - zugeschnitten. Hier konnten sehr gute Verbesserungen verglichen zum Einsatz von baumbasierten, nicht gekoppelten Techniken erzielt werden. Weiterhin werden in [84] Techniken zur Allokation von Programmvariablen in der sehr spezifischen Speicherstruktur des M3 beschrieben.

Rimey & Hilfinger In [107] wird ein Ansatz beschrieben, der lokale Instruktionsanordnung durchführt und dabei in jedem Schritt für eine Operation einen Instruktionszyklus festlegt. Aus der Menge der Maschinenoperationen, die die Operation realisieren können, wird eine feste Maschinenoperation ausgewählt, in der dann die Registerfiles und Funktionseinheiten fest gebunden werden. Es werden notwendige Datentransfers von den Registerfiles, in denen Werte zuerst erzeugt werden, zu den Registerfiles, in denen diese Werte als Operanden benutzt werden, eingefügt. Dabei werden Operationen nur dann dem aktuellen Instruktionszyklus zugeordnet, wenn die notwendigen Datentransfers vor der aktuellen Instruktion eingefügt werden können. Nachteil der Methode ist sicherlich die relativ frühzeitige Bindung von Ressourcen in jedem Schritt sowie die relativ willkürliche Strategie zur Instruktionsauswahl und der Bindung von Datentransferpfaden und anderen Ressorcen. Es werden keine Resultate vorgewiesen.

Wess In [117] wird ein Ansatz zur integrierten Instruktionsauswahl, Instruktionsanordnung und Registerallokation, auf der Basis von *Trellis-Diagrammen*, gezeigt. Die Instruktionsanordnung wird allerdings nur bzgl. eines sequenziellen Modells betrachtet. Die eigentliche Parallelisierung wird durch eine anschliessende Kompaktierung durchgeführt. Es wird von guter Codequalität für eine Reihe von Standard-DSPs berichtet, diese aber leider nicht durch Resultate belegt. Trellis-Diagramme bieten darüber hinaus kein intuitives Modell zur Spezifikation von Befehlssätzen, so dass eine Adaption an neue Prozessoren nicht einfach zu handhaben ist.

Wilson In [119] ist ein ILP-Ansatz zur vollständigen Integration der Codegenerierungsphasen und exakten Codegenerierung vorgestellt worden. Retargierung kann auf der Basis des ILP-Modells vorgenommen werden. Es hat sich allerdings gezeigt, dass dieser Ansatz zu sehr hohen (nicht mehr zu akzeptierenden) Laufzeiten führt. Es wurden keine konkreten Resultate veröffentlicht.

3.5. Zusammenfassung und Konklusion zum Stand der Technik

Es stellt sich heraus, dass Eigenschaften irregulärer Prozessorarchitekturen sehr hohe und neue Anforderungen stellen, die neue Codegenerierungstechniken erfordern. So zeigt sich, dass Techniken, die im Kontext von GPPs zwar für homogene VLIW-artige Prozessoren interessant sein können (wenn auch hier sicherlich Modifikationen notwendig sind), die Probleme von stark irregulären Prozessoren aber nicht lösen

können. Trotzdem ist jeweils zu prüfen, ob Techniken nicht als Ergänzung neuer Methoden einsetzbar sind oder zumindest aus ihnen gelernt werden kann.

Aus der Analyse der Anforderungen, die notwendig sind, um die Eigenschaften von DSPs effektiv zu nutzen, sowie dem Abgleich mit existierenden traditionellen Codegenerierungstechniken, lassen sich folgende Kernforderungen ableiten:

- Um Eigenschaften von DSPs effektiv zu unterstützen sind Techniken notwendig, die eine graphbasierte Mustererkennung von Maschinenoperationen unterstützen. Es ergibt sich die Forderung nach phasengekoppelten Techniken, die die wichtigsten Phasen der Codegenerierung - die Instruktionsauswahl, Registerallokation und Instruktionsanordnung - sinnvoll integrieren.

- Die vielen Randbedingungen von irregulären Prozessorarchitekturen, sowie die Forderung nach der Phasenkopplung von Techniken, machen das Finden guter heuristischer Methoden sehr schwierig. Da es sich um kombinatorische Suchprobleme handelt, besteht ein Bedarf an hocheffizienten suchbasierten Verfahren. Dabei muss ein guter Kompromiss zwischen Codegüte und Effizienz gefunden werden.

- Die Adaption von Techniken an neue Prozessoren erfordert zum einen möglichst retargierbare Techniken, zum anderen aber auch die Erweiterbarkeit eines Compilers. Zur Einhaltung der Codequalität ist es notwendig, klassenspezifische retargierbare Techniken zu entwickeln. Darüber hinaus verlangt die Erweiterbarkeit nach einem transparenten Modell der Zwischenrepräsentation eines Programms während der Codegenerierung, auf das möglichst alle Phasen der Codegenerierung aufsetzen können.

Im Bereich der für eingebettete Prozessoren dedizierten Techniken finden sich gute Lösungen im Bereich der Adressgenerierung wieder. Hier gibt es gute Ansätze zur Berechnung von Speicherlayouts sowie Strategien zur Aufteilung von Variablen auf verteilte Speicherbänke. Weiterhin existieren gute Ansätze zur Instruktionsanordnung unter Berücksichtigung starker Restriktionen.

Mehr oder weniger vollständige Integrationen liegen nur in den Arbeiten von AVIV, Gebotys, Mutation-Scheduling und Wilson vor. Allerdings gibt es zu den Arbeiten von Wilson, Mutation-Scheduling und AVIV keine Resultate. In AVIV und in der Arbeit von Gebotys ist die Handhabung graphbasierter Maschinenoperationsmuster nicht klar. Der Ansatz von Gebotys ist auf sehr spezifische Prozessoren beschränkt und weist darüber hinaus ein unzureichendes Modell der Parallelität auf.

In den übrigen Arbeiten besteht das stärkste Defizit in der Umsetzung graphbasierter Instruktionsauswahl und deren Integration mit der Registerallokation und Instruktionsanordnung. Meistens liegt eine getrennte Phase zur Instruktionsauswahl vor, die zwar graphbasiert ist, aber keinerlei Wechselwirkungen zu den anderen Phasen berücksichtigt. Die ausgewählten Maschinenoperationen sind zwar meist noch nicht an spezifische Registerfiles gebunden, es werden allerdings häufig schon Zuordnungen zu den Funktionseinheiten vorgenommen. Weiterhin liegen bei den Techniken,

die Datentransfers unter Einbeziehung der Instruktionsanordnung mit Parallelität berücksichtigen, keine aussagekräftigen Resultate vor, um diese Ansätze bewerten zu können. Exakte Ansätze bzw. Ansätze, die Resultate aufweisen können, vernachlässigen die Parallelität in der Phasenkopplung oder führen keine Kopplung durch.

Bezüglich der schnellen Adaption an neue Prozessoren sind die meisten der vorgestellten Compiler automatisch retargierbar, wobei ein zentraler Satz an retargierbaren Techniken vorliegt. Nur PROPAN, Trimaran und SPAM bieten eine brauchbare modulare und transparente erweiterbare Struktur im Back-End. Allerdings sind die zu Grunde liegenden LIRs noch nicht für alle Anforderungen eingebetteter Prozessoren und zur Unterstützung aller Codegenerierungsphasen ausreichend und müssen dahingehend erweitert werden.

Es zeigt sich die Tendenz, dass neue Optimierungsmethoden an Relevanz gewinnen, da gute Heuristiken gerade im Kontext der Phasenkopplung von Techniken nur sehr schwer zu finden sind. Interessant sind hier exakte, auf formalen Methoden basierende Techniken, die mit eigenen Lösungs- und Optimierungsmethoden verknüpft sind (z.B. ILP). Diese können optimale Resultate bis zu einer gewissen Programmgröße garantieren und bieten eine handhabbare Methode zur Formulierung phasengekoppelter Techniken. Hier ist man aber noch nicht im Bereich akzeptabler Laufzeiten, um größere Programme unter den relevanten Wechselbeziehungen handhaben zu können. Interessant sind Methoden zur effektiven Beschneidung des Suchraums, die keine optimalen / guten Lösungen ausschliessen. In dieser Hinsicht sind die Arbeiten von AVIV und Philips interessant.

4. Die constraintbasierte Zwischenrepräsentation CoLIR

constraintbasierte Zwischenrepräsentation CoLIR

In diesem Kapitel wird die *constraintbasierte Zwischenrepräsentation CoLIR (CoLIR = constraint based low-level intermediate representation*) spezifiziert. CoLIR umfasst die Darstellung abstrakter Operationen und die Darstellung von Maschinenprogramm-Varianten auf der Grundlage *constraintbasierter, faktorisierter Maschinenoperationen*. CoLIR wurde entwickelt, um die Entwicklung phasengekoppelter Techniken zu unterstützen. Dazu umfasst CoLIR alle notwendigen Informationen von Codegenerierungsphasen, um diese in integrierten Techniken simultan zur Verfügung stellen zu können. Weiterhin bietet CoLIR eine Basis zur schnellen Adaption von Compilern an neue Prozessoren. CoLIR besitzt ein generisches Konzept zur Darstellung maschinenspezifischer Informationen, so dass sie unmittelbar zur Codegenerierung für neue Prozessoren verwendet werden kann und somit die jeweilige Neuimplementierung einer IR bzw. LIR unnötig macht. Weiterhin wird hierdurch die Grundlage zur Entwicklung generischer Techniken geschaffen. CoLIR bietet eine einheitliche Schnittstelle für Codegenerierungsphasen, so dass neue Codegenerierungstechniken einfach in eine bestehende Menge von Techniken integriert werden können.

Eine der zentralen Aufgaben zur Durchführung der Codegenerierung ist, den abstrakten Operationen der IR die möglichen alternativen Maschinenoperationen zuzuordnen. Das Kernproblem der Codegenerierung besteht darin, in jeder Codegenerierungsphase eine Auswahl aus dieser Menge von Maschinenoperationen zu treffen, ohne dass gute Alternativen vorzeitig verloren gehen. Die Schwierigkeiten der traditionellen Codegenerierung haben gezeigt, dass es sinnvoll ist, möglichst viele Maschinenoperationsvarianten nach jeder Codegenerierungsphase beizubehalten. So vermeidet man, dass die nachfolgenden Phasen durch zu restriktive Bindungen von Ressourcen gute Lösungen nicht mehr finden können. In CoLIR können Alternativen mit wechselseitigen Abhängigkeiten auf einfache Weise dargestellt und in handhabbarer Weise über verschiedene Codegenerierungsphasen propagiert werden.

COCOON

LANCE-System

In Abb. 4.1 ist ein Überblick über das Codegenerierungssystem *COCOON* (COnstraintbasierte COdeOptimierungeN) gegeben, das den Rahmen zur Entwicklung der Techniken dieser Arbeit darstellt und in das CoLIR eingebettet ist. Dem System COCOON ist das *LANCE-System*[1] vorgeschaltet. Zunächst werden ANSI-C-Programme durch das Front-End des LANCE-Systems nach LANCE-IR übersetzt. Das LANCE-System umfasst weiterhin ein Middle-End mit einer Menge von maschinenunabhängigen Standardoptimierungen (HLOs[2]) auf der LANCE-IR. Diese wird in COCOON

[1] LANCE=LS12 ANSI-C Environment.
[2] HLOs=High-Level-Optimierungen.

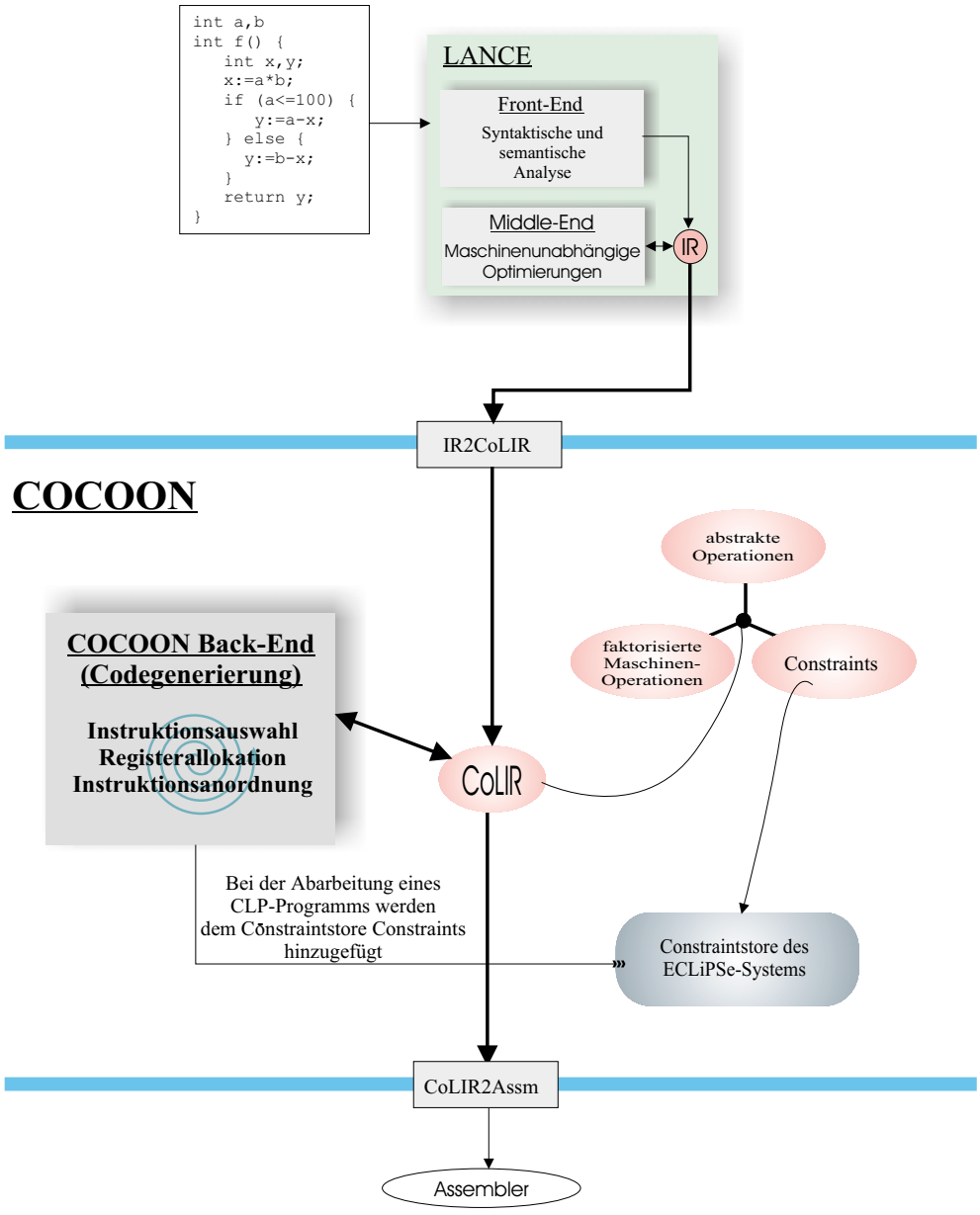


Abbildung 4.1.: Überblick über den Codegenerierungsprozess in COCOON.

zunächst in die IR von CoLIR transformiert und durch die Codegenerierung dann in die LIR von CoLIR umgesetzt. Einer der wesentlichen Schritte dabei ist es, den abstrakten Operationen der IR die constraintbasierten, faktorisierten Maschinenoperationen von CoLIR zuzuordnen.

Die Entitäten der constraintbasierten faktorisierten Maschinenoperationen, denen Ressourcen zugeordnet werden sollen (z.B. eine Funktionseinheit), werden durch Domainvariablen repräsentiert. Die Menge der möglichen alternativen Ressourcen, die einer dieser Entitäten zugeordnet werden kann, ist durch ihren Domain gegeben. Wechselseitige Beziehungen zwischen den Entitäten werden durch Constraints hergestellt, so dass eine Lösung für die Domainvariablen einer faktorisierten Maschinenoperation immer eine legale Maschinenoperation repräsentiert. Während der Codegenerierung werden die Domainvariablen an bestimmte Ressourcen ihrer Domains gebunden und somit die Menge der Alternativen sukzessiv auf eine bestimmte Lösung hin eingeschränkt. In einer Codegenerierungsphase werden nicht immer alle Domainvariablen fest gebunden. Verbleibende wechselseitige Beziehungen zwischen nicht instantiierten Domainvariablen bleiben durch Constraints im Constraintstore erhalten und können so an die nachfolgenden Phasen propagiert werden.

Dieses Kapitels umfasst die folgenden Themen:

- Eine Einführung in weitere notwendige Begriffe wird in Kapitel 4.1 gegeben.
- Kapitel 4.2 umfasst die Spezifikation der Struktur von CoLIR-Programmen, die Repräsentation der abstrakten Operationen in CoLIR und die Definition der constraintbasierten faktorisierten Maschinenoperationen, die als Spezialisierung der in Kapitel 2.1 eingeführten allgemeineren faktorisierten Maschinenoperationen definiert werden.
- Kapitel 4.3 beinhaltet eine formale Einführung der Begriffe und Notationen von graphbasierten Repräsentationen, wie Kontrollflussgraphen, Datenabhängigkeitsgraphen und Datenflussgraphen und deren Adaption an die spezifischen Eigenschaften von CoLIR.
- In Kapitel 4.4 wird ein Überblick über die in LANCE und COCOON integrierten maschinenunabhängigen HLOs gegeben.
- Kapitel 4.5 umfasst Richtlinien zur Propagierung von Constraints über den Constraintstore.
- In Kapitel 4.6 werden Mengen, Notationen und Begriffe definiert, die im Verlauf der weiteren Arbeit benutzt werden.

4.1. Grundlegende Begriffe

4.1.1. Benannte Verbünde

In dieser Arbeit wird eine besondere Form von Tupeln bzw. von Verbünden (in Programmiersprachen auch Records genannt) verwendet, die *benannte Verbünde* genannt

benannte Verbünde

werden und von der Form

$$id\ with\ [label_1 : K_1, \ldots, label_n : K_n]$$

sind, wobei id die Verbünde eines bestimmten Verbundtyps kennzeichnet. Jedes K_i ($1 \leq i \leq n$) ist eine Komponente eines bestimmten Typs, und $label_i$ dient der Benennung der Komponente K_i. Im Unterschied zu Tupeln können die Komponenten der benannten Verbünde in beliebiger Reihenfolge notiert und für den jeweiligen Kontext irrelevante Komponenten weggelassen werden.

Beispiel: Ein benannter Verbund einer Liste

$$list\ with\ [head : H,\ tail : L]$$

besteht aus dem Kopf der Liste H (benannt mit $head$) und dem Rest der Liste L (benannt mit $tail$). Ist von einer Liste $List$ nur der Kopf von Interesse, kann dieser durch $List = list\ with\ [head : H]$ referenziert werden.

4.1.2. Setzungen und Benutzungen

Setzung und Benutzung

In einer abstrakten Operation der Form $id_0 := op(id_1, ..., id_n)$ werden Variablen als Container zur Ablage von Werten interpretiert (im Unterschied zu funktionalen Sprachen, in denen die Bezeichner die Werte darstellen). Das Resultat der Operation op wird in der Programmvariablen id_0 abgelegt und der i-te Operand wird aus der Variablen id_i gelesen. Es wird somit von einem konkreten Speichermodell abstrahiert. Im Folgenden wird id_0 als die *Setzung* und jedes $id_1, ..., id_n$ eine *Benutzung* bzw. *Benutzung des i-ten Operanden* genannt.

Im Falle von Maschinenoperationen bezeichnen Setzungen und Benutzungen spezifische Ressourcen, deren Inhalte durch eine Maschinenoperation entweder geschrieben oder gelesen werden. Hier liegt also ein konkretes Speichermodell vor. In [118, S. 564] sind mögliche Setzungen gegeben durch:

- Modifikation von Registerinhalten durch Resultatablage von LOAD-Operationen, arithmetischen oder logischen Operationen und Adressoperationen.
- Modifikationen von Speicherinhalten durch Speichern von Registerinhalten oder direkten Datentransfers innerhalb des Speichers.
- Modifikation des Bedingungscodes wie das Setzen von Überlauf-, Unterlauf- oder Übertragsbits und Setzungen von Registerinhalten zur Residual-Control (die auch als Modifikation von Registerinhalten interpretiert werden können).
- Modifikation des Stackpointers.

Mögliche Benutzungen sind z.B.:

- Benutzung von Registerinhalten als Argumente in Operationen und beim Speichern von Registerinhalten.

- Zugriff auf die Inhalte von Speicherzellen.
- Abfragen des Bedingungscodes.

Für die Registertransferoperation rf_0:=*op*(rf_1,...,rf_n) einer Maschinenoperation wird in Analogie zu den abstrakten Operationen rf_0 als die Setzung und die rf_1,...,rf_n als Benutzungen bezeichnet. In einer faktorisierten Maschinenoperation repräsentieren die in der Registertransferoperation RF_0:=*op*(RF_1,...,RF_n) vorkommenden RF_i alternative Registerfiles. RF_0 wird als Setzung und die RF_1, ..., RF_n als Benutzungen bezeichnet. Die mit den $RF_0, ..., RF_n$ assoziierten alternativen Registerfiles werden Setzungs- bzw. Benutzungsalternativen genannt.

Setzungs- und Benutzungsalternativen

4.2. CoLIR

In CoLIR können IR, LIR und deren Zwischenstufen in einer Repräsentation dargestellt werden. Zunächst wird die Struktur eines CoLIR-Programms erläutert und dann auf die Repräsentation abstrakter Operationen und constraintbasierter faktorisierter Maschinenoperationen eingegangen.

Definition: Die Repräsentation eines Maschinenprogramms durch CoLIR ist von der Form

CoLIR

$$(stab, [fun_1, .., fun_n], C).$$

$stab$ ist eine Symboltabelle, die u.a. Informationen zum Typ der symbolischen Bezeichner eines Programms (z.B. Programmvariablen und Funktionsnamen) umfasst.
Jedes fun_i ist die Repräsentation einer *CoLIR-Funktion* mit

CoLIR-Funktion

$$fun_i = (fid, [b_1, ..., b_k]).$$

fid ist der Bezeichner der Funktion und jedes der b_i ist ein *CoLIR-Basisblock* der Form

CoLIR-Basisblock

$$b_i = bb\ with\ [label : L,\ in : V_{in},\ out : V_{out},\ mis : [mi_1, ..., mi_k]].$$

L ist eine symbolische Sprungadresse, die innerhalb eines CoLIR-Programms eindeutig vergeben wird. V_{in} ist die Menge der Programmvariablen, die im Basisblock benutzt werden und vor ihrer Benutzung nicht gesetzt wurden. V_{out} ist die Menge der Programmvariablen, die im Basisblock gesetzt werden und bei Austritt aus dem Basisblock noch lebendig sind (der Wert wird also in einem gemäß des Kontrollflusses nachfolgenden Basisblock noch benutzt). $[mi_1, ..., mi_k]$ ist eine Sequenz von *Maschineninstruktionen,* wobei jede Maschineninstruktion mi_i ($1 \leq i \leq k$) von der Form

$$[o_{1_i}, \ldots, o_{n_i}]$$

ist. Jedes $o_{1_i}, ..., o_{n_i}$ ist entweder eine abstrakte Operation oder eine faktorisierte Maschinenoperation aus CoLIR. C ist die Menge der Constraints über den Domainvariablen eines CoLIR-Programms und ist gegeben durch den aktuellen Inhalt des Constraintstores.

Die Definition der Maschineninstruktionen von CoLIR erlaubt es, dass abstrakte Operationen und faktorisierte Operationen gemischt vorkommen. Dies kann sinnvoll sein, wenn Teile eines Programms noch nicht in Code übersetzt werden sollen, um bestimmte Abstraktionsstufen beizubehalten.

Im ECLiPSe-System besteht die Möglichkeit, sich den Inhalt des Constraintstores in Form einer Liste $[c_1, ..., c_n]$ ausgeben zu lassen. Jedes c_i ($1 \leq i \leq n$) entspricht dabei der Termdarstellung eines Constraints aus dem Constraintstore. Somit ist die Möglichkeit gegeben, Constraints in Analysen und Optimierungen explizit einzubeziehen. Weiterhin besteht die Möglichkeit, Constraints aus dem Constraintstore zu eliminieren. Die Elimination von Constraints bietet die Möglichkeit, gegebenfalls zu restriktiv gefasste Bedingungen wieder zu lösen. Somit ist der Constraintstore eine transparente Struktur, die bei der Entwicklung von Optimierungstechniken einbezogen werden kann.

4.2.1. Abstrakte Operationen in CoLIR

Abstrakte Operationen in CoLIR reflektieren die elementaren Konstrukte zur Unterstützung imperativer Sprachen:

- *arithmetische/logische Operationen,*
- *Datentransfer-Operationen,*
- *Operationen zur Ausführung bedingter und unbedingter Sprünge* und
- *Funktionsaufrufe und Rücksprünge aus Funktionen.*

Diese sind der LANCE-IR ähnlich, werden aber in CoLIR allgemeiner gehalten. So besteht keine feste Bindung an einen Satz von Operatoren, wie es in der LANCE-IR der Fall ist. Die erlaubten Operatoren werden durch eine Signatur Σ_{IR} spezifiziert. Die in den abstrakten Operationen vorkommenden Programmvariablen sind durch eine abzählbar unendliche Menge Id gegeben. Typinformationen der Programmvariablen sind durch die mit einem CoLIR-Programm assoziierte Symboltabelle gegeben. Es wird davon ausgegangen, dass Programmvariablen typverträglich verwendet werden.

Klassen abstrakter Operationen

Die Operatoren sind in Klassen unterteilt, die eine spezifische Funktionalität widerspiegeln:

- *Arithmetische/logische Operationen* sind der Klasse alo zugeordnet und von der Form

$$id_0 := op(id_1, ..., id_n),$$

in der op ein n-stelliger arithmetischer oder logischer Operator ist.

- Operationen zur *Zuweisung von Konstanten* der Klasse $const$ der Form

$$id_0 := c,$$

wobei die Konstante c durch Terme wie $int(c')$, $real(c')$, etc. repräsentiert wird. Durch $\&(id)$ wird die symbolische Adresse der Speicheradresse einer Programmvariablen id dargestellt.

- Operationen zum Kopieren von Programmvariablen der Klasse $copy$ der Form

$$id_0 := cp(id_1)$$

- *LOAD-Operationen* der Klasse $load$ von der Form

$$id := ld(addr).$$

Darin ist $addr$ die Programmvariable, die den Wert der zu ladenden Adresse enthält.

- *STORE-Operationen* der Klasse $store$ von der Form

$$st(addr) := id$$

$addr$ ist wiederum die Programmvariable, der die Adresse zugewiesen wurde.

- *Stackoperationen* der Klassen $push$ und pop von der Form

$$push(id)$$

und

$$id := pop().$$

Diese existieren nicht in der LANCE-IR.

- *Unbedingte und bedingte Sprünge* der Klassen jmp und $cjmp$ von der Form

$$goto(label)$$

und

$$if\_goto(id, label),$$

in der id die Variable ist, die den Wert der booleschen Operation der Sprungbedingung enthält. $label$ muss einer mit den Basisblöcken assoziierten symbolischen Sprungadressen entsprechen.

- *Operationen zur Modellierung von Zero-Overhead-Loops* mit den Klassen $zloop$ und $zjmp$ sind von der Form

  $$zloop(id, label)$$

  und

  $$if\_nz\_goto(label).$$

  Es wird davon ausgegangen, dass dem Bezeichner id im ersten Argument von $zloop(id, label)$ eine Konstante zugeordnet wird, die die Anzahl der Schleifendurchläufe spezifiziert. $label$ ist die logische Adresse der ersten auszuführenden Maschineninstruktion nach der Schleife. $if\_nz\_goto(label')$ ist eine Dummy-Operation, die den Rücksprung an den Beginn der Schleife (bei $label'$) bzw. den Austritt aus der Schleife repräsentiert. Im Maschinenprogramm ist diese Operation nicht explizit präsent. Sie wird implizit während der Ausführung der letzten Maschineninstruktion der Schleife durchgeführt und umfasst die Dekrementierung des Schleifenzählers und das Prüfen der Abbruchbedingung der Schleife.

- *Operationen zum Aufruf von Funktionen der Klasse $call$ von der Form*

  $$id_0 := call(label, id_1, ..., id_n).$$

  Das erste Argument $label$ ist die symbolische Sprungadresse einer n-stelligen benutzerdefinierten Funktion (daher hat $call$ $n + 1$ Argumente). Es wird davon ausgegangen, dass durch $label$ auf die verfügbaren Informationen der Funktion zugegriffen werden kann. Alle weiteren Bezeichner stellen die Argumente dar. id_0 wird der Rückgabewert der Funktion zugewiesen. Prozeduren sind im Folgenden als Funktionen mit Rückgabewert *void* zu verstehen, und es wird nicht weiter zwischen Funktionen und Prozeduren differenziert.

- *Operationen zum Rücksprung aus Funktionen der Klasse $return$ der Form*

  $$ret(id)$$

  und

  $$ret.$$

  Dem Argument in $ret(id)$ ist der Rückgabewert der Funktion zugeordnet. Besitzt die Funktion keinen Rückgabewert, wird ret verwendet.

Abstrakte Operationen einer Klasse haben immer die gleiche Struktur und Bedeutung. Durch die Einführung der Klassen wird eine Unabhängigkeit von den konkreten Operatorsymbolen erreicht, so dass später eingeführte Analysen auch bei Adaption an andere Operatorsymbole einsetzbar bleiben.

Die Klassen der abstrakten Operationen werden im Folgenden durch die Menge

$$\mathcal{OC} = \{alo, const, copy, load, store, jmp, cjmp, zloop, zjmp, call, push, pop, return\}$$

präsentiert. Es sei hier schon bemerkt, dass auch jede faktorisierte Maschinenoperation aus CoLIR einer dieser Klasse zugeordnet wird.

Klasse	abstrakte Operation		ECLiPSe-Term
alo	$id_0 := op(id_1, \ldots, id_n)$	$\leftrightarrow$	$aop(alo, op,\ id_0 := [id_1, \ldots, id_n])$
$const$	$id_0 := c$	$\leftrightarrow$	$aop(const, c,\ id_0 := [])$
$copy$	$id_0 := cp(id_1)$	$\leftrightarrow$	$aop(copy, cp,\ id_0 := [id_1])$
$load$	$id_0 := ld(addr)$	$\leftrightarrow$	$aop(load, ld,\ id_0 := [addr, \_])$
$store$	$st(addr) := id$	$\leftrightarrow$	$aop(store, st,\ \_ := [addr, id])$
$push$	$p(id)$	$\leftrightarrow$	$aop(push, p,\ \_ := [id])$
pop	$id := p()$	$\leftrightarrow$	$aop(pop, p,\ id := [\_])$
jmp	$goto(label)$	$\leftrightarrow$	$aop(jmp, goto,\ \_ := [label])$
$cjmp$	$if\_goto(id, label)$	$\leftrightarrow$	$aop(cjmp, if\_goto,\ \_ := [id, label])$
$zloop$	$zloop(id, label)$	$\leftrightarrow$	$aop(zloop, zloop,\ \_ := [id, label])$
$zjmp$	$if\_nz\_goto(label)$	$\leftrightarrow$	$aop(zjmp, if\_nz\_goto,\ \_ := [label])$
$call$	$id_0 := call(label, id_1, \ldots, id_n)$	$\leftrightarrow$	$aop(call, call, id_0 := [label, id_1, \ldots, id_n])$
$return$	$ret(id)$	$\leftrightarrow$	$aop(return, ret,\ \_ := [id])$
$return$	ret	$\leftrightarrow$	$aop(return, ret,\ \_ := [])$

Abbildung 4.2.: Abstrakte Operationen in ECLiPSe.

Darstellung abstrakter Operationen in ECLiPSe

Komplexe Datenstrukturen wie Tupel, Listen und Verbünde (Records) werden in logischen Sprachen durch Terme dargestellt[3]. Eine abstrakte Operationen `x:=y+z` wird in ECLiPSe durch den folgenden Term notiert:

```
aop(alo,'+',x:=[y,z])
```

In dieser Darstellung besteht die Repräsentation einer abstrakten Operation aus drei Komponenten:

1. Die Klasse der abstrakten Operation (`alo`).
2. Das Operatorsymbol (`+`).
3. Die Programmvariablen des Resultats (`x`) und der Operanden (`y` und `z`) werden in Form einer Bezeichnerzuordnung `x:=[y,z]` repräsentiert. Die Notation `x:=[y,z]` repräsentiert den Term `:=(x,[y,z])`. Symbole wie `':='` können in ECLiPSe (und Prolog) als Funktoren verwendet und als infix-Operator deklariert werden.

Die Bezeichnerzuordnungen der abstrakten Operationen werden in einer einheitlichen Form $id_0 := [id_1, \ldots, id_n]$ dargestellt. Abstrakte Operationen, die keinen expliziten Bezeichner für das Resultat besitzen, werden in der Form $\_ := [id_1, \ldots, id_n]$ repräsentiert. Das ist der Fall für Operationen der Klassen jmp, $cjmp$, $zloop$, $zjmp$,

[3]Tupel und Listen besitzen dabei spezielle Darstellungsformen, da sie sehr häufig verwendet werden und sie durch die spezielle Repräsentation effizienter umgesetzt werden können.

$return$, $push$ und $store$ (so wird z.B. eine Operation der Klasse $store$ in der Form $\_ := st(addr, id)$ notiert).

Operationen der Klasse $const$ besitzen kein Argument ($id := []$), da die Konstante als 0-stelliger Operator interpretiert wird. Die Konstante c wird durch den Operator der ECLiPSe-Repräsentation abstrakter Operationen gegeben. Die Abb. 4.2 zeigt die Repräsentation abstrakter Operationen in ECLiPSe.

Die Bezeichnerzuordnung der LOAD-Operation besitzt in der ECLiPSe-Darstellung zwei Operanden. Dies wurde zur einheitlichen Handhabung von Analysen über der IR und der LIR gemacht, da in den faktorisierten Maschinenoperationen die LOAD-Operationen zwei Operanden besitzen. Der zweite Operand erhält durch die Registerfilezuordnung einen Sinn, da durch ihn die möglichen Speicherbänke, aus denen geladen werden kann, explizit repräsentiert werden. Auch den Setzungen von faktorisierten Maschinenoperationen der Klasse jmp, $cjmp$, $zloop$, $zjmp$, $return$, $push$ und $store$ werden ST-Komponenten zugeordnet. Bei den Operationen der Klasse $store$ ist das auch eine der Speicherbänke. Bei den Klassen jmp, $cjmp$, $zloop$, $zjmp$, $return$, $push$ kann die Setzung durch den Programmzähler gegeben werden.

ECLiPSe bietet eine Spracherweiterung von Prolog, um die Struktur von Termen im Sinne benannter Verbünde zu spezifizieren. Die Datenstruktur der abstrakten Operationen wird durch folgende Anweisung an das ECLiPSe-System spezifiziert:

```
:- export struct(aop(class,op,bs)).
```

Das partielle Programm

```
x=a+b;
y=x+b;
ret y;
```

kann jetzt in einem ECLiPSe-Programm wie folgt notiert werden

```
:- S1=aop with [ class : alo,
                 op   : +,
                 bs   : x:=[a,b]],
   S2=aop with [class : alo,
                 op   : +,
                 bs  : y:=[x,b]],
   S3=aop with [class : return,
                 op   : ret,
                 bzo  : _:=[y]].
```

Bzgl. der Bezeichnerzuordnungen liegt ein CoLIR-Programm in lokaler SSA-Form vor. Das bedeutet, dass die Setzungen der Bezeichnerzuordnungen innerhalb eines Basisblocks eindeutig benannt werden. In Abb. 4.3 mussten dazu nur die Setzungen der Operationen in den Zeilen S1 und S4 (und deren Benutzungen) umbenannt werden.

```
S1   y=a*b;                  y1=a*b;  !!!
S2   y=a+y;                  y=a+y1;
S3   if (y>2){       →       if (y>2) {
S4      y=y+2;                  y2=y+2
S5      y=y+a;                  y=y2+a;
S6   } else {                } else {
S7      y=2;                    y=2;
S8   }                       }
S9   x=y+2;                  x=y+2;
```

Abbildung 4.3.: Transformation in lokale SSA-Form.

In normaler SSA-Form hätten auch Setzungen der Operationen in den Zeilen S2, S5 und S7 umbenannt werden müssen, und vor Zeile S9 hätte ein $\phi - Knoten$ eingefügt werden müssen. Lokale SSA-Form ist hinreichend, um unnötige Abhängigkeiten innerhalb der Basisblöcke zu vermeiden und erübrigt die explizite Handhabung von ϕ-Knoten.

In CoLIR-Programmen werden alle Referenzen auf statische und globale Variablen schon auf der IR-Ebene (in CoLIR) durch explizite LOAD- und STORE-Operationen umgesetzt. Weiterhin werden auch lokale Variablen, die per Pointer referenziert werden im Speicher abgelegt. Auch hier werden die entsprechenden Referenzen durch LOAD- und STORE-Operationen umgesetzt[4]. Da alle Speicherreferenzen explizit gegeben sind, können alle weiteren Bezeicher der Bezeichnerzuordnungen an Registerfiles gebunden werden (natürlich bis auf die Setzungen von STORE-Operationen und die Benutzungen des zweiten Operanden von LOAD-Operationen). Hierdurch wird die Codegenerierung davon befreit, jeweils bei Bezeichnern eine Fallunterscheidung durchzuführen ob ein Bezeichner gegebenenfalls durch eine LOAD- bzw. STORE-Operation umzusetzen ist. Redundantes Laden und Speichern kann durch bekannte Analysen und Optimierungen eliminiert werden (siehe [94]).

4.2.2. Constraintbasierte faktorisierte Maschinenoperationen

Constraintbasierte faktorisierte Maschinenoperationen repräsentieren alternative Möglichkeiten die Operationen eines Programms in korrespondierende Maschinenoperationen umzusetzen. Alternative Maschinenoperationen unterscheiden sich in der Verwendung der Ressourcen, die zur Umsetzung einer bestimmten Operation auf einem Prozessor genutzt werden können. Ressourcen für Setzungen, Benutzungen, Funktionseinheiten und Instruktionstypen werden in faktorisierten Maschinenoperationen daher durch Domainvariablen dargestellt. Die Domains repräsentieren die alternativen Ressourcen, die zur Bildung einer Maschinenoperation erlaubt sind. Die Ressourcen eines Zielprozessors werden durch die folgenden charakteristischen Mengen

[4]Existieren Pointer, die potenziell auf alle lokalen Variablen zeigen können müssen auch alle Referenzen lokaler Variablen (nicht die Temporären) einer Funktion durch LOAD- und STORE-Operationen realisiert werden.

gegeben:

- Die Menge der ST-Komponenten $\mathcal{ST} = \mathcal{RF} \cup \mathcal{M} \cup \mathcal{T}$ ist gegeben durch die Menge der Registerfiles $\mathcal{RF}$, der Speicherbänke $\mathcal{M}$ sowie der Menge $\mathcal{T}$ der flüchtigen Komponenten.
- Die Menge der Register
$$\mathcal{R} = \bigcup_{rf_i \in \mathcal{RF}} regs(rf_i)$$
wobei $regs(rf)$ die Register des Registerfiles rf bezeichnet.
- Die Menge $\mathcal{FE}$ der Funktionseinheiten des Zielprozessors.
- Die Menge $\mathcal{IT}$ der Instruktionstypen des Zielprozessors.

Jede der Mengen $\mathcal{RF}$, $\mathcal{M}$, $\mathcal{FE}$ und $\mathcal{IT}$ enthält jeweils eine (möglicherweise leere) Teilmenge *virtueller Ressourcen* $\mathcal{VRF} \subset \mathcal{RF}$, $\mathcal{VM} \subset \mathcal{M}$, $\mathcal{VT} \subset \mathcal{T}$, $\mathcal{VFE} \subset \mathcal{FE}$ und $\mathcal{VIT} \subset \mathcal{IT}$, die Modellierungszwecken dienen.

---

constraintbasierte faktorisierte Maschinenoperation (FMO)

Definition: Eine *constraintbasierte faktorisierte Maschinenoperation (FMO)* ist ein benannter Verbund des Typs

fmo with [class: $Class$,
op: Op,
ops: Ops,
bs: $id_0 := (id_1, ..., id_n)$,
rfs: $RF_0 := (RF_1, \ldots, RF_n)$,
rs: $Reg_0 := (Reg_1, \ldots, Reg_n)$,
fe: FE,
it: IT,
sos: _ := $(so_1, ..., so_n)$,
spec: C
]

- $Class \in \mathcal{OC}$ ist die Klasse einer FMO, die der Klassifizierung analog zu den abstrakten Operatoren entspricht.
- Op ist das Operatorsymbol der auszuführenden Operation.
- Ops ist eine Menge alternativer Operatoren zur Umsetzung von Op.

Bezeichnerzuordnung

- $id_0 := (id_1, ..., id_n)$ ist eine *Bezeichnerzuordnung* und ordnet dem Resultat und den Operanden symbolische Bezeichner zu, die in einem Programm die Datenflussbeziehungen zwischen den FMOs definieren.

Registerfilezuordnung

- $RF_0 := (RF_1, \ldots, RF_n)$ ist eine *Registerfilezuordnung*, in der $RF_0, ..., RF_n$ Domainvariablen mit $D(RF_0) \subseteq \mathcal{ST}$, $D(RF_1) \subseteq \mathcal{ST}$, ..., $D(RF_n) \subseteq \mathcal{ST}$ sind. RF_0 ist die Setzung zum Schreiben des Resultats der Operation Op. Jedes RF_i $(1 \leq i \leq n)$ repräsentiert die Benutzung des i-ten Operanden.

- $Reg_0 := (Reg_1, \ldots, Reg_n)$ ist eine *Registerzuordnung*(mit $D(Reg_0) \subseteq \mathcal{R}$, $D(Reg_1) \subseteq \mathcal{R}$, ..., $D(Reg_n) \subseteq \mathcal{R}$), die zu jedem RF_i ($0 \leq i \leq n$) die zur Auswahl stehenden Register spezifiziert. *Registerzuordnung*
- Für die Domainvariable FE ist $D(FE) \subseteq \mathcal{FE}$ die Menge alternativer Funktionseinheiten, auf der der Operator Op ausgeführt werden kann.
- Für die Domainvariable IT ist $D(IT) \subseteq \mathcal{IT}$ die Menge der alternativen Instruktionstypen von Maschineninstruktionen, in denen die durch die FMO repräsentierten Maschinenoperationen vorkommen können.
- $\_ := (so_1, \ldots, so_n)$ ist eine Zuordnung der Operanden zu *Sub-Operationen* und dient der Modellierung komplexer Maschinenoperationen. Sub-Operationen werden ebenfalls durch FMOs repräsentiert und werden *Sub-FMOs* genannt. so_i ist $none$, wenn mit dem i-ten Operand keine Sub-Operation assoziiert ist. Ist dem i-ten Operand eine Sub-Operation zugeordnet, so sind die Setzungsalternativen von RF_i auf flüchtige Komponenten der Menge $\mathcal{T}$ beschränkt: $D(RF_i) \subseteq \mathcal{T}$. *Sub-FMOs*
- C ist ein Constraint, das die wechselseitigen Abhängigkeiten zwischen den Ressourcen der FMO und der Sub-FMOs definiert.

FMOs werden in ECLiPSe ebenfalls durch benannte Verbünde dargestellt. Die Menge der Constraints, die mit einer FMO assoziiert sind, ist durch den jeweils aktuellen Zustand des Constraintstores des ECLiPSe-Systems gegeben. Die Erzeugung einer FMO während der Codegenerierung besteht im wesentlichen aus der Initialisierung der Domainvariablen und der Generierung der Constraints. Das Konzept wird anhand der Regel eines benutzerdefinierten Constraints $gen\_fmo(Op, FMO)$ illustriert:

```
gen_fmo(+,F) :-
   F = fmo with [class: alo,
                 op:     '+',
                 rfs:    Def:=[A1,A2],
                 it:     IT],
   Def::[ar,af],
   A1::[ar,sr,mr,ax],
   A2::[ay,af],
   IT::[1,4,5,8,9],
   IT#=1 #=> Def#=ar.
```

Darin werden die Domainvariablen des FMOs F gemäß der Addition initialisiert, wobei der Befehlssatz der ADSP2100-Familie zu Grunde gelegt wird. Die Notation $Def :: [ar, af]$ ist die ECLiPSe-Notation für $Def \in \{ar, af\}$. Das Constraint $IT = 1 \Rightarrow Def = ar$ spezifiziert die Restriktion bzgl. der Parallelausführung innerhalb des Instruktionstypen 1 (siehe Kapitel 1.3 Seite 11) und wird bei Ausführung von $gen\_fmo/2$ im Constraintstore abgelegt. Der constraintbasierte Ansatz erlaubt eine recht natürliche Formulierung der alternativen Ressourcen und der wechselseitigen Beziehungen zwischen ihnen. Die Anwendung des Constraints $gen\_fmo/2$ kann in einem CLP-Programm wie folgt aussehen:

```
p1(F) :-

   ...,
   F=fmo with [op:'+'],
   gen_fmo(+,F),
   ....

p2(F) :-

   ...,
   F=fmo with [it:IT],
   IT#=1,
   ....

cg(F):-
    p1(F),
    p2(F).
```

Es wird angenommen, dass die Prädikate $p1/1$ und $p2/1$ unterschiedliche Codegenerierungsphasen implementieren und im Prädikat $cg/1$ nacheinander aufgerufen werden. Jedem dieser Prädikate wird das FMO F als Argument übergeben. Innerhalb von $p1/1$ wird $gen\_fmo(+, F)$ aufgerufen, welches F initialisiert und das primitive Constraint $IT = 1 \Rightarrow Def = ar$ im Constraintstore ablegt, das dort auf seine Re-Aktivierung wartet. In $p2/1$ wird nun der Instruktionstyp von FMO F an den Typ 1 gebunden. Dadurch wird das Constraint $IT = 1 \Rightarrow Def = ar$ reaktiviert und führt zu der Bindung von Def an ar.

4.2.3. Verschmelzung von abstrakten Operationen und FMOs

Die Repräsentation der FMOs umfasst alle Komponenten, die zur Darstellung abstrakter Operationen notwendig sind. Daher werden abstrakte Operationen in CoLIR ebenfalls durch FMOs repräsentiert. Die Unterscheidung zwischen abstrakten Operationen und FMOs ist durch die unterschiedlichen Operatoren der IR und der LIR möglich. Die Ressourcen der FMOs können für abstrakte Operatoren genutzt werden, um für die Setzungen und Benutzungen schon mögliche Ressourcen festzulegen. Es wird im Folgenden der Begriff *CoLIR-Operation* verwendet, wenn nicht zwischen FMO und abstrakter Operation unterschieden werden muss.

CoLIR-Operation

IR-reine, FMO-reine und elementare CoLIR-Programme

Ein CoLIR-Programm wird *IR-rein* genannt, wenn es nur abstrakten Operationen der IR enthält. Es wird *FMO-rein* genannt, wenn es nur FMOs enthält, und es heißt *elementar*, wenn alle FMOs keine Sub-FMOs enthalten.

4.2.4. Eingangs- und Ausgangsvariablen

Eingangs- und Ausgangsvariablen

Die Elemente der Mengen V_{in} und V_{out} eines Basisblocks werden als *Eingangs-* und *Ausgangsvariablen* bezeichnet. Jedes Element $v \in V_{in}$ bzw. $v \in V_{out}$ ist ein benannter Verbund des Typs

$$var\ with\ [\quad rf : RF_0 \\ reg : Reg_0 \\ bs : id_0 \qquad]$$

Wie schon erwähnt, dienen die Eingangs- und Ausgangsvariablen zur zusätzlichen Spezifikation von ST-Komponenten, in denen sich deren Programmvariablen bei Eintritt und bei Austritt aus einem Basisblock befinden müssen. Die Verbundkomponenten rf, reg und bs werden als *Eingangs- bzw. Ausgangssetzungen bzgl. der Registerfilezuordnung, Registerzuordnung oder Bezeichnerzuordnung* bezeichnet. Zur Spezifikation, welche Programmvariablen eines CoLIR-Programms Eingangs- und Ausgangssetzungen in bestimmten Basisblöcken besitzen, werden zunächst die Begriffe Pfad, setzungsfreier Pfad und benutzungsfreier Pfad eingeführt.

Eingangs- und Ausgangssetzungen

Definition: Ein *Pfad eines Basisblocks* $bb\ with\ [mis : [mi_1, ..., mi_n]]$ ist eine beliebige geschlossene Teilsequenz aus $[mi_1, ..., mi_n]$.

Pfad eines Basisblocks

Definition: Es wird gesagt, dass ein Pfad $[mi_i, ..., mi_j]$ in einem Basisblock bzgl. einer Ressource bzw. einer Programmvariablen *setzungsfrei* ist, wenn die Ressource in keiner CoLIR-Operation des Pfades gesetzt wird. Analog sagen wir, dass der Pfad bzgl. einer Ressource bzw. einer Programmvariablen *benutzungsfrei* ist, wenn die Ressource in keiner CoLIR-Operation in diesem Pfad benutzt wird.

setzungsfreier und benutzungsfreier Pfad

In einem Basisblock $b = bb\ with\ [mis : MIs, in : V_{in}, out : V_{out}]$ mit $MIs = [mi_1, ..., mi_n]$ existiert zu einer Programmvariablen id eine Eingangsvariable $var\ with\ [bs : id] \in V_{in}$, wenn es mindestens einen für id setzungsfreien Pfad $[mi_1, \ldots, mi_j]$ in MIs gibt, und v in einer CoLIR-Operation $cop \in mi_j$ benutzt wird. Zu jeder Programmvariablen id existiert eine Ausgangsvariable $var\ with\ [bs : id] \in V_{out}$, wenn es einen Pfad $(mi_j, mi_{j+1}, \ldots, mi_n)$ im Basisblock gibt, so dass id durch eine CoLIR-Operation $cop \in mi_j$ gesetzt wird, und der Pfad $[mi_{j+1}, \ldots, mi_n]$ für v setzungsfrei ist.

4.2.5. Positionen

Positionen werden benötigt, um bestimmte Domainvariablen in FMOs und in Eingangs- und Ausgangsvariablen präzise referenzieren zu können.

Position

Definition: Eine *Position* ist eine Sequenz der Form $[p_1, ..., p_n]$ mit $p_1, ..., p_n \in N_0^+$. Eine Position referenziert eine Komponente in der Registerfile-, Register-, Bezeichner- oder Sub-FMO-Zuordnung einer FMO. Für eine gegebene FMO f mit

$$f = fmo\ with\ [\quad \begin{array}{l} \text{rfs: } RF_0 := (RF_1, .., RF_n) \\ \text{rs: } Reg_0 := (Reg_1, .., Reg_n) \\ \text{bs: } id_0 := (id_1, .., id_n) \\ \text{sos: } \_ := (so_1, .., so_n) \qquad] \end{array}$$

und Position pos spezifiziert der Ausdruck f^k_{pos} ($k \in \{rfs, rs, bs\}$) die folgende Komponente:

1. Ist $pos = []$ so ist $f^k_{pos} = \perp$ ([] ist die leere Liste und $\perp$ ist das undefinierte Ergebnis).
2. Ist $pos = [p]$ und $0 \leq p \leq n$ dann ist
   a) $f^{rfs}_{pos} = RF_p$
   b) $f^{rs}_{pos} = Reg_p$
   c) $f^{bs}_{pos} = id_p$

   sonst ist $f^k_{pos} = \perp$.
3. Für $pos = [p_1, p_2, ..., p_n]$ ist $f^k_{pos} = (so_{p_1})^k_{[p_2,...,p_n]}$ falls $1 \leq p_1 \leq n$ und $so_{p_1} \neq none$. Ansonsten ist $f^k_{pos} = \perp$.

Für Eingangs- und Ausgangsvariablen aus V_{in} und V_{out} wird der Begiff der Position wie folgt definiert. Für v mit

$$v = var\ with\ [\quad \begin{array}{l} rf : RF_0 \\ reg : Reg_0 \\ bs : id_0 \qquad] \end{array}$$

und Position pos spezifiziert der Ausdruck v^k_{pos} ($k \in \{rfs, regs, bs\}$) die folgende Komponente:

1. Ist $pos = [0]$ dann ist
   a) $v^{rfs}_{pos} = RF_0$
   b) $v^{regs}_{pos} = Reg_0$
   c) $f^{bs}_{pos} = id_0$
2. sonst ist $v^k_{pos} = \perp$.

4.3. Graphbasierte Zwischenrepräsentationen

Graphbasierte Zwischenrepräsentationen zu CoLIR-Programmen stellen besondere Beziehungen zwischen den CoLIR-Operationen des Programms heraus. Im Folgenden werden vorwiegend knotenmarkierte und kantenmarkierte Graphen betrachtet. Dabei werden bei knotenmarkierten Graphen die Knoten und deren Markierungen als Synonyme verwendet. Graphrepräsentationen werden über Maschineninstruktionen definiert, die gleichzeitig sowohl FMOs als auch abstrakte Operationen enthalten können.

4.3.1. Kontrollflussgraphen

Der Kontrollflussgraph reflektiert unmittelbar die Ausführungsreihenfolge von Maschineninstruktionen sowie die Verzweigungsstruktur eines Programms, impliziert durch bedingte und unbedingte Sprünge (nicht aber durch Funktionsaufrufe). Die Standarddefinition von Kontrollflussgraphen korrespondiert zu der in [118]. Maschineninstruktionen sind gemäß dieser Standarddefinition nur durch jeweils eine Maschinenoperation gegeben.

Kontrollflussgraph

Definition: Der *Kontrollflussgraph* einer Funktion ist ein gerichteter knoten- und kantenmarkierter Graph $CFG = (N, E, n_{start}, n_{stop})$ mit folgenden Eigenschaften:

- Zu jeder Maschineninstruktion mi existiert ein Knoten $n \in N$, der mit mi markiert ist.
- Es existieren zwei ausgezeichnete, nichtmarkierte Knoten n_{start} und n_{stop}. n_{start} ist der eindeutig definierte Eintrittsknoten in den Graphen und n_{stop} der eindeutig definierte Endknoten des Graphen.
- $E \subset (N \cup \{n_{start}\}) \times \{true, false, none\} \times (N \cup \{n_{stop}\})$ ist eine Menge von Kanten, die den Kontrollfluss zwischen den Instruktionen widerspiegeln mit Kantenmarkierungen aus der Menge $\{true, false, none\}$.

Es gilt $(n_1, none, n_2) \in E$, wenn n_1 und n_2 in der Programmausführung strikt aufeinander ausgeführte Maschinenstruktionen sind, d.h. n_2 immer unmittelbar auf n_1 ausgeführt wird. Es gilt $(n_{start}, none, n) \in E$, wenn n die erste auszuführende Maschineninstruktion der Funktion ist. $(n, none, n_{stop}) \in E$ gilt für alle Knoten n, die als letzte Maschineninstruktion in der Funktion ausgeführt werden. Knoten, die einen bedingten Sprung der Klasse $cjmp$, $zloop$ und $zjmp$ darstellen, haben zwei ausgehende Kanten $(n_1, true, n_2) \in E$ und $(n_1, false, n_3) \in E$ respektive der Auswertung der Bedingung zu $true$ oder $false$.

Die folgende Definition spiegelt eine gröbere Sichtweise des Kontrollflusses einer Funktion wider, und gibt nun auch die formale Definition eines Basisblocks wieder.

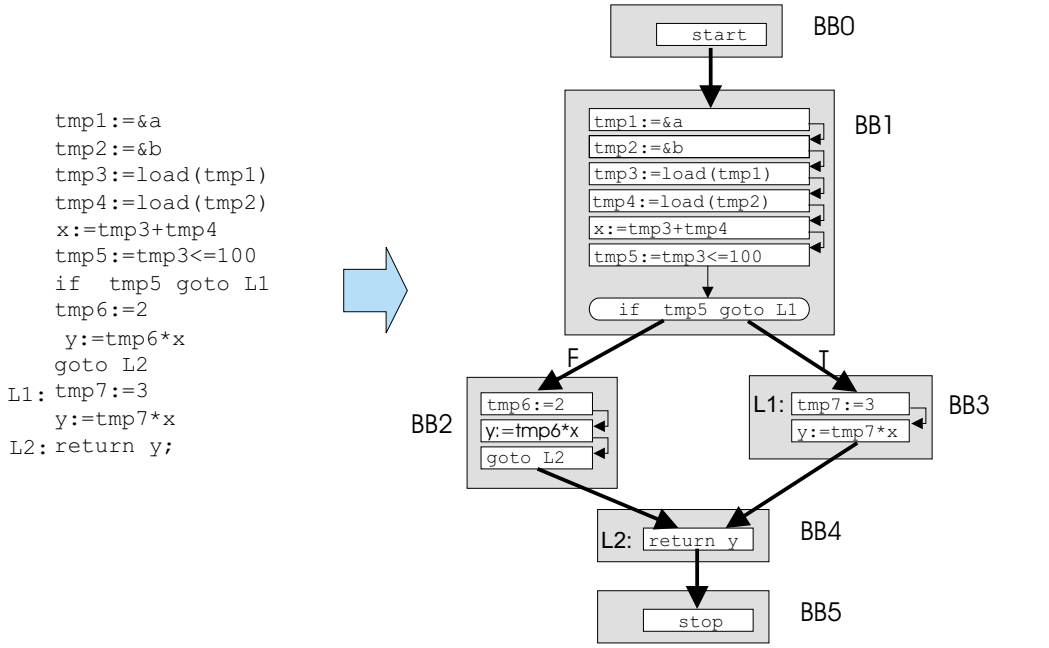


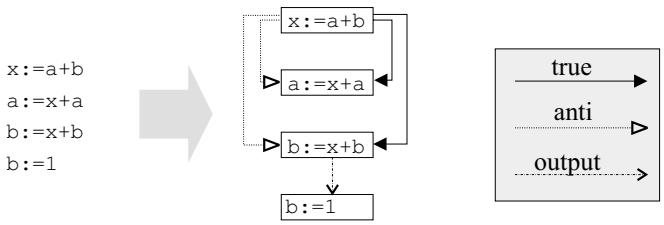
Abbildung 4.4.: Kontrollflussgraph und Basisblockgraph.

Basisblock und Basisblockgraph

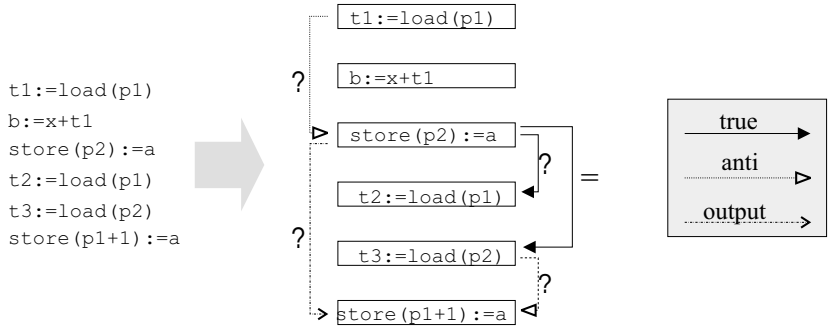
Definition: Ein *Basisblock* eines Kontrollflussgraphen CFG ist gegeben durch die Knoten n_{start} oder n_{stop} oder einen Pfad $(n_1, ..., n_m)$ maximaler Länge mit $n_1 \neq n_{start}$ und $n_m \neq n_{stop}$, so dass nur n_1 mehr als eine eingehende Kante besitzt und nur n_m mehr als eine ausgehende Kante aufweist. Der *Basisblockgraph* einer Funktion entsteht aus dem Kontrollflussgraphen CFG durch sukzessive Reduktion der Basisblöcke auf jeweils einen Knoten und der entsprechenden Anpassung der Kantenbeziehungen.

In Abb. 4.4 ist der Kontrollflussgraph und der Basisblockgraph zu einer Funktion gezeigt. Die Knoten des Kontrollflussgraphen sind durch weiße Kästchen gegeben. Die Knoten eines Basisblockgraphen umfassen die Knoten des Kontrollflussgraphen und sind durch die grauen Kästchen dargestellt. Die Kanten des Basisblockgraphen sind fett hervorgehoben.

In Kontrollflussgraphen, die parallel ausführbare CoLIR-Operationen umfassen, beinhalten die entsprechenden Maschineninstruktionen nur jeweils eine Verzweigungsoperation. Die Definition von Basisblöcken bedarf keiner Modifikation. Bezüglich der Definition von Pfaden sagen wir, dass es einen *Pfad von einer CoLIR-Operation cop nach cop'* im Basisblock gibt, wenn es einen Teilpfad (mi_i,. . .,mi_j) mit $i < j$ im Basisblock gibt, so dass cop in Maschineninstruktion mi_i und cop' in Maschineninstruktion mi_j enthalten ist.



(a) Datenabhängigkeiten zwischen Programmvariablen



(b) Datenabhängigkeiten zwischen LOAD- und STORE-Operationen

Abbildung 4.5.: Datenabhängigkeitsgraph.

4.3.2. Datenabhängigkeitsgraphen

Die Maschineninstruktionen eines Basisblocks geben gemäß ihrer Anordnung eine bestimmte Reihenfolge zur Ausführung von CoLIR-Operationen vor (bei denen alle CoLIR-Operationen innerhalb einer Maschineninstruktion parallel ausgeführt werden). Die CoLIR-Operationen eines Basisblocks können in der Regel umgeordnet werden, wenn bestimmte Abhängigkeiten - die Datenabhängigkeiten - zwischen ihnen berücksichtigt werden.

Unter Betrachtung alternativer Ressourcen können Setzungen und Benutzungen nicht unmittelbar auf bestimmte Ressourcen zurückgeführt werden. Daher werden Datenabhängigkeiten auf der Basis der Bezeichnerzuordnungen abgeleitet (Programmvariablen können ja durchaus als abstrakte Ressource betrachtet werden). Im Unterschied zur Standarddefinition von Datenabhängigkeitsgraphen sind die Kanten hier mit der Position von Benutzungen markiert (die Menge der Positionen wird mit Pos bezeichnet). Ist keine sinnvolle Angabe einer Position möglich, wird eine Kante mit $none$ markiert. Weiterhin tragen die Kanten eine Markierung $m \in \{t, f, o\}$, die Aufschluss über die Art der Abhängigkeit gibt (s.u.).

Datenabhängigkeits-graph

Definition: Der *Datenabhängigkeitsgraph* eines Basisblocks ist ein gerichteter knoten- und kantenmarkierter Graph $DDG = (N, E)$. Zu jeder CoLIR-Operation *cop* des Basisblocks gibt es einen Knoten $n \in N$, der mit *cop* markiert ist. Für eine Kante $(n_i, m, n_j) \in E$ mit Markierung $m \in \{t, f, o\} \times (Pos \cup \{none\})$ existiert im Basisblock ein Pfad $(n_i, n_{i+1}, \ldots, n_{j-1}, n_j)$ von n_i nach n_j mit einer der folgenden Eigenschaften:

1. n_i setzt den Inhalt einer Ressource, n_j benutzt diesen Inhalt an Position p, und der Pfad von n_{i+1} nach n_{j-1} ist setzungsfrei (siehe S. 91). Diese Art der Abhängigkeit wird *Datenflussabhängigkeit (true dependency)* genannt. Dann ist $m = (t, p)$, und es wird auch $n_i \hookrightarrow^{t,p} n_j$ geschrieben.

Datenflussabhängigkeit

2. n_i benutzt den Inhalt einer Ressource an Position p, n_j setzt diesen Inhalt, und der Pfad von n_{i+1} nach n_{j-1} ist setzungsfrei. Diese Art der Abhängigkeit wird *Antiabhängigkeit (false dependency)* genannt. Dann ist $m = (f, p)$, und es wird auch $n_i \hookrightarrow^{f,p} n_j$ geschrieben.

Antiabhängigkeit

3. n_i setzt den Inhalt einer Ressource, und auch n_j setzt diesen Inhalt, und der Pfad von n_{i+1} nach n_{j-1} ist setzungsfrei (in [118] wird hier auch die Benutzungsfreiheit (siehe S. 91) gefordert). Diese Art der Abhängigkeit wird *Ausgabeabhängigkeit (output dependency)* genannt. Dann ist $m = (o, none)$, und es wird auch $n_i \hookrightarrow^{o,none} n_j$ geschrieben.

Ausgabeabhängigkeit

Für Programmvariablen sind die Datenabhängigkeiten einfach bestimmbar. Generell ist die Bestimmung der Datenabhängigkeiten aber nicht immer genau möglich. Für Speicherzugriffe, deren Adressen durch arithmetische Ausdrücke oder beliebige Zeiger gegeben ist, ist generell nicht mehr entscheidbar, ob die Adressen gleich sind (Alias-Problem). Um die Korrektheit des Programms zu garantieren, müssen Adressen, die nicht definitiv als gleich oder ungleich erkannt werden können, als potenziell gleich angesehen werden. Die Abhängigkeiten zwischen den Setzungen und Benutzungen von Speicherzelleninhalten basieren oft auf der konservativen Annahme, dass der Speicher eine singuläre Ressource ist. Somit existieren Abhängigkeiten zwischen allen STORE- und LOAD-Operationen. In dieser Arbeit werden einfache Algorithmen eingesetzt, die für bestimmte einfache Fälle auf Gleichheit bzw. auf Ungleichheit schliessen können (z.B. für a und a+1). Hierdurch kann ein Grossteil der bei konservativer Annahme bestehenden Abhängigkeiten eliminiert werden. In Abb. 4.5 sind die Datenabhängigkeiten einmal für abstrakte Operationen, die keine LOAD- und STORE-Operationen umfassen, gezeigt und einmal für abstrakte Operationen inklusive LOAD- und STORE-Operationen. Für den letzten Fall sind die Kanten mit einem '=' markiert, für die die Gleichheit der Adressen definitiv bestimmbar sind, und ansonsten mit einem '?'.

Datenabhängigkeitsgraphen umfassen auch Abhängigkeiten, die nicht nur durch die Resultate und Operanden ausgedrückt werden. Die in Kapitel 4.1 eingeführten Begriffe der Setzung und Benutzung umfassen auch solche, die implizit durch eine Maschinenoperation gegeben sind (wie z.B. das Setzen und Benutzen von Bedingungscode). Man spricht hier auch von Seiteneffekten der Maschinenoperationen. Diese werden

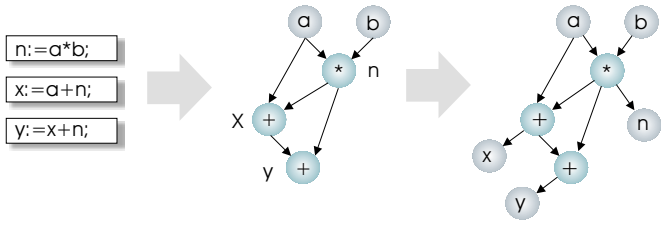


Abbildung 4.6.: Datenflussgraph.

aber durch die allgemeine Formulierung der Datenabhängigkeiten erfasst. Die Positionen sind in diesem Fall immer $none$, da kein expliziter Operand der Maschinenoperationen bzw. FMOs referenziert werden kann. Es wird angenommen, dass die Menge der Kanten des Datenabhängigkeitsgraphen in Klassen unterteilt ist, die widerspiegeln, ob es sich um Abhängigkeiten über Programmvariablen, LOAD-/STORE-Operationen oder implizite Setzungen und Benutzungen von Ressourcen handelt.

4.3.3. Datenflussgraphen

Nach der Standarddefinition sind Datenflussgraphen gerichtete azyklische Graphen und reflektieren den reinen wertebasierten Datenfluss zwischen abstrakten Operationen eines Basisblocks (siehe z.B. [3, 93]). Es werden zusätzliche Knoten für die vorkommenden Programmvariablen und Konstanten eingeführt, die dann die Blätter des Graphen bilden. Die inneren Knoten des Graphen sind mit den Operatoren der abstrakten Operationen markiert. Eine Kante von Knoten n nach Knoten n' kennzeichnet, dass n' einen Wert konsumiert, den n produziert (Variablen- und Konstantenknoten werden ebenfalls als produzierende Knoten betrachtet).

Die Standarddefinition wird im folgenden Sinne erweitert: Knoten, die mit Programmvariablen markiert sind, können auch Wurzeln des Graphen sein (siehe Abb. 4.6). Die Intention ist es, sowohl Eingangs- als auch Ausgangsvariablen explizit darzustellen. Weiterhin werden nicht nur die Operatoren mit den Knoten assoziiert, sondern ganze CoLIR-Operationen (u.a. sind auch komplexe FMOs erlaubt, wodurch ein einzelner Knoten eine komplexe Operation repräsentiert).

lokaler Datenflussgraph

Definition: Der *lokale Datenflussgraph* zu einem Basisblock $b = bb\ with\ [mis : MIs, in : V_{in}, out : V_{out}]$ mit $MIs = [mi_1, ..., mi_n]$ ist ein gerichteter, knoten- und kantenmarkierter azyklischer Graph $DFG = (N, V'_{in}, V'_{out}, E)$ mit $E \subseteq N' \times Pos \times N'$ und $N' = N \cup V'_{in} \cup V'_{out}$, für den gilt:

- Zu jeder CoLIR-Operation cop des Basisblocks gibt es einen Knoten $n \in N$, der mit cop markiert ist. Es existiert eine Kante $(n, p, n') \in E$ mit $n' \in N$, wenn $n \hookrightarrow^{t,p} n'$ im Datenabhängigkeitsgraphen eine Datenflussabhängigkeit bzgl. der Programmvariablen darstellt.

- Zu jedem $v \in V_{in}$ existiert ein $n_v \in V'_{in}$, der mit v markiert ist. Für jede CoLIR-Operation $n \in mi_j$, die v an Position pos benutzt und $[mi_1, \ldots, mi_j]$ ein setzungsfreier Pfad für v ist, gibt es eine Kante $(n_v, p, n) \in E$.
- Zu jedem $v \in V_{out}$ existiert ein $n_v \in V'_{out}$, der mit v markiert ist. Für jede CoLIR-Operation $n \in mi_j$, die v setzt und $[mi_j, \ldots, mi_n]$ ein setzungsfreier Pfad für v ist, gibt es eine Kante $(n, [0], n_v) \in E$.

Es ist weiterhin notwendig, auch den globalen Datenfluss zwischen Programmvariablen zu betrachten.

globaler Datenflussgraph

Definition: Der *globale Datenflussgraph* ist definiert über dem Basisblockgraphen und der Menge der lokalen Datenflussgraphen einer CoLIR-Funktion. Zu je zwei gegebenen Basisblöcken BB_1 und BB_2 mit lokalen Datenflussgraphen $DFG_1 = (N_1, V_{in_1}, V_{out_1}, E_1)$ und $DFG_2 = (N_2, V_{in_2}, V_{out_2}, E_2)$ existiert genau dann eine Kante zwischen $v_1 \in V_{out_1}$ und $v_2 \in V_{in_2}$, wenn $v_1 = v_2$ und es im Basisblockgraph der Funktion einen Pfad von BB_1 nach BB_2 gibt, so dass alle Basisblöcke ungleich BB_1 und BB_2 setzungsfrei bzgl. v_1 sind.

4.4. HLOs im Compilersystem COCOON

Nachfolgend wird ein Überblick über HLOs gegeben, die in LANCE und COCOON integriert sind. Die Qualität der Codegenerierung hängt wesentlich von den eingesetzten LLOs[5] ab. Wichtig ist aber auch der Einsatz von HLOs, ohne die eine hohe Codequalität oftmals nicht möglich ist, da ansonsten die Ausgangsbasis der Codegenerierung noch viel redundanten Code enthalten kann. Es ist weiterhin wichtig, in einem Compilersystem auch für die Codegenerierung HLOs verfügbar zu machen, da diese nach der Codegenerierung oder zwischen den Codegenerierungsphasen wiederum sinnvoll eingesetzt werden können und die Codegüte noch verbessern können. In der Codegenerierung fließen maschinenspezifische Aspekte in die HLOs ein. Die Optimierungsziele der HLOs bleiben i.d.R. aber gleich. Die Constant-Propagation führt beispielsweise die folgende Transformation durch:

```
x:=100;          ⟶      z:=100+y;
z:=x+y;
```

Setzt man diese Technik innerhalb der Codegenerierung ein, muss zusätzlich geprüft werden, ob eine Maschinenoperation überhaupt Konstanten als Argumente erlaubt und ob die zu substituierende Konstante im Wertebereich der möglichen Konstanten der Maschinenoperationen liegt. Es müssen beim Einsatz von HLOs also zusätzliche maschinenspezifische Bedingungen erfüllt sein, um eine bestimmte Transformation

[5] LLO=Low-Level-Optimierung (siehe auch Kapitel 2.2).

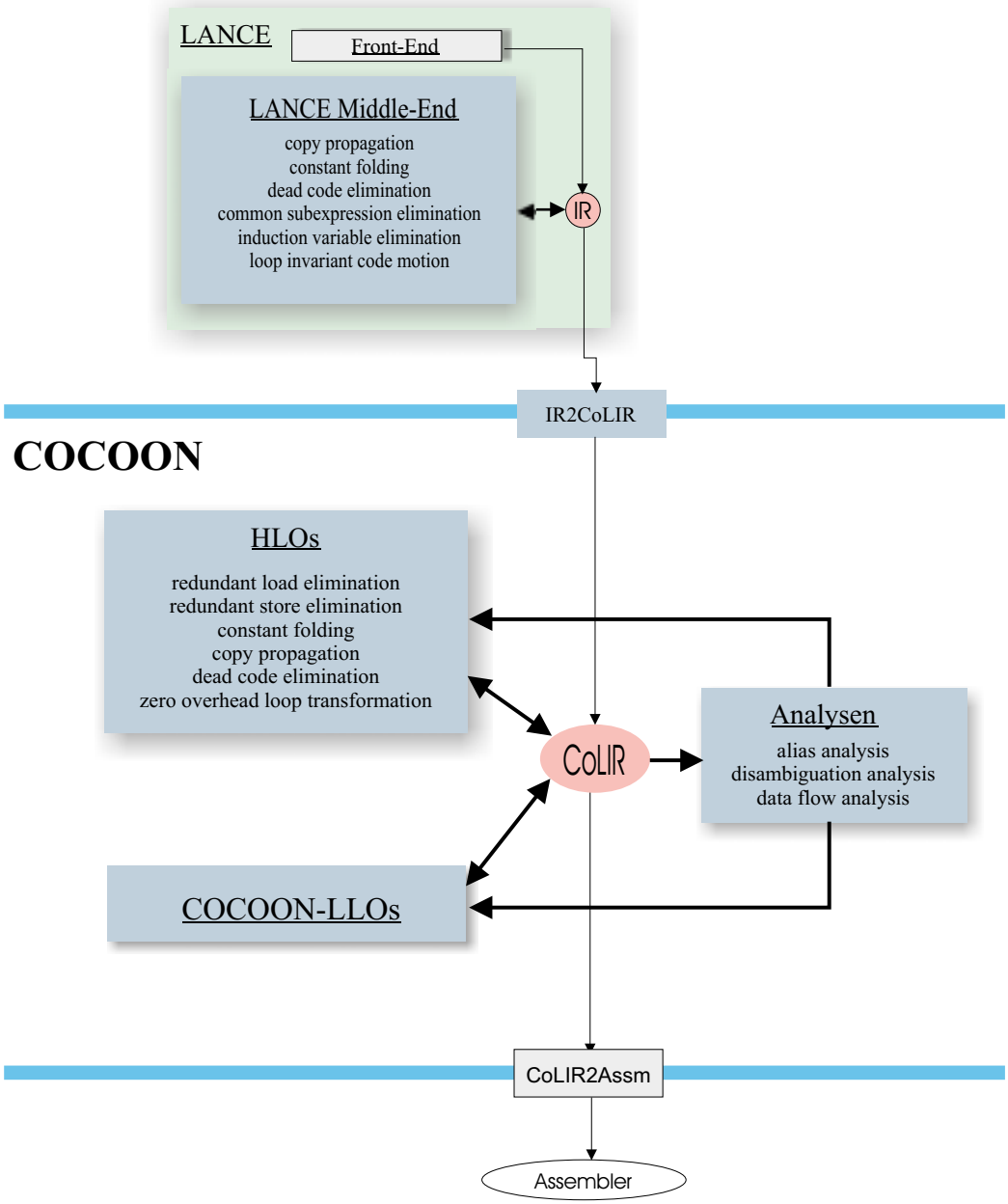


Abbildung 4.7.: HLOs in LANCE und COCOON.

durchführen zu können. Ziel bei der Entwicklung von HLOs sollte sein, diese für möglichst viele Klassen von Prozessoren einsetzen zu können und sie auch als reine HLO auf den abstrakten Operationen zu verwenden. CoLIR ermöglicht durch seine einheitliche Repräsentation von abstrakten Operationen und Maschinenoperationen eine einheitliche Entwicklung von HLOs, die auf der Ebene der abstrakten Operationen, sowie auf den FMOs einsetzbar sind.

Innerhalb dieser Arbeit liegt der Schwerpunkt auf der Entwicklung von Codegenerierungstechniken (also LLOs). Daher wurden hier die HLOs von LANCE ausgenutzt und innerhalb von COCOON um einige nicht in LANCE vorkommende HLOs ergänzt. Diese sind allerdings für IR-reine CoLIR-Programme ausgelegt und werden daher ausschließlich vor Ausführung der eigentlichen Codegenerierung aufgerufen.

In Abb. 4.7 sind die in LANCE und COCOON integrierten HLOs gezeigt. Auf eine Beschreibung der HLOs wird hier verzichtet. Der interessierte Leser findet eine sehr gute Beschreibung in [93]. In der Abbildung ist weiterhin noch eine Teilmenge der eingesetzten Analysen gezeigt. Neben den Datenflussanalysen ist die Erkennung gleicher und ungleicher Speicherreferenzen wichtig (*Alias*- und *Disambiguation*-Analysen), um unnötige Abhängigkeiten zwischen LOAD- und STORE-Operationen zu vermeiden und redundante LOAD- und STORE-Operationen eliminieren zu können. Es hat sich gezeigt, dass dies notwendig ist, um die Qualität von handgenerierten Codes zu erreichen. Eine wichtige HLO war die Transformation von Schleifen in Zero-Overhead-Loops, da hierdurch für die betrachteten Schleifen sowohl die Instruktionen zum Modifizieren der Schleifenzähler als auch die Instruktionen zum Abfragen der Schleifenbedingung entfallen (diese Aufgaben werden implizit durch die Hardware der Prozessoren parallel zur Ausführung des Schleifencodes durchgeführt). Diese Optimierung ist ebenfalls von zentraler Bedeutung, da sie auch Voraussetzung ist, um an die Codegüte von handgeneriertem Assemblercode heranzukommen. Diese HLOs sind üblicherweise schon in Standardcompilern für DSPs integriert, so dass hier keine neuen Techniken eingesetzt werden, die explizit erläutert werden müssten.

4.5. CSP-Modelle über CoLIR

Die Domainvariablen eines CoLIR-Programms zusammen mit den Constraints im Constraintstore repräsentieren ein CSP. Die Menge der alternativen Maschinenprogramme, die durch ein CoLIR-Programm repräsentiert wird, werden im Folgenden auch *alternative Überdeckungen* eines CoLIR-Programms genannt. Jede einzelne Alternative ist eine Lösung bzgl. der mit dem CoLIR-Programm assoziierten Constraints.

alternative Überdeckungen

Wird ein CoLIR-Programm CP ohne assoziierte Constraints betrachtet, so definiert eine bestimmte Codegenerierungsphase, die ihre Aufgabe durch eine Menge von Constraints über den Domainvariablen von CP spezifiziert, ein bestimmtes CSP. Das Modell zur Generierung der Constraints einer Codegenerierungsphase wird im Folgenden als *CSP-Modell einer Codegenerierungsphase* oder einfach nur als *CSP-Modell* bezeichnet. Ein CSP ist dann die Variablen- und Constraintmenge einer konkreten Probleminstanz.

CSP-Modell einer Codegenerierungsphase

Ein CSP-Modell kann dabei die gesamten Constraints der Phase umfassen oder selbst

wieder aus Teil-CSP-Modellen zusammengesetzt sein. Das CSP-Modell der Instruktionsanordnung kann dann z.B. ein CSP-Modell beinhalten, das die zeitlichen Abhängigkeiten zwischen den Maschinenoperationen spezifiziert, und ein weiteres, das die Restriktionen bzgl. der Parallelausführung von FMOs festlegt. CSP-Modelle können genutzt werden, um bestimmte Anforderungen einer Codegenerierungsphase an die Eingabe-Programme zu prüfen. Durch das Prüfen auf Existenz einer Lösung kann getestet werden, ob die Anforderungen gegeben sind. ECLiPSe bietet gute Konzepte, um die hierdurch durchgeführten Bindungen wieder rückgängig zu machen und somit nicht an Freiheitsgraden und Flexibilität einzubüßen. So kann z.B. in einer Phase getestet werden, ob die FMOs in einer Maschineninstruktion den Restriktionen bzgl. der Instruktionstypen genügen. Hierzu kann dann das entsprechende CSP-Modell der Instruktionsanordnung gegebenenfalls wiederverwertet werden. Durch das Prüfen von Anforderungen ist es möglich zu testen, ob gewisse Codegenerierungsphasen schon durchgeführt wurden und ob eine sinnvolle Eingabe für eine Optimierung vorliegt.

Eine gute Modularisierung einer Codegenerierungsphase in Teil-CSP-Modelle kann genutzt werden, um diese bei der Spezifikation neuer Codegenerierungsphasen wiederzuverwenden. Gerade im Hinblick auf die Phasenkopplung können CSP-Modelle einer Phase mit den CSP-Modellen anderer Phasen einfach kombiniert werden, in dem die Constraints der CSP-Modelle simultan generiert werden. Teil-CSP-Modelle können dazu genutzt werden, schnell unterschiedliche Kopplungsgrade von Phasen zu modellieren und auszutesten. Auf dieser Basis erhält man eine Methode zur modularen Spezifikation der einzelnen Codegenerierungsphasen und die Möglichkeit, Phasen schnell in einer handhabbaren Form zu integrieren.

Eine Eigenschaft von CoLIR ist, dass CSP-Modelle über mehrere Codegenerierungsphasen hinweg propagiert werden können, wodurch sich eine weitere Möglichkeit zur Kopplung von Phasen ergibt. Entscheidungen einer Phase können hierdurch in die nachfolgenden Phasen verlagert werden. Von dieser Eigenschaft sollte aber kein uneingeschränkter Gebrauch gemacht werden, da dies in nachfolgenden Phasen zu unabsehbaren Effekten führen kann. So sollten den nachfolgenden Phasen die potenziellen Effekte der propagierten Constraints zumindest bekannt sein. Als Beispiel kann die in Kapitel 6 vorgestellte graphbasierte Instruktionsauswahl betrachtet werden. Hier werden Kosten für notwendige Datentransferpfade mit den Domainvariablen der Registerfilezuordnungen von datenflussabhängigen Setzungen und Benutzungen in Beziehung gesetzt. Nach der Instruktionsauswahl sind bestimmte Datentransferkosten auf Null gesetzt. Das damit verbundene Constraint bedingt jetzt, dass die entsprechenden Setzungen und Benutzungen immer derselben ST-Komponente zugeordnet werden müssen. In einer nachfolgenden Phase kann es aber notwendig werden, zwischen Setzungen und Benutzungen weitere Datentransfers einzufügen und der Setzung und Benutzung unterschiedliche ST-Komponenten zuzuordnen. Daher kann es sinnvoll sein, Constraints, die in einer Phase generiert wurden, wieder zu entkoppeln.

Um die Modularität zu gewährleisten und Phasen möglichst unabhängig voneinander entwickeln zu können, werden folgende Richtlinien zur Propagierung von Constraints eingeführt:

1. Voneinander unabhängig entwickelte Phasen propagieren keine Constraints über

den Constraintstore. Anforderungen an das Eingabe-CoLIR-Programm werden durch entsprechende CSP-Modelle in der jeweiligen Codegenerierungsphase gehandhabt.

2. Der Constraintstore wird nur zur Propagierung genutzt, wenn die nachfolgenden Phasen die Menge der propagierten Constraints kennt. So werden unerwartete Effekte vermieden und zu restriktive Forderungen können durch die Kenntnis, welche Constraints diese erzeugen, sogar eliminiert werden. Auf diese Weise kommunizierende Teilphasen werden selbst wieder in einer Phase gekapselt.

Diese Richtlinien wahren die Transparenz eines Compilers. So ist jeweils genau klar, wer welche Anforderungen an seine Eingabe stellt, und unter welchen Anforderungen es in einer Phase evtl. zu keiner Lösung kommt.

4.6. Mengen und Notationen

Im Verlauf der weiteren Arbeit werden die im Folgenden eingeführten Begriffe, Notationen und Mengen verwendet.

Notation und Begriffe für die Komponenten von FMOs, Eingangs- und Ausgangsvariablen

- $fun, fun_1, ..., fun_n$ sind Bezeichner für Funktionen.
- $b, b_1, ..., b_n$ sind Bezeichner für Basisblöcke.
- $f, f_1, ..., f_n$ sind Bezeichner für FMOs eines Basisblocks b.
- $AOP(b)$ ist die Menge aller abstrakter Operationen innerhalb eines Basisblocks b.
- $FMO(b)$ ist die Menge aller FMOs innerhalb eines Basisblocks b.
- $VAR_{in}(b)$ ist die Menge V_{in} eines Basisblocks b.
- $VAR_{out}(b)$ ist die Menge V_{out} eines Basisblocks b.
- $FV(b) = FMO(b) \cup VAR_{in}(b) \cup VAR_{out}(b)$.
- $CSE(b)$ ist die Menge der CSEs des Basisblocks b:

$$CSE(b) = \{fv | fv \in FMO(b) \cup VAR_{in}(b) \; und \; |edges(fv, b)| > 1\}$$

- $v, v_1, ..., v_n$ sind die Bezeichner für Elemente aus $VAR_{in}(b)$ und $VAR_{out}(b)$ eines Basisblocks b.
- $fv, fv_1, ..., fv_n$ sind Bezeichner für Elemente aus $FV(b)$ eines Basisblocks b.

- Zur Referenzierung von Setzungen und Benutzungen der Registerfilezuordnungen wird im Folgenden anstelle von fv_{pos}^{rfs} einfach kurz fv_{pos} geschrieben. Zugriffe auf Programmvariablenzuordnungen oder Sub-FMOs werden weiterhin mit fv_{pos}^{bs} bzw. fv_{pos}^{sos} notiert.

- Setzungen und Benutzungen von Registerfilezuordnungen werden nur noch als Setzungen und Benutzungen bezeichnet. Ansonsten wird explizit gesagt, ob es sich um eine Bezeichner- oder Registerzuordnung handelt.

- Die Registerfilezuordnungen von Eingangs- und Ausgangsvariablen werden als Eingangs- bzw. Ausgangssetzungen bezeichnet.

- Die Domainvariable FE einer FMO f wird auch mit FE_f bezeichnet.

- Der Domainvariable IT einer FMO f wird auch mit FE_f bezeichnet.

Mengen über Domainvariablen von Basisblöcken

- $STK_{def}(b)$ ist die Menge der Setzungen (bzgl. der Registerfilezuordnungen) eines Basisblocks b:

  $$STK_{def}(b) = \{fv_{[0]} \mid fv \in FV(b)\}$$

  Elemente dieser Menge werden einfach nur als Setzung bezeichnet.

- $STK_{use}(b)$ ist die Menge der Benutzungen des Basisblocks b:

  $$STK_{use}(b) = \{RF_i | fmo\ with\ [rfs : \_ := (..., RF_i, ..)] \in FMO(b)\}$$

  Elemente dieser Menge werden einfach nur als Benutzung bezeichnet.

- $STK(b) = STK_{def}(b) \cup STK_{use}(b)$. Elemente dieser Menge werden auch als *STK-Variablen* bezeichnet. *STK-Variablen*

- $FE(b)$ ist die Menge der Domainvariablen, die die alternativen Zuordnungen von FMOs zu Funktionseinheiten spezifizieren:

  $$FE(b) = \{FE | fmo\ with\ [fe : FE] \in FMO(b)\}$$

  Elemente dieser Menge werden auch als *FE-Variablen* bezeichnet. *FE-Variablen*

- $IT(b)$ ist die Menge der Domainvariablen, die die alternativen Zuordnungen von FMOs zu Instruktionstypen spezifizieren:

  $$IT(b) = \{IT | fmo\ with\ [it : IT] \in FMO(b)\}$$

  Elemente dieser Menge werden auch als *IT-Variablen* bezeichnet. *IT-Variablen*

Notationen für Graphen

- $n \in G$ für einen Graphen G besagt, dass n ein Knoten des Graphen G ist.
- $n \rightarrow^{attr} n' \in G$ für einen Graphen G besagt, dass $n \rightarrow^{attr} n'$ eine Kante des Graphen G mit Kantenmarkierung $attr$ ist.
- $DFG(b)$ ist der lokale Datenflussgraph zu Basisblock b und $gDFG(b_1, ..., b_n)$ ist der globale Datenflussgraph zu den Basisblöcken einer Funktion mit den Basisblöcken $b_1, ..., b_n$.
- $DDG(b)$ ist der lokale Datenabhängigkeitsgraph zu Basisblock b.

Ebenen von Operationen

Im Folgenden werden die Ebenen von abstrakten Operationen, Maschinenoperationen und deren FMOs begrifflich getrennt:

- Maschinenoperationen sind die auf einem Prozessor verfügbaren Operationen, wobei alle Ressourcen einer bestimmten Maschinenoperation determiniert sind.

$class$- und op-Operationen

- Unter *CONST-Operationen* werden alle Maschinenoperationen subsumiert, die das Laden von Konstanten umsetzen. Analoges gilt für alle weiteren CoLIR-Operationen einer Klasse $class$ aus der Menge $\mathcal{OC}$. Es handelt sich hier also um eine Klassifizierung der Maschinenoperationen gemäß der in CoLIR eingeführten Klassen aus $\mathcal{OC}$. Die zu einer Klasse $class \in \mathcal{OC}$ (z.B. $const$) gehörenden FMOs werden auch FMOs der $class$-Operationen genannt, wobei $class$ in Großbuchstaben notiert wird. (wie z.B. CONST-Operationen). Weiterhin können auch feinere Klassifizierungen wie z.B. die FMOs zu einem bestimmten Operator oder die FMOs einer bestimmten Funktionseinheit (z.B. FMOs der AGU-Operationen) angegeben werden. In diesem Fall wird bei einem Operator op von den FMOs der op-Operationen oder einfach nur von den op-Operationen gesprochen.

Spezifikation und Definition von FMOs

- Die Spezifikation der CLP-Regeln, die die Domains und Constraints einer bestimmten Klasse $class$ von FMOs festlegt, wird auch die Definition der $class$-Operationen genannt. Bei der informalen Spezifikation von Maschinenoperationen wird auch nur von der Spezifikation der $class$-Operationen gesprochen.

Darstellung von FMOs in Graphiken

FMOs werden in Abbildungen häufig durch die folgenden Ausprägungen von faktorisierten Registertransferoperationen dargestellt:

- In $D_1 := op(A_1, ..., A_n)$ wird der Operator op der FMO dargestellt, und die Setzung und Benutzungen werden durch die entsprechenden Domainvariablen D_1 und $A_1, ..., A_n$ der Registerfilezuordnung wiedergegeben.

- In $rf_{0,1}|...|rf_{0,n_0} := op(rf_{1,1}|...|rf_{1,n_1}, ..., rf_{m,1}|...|rf_{m,n_m})$ werden die Setzungs- und Benutzungsalternativen explizit repräsentiert.

Diese beiden Darstellungen können auch kombiniert verwendet werden.

4.7. Zusammenfassung

Es wurde die constraintbasierte Zwischenrepräsentation CoLIR spezifiziert, die die Darstellung abstrakter Operationen einer IR und die Darstellung von Maschinenprogrammvarianten auf der Grundlage constraintbasierter faktorisierter Maschinenoperationen umfasst. Ziel bei der Entwicklung von CoLIR war es, eine gemeinsame und generische Schnittstelle für IRs, Maschinenprogramme und deren Zwischenstufen zu geben. Um eine gemeinsame Basis für phasengekoppelte Techniken zu schaffen, umfasst CoLIR alle notwendigen Informationen von Codegenerierungsphasen, um diese in integrierten Techniken simultan zur Verfügung stellen zu können.

Auf der Grundlage faktorisierter Maschinenoperationen wurde ein intuitives und handhabbares Konzept zur Repäsentation alternativer Maschinenprogramme eingeführt. Hierdurch werden folgende Dinge ermöglicht:

- Das Binden von Ressourcen kann in einer bestimmten Phase verzögert werden, um möglichst viele Freiheitsgrade für nachfolgende Phasen beizubehalten.
- Der constraintbasierte Ansatz erlaubt es, Alternativen zu repräsentieren, welche wechselseitige Abhängigkeiten zwischen den Ressourcen besitzen.
- Der Lösungsmechanismus von ECLiPSe erlaubt es, diese wechselseitigen Abhängigkeiten über aufeinanderfolgende Codegenerierungsphasen zu propagieren, ohne dass die nachfolgenden Phasen eine besondere Handhabung der propagierten Constraints spezifizieren müssen.

Weiterhin wird durch CoLIR eine Basis zur schnellen Adaption von Compilern an neue Prozessoren geschaffen:

- CoLIR besitzt ein generisches Konzept zur Darstellung maschinenspezifischer Informationen, so dass sie unmittelbar für die Codegenerierung neuer Prozessoren verwendet werden kann und somit die jeweilige Neuimplementierung einer IR bzw. LIR vermeidet.
- Das generische Ressourcenmodell von CoLIR soll die Entwicklung retargierbarer Codegenerierungstechniken unterstützen, so dass Techniken für möglichst große Klassen ähnlicher Prozessoren wiederverwendet werden können. Dies wird zusätzlich durch die generische Klassifizierung und Struktur von CoLIR-Operationen unterstützt, so dass Optimierungen und Analysen auch unabhängig von den Befehlssätzen und spezifischen Operatoren von Prozessoren allgemein formulierbar sind.

- Durch eine einheitliche Schnittstelle für alle Codegenerierungsphasen können neue Codegenerierungstechniken einfach in eine bestehende Menge von Techniken integriert werden.
- Unter der Verwendung bestehender generischer Techniken kann somit ein Compiler in modularer Weise zusammengefügt und um neue Techniken erweitert werden.
- Die Verschmelzung von abstrakten Operationen und Maschinenoperationen in einer Darstellung soll erlauben, maschinenunabhängige HLOs auch in der Codegenerierung wiederzuverwenden und darüber hinaus die Möglichkeit bieten, HLOs an maschinenspezifische Eigenschaften anzupassen.

Ein weiterer wichtiger Aspekt eines gemeinsamen Modells für die Codegenerierung ist, dass Compilerentwickler ein und dieselbe Sichtweise auf den Codegenerierungsprozess haben, so dass eine Basis zur einheitlichen Kommunikation geschaffen wurde. Die Wirkungsweise von Codegenerierungsphasen läßt sich mittels CoLIR präzise beschreiben, da genau spezifiziert werden kann, welche Phase welche Ressourcen bindet, und welche Klassen von Operationen von einer Phase hinzugefügt bzw. eliminiert werden. Die mittlerweile umfangreichen und äußerst unterschiedlichen Ausprägungen von Techniken wie z.B. zur Instruktionsauswahl oder zur Registerallokation lassen sich so wesentlich genauer erfassen und differenzieren, als einfach nur von Instruktionsauswahl bzw. von Registerallokation zu reden.

CoLIR ist noch kein umfassendes Modell, das die Darstellung von Eigenschaften aller Prozessorklassen unterstützt. Das Modell ist aber bereits für eine grosse Klasse von Prozessoren einsetzbar, und die Grundkonzepte erlauben eine einfache Erweiterung um weitere spezifische Eigenschaften von Prozessoren. Dabei sollte aber jeweils darauf geachtet werden, ob eine Eigenschaft nicht schon mit den gezeigten Konzepten und den im nachfolgenden Kapitel noch vorgestellten Konzepten zur Modellierung von FMOs umgesetzt werden kann.

5. Alternative Überdeckungen - Das CSP-Modell $COVER$

Eine der zentralen Aufgaben der Codegenerierung ist die Instruktionsauswahl. Die wesentliche Aufgabe hiervon ist, die abstrakten Operationen der IR auf Operationen des Zielprozessors abzubilden, mit dem Ziel, eine möglichst kostengünstige Realisierung des Quellprogramms für den Prozessor zu erhalten. Diese Aufgabe umfasst die folgenden beiden Teilaufgaben:

- *Mustererkennung*: Hier findet die Abbildung der abstrakten Operationen auf die Operationen der Zielarchitektur statt. Dabei ist es notwendig, dass möglichst alle alternativen Möglichkeiten, Operatoren auf Maschinenoperationen abzubilden, erfasst werden. Wesentlich dabei ist die Erkennung von komplexen Maschinenoperationsmustern. Da es i.d.R. alternative Muster zur Umsetzung bestimmter abstrakter Operationen gibt und sich die Muster komplexer Maschinenoperationen überschneiden können, ergeben sich alternative Zuordnungen eines Programms zu Überdeckungen mit Maschinenoperationsmustern.
- *Selektion einer Überdeckung*: Hier besteht die Aufgabe darin, aus der Menge der Überdeckungen eines Programms mit Maschinenoperationsmustern die kostengünstigste auszuwählen.

Für die in dieser Arbeit betrachtete Klasse von Prozessoren ist die Erkennung komplexer und graphbasierter Muster wichtig, um guten Code generieren zu können. Weiterhin spielt die Integration der Instruktionsauswahl in andere Phasen der Codegenerierung eine große Rolle. In diesem Kapitel wird ein generisches CSP-Modell zur Instruktionsauswahl spezifiziert, das mit $COVER$ bezeichnet wird und die folgenden Eigenschaften hat:

- Die hier vorgestellten Modellierungskonzepte ermöglichen es, zu einem IR-reinen CoLIR-Programm die komplexen und graphbasierten Maschinenoperations-Muster mit dem gleichen Mechanismus zu erkennen und darzustellen. Die sich daraus ergebenden alternativen Überdeckungen können selbst wieder durch ein CoLIR-Programm dargestellt werden und über Codegenerierungsphasen hinweg beibehalten werden. Dabei können simultan sich überlappende alternative komplexe Muster und deren Umsetzungen durch Einzeloperationen repräsentiert werden. Jede Lösung von $COVER$ enthält nur legale Maschinenoperationen.

- Für datenflussabhängige Setzungen und Benutzungen ist garantiert, dass es für jede Setzungsalternative s mindestens eine Benutzungsalternative b gibt, für die ein Datentransferpfad existiert, so dass der Inhalt von s nach b transportiert werden kann.
- Neben der reinen Existenz von Datentransferpfaden können alternative Datentransferpfade explizit, durch eine Sequenz von FMOs, dargestellt werden. Durch eine einzige Sequenz von n FMOs werden alle alternativen Datentransferpfade zwischen einer Setzung und Benutzung repräsentiert, die maximal die Länge n haben.

Das Modell umfasst kein Kostenmodell zur Bewertung einzelner Alternativen. Es dient als generische Grundlage zur Adaption alternativer Kostenmodelle und Selektionsstrategien, die adaptiert werden können, ohne die Notwendigkeit, das CSP-Modell modifizieren zu müssen. Die Intention bei der Entwicklung von $COVER$ war, ein CSP-Modell zur Verfügung zu stellen, das die schnelle Umsetzung von Techniken zur Instruktionsauswahl und die Integration der Instruktionsauswahl in andere Codegenerierungsphasen ermöglicht. Dies wird noch zusätzlich durch den Aspekt begünstigt, dass die hier vorgestellte Methode zur Modellierung von FMOs gleichzeitig eine elegante und kompakte Methode zur Spezifikation des Befehlssatzes bietet und die Techniken zur Generierung der CSPs zu $COVER$ vollkommen unabhängig von der Spezifikation der FMOs eines bestimmten Prozessors sind.

Der Inhalt des Kapitels gliedert sich wie folgt:

- In Kapitel 5.1 werden die grundlegenden Konzepte zur Modellierung von FMOs erläutert.
- In Kapitel 5.2 wird gezeigt, wie mittels Constraints die Existenz von Datentransferpfaden sichergestellt werden kann und wie alternative Datentransferpfade durch eine einzige Sequenz von FMOs repräsentiert werden können.
- Kapitel 5.3 umfasst die Modellierungskonzepte von FMOs zur Darstellung komplexer und graphbasierter Maschinenoperationen und erläutert, wie dieses Konzept auch gleichzeitig der Erkennung dieser Maschinenoperationen dient.
- In Kapitel 5.5 wird dargestellt, wie IR-reine CoLIR-Programme auf die alternativen Überdeckungen von $COVER$ abgebildet werden. Es werden Erweiterungen von $COVER$ eingeführt, die stärkere Restriktionen umfassen.

5.1. Modellierung von FMOs

Es wird im Folgenden gezeigt, welche Konzepte ECLiPSe bietet, um Constraints über den Domainvariablen einer FMO zu spezifizieren, deren Lösungen legale Maschinenoperationen liefern. Die Modellierung wird am Beispiel der ADSP2100-Familie erläutert. Das benutzerdefinierte Constraint

$$fmo(Class_f, Op_f, RF_{f,0} := [RF_{f,1}, ..., RF_{f,n}], FE_f, IT_f)$$

setzt die Domainvariablen einer FMO f derart in Beziehung, dass eine Variablenbelegung $\theta_{vars(f)}$ eine Lösung einer FMO f ist, wenn $\theta_{vars(f)}(f)$ eine legale Maschinenoperation darstellt. Die Registerzuordnungen der FMOs lassen sich allgemein für die in der Arbeit betrachteten Prozessoren für jede FMO f durch ein Constraint $D(f_{pos}^{regs}) \cap regs(rf) \neq \emptyset \Rightarrow rf \in D(f_{pos}^{rfs})$ ausdrücken. Da diese i.d.R. erst relativ spät verwendet werden, reicht es, die Constraints auch erst zu den benötigten Zeitpunkten zu erzeugen. Für jede Klasse $class \in \mathcal{OC}$ umfasst $fmo/5$ eine Regel, in der ein klassenspezifisches Constraint $class\_fmo(Op, RF, [RF_1, ..., RF_n], FE, IT)$ aufgerufen wird:

```
fmo(alo,Op,Def:=[Arg1,Arg2],FE,IT) :-
   alo_fmo(Op,Def,[Arg1,Arg2],FE,IT).
fmo(const,Op,Def:=[],FE,IT) :-
   const_fmo(Op,Def,[],FE,IT).
fmo(copy,Op,Def:=[Arg1],FE,IT) :-
   copy_fmo(Op,Def,[Arg1],FE,IT).
fmo(load,Op,Def:=[Arg1,Arg2],FE,IT) :-
   load_fmo(Op,Def,[Arg1,Arg2],FE,IT).
fmo(store,Op,Def:=[Arg1,Arg2],FE,IT) :-
   store_fmo(Op,Def,[Arg1,Arg2],FE,IT).
...
fmo(return,ret,_:=[],bus,IT) :-
   return_fmo(ret,[],FE,IT).
```

Hierdurch ergibt sich nach außen eine eindeutige Schnittstelle zur Generierung der Constraints und nach innen die Möglichkeit einer modularen Definition der einzelnen FMO-Klassen. Die grundlegenden Modellierungsmethoden der Constraints werden im Folgenden am Beispiel von arithmetischen Operationen und der Datentransfer-Operationen erläutert[1].

5.1.1. Modellierung arithmetisch/logischer FMOs

In Abb. 5.1 ist die Spezifikation der Maschinenoperationen der Addition und der Multiplikation der ADSP2100-Familie gezeigt. In Abb. 5.2 sind die entsprechenden Instruktionstypen nocheinmal aufgezeigt. Die Addition ist dort durch drei Regeln spezifiziert worden, die unmittelbar durch die folgenden drei Regeln des Prädikats $alo\_fmo/5$ realisiert werden können:

```
alo_fmo(+,Def,[Arg1,Arg2],alu,IT) :-    Regel 1
    Def::[ar,af],
    Arg1::[ar,ax,mr,sr],
    Arg2::[ay,af],
    IT::[1,4,5,8,9],
    IT#=1#=>Def#=ar#.
alo_fmo(+,Def,[Arg1,yconst],alu,9) :-   Regel 2
    Def::[ar,af],
    Arg1::[ar,ax,mr,sr].
```

[1]Die darin eingeführten Konzepte werden auch zur Modellierung der Constraints für die weiteren FMO-Klassen von CoLIR-Operationen genutzt.

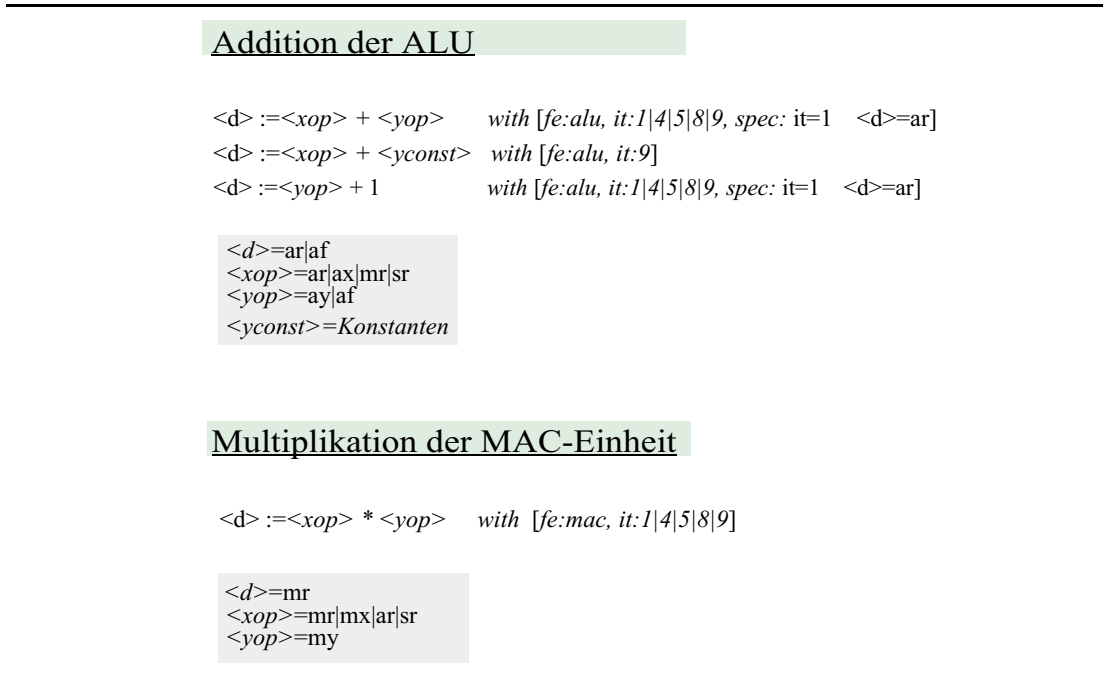


Abbildung 5.1.: FMOs der Addition und der Multiplikation der ADSP2100-Familie.

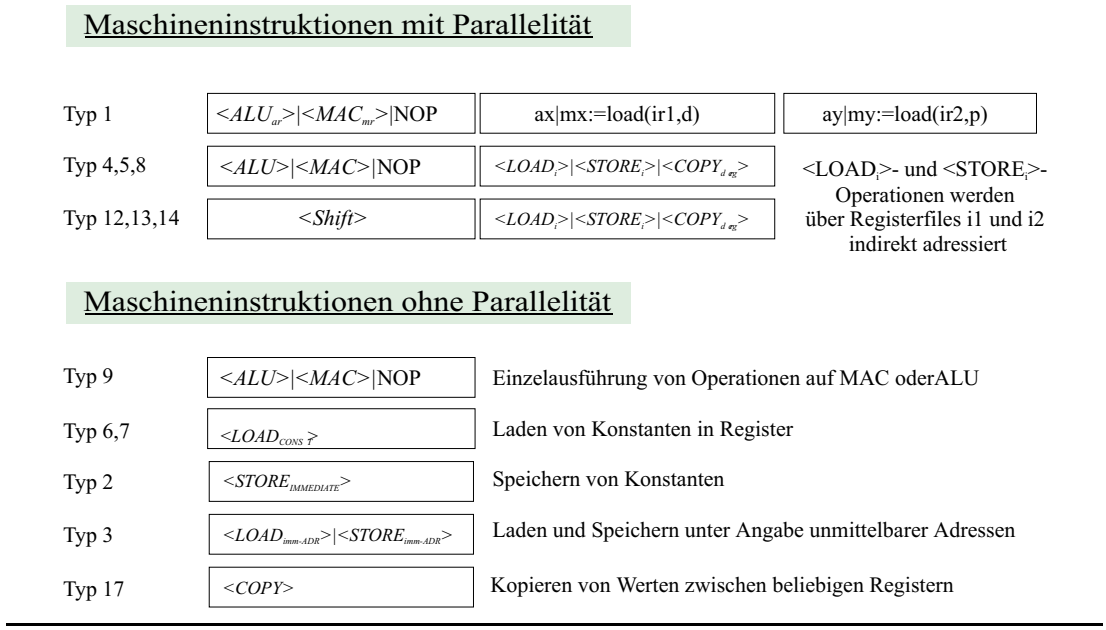


Abbildung 5.2.: Teilmenge der Instruktionstypen der ADSP2100-Familie.

```
alo_fmo(+,Def,[Arg1,'1'],alu,IT) :-      Regel 3
    Def::[ar,af],
    Arg1::[ay,af],
    IT::[1,4,5,8,9],
    IT#=1#=>Def#=ar.
```

Man erkennt die relativ einfache und intuitive Umsetzung der FMOs in ECLiPSe. Die Multiplikation wird entsprechend einfach durch die folgende Regel realisiert:

```
alo_fmo(*,Def,[Arg1,Arg2],mac,IT) :-     Regel 4
    Def#=mr,
    Arg1::[ar,mr,sr,mx],
    Arg2#=my,
    IT::[1,4,5,8,9].
```

Konstanten, die als Argumente von Maschinenoperationen vorkommen können, werden in den FMOs durch flüchtige Komponenten modelliert. So wird die Verwendung der Konstante 1 durch die ST-Komponente $'1'$ repräsentiert. $yconst$ repräsentiert eine ganze Menge von Konstanten, die im zweiten Argument als Operand vorkommen dürfen.

Auf die Anfrage

```
:- fmo(alo,+,X:=[Y,Z],FE,IT]).
```

antwortet das ECLiPSe-System mit:

```
X=X{[ar,af]}
Y=Y{[ar,ax,mr,sr]}
Z=Z{[ay,af]}
FE=alu
IT=IT{[1,4,5,8,9]}
There are delayed Goals
More? (;)
```

Die Evaluierung des Goals führt zur Anwendung der ersten Regel von $alo\_fmo/5$, und die Domains der Variablen werden gemäß dieser Regel gebunden. Die Variablen des Goals werden zusammen mit ihren Domains als Antwort ausgegeben. Die Ausgabe `X=X{[ar,af]}` bedeutet, dass $D(X) = \{ar, af\}$ ist, und die Ausgabe von `'There are delayed Goals'` gibt Hinweis darauf, dass noch reaktivierte Constraints im Constraintstore existieren. In diesem Fall ist es das durch Anwendung der ersten Regel von $alo\_fmo/4$ generierte Constraint $IT = 1 \Rightarrow X = ar$. Bei Eingabe von `';'` führt das ECLiPSe-System intern Backtracking durch und wendet die zweite Regel von $alo\_fmo/5$ an. Dies führt zur Bindung der Variablen an die folgenden Domains:

```
X=X{[ar,af]}
Y=Y{[ar,mr,sr,ax]}
Z=yconst
FE=alu
IT=9
There are delayed Goals
```

Die folgende Anfrage

```
:- fmo(alo,+,X:=[Y,Z],FE,IT), IT#=1.
```

führt zu der Antwort

```
X=ar                          % vorher X ∈ {ar, af}
Y=Y{[ar,ax,mr,sr]}
Z=Z{[ay,af]}
FE = alu
IT = 1
More? (;)
```

Die Evaluierung des Goals führt wiederum zunächst zur Anwendung der ersten Regel von $alo\_fmo/5$ und die Domains der Variablen werden wieder gemäß dieser Regel gebunden und das Constraint $IT = 1 \Rightarrow X = ar$ im Constraintstore abgelegt. Das primitive Constraint $IT = 1$ führt zur Modifikation der Variablenbelegung von IT, und dies zur Re-Aktivierung des Constraints $IT = 1 \Rightarrow X = ar$. Die Prämisse ist erfüllt und impliziert somit auch die Erfüllung der Konklusionen $X = ar$, so dass X an ar gebunden wird.

Betrachtet man die FMOs unterschiedlicher Operatoren, so weisen diese oft grosse Ähnlichkeiten in den möglichen Ressourcenbelegungen auf. Die Subtraktion umfasst z.B. die gleichen Registerfilezuordnungen wie die Addition. In der Spezifikation des Prädikates kann dieser Aspekt durch Faktorisierung der Operatorsymbole ausgenutzt werden. Hierdurch können die Addition und die Subtraktion in gemeinsamen Regeln formuliert werden:

```
alo_fmo(Op,Def,[Arg1,Arg2],alu,IT):-   Regel a
    Op::[+,-],
    Def::[ar,af],
    Arg1::[ar,ax,mr,sr],
    Arg2::[ay,af,yconst],
    IT::[1,4,5,8,9],
    IT#=1#=>Def#=ar#,
    Arg2#=yconst#=>IT#=9.
alo_fmo(Op,Def,[Arg1,'1'],alu,IT) :-   Regel b
    Op::[+,-],
    Def::[ar,af],
    Arg1::[ay,af],
    IT::[1,4,5,8,9],
    IT#=1#=>Def#=ar.
```

Man beachte, dass hier zunächst die ersten beiden Regeln der Addition zu einer Regel zusammengefasst wurden (Regel a setzt jetzt die Regeln 1 und 2 um). Durch das Constraint $Op \in \{+, -\}$ greift jetzt diese Regel sowohl für $Op = +$ als auch für $Op = -$. Die Maschinenoperationen der Subtraktion umfassen weiterhin die Möglichkeit, Operanden in Registerfiles des ersten Eingangsports der ALU von den Inhalten der Registerfiles des zweiten Eingangsports der ALU zu subtrahieren, was durch die folgenden zusätzlichen beiden Regeln ausgedrückt wird:

```
alo_fmo(-,Def,[Arg1,Arg2],alu,IT) :-   Regel c
    Def::[ar,af],
    Arg1::[ay,af,yconst],
    Arg2::[ar,ax,mr,sr],
    IT::[1,4,5,8,9],
    IT#=1#=>Def#=ar#,
    Arg1#=yconst#=>IT#=9.
alo_fmo(-,Def,[Arg1,'1'],alu,IT) :-   Regel d
    Def::[ar,af],
    Arg1::[ay,af],
    IT::[1,4,5,8,9],
    IT#=1#=>Def#=ar.
```

Es werden später noch Methoden vorgestellt, die eine elegantere Modellierung dieser zusätzlichen Regeln erlauben.

Vermeidung von Nichtdeterminismus und Backtracking

Die Addition wurde im letzten Beispiel durch zwei Regeln implementiert. Der wesentliche Grund, alternative Regeln zu verwenden, ist, dass die wechselseitigen Beziehungen zwischen den Variablen so in klarer und intuitiver Form spezifiziert werden konnten. Ein Nachteil dieser Methode ist, dass bei Anwendung des Prädikats $alo\_fmo/5$ immer nur eine Teilmenge der möglichen alternativen Maschinenoperationen simultan zur Auswahl steht. Bei der Evaluierung des Goals

```
:- alo_fmo(+,A:=[B,C],_,_),
   alo_fmo(+,D:=[E,F],_,_),
   C#=F,
   C#='1'.
```

wird zunächst die erste Regel a) von $alo\_fmo/5$ sowohl für *alo_fmo(+,A:=[B,C],_,_)* als auch für *alo_fmo(+,D:=[E,F],_,_)* ausgeführt. Das Constraint $C = 1$ führt zum Backtracking, da der Domain von C gemäß der ersten Regel von $alo\_fmo/5$ an die Menge $\{ay, af, yconst\}$ gebunden ist. Backtracking führt zur Anwendung der zweiten Regel b) für *alo_fmo(+,D:=[E,F],_,_),* wodurch jetzt das Constraint $C = F$ verletzt wird. Backtracking führt jetzt dazu, dass für *alo_fmo(+,A:=[B,C],_,_)* die Regel b) und für *alo_fmo(+,D:=[E,F],_,_)* die erste Regel a) angewandt wird, was ebenfalls $C = F$ verletzt. Erst die Anwendung der Regel b) sowohl für *alo_fmo(+,A:=[B,C],_,_)* als auch für *alo_fmo(+,D:=[E,F],_,_)* erfüllt die Constraints $C = F$ und $C = 1$.

Backtracking kann vermieden werden, wenn die Addition in einer einzigen Regel verschmolzen wird:

```
alo_fmo(+,Def,[Arg1,Arg2],alu,IT) :-
    Def::[ar,af],
    Arg1::[ar,ax,mr,sr,ay,af],
    Arg2::[ay,af,yconst,1],
    IT::[1,4,5,8,9],
```

```
IT#=1#=>Def#=ar#,
IT#=1#=>Arg2##yconst,
Arg1::[ay,af]#<=>Arg2#='1'.
```

Diese Formulierung ist nicht mehr so intuitiv wie die erste Version. Vor allem, wenn es wesentlich mehr Regeln zur Spezifikation einer Operation gibt, wird ein Verschmelzen immer schwieriger und die Regel schnell undurchsichtig. Allerdings benötigt die Evaluierung des oben gezeigten Goals jetzt keinen einzigen Backtrackingschritt mehr.

generalisierte Propagierung

Eine wesentlich elegantere Methode als das Verschmelzen von Regeln bietet die *generalisierte Propagierung*. Diese führt das Verschmelzen einer gegebenen Menge von Regeln selbstständig durch. Das Konzept lässt sich am besten an einem Beispiel erläutern. Dazu sei ein Prädikat $p/3$ mit den folgenden Fakten gegeben:

```
p(ar,ar,ay).
p(af,ar,af).
```

Zu einer Anfrage

```
:- p(X,Y,Z).
```

gibt das System wie gewohnt die alternativen Variablenbelegungen gemäß der beiden Fakten aus:

```
X=ar
Y=ar
Z=ay
More?(;)

X=af
Y=ar
Z=af
yes
```

Die Bibliothek *Propia* des ECLiPSe-Systems beinhaltet das Prädikat $infers/2$, das wie folgt aufgerufen wird:

```
:- p(X,Y,Z) infers fd[2].
```

Die Antwort des Systems ist jetzt

```
X={[ar,af]}
Y=ar
Z={[ay,af]}
yes
```

[2] Die generalisierte Propagierung ist nicht auf endliche Bereiche beschränkt. Das zweite Argument (fd=finite domain) spezifiziert hier daher, dass die Propagierung über den endlichen Wertebereichen durchgeführt werden soll.

Durch $p(X,Y,Z)\ infers\ fd$ wird ein Constraint generiert, das die einzelnen Regeln von $p/3$ ausführt, sich die Domains der Argumente von $p/3$ für die erfolgreichen Anwendungen merkt, und die Domains dann argumentweise vereinigt. Weiterhin wird ein Constraint ***p(X,Y,Z) infers fd*** im Constraintstore abgelegt, das bei Veränderungen der Domains der Variablen den Prozess wiederholt. Die Anfrage

```
:- p(X,Y,Z) infers fd, Z##ay.
```

führt zu der Antwort

```
X=af
Y=ar
Z=af
yes
```

Durch das Constraint $Z \neq ay$ wird das Constraint $p(X,Y,Z)\ infers\ fd$ reaktiviert. Da ay nicht mehr im Wertebereich der Variablen Z liegt, führt die erste Regel zu keiner Lösung mehr und wird daher nicht mehr berücksichtigt.

Die generalisierte Propagierung erlaubt, die ursprünglich gezeigte Spezifikation der Addition beizubehalten. Lediglich der Aufruf von $alo\_fmo(...)$ in $fmo/5$ muss durch $alo\_fmo(...)\ infers\ fd$ ersetzt werden. Welche Form der Modellierung vorzuziehen ist, hängt letztendlich davon ab, ob der Schwerpunkt auf einer intuitiven oder eher auf einer effizienten Modellierung liegt und ob die Maschinenoperationsmuster evtl. automatisch generiert werden sollen. Im letzten Fall ist die Variante der Aufzählung der einzelnen Muster in Kombination mit $infers/2$ sicher die geeignetste.

5.1.2. FMOs der Datentransfer-Operationen

In Abb. 5.3 ist die Spezifikation der Datentransfer-Operationen der ADSP2100-Familie zu sehen (siehe auch die Spezifikation der Instruktionstypen in Abb. 5.2). Diese sind gemäß der Klassifizierung von abstrakten Operationen in vier Gruppen unterteilt:

1. Laden von Konstanten in Register (CONST-Operationen).
2. Datentransfers zwischen Registern (COPY-Operationen).
3. Laden von Speicherinhalten in Register (LOAD-Operationen).
4. Sichern von Registerinhalten im Speicher (STORE-Operationen).

In der Abbildung 5.3 ist die starke Abhängigkeit zwischen den Instruktionstypen und den erlaubten ST-Komponenten für Setzungen und Benutzungen zu sehen.

Es werden zunächst die Regeln zur Definition der CONST-Operationen betrachtet. Die Definition kann fast analog zu der in der Abbildung gezeigten Spezifikation umgesetzt werden und ist in Abb. 5.4 zu sehen. Man erinnere sich, dass die Werte der Konstanten in den FMOs der Klasse $const$ durch den Operator repräsentiert werden

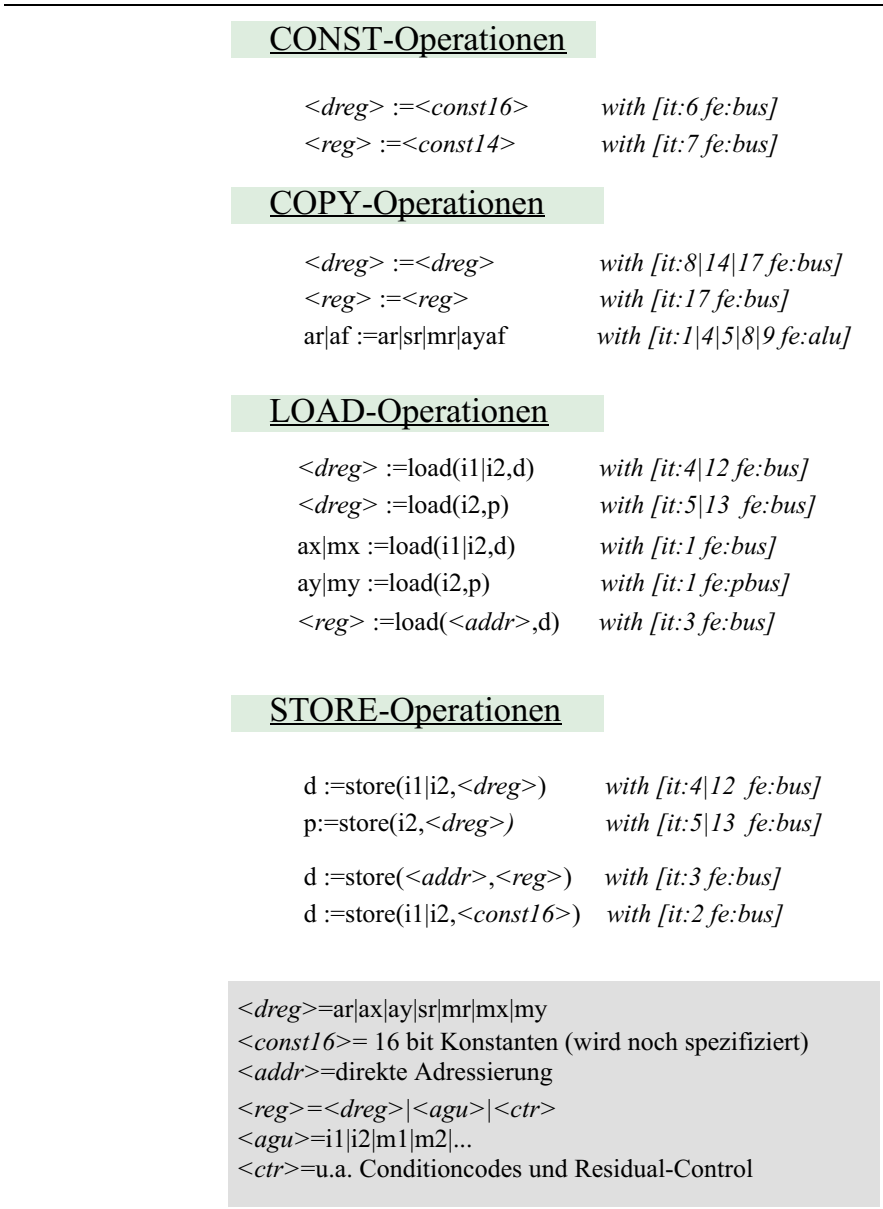


Abbildung 5.3.: FMOs der Datentransfer-Operationen der ADSP2100-Familie.

```
const_fmo(int(C),D,[],bus,6) :-     Regel 1
  const(C,16),
  dreg(D).
const_fmo(int(C),D,[],bus,7) :-     Regel 2
   const(C,14),
   reg(D).
const_fmo(int(C),D,[],bus,_) :-     Regel 3
   yconst(C),
   D#=yconst.
const_fmo(int(C),D,[],bus,_) :-     Regel 4
   const(C,14),
   D#=addr.


dreg(D):-
   D::[ar,ax,ay,sr,mr,mx,my].
reg(D):-
   D::[i1,i2,m1,m2,ar,ax,ay,sr,mr,mx,my].
mem(D):-D::[p,d].


const(C,N):-
   Max is (2^(N-1))-1,
   abs(C)=< Max.
yconst(C):-
   zp(['217x','218x','21msp58/59']),
   C::[0,1,2,4,8,16,32,..,32767,
       -2,-3,-5,-9,-17,..,-32768].
```

Abbildung 5.4.: Definition des Prädikats $const\_fmo/5$.

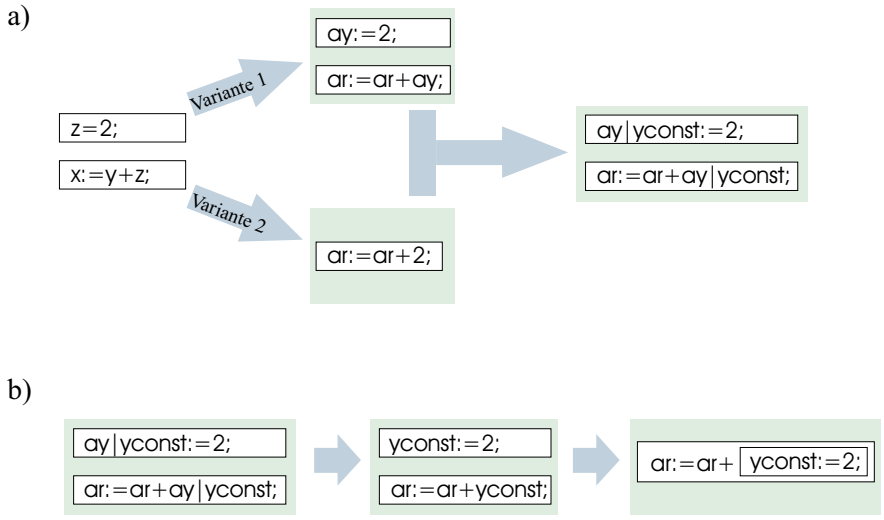


Abbildung 5.5.: CoLIR-Modell zur Darstellung von Konstanten.

und die Menge der Argumente leer ist (z.B. *ar:=const(1)*, wobei *const(1)* als Operator zur Darstellung der Konstante 1 zu interpretieren ist). Die ersten beiden Regeln (Regel 1 und 2) von $cons\_fmo/4$ entsprechen der Spezifikation in Abb. 5.3. Die Prädikate $dreg/1$ und $reg/1$ spiegeln die Abstraktionen `<reg>` und `<dreg>` wider, die die Setzung auf die möglichen ST-Komponenten beschränken, die für CONST-Operationen möglich sind. Das Prädikat $const/2$ prüft, ob eine vorzeichenbehaftete Integerkonstante C durch Bitbreite N darstellbar ist. Die Regeln 3 und 4 von $cons\_fmo/4$ stellen das Modellierungskonzept von CoLIR dar, Konstanten als unmittelbares Argument in FMOs zu benutzen. Diese Modellierung dient der simultanen Darstellung alternativer Umsetzungen von Konstanten. Als Beispiel ist das Programm `z=2;x=y+z;` in Abb. 5.5a) gezeigt, für die es u.a. die folgenden zwei Umsetzungen in Maschinenoperationen gibt:

- *Variante 1*: Eine Möglichkeit, den beiden abstrakten Operationen Maschinenoperationen zuzuordnen, ist wie folgt gegeben:

  ```
  ay:=2
  ar:=ar+ay
  ```
  Es wird also zunächst ein Laden der Konstante 2 in ein Register des Registerfiles ay durchgeführt und dieser Registerinhalt dann in der Maschinenoperation ar:=ar+ay benutzt.

- *Variante 2*: Da die Konstante 2 auch als zweites Argument der Addition erlaubt ist, ist auch die Umsetzung durch eine einzige Maschinenoperation `ar:=ar+2` möglich.

Modellierung von Konstanten

Um nun beide Möglichkeiten innerhalb eines CoLIR-Programms repräsentieren zu

```
copy_fmo(mv,D,[S],bus,t17) :- %Regel 1
   reg(D),reg(S).
copy_fmo(mv,D,[S],bus,IT) :-  %Regel 2
   dreg(D),dreg(S),
   IT::[t8,t14].
copy_fmo(mv,D,[S],alu,IT) :-  %Regel 3
   D::[ar,af],
   S::[ar,sr,mr,ay,af],
   IT::[t1,t4,t5,t8,t9].



load_fmo(ld,D,[A,d],bus,IT) :-
   A::[i1,i2],
   dreg(D),
   IT::[t4,t12].
load_fmo(ld,D,[i2,p],bus,IT) :-
   dreg(D),
   IT::[t5,t13].
load_fmo(ld,D,[A,d],bus,t1) :-
   A::[i1,i2],
   D::[ax,mx]
load_fmo(ld,D,[i2,p],pbus,t1) :-
   D::[ay,my].
load_fmo(ld,D,[addr,d],bus,t3) :-
   reg(D).
```

Abbildung 5.6.: Definition der COPY- und LOAD-Operationen der ADSP2100-Familie.

können, werden für Konstanten flüchtige Komponenten eingeführt. Regel 3 von Prädikat $const\_fmo/5$ spezifiziert Konstanten der Menge $yconst$, die unmittelbar im zweiten Argument der Maschinenoperationen der Addition vorkommen können. Die Umsetzung beider gezeigter Varianten ist durch die folgenden beiden FMOs gegeben (siehe auch Abb. 5.5b):

```
ay|yconst:=2
ar:=ar+ay|yconst
```

Wird nun der Setzung der FMO ay|yconst:=2 die ST-Komponente $yconst$ zugeordnet, werden die beiden FMOs zu einer komplexen FMO `ar:=ar+(yconst:=2)` zur Darstellung von `ar:=ar+2` zusammengefasst (siehe Abb. 5.5 b). Der Aufruf von Prädikat $yconst/1$ in Regel 3 von $const\_fmo/5$ prüft, ob die Integerkonstante aus dem Bereich der erlaubten Konstanten in ALU-Operationen ist. Der darin vorkommende Aufruf von $zp/1$ prüft, ob der aktuell betrachtete Zielprozessor der ADSP2100-Familie in der angegebenen Liste vorkommt. Nur die dort angegebenen Prozessoren der ADSP2100-Familie unterstützen Konstanten der Klasse $yconst$.

Die Regeln zur Definition der COPY- und LOAD-Operationen sind in Abb. 5.6 gezeigt. Man erkennt, dass die Anwendung der generalisierten Propagierung eine nahezu analoge Formulierung zur Spezifikation in Abb. 5.3 erlaubt und gleichzeitig die

simultane Repräsentation aller durch die Regeln repräsentierten alternativen Maschinenoperationen ermöglicht. Die Definition der STORE-Operationen kann analog erfolgen und wird hier nicht mehr angegeben. Eine Besonderheit ist noch bei den COPY-Operationen hervorzuheben: Einige der Datentransfers werden durch die ALU durchgeführt. Dies wird durch die dritte Regel von $copy\_fmo/4$ ausgedrückt, was in den entsprechenden Maschinenoperationen zu einer Zuordnung der Funktionseinheit zu alu führt. Es ist hierdurch potenziell möglich, die Datentransfers über die ALU parallel zu anderen Datentransfers auszuführen.

Richtlinien zur Verwendung von LOAD- und STORE-Operationen

Zugriffe auf den Speicher werden in CoLIR immer durch LOAD- bzw. STORE-Operationen modelliert. Speicherreferenzen kommen also nur in den Benutzungsalternativen des zweiten Operanden der Registerfilezuordnungen von LOAD-Operationen oder als Setzungsalternativen von STORE-Operationen vor. Weiterhin können sie noch in den Eingangs- und Ausgangssetzungen vorkommen. Somit werden Speicherzugriffe immer gleich modelliert, was es einfacher macht, generische Analysen und Optimierungen zu entwickeln. Weitere Konventionen zur Modellierung von LOAD- und STORE-Operationen werden noch im Kapitel 5.3 erläutert, wo es um die Darstellung komplexer Maschinenoperationen geht; dort wird die Handhabung komplexer Adressierungsmodi sowie der unmittelbare Zugriff auf den Speicher von Benutzungen, innerhalb von beliebigen Maschinenoperationen, gezeigt.

5.1.3. Weitere Klassen von CoLIR-Operationen

Die CoLIR-Operationen der anderen Klassen aus $\mathcal{OC}$ weisen keine Besonderheiten auf und können mit den bislang beschriebenen Konzepten modelliert werden.

5.2. Datentransferpfade

Die Handhabung von Datentransferpfaden ist im Kontext irregulärer Prozessoren von zentraler Bedeutung. Bei der Instruktionsauswahl muss gewährleistet sein, dass Datentransferpfade zwischen den datenflussabhängigen Setzungen und Benutzungen existieren. Um die frühzeitige Bindung von Ressourcen zu vermeiden, ist die Beibehaltung alternativer Datentransferpfade - über Phasen hinweg - ein wichtiger Aspekt. Dazu ist eine kompakte und handhabbare Repräsentation wichtig. In diesem Kapitel wird zuerst gezeigt, wie die Existenz von Datentransferpfaden mittels Constraints sichergestellt werden kann. Dann wird die explizite Darstellung alternativer Datentransferpfade gezeigt, die durch unsere Modellierung mittels einer einzigen Sequenz von FMOs der Klasse $copy$ realisiert werden kann.

5.2.1. Existenz von Datentransferpfaden

Lösungen von $COVER$ umfassen nur Maschinenoperationen, für die die Existenz von Datentransferpfaden von den Setzungen zu den korrespondierenden Benutzungen in-

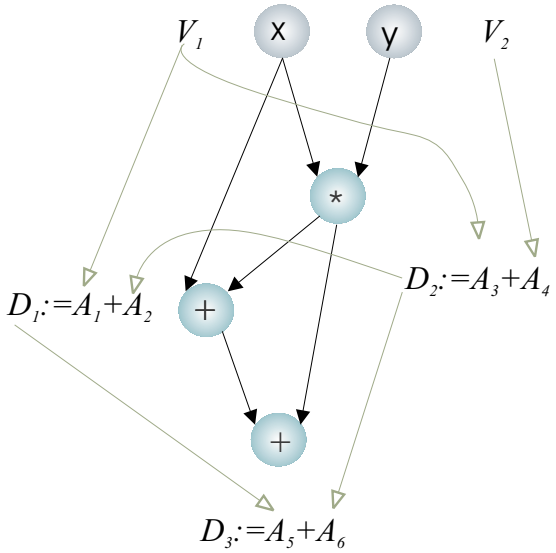


Abbildung 5.7.: Existenz von Datentransferpfaden.

nerhalb eines Basisblocks garantiert werden kann. In Abb. 5.7 sind die Abhängigkeiten zwischen den datenflussabhängigen Setzungen und Benutzungen gezeigt. Dabei müssen auch Datentransfers von den Eingangssetzungen zu den Benutzungen in den FMOs, sowie die Setzungen von FMOs zu den Ausgangssetzungen eines Basisblocks berücksichtigt werden. Jeder Knoten der Menge $VAR_{in}(b)$ und der Menge $VAR_{out}(b)$ ist daher mit einer STK-Variablen assoziiert, deren Domain die möglichen ST-Komponenten bei Eintritt in den Basisblock bzw. bei Austritt aus dem Basisblock umfasst.

Das benutzerdefinierte Constraint

$$X \longrightarrow^* Y$$

stellt die Setzung X und Benutzung Y derart in Beziehung, dass für jede Lösung $\theta_{\{X,Y\}}$ ein Datentransferpfad von $\theta(X)$ nach $\theta(Y)$ existiert. In der ADSP2100-Familie gibt es für die sequenziellen Komponenten bzgl. der Existenz von Datentransferpfaden keine Einschränkungen (wenn man die Realisierung von COPY-Operationen durch arithmetische Operationen mit einbezieht: a:=cp(b) enspricht auch a:=b+0). Dies ist in der Definition von $X \longrightarrow^* Y$ in Abb. 5.8 durch Regel 1 realisiert worden. Darin wird durch das Prädikat $sk/1$ gerade die Menge der Registerfiles und der Speicherbänke mit der Setzung X und der Benutzung Y assoziiert ($X, Y \in \mathcal{RF} \cup \mathcal{M}$). Wenn man flüchtige Komponenten einbezieht, existieren für diese keine Datentransfers zu anderen ST-Komponenten. In diesem Fall muss die entsprechende Benutzung gleich der Setzung sein. Dies ist durch Regel 2 von $\rightarrow\ /2$ realisiert worden, worin durch das Prädikat $tk/1$ gerade die Menge der flüchtigen Komponenten mit X verbunden wird ($X \in \mathcal{T}$).

Um die Existenz der Datentransferpfade zwischen den Setzungen und korrespondierenden Benutzungen zu gewährleisten, wird für jede Kante $fv \rightarrow^{pos} fv' \in FV(b)$ ein Constraint $fv_{[0]} \longrightarrow^* fv'_{pos}$ generiert.

```
X -->* Y :- X --> Y infers fd.

X --> Y :- sk(X),sk(Y).     Regel 1
X --> X :- tk(X).           Regel 2


sk(X) :- sk1(X) infers most.
sk1(X):-reg(X).
sk1(X):-mem(X).

tk(X) :- X::['*(X,Y),1,yconst,...].
```

Abbildung 5.8.: Definition von $\rightarrow^* /2$, die hier in infix Notation definiert wird.

Für die sequenziellen Komponenten der ADSP2100-Familie bestehen Restriktionen bzgl. der möglichen erreichbaren sequenziellen Komponenten mittels einer einzigen COPY-Operation. In das Feedbackregister af kann z.B. nur direkt aus den Registerfiles ar, mr, sr, ax und ay kopiert werden. Werte aus af können nur direkt in ar kopiert werden. Solche Restriktionen werden durch die explizite Repräsentation von Datentransferpfaden erfasst, deren Modellierung im folgenden Abschnitt gezeigt wird.

5.2.2. Alternative Datentransferpfade dynamischer Länge

Datentransferpfade zwischen einer Setzung S_0 und einer Benutzung B_n können in CoLIR auch explizit durch eine Sequenz von n FMOs der Klasse $copy$ dargestellt werden:

$$\begin{aligned} D_1 &:= cp(S_0) \\ D_2 &:= cp(D_1), \\ &\vdots \\ B_n &:= cp(D_{n-1}) \end{aligned}$$

Durch diese Sequenz kann die Menge aller Datentransferpfade der Länge n von Setzungsalternativen aus D_0 zu Benutzungssalternativen aus D_n dargestellt werden. Dies ist möglich, da durch eine einzige FMO der Klasse $copy$ alle COPY-Operationen und somit alle Datentransferpfade der Länge 1 repräsentiert werden können.

dynamische Datentransferpfade

Mit einem einfachen Modellierungstrick ist es möglich, dass die oben gezeigte Sequenz von FMOs die Menge aller Datentransferpfade der Länge kleiner gleich n repräsentiert. Diese werden *dynamische Datentransferpfade* genannt. Der Trick besteht darin, COPY-Operationen der Form $rf := cp(rf)$ zuzulassen, die dann während der Codegenerierung als Dummy-COPY-Operationen mit Kosten 0 interpretiert werden. Um diese nicht mit den realen Datentransfers zu verwechseln, wird diesen eine virtuelle Funktionseinheit $none$ zugeordnet. Die Definition der COPY-Operationen muss dazu nur um die folgende Regel erweitert werden:

```
copy_fmo(mv,D,[D],none,_).
```

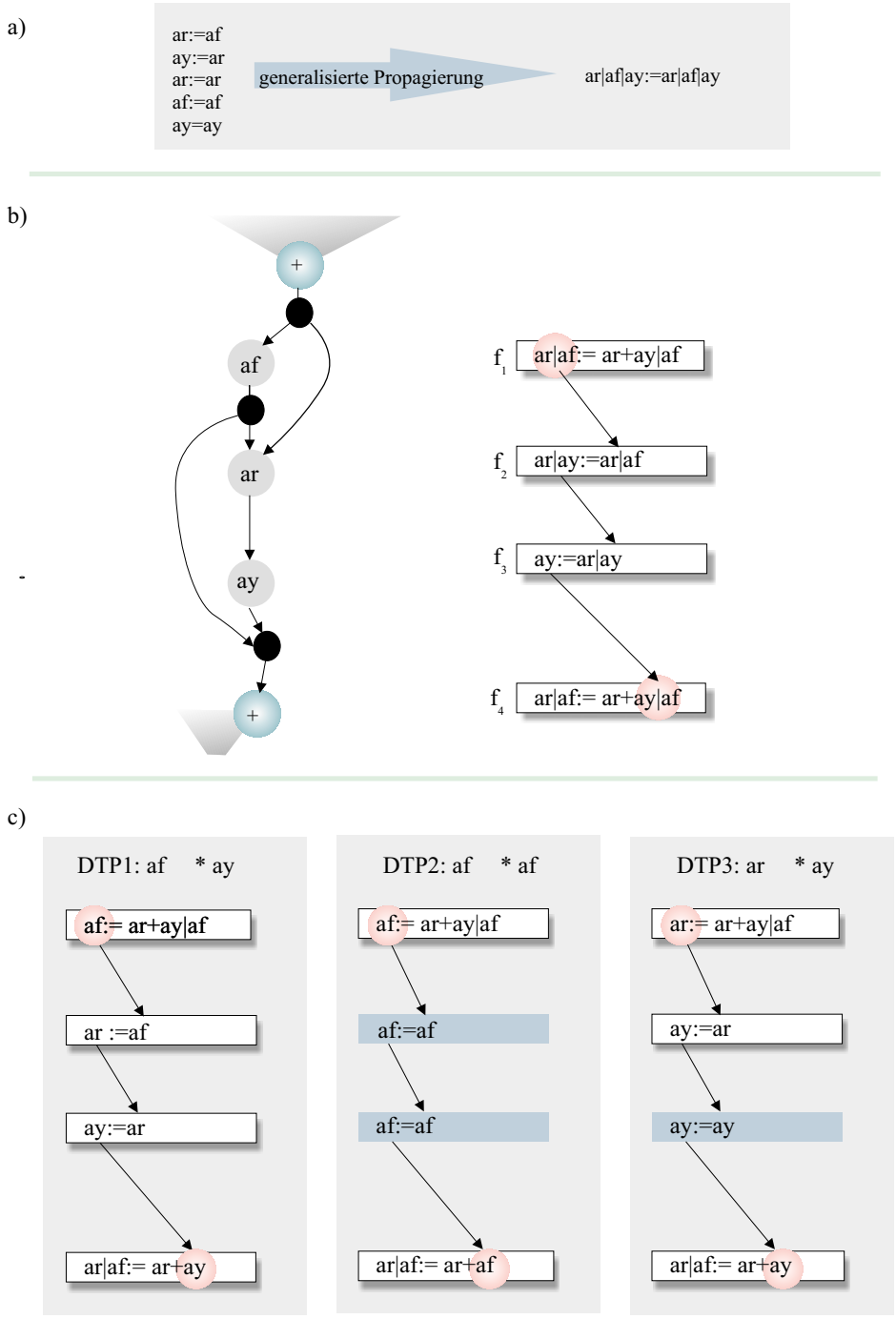


Abbildung 5.9.: Dynamische Datentransferpfade.

Ein Beispiel ist in Abb. 5.9a-c) gezeigt. Darin werden die minimal notwendigen Datentransfers zwischen einer Setzung $(f_1)_{[0]}$ mit $D((f_1)_{[0]}) = \{ar, af\}$ und deren Benutzung $(f_4)_{[2]}$ mit $D((f_4)_{[2]}) = \{ay, af\}$ dargestellt. In Abb. 5.9a) sind die möglichen COPY-Operationen gegeben, die die entsprechenden Datentransfers umsetzen können. Weiterhin ist die entsprechende faktorisierte Registertransferoperation dieser Maschinenoperationen gezeigt. Abb. 5.9b) umfasst einen partiellen Datenflussgraphen mit der oben angegeben Setzung und Benutzung. Dazwischen sind die minimal möglichen Datentransferpfade abgebildet, um einen Wert aus af oder ar nach ay bzw. nach af zu bewegen. Der entsprechende dynamische Datentransferpfad zur Umsetzung dieser alternativen Datentransferpfade ist durch die FMOs f_2 und f_3 gegeben. Die Abbildung auf die abgebildeten drei minimal notwendigen Datentransferpfade ($DTP1$, $DTP2, DTP3$) ist in Abb. 5.9c) gegeben. Die Dummy-COPY-Operationen sind grau unterlegt und würden im weiteren Verlauf der Codegenerierung eliminiert.

Es ist weiterhin möglich, auch alternative Datentransferpfade darzustellen, die sowohl aus COPY-, LOAD- und STORE-Operationen gebildet werden können. Dies wird erreicht, indem die generalisierte Propagierung über die Regeln von $copy\_fmo/5$, von $load\_fmo/5$ und von $store\_fmo/5$ angewandt wird. Dazu ist zunächst ein weiterer Modellierungstrick notwendig, so dass diese Prädikate alle die gleiche Struktur bzgl. ihrer Argumente aufweisen. Dies ist notwendig, damit die generalisierte Propagierung die Domains der einzelnen Argumente von $copy\_fmo/5$, von $load\_fmo/5$ und $store\_fmo/5$ (d.h. die Setzung, die Benutzungen der Operanden, die FE- und die IT-Variable) korrekt zusammenfügen kann. Dazu ist die Modifikation der Regeln von $copy\_fmo/5$ notwendig, da COPY-Operationen im Unterschied zu den LOAD- und STORE-Operationen nur einen Operanden in den Registerfilezuordnungen besitzt. Hier wird ein Dummy-Argument an Position [1] eingefügt, so dass der ursprünglich zu kopierende Operand jetzt an Position [2] steht. Das Dummy-Argument steht jetzt an der Position, wo in den LOAD- und STORE-Operationen die Adresse der Speicherreferenz steht.

5.3. Modellierung komplexer und graphbasierter FMOs

Es wird zunächst die Modellierung der komplexen FMOs und dann die Modellierung graphbasierter FMOs erläutert, mit der die Erkennung und die Darstellung entsprechender Maschinenoperationsmuster in einem einheitlichen Konzept möglich ist.

5.3.1. Modellierung komplexer FMOs

Komplexe Maschinenoperationen setzen arithmetische Ausdrücke um, die aus mehr als einem Operator zusammengesetzt sind. Im Datenflussgraphen sind dies Muster, die mehr als einen Knoten umfassen. Dies ist für das Beispiel einer MAC-Operation in Abb. 5.10 gezeigt. Dabei gibt es häufig zu einem Datenflussgraphen mehrere Möglichkeiten, ein komplexes Muster zu bilden. In der Abbildung gibt es zwei Möglichkeiten, eine MAC-Operation zu bilden, die sich im Knoten $+_7$ überlappen. Weiterhin besteht die Möglichkeit, alle drei Operationen separat auszuführen. Die FMOs, die mit den

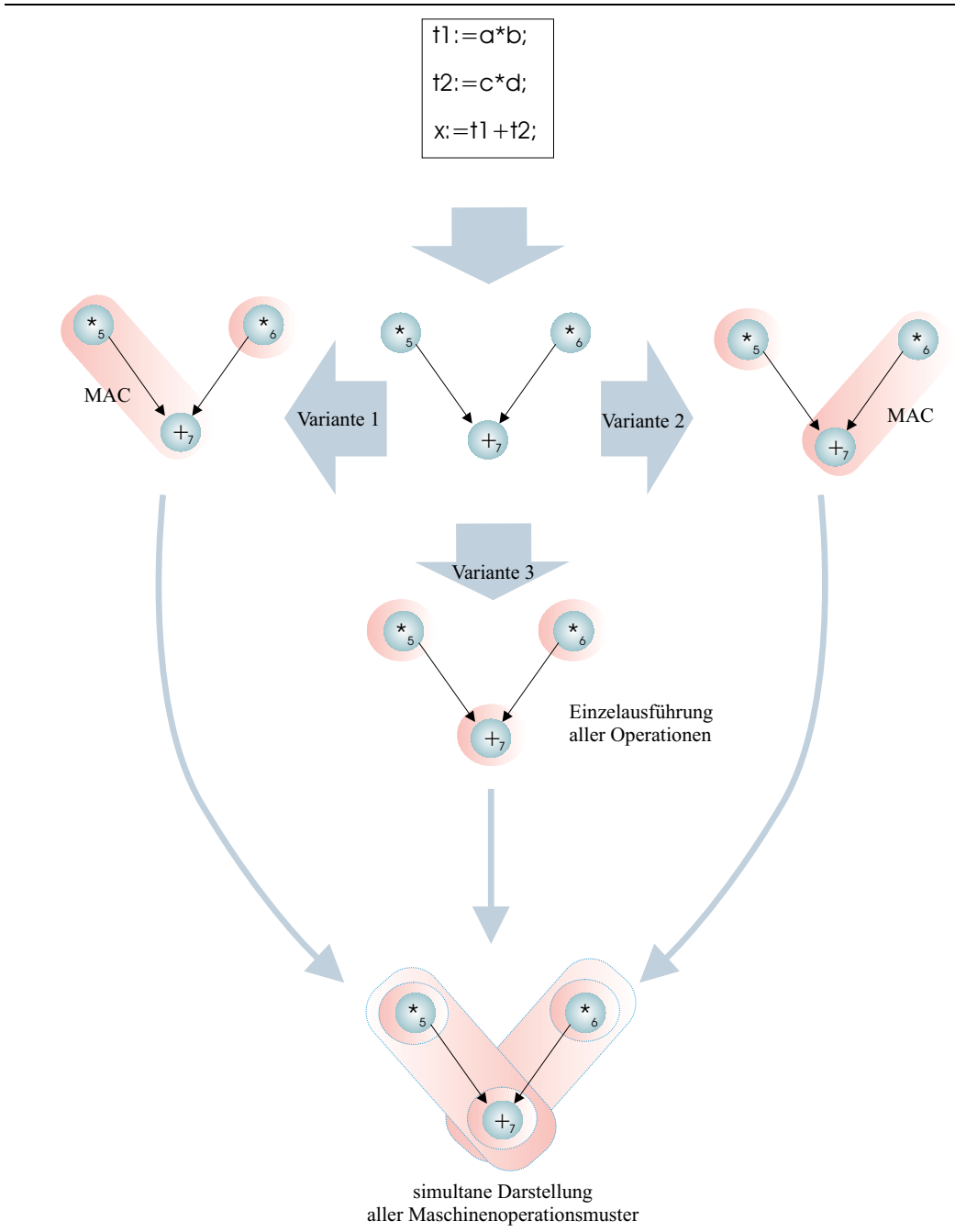


Abbildung 5.10.: Alternative komplexe und nicht komplexe Operationen.

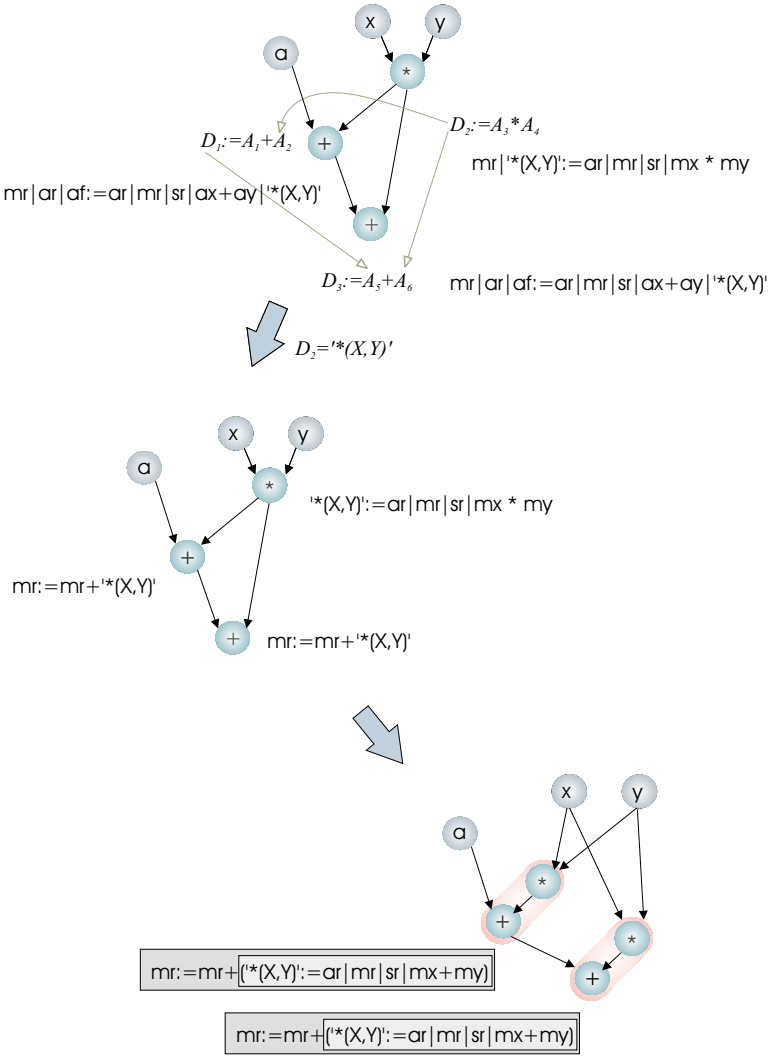


Abbildung 5.11.: Handhabung der komplexen MAC-Operation.

einzelnen Knoten des Datenflussgraphen assoziiert sind, müssen die Möglichkeit bieten, alle drei Varianten simultan darzustellen, da es das Ziel ist, in den alternativen Überdeckungen alle alternativen Möglichkeiten zu subsummieren.

Die Modellierung komplexer Maschinenoperationen basiert auf der Verwendung flüchtiger Komponenten in den Setzungs- und Benutzungsalternativen. Eine Operation, die Sub-Operation einer komplexen Maschinenoperation sein kann, wird durch eine partielle Maschinenoperation modelliert, die ihr Resultat in eine flüchtige Komponente schreibt (die Modellierung der Konstanten basiert auf dem gleichen Prinzip). Die Definitionen der FMOs werden so erweitert, dass sie entsprechende partielle Maschinenoperationen als Alternativen umfassen.

Die MAC-Funktionseinheit der ADSP-2100-Familie besteht aus einem Multiplizierer und einem Addierer / Subtrahierer (siehe Abb. 5.12). Die Signalleitung zwischen Mul-

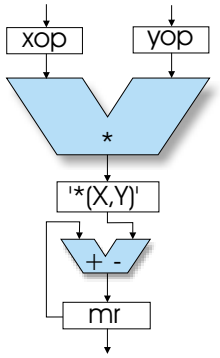


Abbildung 5.12.: Schema der MAC-Einheit.

tiplizierer und Addierer wird als flüchtige Komponente '$*(X,Y)$' modelliert. Die Regeln der Addition, Subtraktion und der Multiplikation werden wie folgt ergänzt:

```
alo_fmo(Op,mr,[mr,'*(X,Y)'],mac) :-
   Op::[+,-].
alo_fmo(*,'*(X,Y)',[A1,my],mac) :-
   A1::[ar,mr,sr,mx].
```

Es existieren keine Datentransferpfade von '$*(X,Y)$' zu anderen Registerfiles und ebenfalls nicht von anderen Registerfiles zu '$*(X,Y)$'. Die Zuordnung einer Setzung zu '$*(X,Y)$' führt zur zwingenden Bindung der entsprechenden Benutzung an '$*(X,Y)$' und repräsentiert somit die komplexe FMO zur Umsetzung einer MAC-Operation. Dies wird durch die Definition von $\rightarrow^*\ /2$ bereits erfasst. In Abb. 5.11 besteht die Möglichkeit, die beiden Additionen und die Multiplikation als eigenständige Maschinenoperationen oder auch als MAC-Operation umzusetzen, da D_2 sowohl an mr als auch an '$*(X,Y)$' gebunden werden kann. Die Bindung von D_2 an '$*(X,Y)$' führt zur Bindung von A_2 und von A_6 an '$*(X,Y)$' (dies wird durch die Constraints $D_2 \longrightarrow^* A_2$ und $D_2 \longrightarrow^* A_6$ erreicht). Es entstehen zwei MAC-Operationen, wie es unten in Abb. 5.11 gezeigt ist. Dabei muss der Multiplikationsknoten bzw. die korrespondierende FMO dupliziert werden.

5.3.2. Modellierung graphbasierter FMOs

Graphbasierte Maschinenoperationen müssen nicht (können aber) komplex sein. Die SQUARE-Operation des ADSP2100 ist z.B. eine nicht komplexe graphbasierte Maschinenoperation. Sie erfordert nur, dass die Operanden gleich sind, im Datenflussgraphen also vom gleichen Knoten herrühren (siehe Abb. 5.13 a). Unter Berücksichtigung der Graphstruktur kann die Menge der Maschinenoperationen, die die MAC-Operation sowie die Multiplikation realisieren, um die in Abb. 5.13 (b,c,d) gezeigte Menge erweitert werden. Damit ist die mehrfache Benutzung des gleichen Registerfiles zulässig.

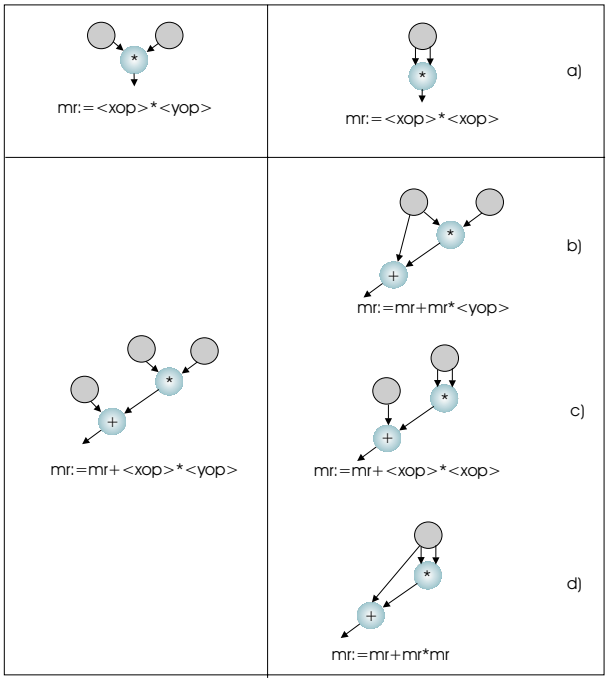


Abbildung 5.13.: Graphbasierte Maschinenoperationen.

Zur Modellierung graphbasierter Maschinenoperationen müssen die Regeln des Prädikats $fmo/5$ gemäß der gezeigten Maschinenoperationen in Abb. 5.13 erweitert werden. Es werden zusätzliche Constraints über der Graphstruktur auferlegt, die bei Nichtvorhandensein des Graphmusters dafür sorgen, dass auch die korrespondierenden Maschinenoperationen aus der Lösungsmenge eliminiert werden. Sind z.B. die beiden Argumentknoten einer Multiplikation nicht identisch, wird für die Argumente des Registertransferpattern die Ungleichheit gefordert, womit die Registerzuordnung $mr := [X, X]$ nicht mehr in Frage kommt. Für das komplexe Graphmuster aus Abb. 5.13 b) wird im Falle der vorliegenden Struktur von Addition und Multiplikation geprüft, ob der erste Operand der Addition und der Multiplikation durch den gleichen Graphknoten repräsentiert werden. Falls nicht, wird das Registerfile mr aus den Benutzungsalternativen des ersten Operanden der Registerfilezuordnung des korrespondierenden FMOs eliminiert.

5.3.3. Richtlinien zur Modellierung von komplexen Maschinenoperationen

Durch komplexe Maschinenoperationen können nicht nur komplexe arithmethische Ausdrücke sondern auch komplexe Adressierungsmodi dargestellt werden. Sobald komplexe Maschinenoperationen einmal erkannt werden, können diese auch durch elementare FMOs repräsentiert werden. Dies kann erreicht werden, indem für die komplexen Operationen entsprechende Operatoren eingeführt werden (z.B. mac zur Darstellung der MAC-Operation in der Form $mr := mac(mr, mx, my)$). In CoLIR ist

dies grundsätzlich möglich, wird aber aus den folgenden Gründen nicht umgesetzt:

- Die Beibehaltung der komplexen FMOs ist wesentlich transparenter, da sie die umzusetzenden komplexen Ausdrücke unmittelbar reflektieren. Außerdem spart man sich die Einführung zusätzlicher Operatoren.
- Insbesondere Adressmodi können durch komplexe FMOs wesentlich klarer dargestellt und analysiert werden, was gerade in den $Alias-$ und $Disambiguations$-Analysen wichtig ist. Es können wesentlich einfacher generische Analysen und Optimierungen realisiert werden, da diese nicht an spezifische Operatorsymbole eines Prozessors angepasst werden müssen.
- Komplexe FMOs können wieder einfach in ihre Sub-FMOs zerlegt werden, was ebenfalls wieder in generischer Form realisiert werden kann.

Darüber hinaus werden noch folgende Richtlinien bzgl. der Verwendung von LOAD- und STORE-Operationen eingehalten:

- Zugriffe auf den Speicher werden immer durch eine FMO der Klasse $load$ oder $store$ umgesetzt. ST-Komponenten $st \in \mathcal{M}$ kommen daher nur in den Benutzungen des zweiten Operanden von LOAD-Operationen vor oder in den Setzungen von STORE-Operationen. Ist es also möglich, dass eine Maschinenoperation direkt aus dem Speicher lesen kann (z.B. bei DSPs von Texas Instrument der TMS320C1x/C2x/C5x-Familie [60]), wird dies immer durch eine komplexe FMO realisiert, die an der entsprechenden Argumentposition eine LOAD-Operation als Sub-FMO besitzt.
- Adressmodi werden explizit durch die entsprechenden Operationen umgesetzt.

Durch die Einhaltung dieser Richtlinien wird die Entwicklung generischer Optimierungen und Analysen erheblich vereinfacht.

5.4. Konventionen

Es ist notwendig, bestimmte Bedingungen und Konventionen zu erfüllen, die nicht unmittelbar mit dem Befehlssatz eines Prozessors in Bezug stehen. Für die ADSP2100-Familie gibt es ein paar zusätzlich zu beachtende Bedingungen und Konventionen. Diese werden durch die Generierung weiterer Constraints gehandhabt:

- Alle Speicherreferenzen, die auf dieselben Arrays zeigen, müssen denselben Speicherbänken zugeordnet werden. Referenzen, bei denen nicht zugeordnet werden kann, auf welche Daten sie zeigen, werden dem d-Speicher zugeordnet. Existieren solche Referenzen, werden alle weiteren Speicherreferenzen ebenfalls dem d-Speicher zugeordnet.

- Globale Variablen und Arrays werden grundsätzlich dem d-Speicher zugeordnet. Dies ist notwendig, da globale Variablen auch von anderen Modulen eines Programms aus referenziert werden können. Um hier eine unabhängige Compilierung von Modulen zu gewährleisten, muss eine eindeutige Speicherbank definiert sein.

- Statische Variablen und statische Arrays können beiden Speicherbänken (d-Speicher und p-Speicher) zugeordnet werden, da diese nur aus der Funktion heraus referenziert werden können, in der sie deklariert wurden. Lokale Variablen, die gespilled werden müssen, können ebenfalls beiden Speichern zugeordnet werden, um hier parallele LOAD-Operationen beim Wiedereinlagern der Werte in Register zu ermöglichen. Allerdings kann hierdurch ein erhöhter Speicherbedarf entstehen, da diese Variablen, die dem p-Speicher zugeordnet werden, nicht mehr auf dem Funktionsstack alloziert werden können[3].

- Die Eingangs- und Ausgangssetzungen der Basisblöcke werden an sequenzielle Komponenten gebunden. Hierdurch werden komplexe Maschinenoperationsmuster, die sich über Basisblockgrenzen hinwegstrecken, ausgeschlossen.

- Es werden keine FMOs an die AGU-Einheiten gebunden, die nicht an Adressoperationen beteiligt sind. Die Ausnutzung der AGUs kann erst effektiv nach der Bestimmung der Instruktionsanordnung erfolgen, da hier erst die Zugriffsreihenfolge auf Programmvariablen im Speicher feststeht und diese Reihenfolge Einfluss auf die effektive Nutzung von Autoinkrement- und Autodekrementoperationen hat. Hier sollen nicht im Vorfeld entsprechende Ressourcen belegt werden. Für jede FMO $f \in FMO(b)$, die an keiner Adressberechnung beteiligt ist, wird ein Constraint $FE_f \in \{alu, mac, shifter\}$ erzeugt, unter der Voraussetzung, dass $D(FE_f) \cap \{alu, mac, shifter\} \neq \emptyset$ gilt.

Das Binden von Speicherreferenzen an einen eindeutigen Speicher ist äußerst relevant, daher wird davon ausgegangen, dass es i.d.R. mindestens eine eindeutige Speicherbank in den ST-Komponenten gibt, denen grundsätzlich alle Programmvariablen zugeordnet werden können.

5.5. Generierung der CSPs von $COVER$

$COVER(fun)$

Im Folgenden wird die Generierung des CSPs einer CoLIR-Funktion gemäß des CSP-Modells $COVER$ beschrieben, welches mit $COVER(fun)$ bezeichnet wird. Da keine Constraints zwischen Domainvariablen unterschiedlicher Funktionen erzeugt werden, ist es hinreichend, sich auf die Betrachtungvon CSPs einzelner Funktionen zu beschränken (diese bilden in sich abgeschlossene Teil-CSPs eines CoLIR-Programms). Ein CSP $COVER(fun)$ kann auf folgende unterschiedliche Weisen generiert und genutzt werden:

[3] Hier gibt es allerdings wieder Verfahren, die den notwendigen Speicherbedarf minimieren können (siehe [110]).

- Generierung des CSPs zu einer IR-reinen CoLIR-Funktion fun, was i.d.R. der ersten Generierung von alternativen Überdeckungen während der Codegenerierung entspricht. Dazu müssen zuerst die abstrakten Operationen auf FMOs abgebildet und dann die notwendigen Constraints gemäß $COVER$ generiert werden.
- Die ausschließliche Generierung der Constraints für $COVER(fun)$ zu einem FMO-reinen CoLIR-Programm (die Domainvariablen sind schon durch das CoLIR-Programm gegeben). Dies wird bei der Generierung der CSP-Modelle einer Codegenerierungsphase genutzt, um die Constraints der Instruktionsauswahl in die entsprechende Phase zu integrieren.
- Expandierung der FMOs eines FMO-reinen CoLIR-Programms, um mehr Flexibilität in einer Phase zu erreichen. In diesem Fall werden die Operatoren der LIR wieder auf frische FMOs abgebildet, so dass man eine neue CoLIR-Funktion fun' erhält und dazu die Constraints $COVER(fun')$ generiert.

Es wird zuerst die Generierung von CSPs $COVER(fun)$ unter diesen drei Verwendungszwecken beschrieben und es wird danach auf die Existenz von Lösungen zu $COVER(fun)$ eingegangen. Am Beispiel der ADSP2100-Familie werden die Laufzeiten für die Generierung von $COVER(fun)$ für Benchmarks des DSPStone-Benchmarksets (s.u.) gezeigt. Weiterhin werden Erweiterungen des Modells gezeigt, die u.a. dynamische Datentransferpfade zur expliziten Repräsentation alternativer Datentransferpfade umfassen. Abschließend wird reflektiert, wie dass CSP-Modell $COVER$ und dessen Erweiterungen entscheidend zur Vermeidung von Backtracking während der Suche nach Lösungen beitragen.

5.5.1. Generierung von $COVER(fun)$

Die Generierung der CSPs $COVER(fun)$ wird für jede Funktion einzeln betrachtet. Die Menge der Domainvariablen eines CSPs $COVER(fun)$ ist gegeben durch die Menge

$$\bigcup_{b \in fun} STK(b) \cup FE(b) \cup IT(b)$$

Die Generierung der Constraints von $COVER(fun)$ wird weiter unten beschrieben und jetzt zuerst die Abbildung von abstrakten Operatoren eines IR-reinen Programms auf FMOs betrachtet. Dazu müssen zunächst alle abstrakten Operationen auf *frische FMOs* abgebildet werden. Frische FMOs sind FMOs, bei denen alle Domainvariablen neu und verschieden sind und deren Domains die maximal möglichen Wertebereiche umfassen. Frische FMOs sind mit keinen Constraints assoziiert. Lösungen eines solchen CoLIR-Programms enthalten also durchaus illegale Maschinenoperationen, da alle willkürlichen Kombinationen von Ressourcen erlaubt sind. Über den so generierten alternativen Überdeckungen werden dann unmittelbar die Constraints des CSP-Modells $COVER$ generiert. Zu einem FMO-reinen CoLIR-Programm können die Constraints dann analog generiert werden, ohne dass vorher frische FMOs generiert werden müssen. Die Constraints werden also über den originalen FMOs des CoLIR-Programms erzeugt. Bei der Erweiterung der alternativen Überdeckungen zu einem

frische FMO

FMO-reinen CoLIR-Programm werden zunächst wieder frische FMOs generiert, wobei die Abbildung diesmal von LIR-Operationen auf LIR-Operationen durchgeführt wird.

Es wird jetzt zuerst die Abbildung von abstrakten Operatoren auf frische FMOs betrachtet, die ein unmittelbares Gegenstück auf dem Zielprozessor besitzen, so dass eine Eins-zu-Eins-Abbildung möglich ist. Die Erweiterung auf nicht unmittelbar abzubildende Operatoren ist Gegenstand des nachfolgenden Unterkapitels. Danach wird die Generierung der Constraints beschrieben und Kriterien angegeben, wann es mit Sicherheit Lösungen für ein CSP $COVER(fun)$ gibt.

Abbildung abstrakter Operatoren auf frische FMOs

Zu jedem Basisblock b eines IR-reinen CoLIR-Programms wird ein neuer Basisblock b' generiert, indem zuerst b nach b' kopiert und dann jede abstrakte Operation $f \in AOP(b')$ durch eine frische FMO f'_f substituiert wird. Dazu wird zu Op_f der korrespondierende Operator Op'_f der LIR ermittelt, und die Domainvariablen der neuen FMO werden gemäß des Operators Op'_f durch ein Prädikat $frische\_fmo/5$ initialisiert. Die Initialisierung erfolgt wie in $fmo/5$, nur, dass jetzt keine Constraints generiert werden. Die Zuordnung der FMOs zu einer abstrakten Operation kann durch folgende Regel realisiert werden:

```
transform(AOP,FMO):-
   AOP = fmo with [class: C,
                   op   : IrOp,
                   bs   : B:=Bs
                  ],
   FMO2=fmo with [
                   class: C,
                   bs   : B:=Bs,
                   op   : LirOp,
                   rfs  : RF:=RFs,
                   fe   : FU,
                   it   : IT],
   irop2lirop(IrOp,LirOps),
   args(Bs,RFs),
   frische_fmo(C,LirOps,RF:=RFs,FU,IT).
```

In dieser Transformation erbt die FMO die Bezeichnerzuordnung der abstrakten Operation. Das Prädikat $args/2$ generiert die zu der Bezeichnerzuordnung korrespondierende Anzahl von Operanden der Registerfilezuordnungen des FMOs. Das Prädikat $irop2lirop/2$ gibt zu Operatoren der abstrakten Operation den korrespondierenden Operator der LIR aus.

Bei der Transformation der Operatoren müssen die Typinformationen des Operators miteinfliessen: Es ist relevant zu wissen, ob es sich beim abstrakten Operator z.B. um eine Addition über Integern, Long-Integern oder Reals handelt, da dies vollkommen unterschiedliche Umsetzungen in Operatoren der LIR erfordern kann. Die Typinformation kann schon bei der Generierung der IR explizit in den Operatorsymbolen verankert werden, wie z.B. durch *int_add* oder *real_add*. In diesem Fall kann das gezeigte

Transformationsschema wie gehabt eingesetzt werden. Eine zweite Möglichkeit ist die implizite Darstellung, in der die Typen der Operatoren durch die Typen der Operanden festgelegt werden, und die Typinformationen in die Definition der Regeln von $fmo/5$ einfließen könnten. Dies würde aber die Erweiterung um die entsprechenden Argumente in den Regelköpfen erfordern und zu relativ komplexen und komplizierten Regeln führen. In dieser Arbeit ist die erste Variante gewählt worden. Ob und welche Vorteile die zweite Variante bringen könnte, kann Gegenstand weiterführender Forschung sein.

Es ist weiterhin möglich, dass zu einem abstrakten Operator mehrere korrespondierende LIR-Operatoren existieren. Diese Handhabung ist einfach zu erfassen, indem das Prädikat $irop2lirop/2$ im zweiten Argument eine Domainvariable akzeptiert, mit der durch die Variablenbelegung die entsprechende Menge von LIR-Operatoren assoziiert wird. Die Erfassung alternativer Operatoren wird in $fmo/5$ durch die generalisierte Propagierung automatisch miterfasst.

Durch die hier gezeigte Handhabung alternativer Operatoren werden noch keine algebraischen Transformationen erfasst, da dies die Betrachtung der Argumente von Operatoren einbeziehen müsste. Grundsätzlich können gewisse algebraische Transformationen mit den gezeigten Konzepten modelliert werden. I.d.R. sind dazu aber zusätzliche virtuelle ST-Komponenten und graphbasierte Constraints notwendig, die zu sehr komplizierten Regeln in $fmo/5$ führen würden. In dieser Arbeit sind nur Transformationen auf der Basis der Kommutativität berücksichtigt worden, die sich durch folgende Modifikation des Aufruf von $alo\_fmo/4$ realisieren lässt:

```
...
fmo(alo,Op,RFs,FE,IT):-
   alo_fmo_kommutativ(Op,RFs,FE,IT) infers fd.
...

alo_fmo_kommutativ(Op,Def:=Args,FE,IT):-
   alo_fmo(Op,Def,Args,FE,IT).
alo_fmo_kommutativ(Op,Def:=[X,Y],FE,IT):-
   kommutativ(Op),
   alo_fmo(Op,Def,[Y,X],FE,IT).

kommutativ(Op):-Op::[+,*,...].
```

Die zweite Regel von $alo\_fmo\_kommutativ/4$ bewirkt, dass beim Aufruf die Domainvariablen der Operanden vertauscht werden. Die generalisierte Propagierung bewirkt, dass die entsprechenden Alternativen simultan generiert werden.

Handhabung nicht direkt umsetzbarer Operatoren der IR

Es kann vorkommen, dass Operatoren der IR kein unmittelbares Pendant auf dem Zielprozessor besitzen. So muss zum Beispiel die Addition auf Long-Integern durch Zerlegung der Argumente in Integer und durch eine Addition sowie einer Addition mit Übertrag auf Integern realisiert werden. Es bestehen die folgenden Möglichkeiten, diese Fälle in CoLIR zu handhaben:

- Man kann auf der IR-Ebene entsprechende Transformationen in die notwendigen Operatoren des Zielprozessors durchführen. Die Transformation kann entweder in der Substitution einer solchen abstrakten Operation durch eine entsprechende Sequenz abstrakter Operationen bestehen, die dann direkt umsetzbar sind. Denkbar ist aber auch der Einsatz von traditionellen baumbasierten Verfahren, um größere Kontexte der abstrakten Operationen zu berücksichtigen.

- Die nicht unmittelbar umsetzbaren Operatoren werden bei der Transformation in LIR-Operatoren beibehalten. Jede Lösung eines solchen FMOs repräsentiert dann eine bestimmte Sequenz von FMOs, dessen Ausführung auf dem Prozessor die abstrakte Operation realisiert. Mit einem der noch vorzustellenden Verfahren zur Instruktionsauswahl kann eine optimale Überdeckung eines Basisblocks ermittelt werden. Die FMOs einer solchen Überdeckung, die nicht unmittelbar umsetzbar sind, werden dann durch die korrespondierenden Sequenzen substituiert.

Welche der Methoden zum Einsatz kommen sollte, hängt davon ab, ob es bei der Transformation der nicht unmittelbar umsetzbaren Operatoren auf die Erkennung komplexer Muster ankommt, und ob es gegebenenfalls überlappende Maschinenoperationsmuster gibt, deren Auswahl die Codegüte drastisch beeinflussen kann. Ist dies nicht der Fall, sollten Transformationsregeln für einzelne abstrakte Operatoren verwendet werden, da dies einfacher zu realisieren ist.

Generierung der Constraints von $COVER(fun)$

Die Generierung des CSPs $COVER(fun)$ kann grundsätzlich über jeder FMO-reinen CoLIR-Funktion durchgeführt werden. Gegeben sei die Funktion fun eines FMO-reinen CoLIR-Programms. Für jeden Basisblock $b \in fun$ werden die folgenden Schritte durchgeführt:

- Jede FMO $f \in FMO(b)$ wird durch den Aufruf von $fmo/5$ mit den Constraints, die die Bildung legaler Maschinenoperationen gewährleisten, assoziiert.

- Die Domains der Eingangs- und Ausgangssetzungen von b werden mit den für den globalen Datenfluss zulässigen ST-Komponenten initialisiert.

- Es werden Constraints generiert, die die Existenz von Datentransferpfaden zwischen den datenflussabhängigen Setzungen und Benutzungen fordern. Dies wird durch Generierung des Constraints $fv_{[0]} \rightarrow^* fv'_{pos}$ für jede Kante $fv \rightarrow^{pos} fv' \in DFG(b)$ erreicht.

- Generierung der Constraints, die das Vorhandensein graphbasierter Maschinenoperationen prüfen. Hierzu müssen zur Zeit dedizierte Prädikate formuliert werden. Eine generische Schnittstelle, wie sie durch $fmo/5$ und $\rightarrow^* /2$ gegeben ist, liegt hier noch nicht vor.

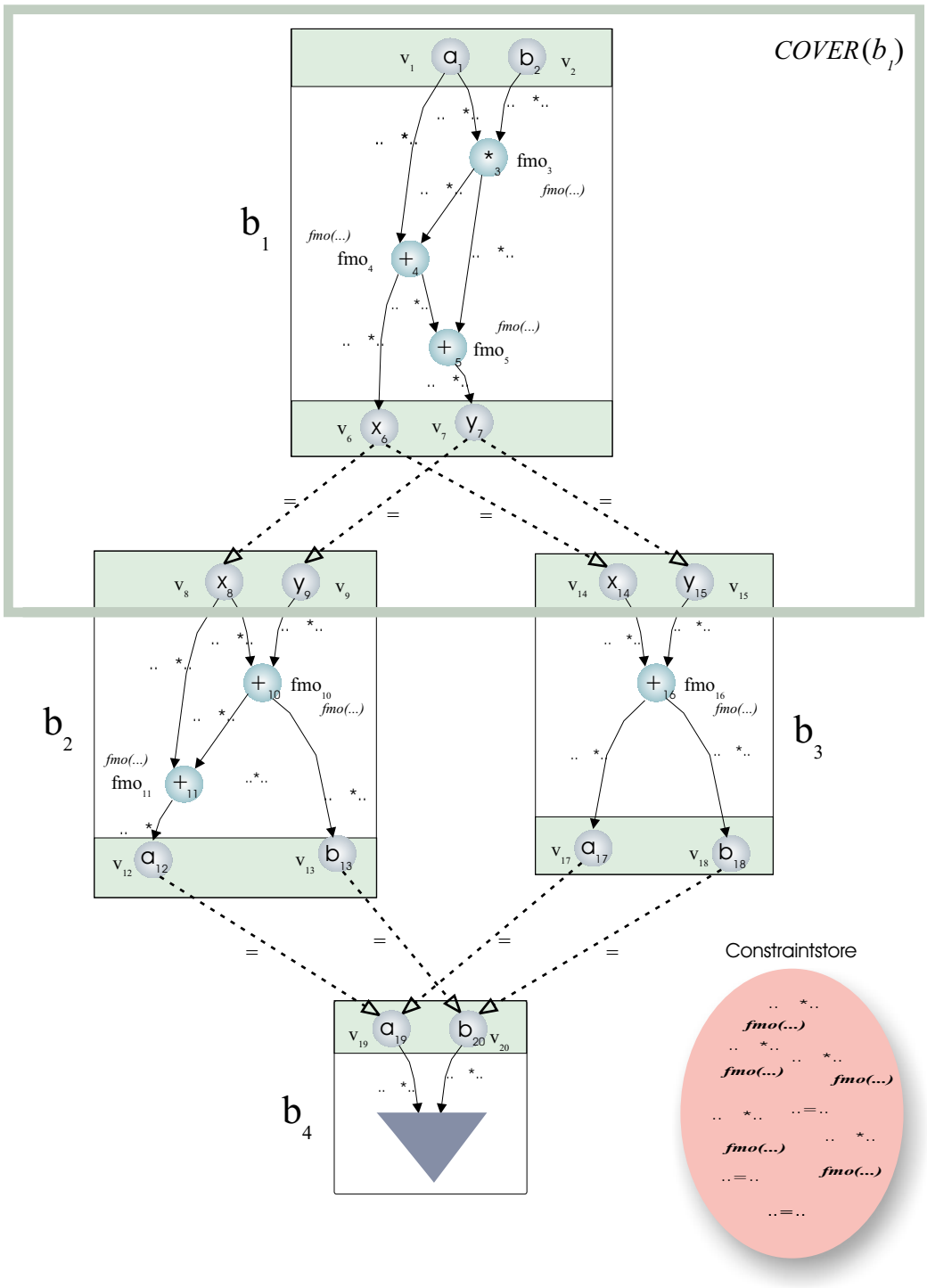


Abbildung 5.14.: Generierte Constraints von $COVER^{fmo^*}_{\rightarrow^*}$ zu einem Datenflussgraphen.

Es verbleibt die Sicherstellung des globalen Datenflusses durch Gleichsetzung aller Ausgangssetzungen mit den Eingangssetzungen gemäß der Kanten des globalen Datenflussgraphen $gDFG(fun)$. Dies wird durch Generierung des Constraints $v_{[0]} = v'_{[0]}$ für jede Kante $v \rightarrow v' \in gDFG(fun)$ erreicht. Somit wird garantiert, dass sich die Programmvariablen bei Verzweigung des Kontrollflusses und dessen Zusammenfließen in den gleichen ST-Komponenten befinden. Weiterhin werden noch die Constraints für die Konventionen erzeugt.

In Abb. 5.14 sind schemenhaft die generierten Constraints zu einer Funktion bestehend aus vier Basisblöcken ($b_1, ..., b_4$) dargestellt. Für die Knoten sind die korrespondierenden FMOs (fmo_i) und die Eingangs- und Ausgangsvariablen (v_i) gezeigt. Die Eingangs- und Ausgangsvariablen sind pro Basisblock nochmal extra hervorgehoben. Weiterhin sind die pro Knoten und Kante aufgerufenen Constraints $fmo/5$ (gegeben durch $fmo(...)$) und $\rightarrow\ /2$ (gegeben durch .. $\rightarrow^*$ ..) dargestellt worden. Die Kanten des lokalen Datenflusses sind durchgezogen und die des globalen Datenflusses sind gestrichelt dargestellt.

Die in dieser Arbeit noch vorgestellten Optimierungstechniken sind alle lokaler Natur, d.h., sie werden jeweils nur auf einen Basisblock angewandt. In diesem Kontext spielt die lokale Sichtweise auf ein CSP $COVER(fun)$ eine Rolle, um dem Begriff einer Lösung einen sinnvollen Bezug zu geben. Zu einer gegebenen Funktion fun und dessen CSP $COVER(fun)$ ist $COVER(b)$ das auf einen Basisblock $b \in fun$ reduzierte CSP. Dieses CSP umfasst alle zu b erzeugten Constraints $fmo(...)$ und .. $\rightarrow *$.. und alle Gleichheitsconstraints, die sich aus dem globalen Datenfluss ergeben, aber ausschließlich aus Eingangs- bzw. Ausgangssetzungen von b resultieren. In Abb. 5.14 sind die Constraints vom CSP $COVER(b_1)$ durch den grauen Rahmen abgegrenzt worden.

$COVER(b)$

5.5.2. Existenz von Lösungen zu $COVER(fun)$

Im Allgemeinen kann schon das Finden einer Lösung für $COVER(fun)$ exponenzielle Laufzeit benötigen, obwohl dies in der praktischen Anwendung von COCOON bislang niemals vorgekommen ist. Viel schwerwiegender gerade für irreguläre Prozessoren ist, dass zu einem Programm nicht einmal eine Lösung für das CSP $COVER(fun)$ bzw. für $COVER(b)$ existieren muss. Ein Grund ist, dass Registerinhalte nicht zwischen beliebigen Registerfiles hin und her bewegt und Operationen nicht auf beliebigen Funktionseinheiten ausgeführt werden können. Als Beispiel nehmen wir an, dass Operation *op* nur von Funktionseinheit fe ausgeführt werden kann und fe das Resultat von *op* nur in Registerfile rf schreiben kann. Es gibt weiterhin eine Funktionseinheit fe', die keinen Zugriff auf rf hat, und es existieren auch keine Transfermöglichkeiten um einen Wert aus rf in ein Registerfile zu bewegen, aus dem fe' lesen kann. Gibt es jetzt in einem Programm eine Operation *op*, die auf fe ausgeführt werden muss, und eine Operation *op'*, die auf fe' ausgeführt werden muss, und *op'* benutzt den Wert den *op* generiert, so gibt es keine Lösung, da kein Datentransferpfad zwischen der Setzung von *op* und der Benutzung von *op'* existiert.

In diesem Unterkapitel werden Kriterien angegeben, in welchem Fall es Lösungen für $COVER(fun)$ bzw. für $COVER(b)$ gibt (was aber eine exponenzielle Laufzeit bei der Suche nach einer Lösung nicht ausschließen kann). Die Kriterien erlauben es, die

Lösungssuche auf Basisblöcke zu beschränken und daraus auf die Existenz einer globalen Lösung zu schließen. Weiterhin geben die Kriterien die Möglichkeit, für ein CSP $COVER(fun)$ bzw. für $COVER(b)$, für das keine Lösung existiert, eine Variante zu konstruieren, die eine Lösung besitzt. Die Kriterien formulieren Bedingungen über den Eigenschaften des Befehlssatzes des Zielprozessors und des jeweilig betrachteten CoLIR-Programms. Zur Spezifikation der Kriterien werden jetzt zuerst *zusammenhängende sequentielle Komponenten* und *erreichbare Maschinenoperationen* eingeführt.

Menge zusammenhängender ST-Komponenten

Definition : Eine nichtleere Menge M von sequenziellen ST-Komponenten heißt *zusammenhängend, falls* zwischen zwei beliebigen Komponenten $sk_1, sk_2 \in M$ ein Datentransferpfad $sk_1 \rightarrow^* sk_2$ existiert[4].

erreichbare Maschinenoperation und erreichbare FMO

Definition: Gegeben sei eine zusammenhängende Menge M von sequenziellen ST-Komponenten. Eine bzgl. M *erreichbare Maschinenoperation* umfasst nur Setzungen und Benutzungen aus M[5]. Eine bzgl. M *erreichbare FMO* besitzt mindestens eine erreichbare Maschinenoperation als Lösung.

Kriterium I: Der Zielprozessor besitzt eine Menge M von zusammenhängenden sequenziellen ST-Komponenten und für jeden Operator des Zielprozessors gibt es mindestens eine bzgl. M erreichbare Maschinenoperation.

In diesem Fall können Operatoren einer IR-reinen bzw. FMO-reinen CoLIR-Funktion durch erreichbare Maschinenoperationen realisiert werden (sofern sich die abstrakten Operatoren auf Operatoren des Zielprozessors abbilden lassen, was man i.d.R. annehmen kann, auch wenn dies nicht immer unmittelbar möglich ist (s.o.)) und es ist gewährleistet, dass dann zwischen allen Setzungen und deren Benutzungen ein Datentransferpfad existiert. Es sei bemerkt, dass die hier betrachteten Prozessoren dieses Kriterium erfüllen.

Kriterium II: Die Existenz einer Lösung für ein CSP $COVER(fun)$ ist für eine Funktion fun unter gegebenem Kriterium I immer dann gewährleistet, wenn die folgenden Bedingungen erfüllt sind:

- Gegeben ist eine Menge M zusammenhängender sequenzieller Komponenten.

[4]Interpretiert man die ST-Komponenten eines Prozessors als Graphknoten eines Graphen G und die möglichen Datentransferoperationen als gerichtete Kanten von G, so sind die erreichbaren Mengen von sequenziellen Komponenten gerade die zusammenhängenden Teilgraphen von G. Allerdings werden hier auch die trivialen einelementigen Knotenmengen zugelassen.

[5]Für komplexe FMOs betrifft dies natürlich nur die nach außen hin sichtbaren Setzungen und Benutzungen. Für eine komplexe MAC-Operation R1:=R2+(T:=R3*R4) sind dies die Setzung R1 und die Benutzungen R2, R3 und R4, nicht aber die Setzungen von Sub-FMOs und die Benutzungen, an deren Positionen Sub-FMOs existieren.

- fun enthält nur bzgl. M erreichbare FMOs.
- Alle Eingangs- und Ausgangssetzungen der Basisblöcke von fun sind aus M .
- Für alle Kanten $v \rightarrow v' \in gDDG(fun)$ gilt $v_{[0]} = v'_{[0]}$.
- Die Constraints der Konventionen (siehe Kapitel 5.4) schließen nie alle erreichbaren Maschinenoperationen aus den Lösungsmengen einer FMO aus fun aus.

Unter den ersten vier Bedingungen von Kriterium II ist per Konstruktion klar, dass alle FMOs mindestens eine Lösung besitzen und weiterhin die Garantie dafür gegeben ist, dass auch die Existenz der Datentransferpfade gewährleistet ist. Die letzte Bedingung ist in der Regel nicht so einfach zu zeigen, da durch die Constraints der Konventionen beliebige wechselseitige Beziehungen zwischen den FMOs erzeugt werden können. Für die ADSP2100-Familie ist hier allerdings unter den folgenden Bedingungen gewährleistet, dass Lösungen existieren:

- Die Menge der Registerfiles der ALU-, MAC-, Shifter- und AGU-Einheiten sowie die Speicherbänke d und b bilden eine zusammenhängende Menge von sequenziellen ST-Komponenten, die nachfolgend mit M_{adsp} bezeichnet wird.
- Die Gleichheitsbeziehungen, die über den Speicherressourcen gegeben sind, sind potenziell einhaltbar, da beide Speicherbänke d und p in der Menge M_{adsp} vorkommen und es erreichbare LOAD- und STORE-Operation für den Zugriff auf beide Speicherbänke gibt. Somit ist die Existenz für LOAD- und STORE-Operationen immer dann gewährleistet, wenn pro Speicherbank mindestens eine entsprechende erreichbare Maschinenoperation in einer FMO der Klasse $load$ oder $store$ vorhanden ist.
- Die Bindungen der Eingangs- und Ausgangssetzungen entsprechen gerade der dritten Bedingung des Kriteriums II erfüllt, da dies gerade die erreichbaren Komponenten aus M_{adsp} sind.
- Das Binden der FE-Variablen an Funktionseinheiten aus $\{alu, mac, shifter\}$ verletzt die oben gestellten Bedingungen ebenfalls nicht, da hieraus nie alle erreichbaren Maschinenoperationen ausgeschlossen werden.

Unter Kriterium II können Lösungen pro Basisblock generiert werden, ohne Gefahr zu laufen, dass man durch die Lösung eines Basisblocks b die Eingangs- und Ausgangssetzungen der anderen Basisblöcke in der Art einschränkt, dass alle Lösungen eines anderen Basisblocks ausgeschlossen werden. Da auch die Eingangs- und Ausgangssetzungen der anderen Basisblöcke auf Setzungsalternativen aus M_{adsp} beschränkt sind und weiterhin auch noch Gleichheitsconstraints bestehen, können diesen beliebige Elemente aus M_{adsp} zugeordnet werden, ohne dass die Existenz von Datentransferpfaden verletzt wird. Wird eine Setzung oder eine Benutzung einer LOAD- oder STORE-Operation an d oder p gebunden, existiert dazu immer eine erreichbare Maschinenoperation.

Gibt es zu einem Basisblock b keine Lösung zu $COVER(b)$, kann unter Kriterium I Korrekturcode generiert werden, indem man die im Basisblock vorkommenden FMOs

benchmark	b	$\#n$	$\#e$	$\#cse$	t_{cover}
ru	b_1	11	9	0	0.3
cm	b_1	18	20	4	0.5
cu	b_1	35	36	4	1.2
bi1	b_1	30	31	3	1.0
co	b_1	8	4	0	0.2
	b_2	4	3	0	< 0.1
	b_3	19	16	2	0.6
	b_4	2	1	0	< 0.1
ruN	b_1	10	5	0	0.2
	b_2	4	3	0	< 0.1
	b_3	27	24	4	0.8
	b_4	2	1	0	< 0.1
cuN	b_1	10	5	0	0.2
	b_2	4	3	0	< 0.1
	b_3	56	60	12	1.9
	b_4	2	1	0	< 0.1
biN	b_1	12	7	0	0.3
	b_2	4	3	0	< 0.1
	b_3	51	56	10	1.7
	b_4	2	1	0	< 0.1
fir	b_1	10	5	0	0.4
	b_2	4	3	0	< 0.1
	b_3	25	22	3	0.8
	b_4	11	10	0	0.4

Tabelle 5.1.: Eckdaten der in der Arbeit betrachteten Benchmarks und Laufzeiten zur Generierung von $COVER^{fmo^*}_{\rightarrow^*}(b)$ (Spalte t_{cover}). Benchmarks: real update (ru), complex multiply (cm), complex update (cu), biquad one section (bi1), convolution (co), N real updates (ruN), N complex updates (cuN), biquad N sections (biN), fir filter (fir).

auf die erreichbaren Pendants abbildet. Dazu müssen gegebenenfalls komplexe FMOs wieder in ihre Bestandteile zerlegt und Maschineninstruktionen mit parallelisierten FMOs wieder in Maschineninstruktionen mit einzelnen FMOs transformiert werden. Die Verifikation, dass keine Lösung für $COVER(b)$ vorliegt, kann im Allgemeinen exponenziell viel Laufzeit benötigen. Um hier immer noch die Möglichkeit zu haben, Korrekturcode zu erzeugen, können Timeouts definiert werden, die angeben, wie lange maximal nach einer Lösung gesucht wird. Danach wird die Suche abgebrochen und angenommen, dass keine Lösung existiert.

5.5.3. Laufzeiten für die Generierung von $COVER(b)$

In der Arbeit werden die DSPStone-Benchmarks betrachtet [120]. Diese Sammlung von Benchmarks umfasst Kernfunktionen der digitalen Signalverarbeitung und hat sich als Standardbenchmarkset der digitalen Signalverarbeitung etabliert. In Tabelle 5.1 sind zu jedem Benchmark größenbezogene Eckdaten der einzelnen Basisblöcke mit Bezug auf die korrespondierenden Datenflussgraphen angegeben: Anzahl der Knoten (Spalte $\#n$), Kanten (Spalte $\#e$) und Anzahl der CSEs[6] (Spalte $\#cse$). In Spalte t_{cover} sind die Laufzeiten für die Generierung von $COVER(b)$ inklusive der Generierung der frischen FMOs gezeigt. Die Zeiten wurden auf einer UltraSparc10 gemessen und sind in CPU-Sekunden angegeben. Dabei wurde die Genauigkeit der Darstellung auf 0.1 Sekunden beschränkt, da diese für die weiteren Betrachtungen in dieser Arbeit hinreichend aussagekräftig sind. Laufzeiten, die kleiner als 0.1 CPU-Sekunden sind, werden durch '< 0.1' gekennzeichnet. Die Zeiten zur Generierung der CSPs sind pro Basisblock aufgezeigt. Die Generierungszeiten der zusätzlich notwendigen globalen Constraints für das gesamte CSP $COVER(fun)$ lagen hier jeweils im Bereich von < 0.1 CPU-Sekunden.

Die ersten vier Benchmarks in Tabelle 5.1 sind rein datenflussorientiert und bestehen jeweils aus nur einem Basisblock. Die weiteren Benchmarks umfassen auch den Kontrollfluss und haben jeweils die gleiche Struktur:

- b_1: Initialisierung von Speicherreferenzen und Schleifenprolog.
- b_2 : Abfrage der Schleifenbedingung.
- b_3 : Schleifenrumpf.
- b_4 : Epilog und Rücksprung aus der Funktion.

Von zentraler Bedeutung für die späteren Vergleiche ist jeweils der Schleifenrumpf von b_3. Daher wird in den nachfolgenden Kapiteln bei der Darstellung von Resultaten für die kontrollflussumfassenden Benchmarks nur noch jeweils b_3 aufgeführt.

5.5.4. Restriktivere Modelle von $COVER$

Zum CSP-Modell $COVER$ gibt es Erweiterungen, die jeweils strengere Anforderungen an die alternativen Überdeckungen stellen. Diese Restriktionen schränken die folgenden beiden Eigenschaften von $COVER$ ein:

1. Alle FMOs dürfen (müssen aber nicht) partielle Maschinenoperationen umfassen.
2. Die Existenz von Datentransferpfaden zwischen Setzungen und Benutzungen wird zwar gefordert, diese müssen aber nicht explizit dargestellt werden müssen.

[6] Zur Erinnerung: CSE=common sub-expression=gemeinsamer Teilausdruck. Im Datenflussgraph sind das gerade die Knoten, die mehr als eine ausgehende Kante besitzen.

Es gibt die folgenden beiden in dieser Arbeit verwendeten Erweiterungen von $COVER$:

- $COVER_{dp}$ fordert die explizite Darstellung von Datentransferpfaden, wobei auch dynamische Datentransferpfade möglich sind. Es gibt also keine Lösungen, in denen noch Datentransfer-Operationen eingefügt werden müssen. Anstelle der Generierung der Constraints $f_{[0]} \rightarrow^* f'_{pos}$ wird hier jeweils das Constraint $f_{[0]} = f'_{pos}$ generiert. $COVER_{dp}$
- $COVER_{ffp}$ fordert die explizite Darstellung von Datentransferpfaden, wobei aber die Länge der Datentransferpfade determiniert ist. D.h., dass keine dynamischen Anteile mehr in Datentransferpfaden und somit auch keine Dummy-COPY-Operationen mehr existieren. Zusätzlich zu $COVER_{dp}$ wird noch pro FMO f ein Constraint $FE_f \neq none$ erzeugt. Weiterhin kann keine FMO zu einer Sub-FMO werden, sofern sie das nicht schon ist. Pro FMO f wird dazu ein Constraint $f_{[0]} \neq \mathcal{T}$ generiert. Hierdurch kann f keiner Sub-FMO zugeordnet werden. $COVER_{ffp}$

Alternative Überdeckungen von $COVER_{dp}$ erlauben es, dass Datentransferpfade durch dynamische Datentransferpfade dargestellt werden, wodurch sich Freiheitsgrade bzgl. der Länge von Datentransferpfaden ergeben können. Man kann CoLIR-Programme, die dem CSP-Modell $COVER_{dp}$ genügen, aus CoLIR-Programmen, die $COVER$ genügen, erzeugen. Für reale Prozessoren kann man davon ausgehen, dass ein n existiert, welches die obere Schranke der notwendigen Datentransfers angibt, mit der man einen Wert von jedem Registerfile rf_1 einer Setzung in jedes von rf_1 erreichbare Registerfile rf_2 bewegen kann. Zwischen jeder Setzung und der korrespondierenden Benutzung fügt man eine Sequenz von n Datentransfer-Operationen ein. Analog kann man bei der Generierung von CoLIR-Programmen vorgehen, die dem CSP-Modell $COVER_{ffp}$ genügen sollen, falls zu einem CoLIR-Programm, das $COVER$ genügt, die Längen aller erforderlichen Datentransferpfade zwischen den Setzungen und ihren Benutzungen bekannt sind.

5.5.5. Vermeidung von Backtracking

Die eingeführten Modellierungskonzepte erlauben eine Definition von $fmo/5$, so dass das Finden jeder Lösung ohne Backtracking auskommt. Dies ist entweder möglich, wenn jede FMO durch eine einzige Regel beschrieben werden kann, oder eine Menge von Regeln, die eine FMO spezifizieren, durch die generalisierte Propagierung zusammengefasst wird. Dieser Aspekt hat größte Relevanz, da er Voraussetzung dafür ist, dass zu einer Menge von n FMOs durch das Constraint

$$fmo(Class_1, Op_1, RFs_1, FE_1, IT_1) \wedge \ldots \wedge fmo(Class_n, Op_n, RFs_n, FE_n, IT_n) \wedge cg(\ldots)$$

alle alternativen Überdeckungen simultan im Prädikat $cg/1$ zur Verfügung stehen. Dies ermöglicht erst die Generierung einer größtmöglichen betrachtbaren Menge alternativer Maschinenoperationen in einer Codegenerierungsphase. Die Modellierungskonzepte erlauben es, dass auch die Lösungen des Constraints $X \longrightarrow^* Y$ backtrackingfrei generierbar sind. Somit ist simultan die Menge aller alternativen Überdeckungen generierbar, deren Lösungen gültige Maschinenoperationen umfassen und die Existenz von Datentransferpfaden garantieren.

5.6. Zusammenfassung

Das CSP-Modell $COVER$ wurde als generische Grundlage zur Implementierung von Instruktionsauswahltechniken sowie zur Integration der Instruktionsauswahl in andere Codegenerierungsphasen entwickelt. Die Lösungen von $COVER$ spiegeln die folgenden Eigenschaften wider:

- Jede Lösung enthält nur legale Maschinenoperationen.
- Jede Maschinenoperation enthält nur Ressourcenzuordnungen gemäß des gewählten Instruktionstypen.
- Für die datenflussabhängigen Setzungen und Benutzungen ist die Existenz eines Datentransferpfades garantiert.

Die eingeführten Modellierungskonzepte von FMOs durch Regeln und unter Hinzunahme der generalisierten Propagierung erlauben eine intuitive und kompakte Spezifikation des Befehlssatzes eines Prozessors. Die Generierung der CSPs zu $COVER$ ist darüber hinaus vollkommen unabhängig von der Spezifikation der FMOs eines bestimmten Prozessors. Hierdurch ist die Basis zur Retargierung des CSP-Modells $COVER$ gegeben.

Das CSP-Modell $COVER$ integriert die folgenden Aufgaben der Instruktionsauswahl in einem einheitlichen, constraintbasierten Konzept:

- *Mustererkennung von Maschinenoperationen und Datentransferpfaden*:
  - Die Mustererkennung von Maschinenoperationen und die Darstellung alternativer Maschinenoperationsmuster findet auf der Basis der Domainvariablen und Constraints von FMOs statt. Dazu zählt auch die Erkennung und Darstellung komplexer und graphbasierter Maschinenoperationen, wobei simultan die Möglichkeit gegeben ist, komplexe Maschinenoperation neben ihrer Umsetzung durch Einzeloperationen darzustellen, was durch partielle FMOs möglich ist.
  - Domains von Setzungen und Benutzungen enthalten nur ST-Komponenten, für die mindestens ein Datentransferpfad von der Setzung zur korrespondierenden Benutzung existiert. Die explizite Darstellung von Datentransferpfaden ist durch eine Sequenz von n FMOs möglich, die alle alternativen Datentransferpfade zwischen einer Setzung und Benutzung repräsentiert, die eine maximale Länge von n haben.
- *Auswahl einer Überdeckung*: Die Auswahl einer Überdeckung mit legalen Maschinenoperationen und unter der Berücksichtigung der Existenz von Datentransferpfaden ist für ein CSP $COVER(fun)$ bzw. $COVER(b)$ schon nur durch das Labeling der Domainvariablen der FMOs aus fun bzw. b gegeben.

Weiterhin werden durch die Constraints der FMOs Wechselbeziehungen zwischen STK-, FE-Variablen und IT-Variablen hergestellt. Hier fließen neben den üblichen Aufgaben der Instruktionsauswahl auch Aufgaben der Instruktionsanordnung mit ein, die Ressourcen von FMOs gemäß der Bindung von Instruktionstypen bei der Parallelisierung und gemäß der Zuordnung zu Funktionseinheiten korrekt auszuwählen.

Die benutzerdefinierten Constraints können so formuliert werden, dass alle Lösungen zu $fmo(Class, Op, RF_o := [RF_1, ..., RF_n], FE, IT)$ ohne Backtracking generiert werden können. Für das Goal

$$fmo(..),\ \ldots,\ fmo(...),\ cg(...).$$

sind somit in $cg(...)$ alle Lösungen für die FMOs simultan verfügbar. Diese Eigenschaft ist äußerst relevant für die Effizienz der in dieser Arbeit beschriebenen Optimierungsverfahren!

$COVER$ umfasst kein Kostenmodell zur Bewertung einzelner Alternativen, sondern bietet eine generische Grundlage zur Adaption unterschiedlichster Kostenmodelle und zur Integration der Instruktionsauswahl in andere Phasen der Codegenerierung. Die Flexibilität des Ansatzes wurde innerhalb verschiedener Ansätze zur Instruktionsauswahl und zur Implementierung phasenintegrierter Techniken genutzt. So wird $COVER$ bei der Entwicklung von GIA (Kapitel 6) und der Techniken von I^3R (Kapitel 7) verwendet, ohne dass dazu das Modell modifiziert werden muss. Weitere Anwendungen sind in [11] und im Anhang A gegeben.

6. Graphbasierte Instruktionsauswahl - Das CSP-Modell GIA_{CSP}

In diesem Kapitel werden Techniken zur graphbasierten Instruktionsauswahl am Beispiel der ADSP2100-Familie betrachtet. Das Ziel ist, aus einer gegebenen Menge alternativer Überdeckungen bzgl. einer gegebenen Kostenfunktion die kostengünstigste Teilmenge (mit gleichen Kosten) auszuwählen. Das Resultat der Techniken ist eine Menge alternativer Überdeckungen, deren Lösungen dem CSP-Modell $COVER_{ffp}$ genügen[1]. Die beschriebenen Techniken bauen alle auf dem gleichen Grundmodell auf, bestehend aus $COVER$ und einem Kostenmodell, und werden unter dem Begriff GIA (GIA = graphbasierte Instruktionsauswahl) zusammengefasst. Die unterschiedlichen Techniken definieren sich durch die Anwendung unterschiedlicher Labelingstrategien (Suchstrategien) in der Optimierung. Gemäß der Verwendung einer Labelingstrategie l wird eine bestimmte Optimierungstechnik aus GIA mit GIA_l bezeichnet. GIA umfasst Techniken zur optimalen Instruktionsauswahl über Basisblöcken, und es wird untersucht, bis zu welchen Basisblockgrößen der Einsatz dieser optimalen Techniken noch realistisch ist. Für Basisblöcke, die diese Größen überschreiten, werden effizientere Techniken eingeführt, die zwar optimale Resultate nicht mehr garantieren, aber in Experimenten gezeigt haben, dass i.d.R. Resultate gleich oder sehr nahe beim Optimum generiert werden. Die Grundidee besteht in der Partitionierung eines Basisblocks in eine Menge kleinerer handhabbarer Teilblöcke. Diese Techniken werden mit GIA_{l+part} bezeichnet.

Die Instruktionsauswahl als separate Phase einzuführen, ist auch im Kontext der Phasenkopplung wohlbegründet:

- Die Techniken aus GIA generieren eine Menge optimaler Überdeckungen, die dem CSP $COVER_{ffp}$ genügen. Aus dieser Menge können die nachfolgenden Phasen bzgl. ihrer Kostenmaße die günstigste Alternative oder wiederum eine kostengünstigste Teilmenge auswählen. Für die ADSP2100-Familie verbleiben noch viele Freiheitsgrade in der Auswahl von Ressourcen, die genügend Flexibilität für nachfolgende Phasen bieten.

- Techniken aus GIA können genutzt werden, um abstrakte Operatoren der IR zunächst auf die auf dem Zielprozessor verfügbaren Operatoren abzubilden.

[1] D.h., dass bzgl. der aus der Optimierung von GIA erhaltenen Kosten komplexe FMOs gebildet und Datentransferpfade eingefügt werden. Es gibt danach keine FMOs mehr, denen partielle Maschinenoperationen zugeordnet werden können und die nicht Sub-FMO sind. Weiterhin müssen keine Datentransfer-Operationen mehr eingefügt werden.

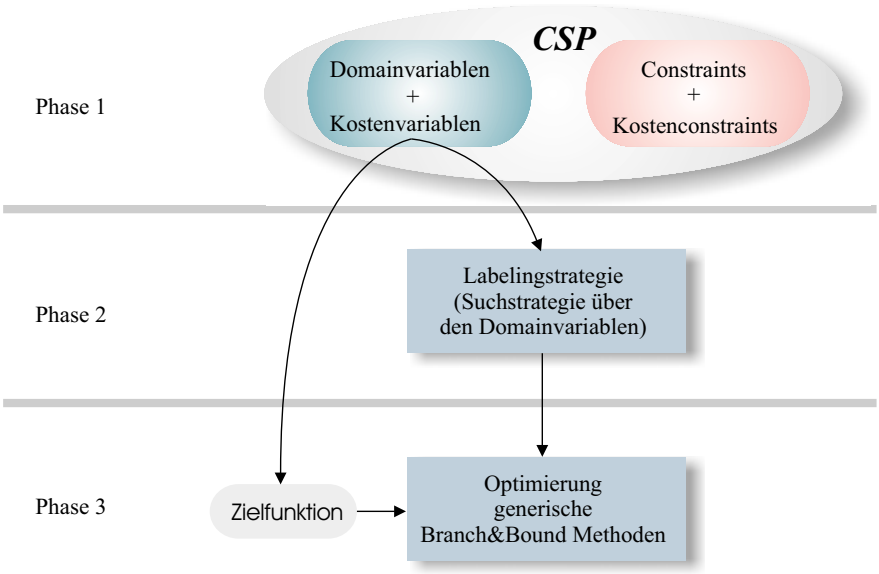


Abbildung 6.1.: Phasen in der Entwicklung von Lösungs- und Optimierungsverfahren in der Constraintlogik-Programmierung.

- Im Allgemeinen kann durch eine separate Instruktionsauswahl eine Menge alternativer Überdeckungen auf eine für nachfolgende Phasen sinnvolle Menge reduziert werden. Ein Beispiel dazu ist, die Anzahl der FMOs durch eine maximale Ausnutzung komplexer FMOs zu minimieren.

Um die weitere Vorgehensweise zu motivieren, wird das Konzept zur Entwicklung von Optimierungsverfahren unter ECLiPSe[2] noch einmal kurz skizziert (siehe auch Abb. 6.1 und Kapitel 2.4.4 Seite 49):

1. *Definition des Problems in Form eines CSPs*: Die Entitäten des Problems werden in Form von Domainvariablen spezifiziert. Weiterhin werden die wechselseitigen Abhängigkeiten zwischen den Entitäten in Form von Constraints formuliert. Ein Kostenmodell muss es ermöglichen, eine bestimmte Lösung des CSPs bewerten zu können. Umfasst das CSP-Modell noch keine Domainvariablen, über denen die Kostenfunktion definiert werden kann, muss das CSP-Modell zunächst mit Kosten in Beziehung gesetzt werden: Die Kosten werden selbst wieder durch Domainvariablen repräsentiert und mit den Domainvariablen des ursprünglichen CSP-Modells mittels neuer Constraints in Beziehung gesetzt. Dies muss nicht von Nachteil sein, da eine allgemeine Formulierung eines CSP-Modells die Möglichkeit bietet, unterschiedlichste Kostenmodelle zu adaptieren.

2. *Spezifikation einer Suchstrategie über den Domainvariablen des CSPs*: Es wird zunächst eine Labelingstrategie (Lösungsstrategie) zur reinen Lösungssuche entwickelt. Diese Strategie gibt vor, in welcher Reihenfolge Variablen während der

[2] Dies ist auch die allgemeine Vorgehensweise bei der Entwicklung von Lösungs- und Optimierungsverfahren in der Constraint-Logikprogrammierung.

Suche bearbeitet werden und in welcher Reihenfolge ihnen die möglichen Werte aus ihren Wertebereichen zugeordnet werden. Die Reihenfolge, in der Variablen gelabelt werden, hat großen Einfluss auf die Reaktivierung der Constraints und auf den Grad der Beschneidung des Suchraums. Dieser Schritt ist äußerst wichtig, da er einen sehr großen Einfluss auf die Laufzeit hat (sowohl bei der Lösungssuche als auch bei der Optimierung).

3. *Optimierung auf der Basis generischer Branch&Bound-Methoden*: ECLiPSe bietet in der FD-Bibliothek generische Optimierungsprozeduren - oder besser Optimierungsprädikate - auf der Basis der Branch&Bound-Suche. Diese erwarten als Argument eine Labelingstrategie $labeling(Vs)$ und eine Kostenfunktion $f(Vs)$ über den Domainvariablen Vs (siehe auch Kapitel 2.4.4). Der Aufruf hat die Form

$$minimize(labeling(Vs), f(Vs)).$$

Das Prinzip der Branch&Bound-Suche ist, zunächst auf der Basis der Labelingstrategie $labeling(Vs)$ irgendeine Lösung zu finden. Diese Lösung impliziert Kosten c. Es wird nun ein Constraint hinzugefügt, das eine Lösung mit geringeren Kosten fordert: $f(Vs) < c$. Die Labelingstrategie wird nicht erneut aufgerufen, sondern es wird implizit ein Backtrackingschritt durchgeführt, der bewirkt, dass die Suche an der nachfolgenden Stelle im Suchbaum fortgesetzt wird, an der die letzte beste Lösung gefunden wurde (dies entspricht dem Algorithmus `back_int_opt` aus Abb. 2.9 Seite 40). Die Lösungssuche wird solange fortgesetzt, bis nachweislich keine bessere Lösung mehr gefunden werden kann.

Das CSP-Modell zur Instruktionsauswahl liegt in Form von $COVER$ vor. Die weiteren notwendigen Schritte bestehen nun in der Entwicklung eines Kostenmodells, dessen Assoziation mit $COVER$ sowie der Entwicklung von Labelingstrategien, die ein schnelles Finden der optimalen Lösung ermöglichen.

Der weitere Inhalt des Kapitels gliedert sich wie folgt:

- In Kapitel 6.1 wird zunächst ein Überblick über das gesamte Konzept und der Teilphasen von GIA gegeben.
- Die Definition des CSP-Modells GIA_{CSP}, dem die Techniken aus GIA unterliegen, ist Bestandteil von Kapitel 6.2. Dieses CSP-Modell besteht aus den Teil-CSP-Modellen $COVER$ und dem Kostenmodell GIA_{cost}.
- In Kapitel 6.3 wird die Entwicklung exakter Labelingstrategien aus GIA_l und aus GIA_{l+part} beschrieben, wobei letztere die Handhabung grosser Basisblöcke in noch akzeptablen Laufzeiten erlauben, aber das Optimum nicht garantiert finden müssen.
- Kapitel 6.4 umfasst Kriterien, unter welchen Bedingungen noch auf die Existenz von Lösungen geschlossen werden kann.
- Das Kapitel wird abgeschlossen mit einer Zusammenfassung und einem Überblick der wesentlichen Eigenschaften des Ansatzes.

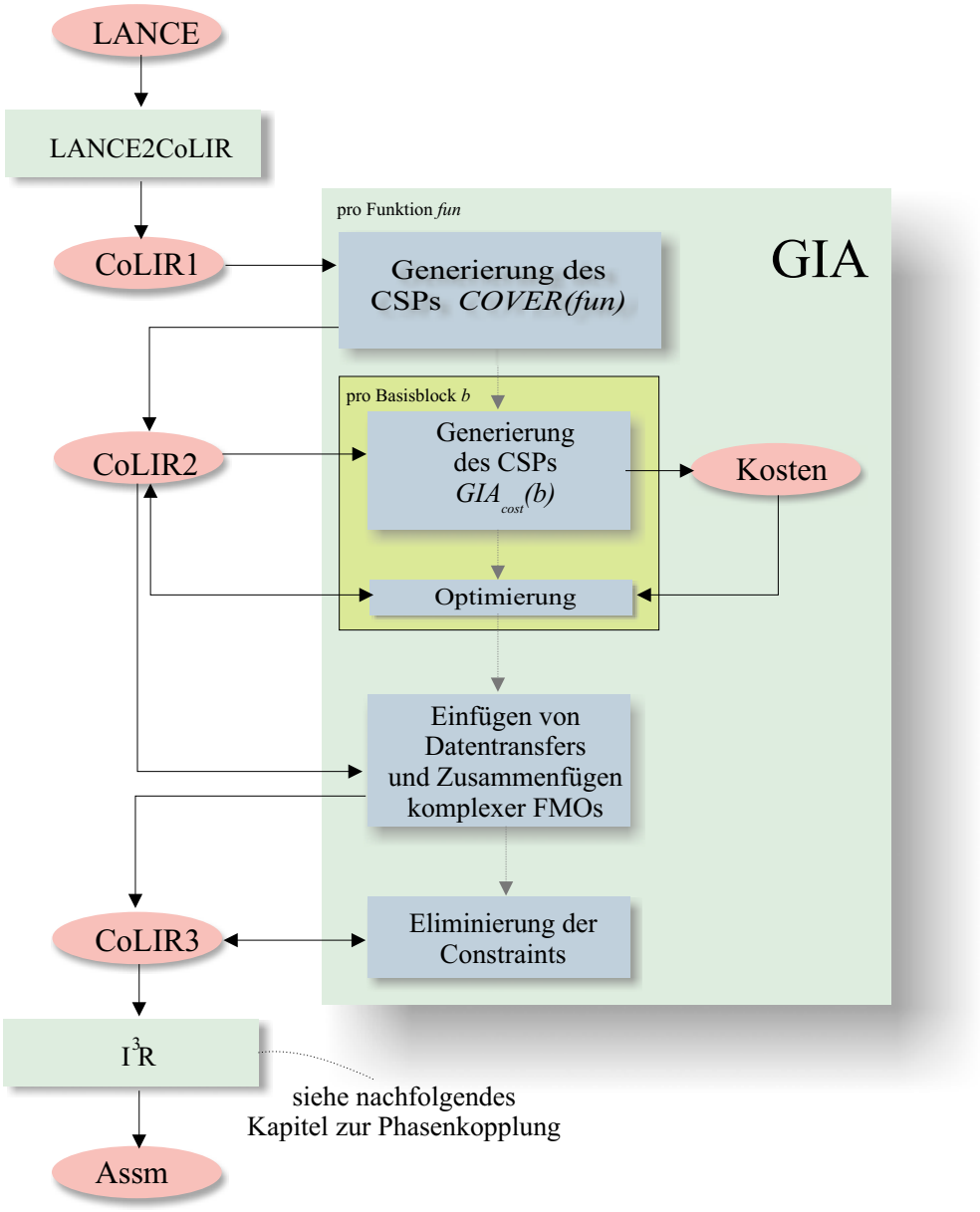


Abbildung 6.2.: Graphbasierte Instruktionsauswahl.

6.1. Überblick des Verfahrens

In Abb. 6.2 sind die Teilphasen und der Datenfluss von GIA gezeigt:

1. Aus der LANCE-IR wird zunächst ein IR-reines CoLIR-Programm CoLIR1 generiert.

2. Zu CoLIR1 werden die alternativen Überdeckungen generiert, die dem CSP-Modell $COVER$ genügen und durch das CoLIR-Programm CoLIR2 repräsentiert werden.

3. Die Optimierung von GIA wird basisblockweise durchgeführt:

   a) Mit jedem Basisblock von CoLIR2 werden Kosten assoziiert. Dazu werden neue Domainvariablen eingeführt und Constraints, die die Beziehung zwischen den Domainvariablen der alternativen Überdeckungen und den Kostenvariablen herstellen (siehe Kapitel 6.2).

   b) Die nächste Teilphase besteht aus der eigentlichen optimierenden Instruktionsauswahl, die auf der beschriebenen Branch&Bound-Methode basiert. Das Resultat der Optimierung spiegelt sich in CoLIR2 wider, in dem die Wertebereiche der Domainvariablen gemäß der besten gefundenen Lösung eingeschränkt worden sind (Kapitel 6.3).

4. Die Menge der alternativen besten Lösungen legt eine Auswahl von komplexen FMOs sowie die Auswahl bestimmter Datentransferpfade fest. In CoLIR2 werden daher entsprechende FMOs zu komplexen FMOs zusammengefasst. Zwischen einer bestimmten Setzung und Benutzung werden gemäß der Transferkosten eine entsprechende Anzahl von Datentransfer-FMOs eingefügt. Wie im Kapitel 5.2 gezeigt, wird hierdurch die Menge der alternativen Datentransferpfade zwischen der Setzung und der Benutzung repräsentiert. Das Einfügen der Datentransferpfade umfasst die Generierung neuer Programmvariablen, die derart erzeugt werden, dass die lokale SSA und der globale Datenfluss erhalten bleiben. Die Ausgabe dieser Teilphase ist ein CoLIR-Programm CoLIR3, das dem CSP-Modell $COVER_{ffp}$ genügt.

5. Im letzten Schritt werden die Constraints, die mit CoLIR3 assoziiert sind, eliminiert. Ziel ist, die durch das Kostenmodell zusätzlich generierten Constraints nicht in die weiteren Phasen zu propagieren. Das Kostenmodell setzt z.B. Setzungen und deren Benutzung mit Transferkosten in Beziehung. Wurden diese im Verlauf der Optimierung zu Null evaluiert, impliziert dies ein Gleichheitsconstraint zwischen der Setzung und der Benutzung, so dass diese immer nur an gleiche ST-Komponenten gebunden werden können. Um hier die Möglichkeit zu haben, evtl. noch Datentransfers einzufügen und unterschiedliche ST-Komponenten zuordnen zu können, werden die Constraints entkoppelt (siehe auch Kapitel 4.5).

GIA ist nicht auf die Instruktionsauswahl für IR-reine CoLIR-Programme beschränkt, sondern kann potenziell an beliebigen Stellen in der Codegenerierung integriert werden. Dies kann sinnvoll sein, um in späteren Phasen für bereits generierten Code eine erneute Instruktionsauswahl zu treffen, wenn dadurch z.B. komplexe oder graphbasierte Maschinenoperationen generiert werden können, die erst durch den neu hinzugefügten Code einer Teilphase präsent werden.

6.2. Das CSP-Modell GIA_{CSP}

Die Techniken aus GIA werden lokal auf jeweils einen Basisblock angewandt. Daher wird auch das CSP-Modell von GIA lokal für jeden Basisblock definiert und mit GIA_{CSP} bezeichnet. GIA_{CSP} umfasst $COVER$ als Teil-CSP-Modell und beinhaltet zusätzlich ein Kostenmodell, das ebenfalls in Form eines Teil-CSP-Modells GIA_{cost} spezifiziert wird (s.u.). Da nur lokale Techniken betrachtet werden, wird das CSP-Modell $COVER$ auf den jeweils gegebenen Basisblock b eingeschränkt. Das zu GIA_{CSP} zugehörige CSP eines Basisblocks b - bezeichnet durch $GIA_{CSP}(b)$ - setzt sich pro Basisblock b jeweils aus zwei Teil-CSPs zusammen:

$$GIA_{CSP}(b) = COVER(b) \cup GIA_{cost}(b)$$

Es wird zunächst das Kostenmodell GIA_{cost} spezifiziert und dann die Generierung des CSPs $GIA_{CSP}(b)$ im Detail beschrieben.

6.2.1. Das CSP-Modell GIA_{cost}

Als Kostenfunktion der graphbasierten Instruktionsauswahl wird die Summe der notwendigen Instruktionszyklen unter einer rein sequenziellen Ausführung der Maschinenoperationen eines Basisblocks betrachtet. Dies entspricht dem Kostenmodell der traditionellen baumbasierten Verfahren. Die hier vorgestellten Techniken können leicht auf komplexere Kostenmodelle erweitert werden, die z.B. auch Abschätzungen bzgl. der Parallelität umfassen. Hierauf wurde für die ADSP2100-Familie verzichtet, da die Techniken aus GIA durch die Beibehaltung einer Menge optimaler alternativer Überdeckungen genügend Freiheitsgrade in der Auswahl der Ressourcen erhalten. Dadurch ergeben sich so gut wie keine Beschränkungen für die nachfolgenden Phasen, die durch vorzeitige Bindung von Ressourcen bedingt sind (siehe auch Resultate der nachgeschalteten Technik I^3R, zur integrierten Instruktionsauswahl, Instruktionsanordnung und Registerallokation - Kapitel 7). So bleiben für die spätere Instruktionsanordnung genügend Freiheitsgrade bzgl. der Ressourcenauswahl, um den Ressourcenrestriktionen der Parallelausführung genügen zu können.

Die graphbasierte Instruktionsauswahl wird jeweils für einen Basisblock durchgeführt. Für eine Lösung der alternativen Überdeckungen eines Basisblocks b mit lokalem Datenflussgraphen $DFG(b)$ wird folgende Kostenfunktion definiert:

$$cost(b) = \sum_{fv \in FV(b)} fmo\_cost(fv) + transfer\_cost(fv, DFG(b))$$

mit

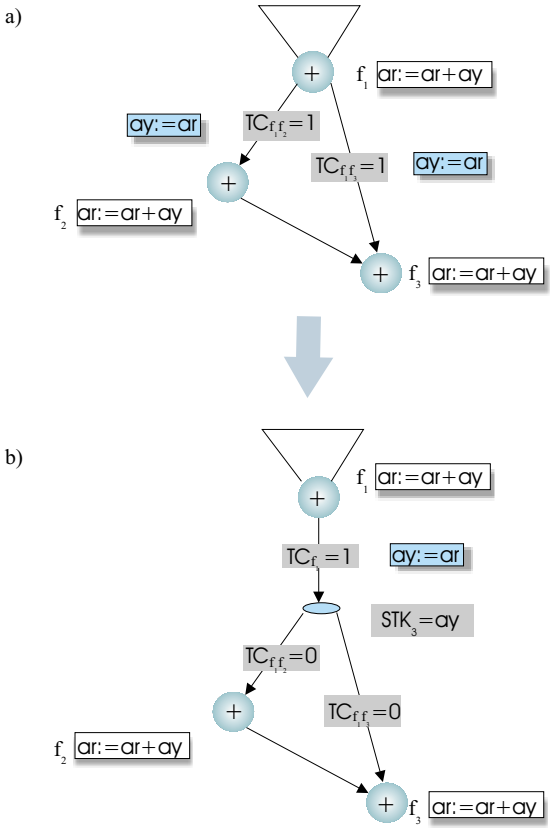


Abbildung 6.3.: Datentransferkosten von CSEs.

- $fmo\_cost(fv)$ ist für $fv \in FMO(b)$ die Anzahl der Instruktionszyklen, die zur Ausführung einer FMO fv notwendig sind. Für $fv \in VAR_{in}(b) \cup VAR_{out}(b)$ ist $fmo\_cost(fv)$ immer gleich Null.
- $transfer\_cost(fv, DFG)$ ist die Anzahl der notwendigen Instruktionszyklen zur Ausführung der Datentransfer-Operationen, um einen Wert von der Setzung von fv zu deren Benutzungen zu bewegen:

$$transfer\_cost(fv, DFG(b)) = \sum_{fv \rightarrow^{pos} fv' \in DFG(b)} tc(fv_{[0]}, fv'_{pos})$$

  $tc(ST_1, ST_2)$ ist die Anzahl der minimal notwendigen Instruktionszyklen, um den Inhalt von einer ST-Komponenten ST_1 in eine ST-Komponente ST_2 zu übertragen. $tc(ST_1, ST_2)$ gibt also jeweils die Kosten des billigsten Pfades zwischen einer Setzung und Benutzung an.

Die Kosten der Maschinenoperationen werden über FMOs gebildet. Da eine FMO eine Menge von Maschinenoperationen darstellen und diese Menge auch entsprechend mit alternativen Kosten assoziiert sein kann, ist es naheliegend, die Kosten einer FMO f durch eine Domainvariable C_f zu repräsentieren, dessen Wertebereich $D(C_f)$ die möglichen alternativen Kosten einer FMO umfaßt. Die Beziehung zwischen C_f und den Domainvariablen von f werden durch Constraints hergestellt (diese werden weiter unten beschrieben). Jede Kante $fv \rightarrow^{pos} fv' \in DFG(b)$ wird mit einer Domainvariablen $TC_{fv,fv'}$ assoziiert, die die Transferkosten $tc(fv_{[0]}, fv'_{pos})$ reflektiert. Die neu eingeführten Variablen des Kostenmodells werden im Folgenden auch als *Kostenvariablen* bezeichnet.

Kostenvariablen

In Abb. 6.3 a) ist ein Teil eines Datenflussgraphen mit den FMOs f_1, f_2 und f_3 gezeigt. f_1 ist eine CSE, deren Wert von f_2 und f_3 benutzt wird. Die Transferkosten TC_{f_1,f_2} und TC_{f_1,f_3}sind jeweils gleich 1, da jeweils ein Datentransfer von der Setzung ar zur Benutzung ay (zweiter Operand von f_2 und f_3) notwendig ist. Allerdings wird hier der Datentransfer $ay := ar$ zweimal durchgeführt, was in diesem Fall nicht notwendig ist.

Um redundante Datentransfers während der Optimierung berücksichtigen zu können, wird das Kostenmodell für die Transferkosten von CSEs verfeinert. Jede CSE $fv \in CSE(b)$ wird mit einer extra Domainvariablen STK_{fv} assoziiert, mit $D(STK_{fv}) \subseteq \mathcal{ST}$. Die Domainvariable STK_{fv} ermöglicht, dass das Zwischenspeichern des Wertes einer CSE in einer bestimmten Menge von ST-Komponenten spezifiziert werden kann. Die Transferkosten für CSEs werden nun wie folgt definiert:

$$transfer\_cost(fv, DFG(b)) = tc(fv_{[0]}, STK_{fv}) + \sum_{fv \rightarrow^{pos} fv' \in DFG(b)} tc(STK_{fv}, fv'_{pos})$$

Während der Optimierung wird durch dieses Kostenmodell erreicht, dass gemeinsame Präfixe der Datentransferpfade zu unterschiedlichen Benutzungen einer CSE zusammengefasst werden. Zur Umsetzung dieses Kostenmodells wird jede CSE fv mit einer weiteren Kostenvariablen TC_{fv} assoziiert, die die Transferkosten von der Setzung von fv zur ST-Komponenten STK_{fv} widerspiegelt. In Abb. 6.3 b) ist die modifizierte Variante des Modells gezeigt. CSE f_1 wird mit der zusätzlichen Kostenvariablen

```
fmo_cost(FMO,C) :-
    FMO=fmo with [rfs:Def:=_],
    fmo_cost1(Def,C) infers fd.

fmo_cost1(D,0) :- fk(D).          Regel 1
fmo_cost1(D,1) :- sk(D).          Regel 2


transfer_cost(FMO1,FMO2,Pos,TC):-
   pos(FMO1,[0],Def),
   pos(FMO2,Pos,Use),
   -->*(Def,Use,TC) infers fd.


-->*(X,Y,1):-                     Regel 1
   X##af,Y##af,
   sk(X),
   sk(Y).
-->*(X,af,1):-                    Regel 2
   X::[ar,ax,ay,sr,mr].
-->*(X,af,2):-                    Regel 3
   X::[mx,my,i1,i2,m1,m2,m,p].
-->*(af,ar,1).                    Regel 4
-->*(af,Y,2):-                    Regel 5
   sk(Y),
   Y##ar.
-->*(X,X,0):-fk(X).               Regel 6

fk(D):-D::['*(X,Y)',const].
sk(D):-D::[ar,ax,ay,...,p,d].
```

Abbildung 6.4.: Constraints des Kostenmodells.

TC_{f_1} assoziiert. Diesmal wird der Datentransfer $ay := ar$ der Kostenvariablen TC_{f_3} zugeordnet und somit nur ein Datentransfer erzeugt.

Eine FMO $f \in FMO(b)$ und C_f werden durch ein Constraint $fmo\_cost(f, C_f)$ miteinander in Beziehung gesetzt. Transferkosten $TC_{fv,fv'}$ einer Kante $fv \rightarrow^{pos} fv' \in DFG(b)$ werden durch das Constraint $transfer\_cost(fv, fv', pos, TC_{fv,fv'})$ in Relation gesetzt. Die Definition dieser Constraints für die ADSP2100-Familie ist in Abb. 6.4 gezeigt. Beide Constraints machen Gebrauch von der generalisierten Propagierung, so dass die Kostenconstraints sauber durch eine Menge von Regeln spezifiziert werden können. $fmo\_cost/2$ benutzt in diesem Sinne das Prädikat $fmo\_cost1/2$ und $transfer\_cost/4$ verwendet $\rightarrow *$/3. Die Regeln dieser Prädikate werden im Folgenden kurz erläutert:

- $fmo\_cost1/2$: Auf der ADSP2100-Familie können alle Maschinenoperationen innerhalb eines Instruktionszyklus abgearbeitet werden (sofern Daten in Registern

oder im On-Chip-Speicher liegen, was für die Betrachtungen dieser Arbeit ausreichend ist). Sub-FMOs komplexer Maschinenoperationen werden Kosten 0 zugeordnet. Die Unterscheidung, ob eine FMO oder eine Sub-FMO vorliegt, kann anhand der Setzung bestimmt werden. Die Setzung einer Sub-FMO ist immer einer flüchtigen Komponente zugeordnet und induziert Kosten 0 für die FMO. Dies wird durch Regel 1 von $fmo\_cost1/2$ in Abb. 6.4 umgesetzt, die mittels Prädikat $fk/1$ prüft, ob die Setzung D eine flüchtige Komponente ist. Ist die Setzung eine sequenzielle Komponente, hat die FMO Kosten 1. Regel 2 von $fmo\_cost1/2$ prüft mittels Prädikat $sk/1$ ob es sich bei der Setzung D um eine sequenzielle Komponente handelt.

- $\rightarrow\ */3$: Die erste Regel des Prädikates $\rightarrow^*\ /3$ in Abb. 6.4 besagt, dass alle Datentransfers zwischen sequenziellen Komponenten, die ungleich Registerfile af sind, durch eine COPY-Operation möglich sind. Die Regeln 2,3,4 und 5 stellen die besondere Bedeutung des Registerfiles af heraus. Ein unmittelbarer Datentransfer nach af ist nur von den Registern ar, ax, ay, mr und sr möglich (Regel 2). Alle weiteren Registerfiles benötigen 2 Datentransfers (Regel 3). Von af aus kann nur nach ar ein unmittelbarer Datentransfer erfolgen (Regel 4). Es sind zwei Datentransfers notwendig, um den Wert aus af in eine beliebige andere sequenzielle Komponente ungleich ar zu kopieren (Regel 5). Wird eine Setzung einer flüchtigen Komponente zugeordnet, so muss dies auch für die korrespondierende Benutzung geschehen, und es ergeben sich Transferkosten 0 (Regel 6).

Es ist zu bemerken, dass das Kostenmodell der ADSP2100-Familie allein durch Beziehungen zwischen Kostenvariablen und STK-Variablen spezifiziert ist. Dies impliziert, dass die Kostenvariablen allein durch eine partielle Variablenbelegung $\theta_{STK(b)}$ determiniert werden.

6.2.2. Generierung des CSPs $GIA_{CSP}(b)$

Die Optimierungstechniken von GIA werden basisblockweise ausgeführt. In unserer Implementierung wird zu einer Funktion fun zunächst das CSP $COVER(fun)$ (für die ganze Funktion fun) generiert. Die weiterhin notwendigen Constraints werden jeweils lokal generiert, so dass jeweils pro Basisblock $b \in fun$ noch das CSP $GIA_{cost}(b)$ erzeugt werden muss. Wie schon oben erwähnt, werden pro Basisblock b die Lösungen bzgl. der lokalen Basisblocksichtweise von $COVER(fun)$ - also zu $COVER(b)$ - erzeugt. Im Folgenden wird noch einmal das CSP-Modell $GIA_{cost}(b)$ zusammenfassend beschrieben.

Die Domainvariablen von $GIA_{cost}(b)$

Jedes $fv \in FV(b)$ eines Basisblocks b wird mit einer Domainvariablen C_{fv} und jede CSE $fv \in CSE(b)$ zusätzlich mit den Domainvariablen TC_{fv} und STK_{fv} assoziiert. Jeder Kante $fv \rightarrow^{pos} fv' \in DFG(b)$ wird eine Domainvariable $TC_{fv,fv'}$ zugeordnet. Die Domainvariablen von $GIA_{cost}(b)$ bestehen somit aus der Menge der zusätzlichen

	b	$\#dfg$	t_{cost}
ru	b_1	11/9/0	0.2
cm	b_1	18/20/4	0.5
cu	b_1	35/36/4	0.9
bi1	b_1	30/31/3	0.8
co	b_3	19/16/2	0.5
ruN	b_3	27/24/4	0.5
cuN	b_3	56/60/12	1.7
biN	b_3	51/56/10	1.6
fir	b_3	25/22/3	0.6

Tabelle 6.1.: Laufzeiten zur Generierung der Kostenconstraints (t_{cost}) in CPU-Sekunden. In Spalte $\#dfg$ sind die Eckdaten #Knoten/#Kanten/#CSEs der Benchmarks wiedergegeben.

ST-Komponenten, die im Folgenden durch die Menge

$$STK_{cse}(b) = \{STK_{fv} | \forall fv \in CSE(b)\}$$

repräsentiert werden, sowie der Menge der Kostenvariablen, die im Folgenden durch die Menge $Cost(b)$ mit

$$Cost(b) = \{C_f | f \in FV(b)\} \cup \{TC_f | f \in CSE(b)\} \cup \{TC_{fv,fv'} | fv \rightarrow^{pos} fv' \in DFG(b)\}$$

gegeben ist.

Die Constraints von $GIA_{cost}(b)$

Für jedes $fv \in FV(b)$ werden die folgenden Constraints generiert:

- Für $fv \in VAR_{in}(b)$ wird ein Constraint $C_{fv} = 0$ generiert.
- Für $fv \in FMO(b)$ wird ein Constraint $fmo\_cost(fv, C_{fv})$ generiert.
- Für $fv \in CSE(b)$ wird das Constraint $\rightarrow *(fv_{[0]}, STK_{fv}, TC_{fv})\ infers\ fd$ generiert.
- Für jede Kostenvariable $TC_{fv,fv'}$ mit $fv \in CSE(b)$ und Kante $fv \rightarrow^{pos} fv' \in DFG(b)$ wird ein Constraint $transfer\_cost(STK_{fv}, fv', pos, TC_{fv,fv'})$ generiert.
- Für jede Kostenvariable $TC_{fv,fv'}$ mit $fv \notin CSE(b)$ und Kante $fv \rightarrow^{pos} fv' \in DFG(b)$ wird ein Constraint $transfer\_cost(fv, fv', pos, TC_{fv,fv'})$ generiert.

Die Laufzeiten zur Generierung der Kostenconstraints für die DSPStone-Benchmarks sind in Tabelle 6.1 gezeigt. Für Benchmarks, die aus mehreren Basisblöcken bestehen, sind nur noch die Laufzeiten der zentralen Schleife gezeigt. Die Laufzeiten der nicht gezeigten Basisblöcke waren jeweils kleiner 0.1 CPU-Sekunden.

	$\|Cost(b)\|$	$\|Cost_\Delta(b)\|$
ru	20	11
cm	42	28
cu	78	40
bi1	64	39
co	37	20
ruN	68	30
cuN	128	77
biN	118	72
fir	60	36

Tabelle 6.2.: Anzahl der Kostenvariablen in $Cost(b)$ und Anzahl der noch nicht instanziierten Domainvariablen gegeben durch $Cost_\Delta(b)$.

Die Kostenfunktion von $GIA_{cost}(b)$

Die Kostenfunktion läßt sich wie folgt definieren:

$$C_{total} = \sum_{C \in Cost(b)} C$$

Die Kostenfunktion wird als Constraint formuliert, in dem die Summe über $Cost(b)$ einer Domainvariablen C_{total} zugeordnet wird. Diese Domainvariable wird dem Prädikat $minimize/2$ als zweites Argument übergeben und dessen Wert minimiert.

Nach der Generierung der Kostenconstraints ist ein Teil der Kostenvariablen festen Werten zugeordnet. Die Menge $Cost(b)$ wird entsprechend in zwei disjunkte Mengen $Cost_{fix}(b)$ (gebundene Domainvariablen) und $Cost_\Delta(b)$ (ungebundene Domainvariablen) aufgeteilt, so dass $Cost(b) = Cost_{fix}(b) \cup Cost_\Delta(b)$. In Tabelle 6.2 ist die Größe der Menge $Cost(b)$ und $Cost_\Delta(b)$ für die DSPStone-Benchmarks gezeigt.

Die Kostenfunktion eines Basisblocks läßt sich wie folgt darstellen:

$$C_{total} = \sum_{c \in Cost_{fix}(b)} c + \sum_{V \in Cost_\Delta(b)} V$$

Dabei ist $\sum_{c \in Cost_{fix}(b)} c$ ein konstanter Anteil, der im Folgenden mit c_{fix} bezeichnet wird. Es ist offensichtlich, dass das Optimum nur noch von $Cost_\Delta(b)$ abhängt. Als Kostenfunktion für die Optimierung ist somit die Funktion $C_\Delta = \sum_{V \in Cost_\Delta(b)} V$ ausreichend, in denen nun der Wert von C_Δ minimiert wird. Im Folgenden wird C_Δ auch als das Δ-Optimum bezeichnet.

6.3. Entwicklung von Labelingstrategien (GIA_l)

Ziel der Techniken aus GIA_l ist, zu einem Basisblock b einer Funktion fun eine Lösung zum CSP $GIA_{CSP}(b)$ zu finden, die bzgl. des eingeführten Kostenmodells minimal ist. Dieses Unterkapitel umfasst die Entwicklung von Labelingstrategien, die

```
gia(BB) :-
    gia_cost(BB,Cs,CSEs),
    cover_vars(BB,STKs,FEs,ITs),
    sumList(Cs,C),
    minimize(l(Cs,CSEs,STKs,FEs,ITs),C).
```

Abbildung 6.5.: Schema des Optimierungsprädikats $gia/1$.

das Labeling über den Domainvariablen aus $GIA_{CSP}(b)$, gegeben durch die Menge $DVs_{GIA}(b) = STK(b) \cup FE(b) \cup IT(b) \cup STK_{cse}(b) \cup Cost_{\Delta}(b)$, durchführen. Darin ist $STK(b) \cup FE(b) \cup IT(b)$ gerade die Menge der Domainvariablen aus $COVER^{fmo^*}_{\rightarrow^*}(b)$ und $STK_{cse}(b) \cup Cost_{\Delta}(b)$ die Menge der Domainvariablen aus $GIA_{cost}(b)$. Die Variablenbelegungen θ sind in diesem Kapitel genau über der Menge $DVs_{GIA}(b)$ definiert.

Das Optimierungsprädikat zu GIA_l ist in Abb. 6.5 dargestellt. Die darin vorkommenden Prädikate haben die folgende Bedeutung:

- *gia_costs*(b, Cs, $CSEs$) generiert zu einem Basisblock b lokal die Constraints des Kostenmodells und liefert die Kostenvariablen aus $Costs_{\Delta}(b)$ in Form einer Liste Cs und die Domainvariablen der Menge $STK_{cse}(b)$ in Form der Liste $CSEs$.
- *cover_vars*(*b*, *STKs*, *FEs*, *ITs*) liefert zum Basisblock b die Menge $STK(b)$, $FE(b)$ und $ITs(b)$ in Form der Listen $STKs$, FEs und ITs.
- *sumList*(Cs, C) generiert zu den Kostenvariablen $Cs = [C_1, ..., C_n]$ das Constraint $C = C_1 + ... + C_n$.
- *minimize(l(Cs, CSEs, STKs, FEs, ITs),C)* führt die Branch&Bound-Suche über einer Labeling-Strategie *l(Cs, CSEs, STKs, FEs, ITs)* durch, mit dem Ziel, den Wert der Domainvariable $Cdelta$ zu minimieren.

Die Labelingstrategie $l/5$ definiert die Suche über den Mengen $STK(b)$, $FE(b)$, $IT(b)$, $C(b)$ und $STK_{cse}(b)$. Das ECLiPSe System bietet vordefinierte Labelingstrategien an, die bei der Entwicklung von Strategien zur Umsetzung des Prädikats $l/5$ genutzt werden (siehe Abb. 6.6):

1. $labeling([V_1, ..., V_n])$*:* Die Sequenz $[V_1, ..., V_n]$ von Domainvariablen wird sukzessiv durchlaufen. Das Constraint $indomain(V)$ wählt zuerst das kleinste Element aus dem Wertebereich $D(V)$ aus und beim Backtracking dann jeweils den nächstgrößeren noch nicht gewählten Wert.
2. $labeleff([V_1, ..., V_n])$*:* Die Sequenz $[V_1, ..., V_n]$ von Domainvariablen wird nicht sukzessiv durchlaufen. Aus der Liste wird stattdessen jeweils die Variable ausgewählt, die den kleinsten Wertebereich hat. $deleteff(Vs0, V, Vs)$ liefert zur Liste $Vs0$ die entsprechende Domainvariable und den Rest der Liste Vs.
3. $labeleffc([V_1, ..., V_n])$*:* Analog zu $labeleff/2$, nur dass bei gleich großen Wertebereichen zweier Domainvariablen die mit der größten assoziierten Menge von Constraints ausgewählt wird.

```
labeling([]):-!.
labeling([V|Vs]):-
   indomain(V),
   labeling(Vs).
labeleff([]):-!.
labeleff(Vs0):-
   deleteff(Vs0,V,Vs),
   indomain(V),
   labeleff(Vs).
labeleffc([]):-!.
labeleffc(Vs0):-
   deleteffc(Vs0,V,Vs),
   indomain(V),
   labeleffc(Vs).
```

Abbildung 6.6.: Vordefinierte Labeling-Strategien der FD-Bibliothek von ECLiPSe.

6.3.1. Intuitive Strategie

Die einfachste und intuitivste Strategie zur Umsetzung der Instruktionsauswahl ist, alle Domainvariablen des CSPs $COVER(b)$ sowie die Domainvariablen der Menge $STK_{cse}(b)$ zu labeln. Durch das Labeln dieser Domainvariablen werden alle Kostenvariablen an feste Werte gebunden. Die Labelingstrategien, die in $minimize/2$ eingesetzt werden, sind durch die Prädikate $ls/5$ (Anwendung von $labeling/1$), $leff/5$ (Anwendung von $labeleff/1$) und $leffc/5$ (Anwendung von $labeleffc/1$) gegeben, die wie folgt definiert sind:

```
ls(Cs,CSEs,STKs,FEs,ITs):-
   flatten([STKs,CSEs,FEs,ITs],L),
   labeling(L).

leff(Cs,CSEs,STKs,FEs,ITs):-
   flatten([STKs,CSEs,FEs,ITs],L),
   labeleff(L).

leffc(Cs,CSEs,STKs,FEs,ITs):-
   flatten([STKs,CSEs,FEs,ITs],L),
   labeleffc(L).
```

Darin glättet $flatten/2$ die Liste von Listen im ersten Argument zu einer flachen Liste.

In Tabelle 6.3 sind die optimalen bzw. besten gefundenen Kosten für eine Teilmenge der DSPStone-Benchmarks gezeigt (Spalte c_{best}). Diese Kosten entsprechen gemäß des Kostenmodells der Anzahl notwendiger FMOs, die sich nach Einfügen der notwendigen Datentransfers und dem Verschmelzen von FMOs zu komplexen FMOs ergibt. Die Laufzeiten für die unterschiedlichen Labelingstrategien sind unter den Spalten t_{ls}, t_{leff} und t_{leffc} dargestellt. Es zeigt sich, dass für alle drei Strategien schon für die relativ kleinen Benchmarks sehr hohe Laufzeiten erforderlich sind (die größeren

	c_{best}	t_{ls}	t_{leff}	t_{leffc}
ru	5	0.9	0.9	0.7
cm	10	44.8	43.3	16.3
cu	<u>15</u>	12h	12h	12h
bi1	<u>17</u>	12h	12h	12h

Tabelle 6.3.: Resultate für die intuitive Labelingstrategie, mit maximal 12 CPU-Stunden Laufzeit.

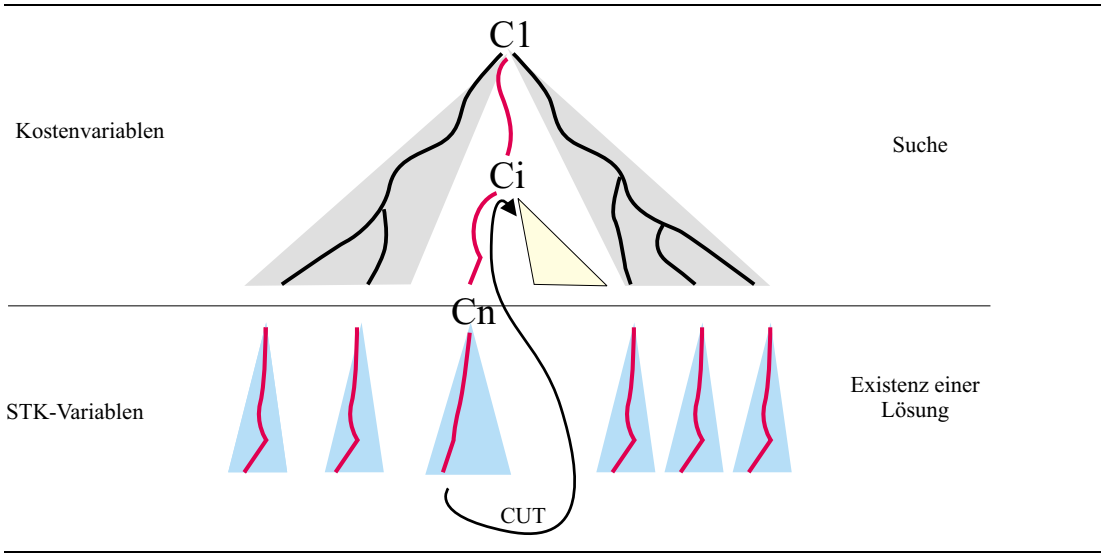


Abbildung 6.7.: Reduktion der Suche.

Benchmarks sind daher erst gar nicht mehr aufgeführt worden). Die Optimierungen wurden nach 12h Laufzeit abgebrochen. Das bis dahin beste gefundene Resultat ist in der Tabelle zu sehen. Bei diesen Resultaten ist nicht sicher, ob sie das Optimum repräsentieren, und sie sind unterstrichen dargestellt.

6.3.2. Reduktion des Suchaufwands

Vergleicht man den Lösungsraum der Domainvariablen aus $SFI(b) = STK(b) \cup STK_{cse}(b) \cup FE(b) \cup IT(b)$ mit dem Lösungsraum $Cost(b)_\Delta$ der noch nicht fix gebundenen Kostenvariablen, so ist letzterer wesentlich geringer, auch wenn er vom Komplexitätsgrad ebenfalls einen exponenziellen Suchraum aufspannt. Die folgende Strategie reduziert den Aufwand der Suche drastisch und führt für alle Benchmarks zu akzeptablen Laufzeiten.

Das Labeling wird über den Kostenvariablen $Cost_\Delta(b)$ durchgeführt, und man erhält somit eine partielle Variablenbelegung $\theta_{Cost_\Delta(b)}$, durch die die Gesamtkosten determiniert sind. Allerdings sind die Variablen von $SFI(b)$ i.d.R. nicht fix gebunden, da sich für bestimmte Kosten mehrere Alternativen für die Ressourcenbelegungen erge-

ben. Es muss noch gezeigt werden, dass $\theta_{Cost_\Delta(b)}$ auch eine partielle Lösung ist. Es ist hier ausreichend, eine einzige Variablenbelegung für die Domainvariablen der Menge $SFI(b)$ zu finden, so dass θ eine Lösung ist. Danach kann unmittelbar mit der weiteren Suche über den Domainvariablen aus $Cost_\Delta(b)$ fortgefahren werden.

Diese modifizierte Suche ist in Abb. 6.7 gezeigt, in der der Suchbaum über den C- und STK-Variablen in stark vereinfachter Form gezeigt ist. Zuerst werden auf jedem Pfad die Domainvariablen aus $Cost_\Delta(b)$ gelabelt. Um eine Lösung für die Domainvariablen der Menge $SFI(b)$ zu finden, muss nur ein einziger Pfad in den gezeigten Unterbäumen gefunden werden. Die weitere Suche nach einer besseren Lösung kann dann oberhalb des Teilbaumes am letzten Backtrackingpunkt erfolgen. Zwar kann die Lösungssuche in den Teilbäumen über $SFI(b)$ immer noch exponenziell viel Zeit in Anspruch nehmen, Untersuchungen haben aber gezeigt, dass es in diesen Teilbäumen so gut wie kein Backtracking gibt und Lösungen sehr schnell gefunden werden.

In logischen Programmiersprachen besteht die Möglichkeit, den Suchraum, der sich durch Backtracking ergibt, explizit abzuschneiden. Dies wird in ECLiPSe (d.h. eigentlich schon in Prolog) durch ein allgemeines Prädikat cut/1 zur Beschneidung des Suchraums formuliert:

```
cut(P):- P,!.
```

Dem Prädikat *cut/1* wird ein Goal *P* übergeben (Goals können in Prolog als Argument übergeben und dann wie gezeigt aufgerufen werden). Durch den Cut wird hier nun nach der Ausführung des Goals P das Abschneiden des noch möglichen verbleibenden Suchraums von P bewirkt.

Auf dieser Grundlage werden nun die folgenden Labelingstrategien eingeführt:

```
lcs(Cs,CSEs,STKs,FEs,ITs):-
   flatten([STKs,CSEs,FEs,ITs],L),!,
   labeling(Cs),
   cut(labeling(L)).

lceff(Cs,CSEs,STKs,FEs,ITs):-
   flatten([STKs,CSEs,FEs,ITs],L),!,
   labeleff(Cs),
   cut(labeling(L)).

lceffc(Cs,CSEs,STKs,FEs,ITs):-(STKs,CSEs,Cs):-
   flatten([STKs,CSEs,FEs,ITs],L),!,
   labeleffc(Cs),
   cut(labeling(L)).
```

$flatten/2$ generiert aus den Listen *STKs*, *CSEs*, *FEs* und *ITs* eine flache Liste *L*. Der Cut nach dem Aufruf von $flatten/2$ ist notwendig, damit nicht während der Suche über dieses Prädikat Backtracking durchgeführt wird.

Für das Labeling der Kostenvariablen werden die vordefinierten Labelingstrategien angewandt. Die Resultate der reduzierten Suche unter Anwendung der Labelingstrategie $lcs/5$ sind in Tabelle 6.4 gezeigt. Man erkennt bei den datenflussdominierten

	$\#dfg$	c_{best}	t_{lcs}	$t_{lcs_{best}}$
ru	9/8/0	5	< 0.1	< 0.1
cm	17/20/4	10	0.3	0.1
cu	33/35/6	15	0.4	0.2
bi1	30/31/3	17	0.6	0.2
co	16/14/2	5	0.2	0.2
ruN	24/23/4	9	0.2	0.2
cuN	53/58/12	27	1.2	0.8
biN	48/54/10	25	1.1	0.6
fir	22/20/3	5	0.2	0.2

Tabelle 6.4.: Resultate der verbesserten Labelingstrategie über den Kostenvariablen mit maximal 24h Laufzeit. Die Spalte $\#dfg$ beinhaltet die Eckdaten der Benchmarks: #Knoten / #Kanten / #CSEs.

	$c_{best_{trad}}$	c_{best}	%
ru	5	5	0%
cm	21	10	52%
cu	21	15	29%
bi1	27	17	37%
co	5	5	0%
ruN	9	9	0%
cuN	30	27	10%
biN	35	25	29%
fir	8	8	0%

Tabelle 6.5.: Vergleich der Codegüte von GIA mit dem traditionellen baumbasierten Verfahren.

Benchmarks die drastische Reduktion der Laufzeiten im Vergleich zur ersten intuitiven Strategie. So konnte die optimale Instruktionsauswahl diesmal für alle betrachteten Benchmarks in weniger als 2 CPU-Sekunden durchgeführt werden. Die Anwendung der weiteren Labelingstrategien $lcseff/5$ und $lcseffc/5$ ergab nur Unterschiede in den Laufzeiten, die im Zehntelsekundenbereich lagen. Daher sind hier nur die Werte von $lcs/5$ in der Spalte t_{lcs} aufgeführt. In Spalte $t_{lcs_{best}}$ sind die Generierungszeitpunkte aufgeführt, die zeigen, wann innerhalb der Optimierung das beste Resultat gefunden wurde.

In Tabelle 6.5 wird die generierte Codegüte des optimalen graphbasierten Verfahrens (Spalte c_{best} enthält die optimale Anzahl von FMOs) mit der Güte des traditionellen baumbasierten Verfahrens (Spalte $c_{best_{trad}}$) verglichen. Zur Generierung der Resultate wurde der traditionelle Ansatz durch den CLP-Ansatz simuliert, indem die Domainvariablen aus $STK_{cse}(b)$ an die Speicherbank (d-Speicher) gebunden und die Bäume jeweils separat optimiert werden. Es werden allerdings keine CSEs an den Speicher ge-

bunden, die ausschließlich Operanden von AGU-Operationen sind. Diese werden an die Registerfiles der AGUs gebunden. In Spalte '%' von Tabelle 6.5 sind die prozentualen Verbesserungen des graphbasierten zum traditionellen baumbasierten Verfahren aufgezeigt: Im Mittel ergibt sich eine Verbesserung von 17.5%, im besten Fall sind es 52% Verbesserung. Für die Benchmarks mit keinen bis zu geringen Verbesserungen ist zu bemerken, dass diese entweder keine CSEs enthielten oder ein Großteil der CSEs den Adressregistern der AGUs zugeordnet waren.

VSV-Labeling: abwechselnde verbessernde Suche und Verifikation

Es soll hier nochmal das unterliegende Konzept reflektiert werden: Die Grundidee ist, eine Menge von Variablen V zu Labeln, welche die Kosten determiniert und dabei einen möglichst geringen Suchraum aufspannt. Die verbleibenden Domainvariablen werden dann ebenfalls gelabelt, um zu zeigen, dass θ_V eine partielle Lösung ist. Sodann kann mit der Suche über der Menge V fortgefahren werden. Es handelt sich hier also um abwechselnde *verbessernde Suche (VS)* und *Verifikation (V)*. Aus diesem Grund wird diese Suchstrategie auch als VSV-Labeling bezeichnet. In Kapitel 7 wird noch demonstriert, dass allein das Finden einer Menge V mit möglichst geringem Suchraum nicht immer hinreichend ist, um auch effiziente Suchstrategien zu erhalten.

6.3.3. Analyse der Laufzeiten

Setzt man die Größen der Basisblöcke einmal in Relation zu den Laufzeiten, ist auffällig, dass die Benchmarks biN und cuN verglichen zu ihren Eckdaten ähnliche Laufzeiten aufweisen, wie die kleinen Basisblöcke, wogegen man größere Laufzeiten gemäß der exponenziellen Natur des Optimierungsproblems erwartet hätte. Der Grund für die geringen Laufzeitunterschiede wird im Folgenden analysiert und erläutert. Dazu wird eine grobe Abschätzung für die obere Anzahl von Pfaden für den Suchbaum über $Cost_\Delta(b)$ und $SFI(b)$ durchgeführt.

Innerhalb der Teilbäume von $SFI(b)$ wird nur nach einer Lösung gesucht, wenn alle Domainvariablen aus $Cost_\Delta(b)$ ohne Konflikte gelabelt werden können. Untersuchungen haben gezeigt, dass innerhalb dieser Teilbäume so gut wie kein Backtracking durchgeführt wird, so dass hier kaum Pfadverzweigungen entstehen. Die Betrachtungen können daher auf Pfade innerhalb des Suchbaums von $Cost_\Delta(b)$ beschränkt und die Lösungen für $SFI(b)$ als reine Pfadverlängerung jeder Lösung von $Cost_\Delta(b)$ interpretiert werden.

Wie schon erwähnt, wird die Kostenfunktion als Constraint $C_\Delta = \sum_{V \in Cost_\Delta(b)} V$ formuliert. In $minimize/2$ wird zunächst nach einer Lösung $C_\Delta = c_{\Delta_1}$ gesucht und die Suche mit dem weiteren Constraint $C_\Delta < c_{\Delta_1}$ nach einem besseren Wert für C_Δ fortgeführt. c_{Δ_1} wird im folgenden auch Δ-Init genant. Ist c_{Δ_i} die i-te gefundene verbesserte Lösung, wird die Suche mit $C_\Delta < c_{\Delta_i}$ fortgesetzt. Dies wird solange durchgeführt, bis feststeht, dass keine bessere Lösung mehr existiert. Das beste gefundene Resultat wird mit $c_{\Delta_{best}}$ bezeichnet, welches dann das Δ-Optimum repräsentiert. Die optimalen Kosten ergeben sich somit zu $c_{best} = c_{fix} + c_{\Delta_{best}}$.

Angenommen, dass der aktuell beste gefundene Wert durch c_{Δ_i} gegeben ist und die Suche nun fortgesetzt wird. Die Constraints $C_\Delta = \sum_{dv \in Cost_\Delta(b)} dv$ und $C_\Delta < c_{\Delta_i}$ bewirken durch Constraintpropagierung, dass, sobald die Summe der Domainvariablen aus $Cost_\Delta(b)$ den Wert $c_{\Delta_i} - 1$ übersteigt, Backtracking ausgelöst wird. Ist die Summe

$c = 1$	1
$c = 2$	2, $1 + 1$
$c = 3$	$2 + 1$, $1 + 1 + 1$
$c = 4$	$2 + 2$,$2 + 1 + 1$,$1 + 1 + 1 + 1$
$c = 5$	$2 + 2 + 1$, $2 + 1 + 1 + 1$, $1 + 1 + 1 + 1 + 1$

Tabelle 6.6.: Mögliche Terme, um einen Wert $c \in \{1, .., 5\}$ aus der Summe der Zahlen aus $\{1, 2\}$ zu bilden.

einer Teilmenge aus $Cost_\Delta(b)$ gleich $c_{\Delta_i} - 1$, wird versucht, alle anderen Variablen auf Null zu setzen. Gelingt das nicht, wird wiederum Backtracking ausgelöst.

Die Domainvariablen aus $Cost_\Delta(b)$ können somit die Summe $c_{\Delta_i} - 1$ während des Labelings nicht überschreiten. Die Anzahl der Pfade im Suchbaum kann also nicht größer sein als die Anzahl der Möglichkeiten, einen Wert kleiner als c_{Δ_i} mit Zahlen aus dem Wertebereich der Domainvariablen aus $Cost_\Delta(b)$ zu bilden. Für das gezeigte Kostenmodell der ADSP2100-Familie haben alle Domainvariablen $V \in Cost_\Delta(b)$ Wertebereiche $D(V) \subseteq \{0, 1, 2\}$. Die Anzahl der Pfade entspricht somit der Anzahl der Möglichkeiten, Werte aus {0,1,2} auf die Kostenvariablen aus $Cost_\Delta(b)$ zu verteilen, so dass die Summe $c_{\Delta_i} - 1$ nicht überschritten wird.

In Tabelle 6.6 sind die Möglichkeiten gezeigt, einen Wert $c \in \{1, .., 5\}$ von Summanden aus $\{1, 2\}$ zu bilden. Für $c = 1$ gibt es einen Term. Für $c = 2$ und $c = 3$ gibt es zwei Terme. Die größte Anzahl von Summanden in einem Term ergibt sich bei der Summenbildung mit Summanden gleich 1. Die Möglichkeiten, die Summe c nur aus Einsen über $Cost_\Delta(b)$ zu bilden, entspricht der Anzahl der Möglichkeiten, c Einsen auf $n = |Cost_\Delta(b)|$ Domainvariablen zu verteilen. Aus der Kombinatorik ist bekannt, dass dies gerade $\frac{n!}{(n-c)!c!}$ (was auch durch $\binom{n}{c}$ notiert wird) entspricht. Da es bei unterschiedlichen Summanden sinnvoll ist, Permutationen der Summanden bei Zuordnung auf die gleiche Menge von Variablen zu betrachten, wird die Anzahl der Möglichkeiten nach oben mit $\frac{n!}{(n-c)!}$ abgeschätzt. Für ein bestimmtes c läßt sich die Anzahl der Pfade durch $(\lfloor \frac{c}{2} \rfloor + 1)(\frac{n!}{(n-c)!})$ nach oben abschätzen, wobei $(\lfloor \frac{c}{2} \rfloor + 1)$ die Anzahl der Terme ist, um die Summe c zu bilden.

Bei der Suche nach einer besseren Lösung für ein c_{Δ_i} müssen im schlimmsten Fall alle Pfade für alle $c_{\Delta_{i+1}} < c_{\Delta_i}$ durchlaufen werden. Die Anzahl der Pfade lässt sich für $c' = c_{\Delta_i} - 1$ mit $c'(\left\lfloor \frac{c'}{2} \right\rfloor + 1)(\frac{n!}{(n-c')!})$ oder noch einfacher mit $(c' + 1)^2 n^{c'}$ nach oben abschätzen.

Ist $c_{\Delta_{best}}$ nahe bei Null, folgt aus der Abschätzung, dass der verbleibende Suchraum handhabbar wird. So ist die Menge der Pfade für $c' = 3$ kleiner $9n^3$, für $c' = 2$ kleiner $4n^2$ und für $c = 1$ kleiner $2n$. Durch die Constraintpropagierung der weiteren Constraints der alternativen Überdeckungen kann die Anzahl der Pfade natürlich noch wesentlich weiter eingeschränkt werden. Für die DSPStone-Benchmarks liegt $c_{\Delta_{best}}$ für alle betrachteten Benchmarks im Wertebereich $\{0, 1, 2\}$ und es wird immer sofort ein Wert $c_{\Delta_1} \leq 2$ innerhalb von Bruchteilen von Sekunden gefunden. Für die weitere Suche ergibt sich durch $c' < 2$ ein höchstens linearer Suchraum. Dass für die DSPStone-Benchmarks immer eine günstige Kostenkonstellation vorliegt, ist sicherlich kein Zu-

	$\#dfg$	$\|Cost(b)\|$	$\|Cost_\Delta(b)\|$	t_{cover}	t_{cost}
l1	23/26/4	53	32	0.9	0.6
l2	27/33/6	66	42	1.2	0.7
l3	25/38/6	73	50	1.3	0.9
l4	36/46/7	89	60	1.6	1.1
l5	35/51/11	97	53	1.7	1.2
l6	37/52/9	98	67	2.0	1.6
l7	50/78/13	131	106	3.0	2.4
l8	83/144/21	248	188	6.7	6.4

Tabelle 6.7.: Eckdaten der internen Benchmarks sowie die Laufzeiten für die Generierung der alternativen Überdeckungen und der Generierung der Kostenconstraints.

fall, bedenkt man, dass die ADSP2100-Familie ja gerade auf diese Applikationen hin ausgerichtet ist. Es ist zu bemerken, dass es natürlich auch wichtig ist, jeweils eine erste Lösung mit einem Δ-Init zu finden, dass möglichst nahe bei $c_{\Delta_{best}}$ liegt, da sich bei großen Werten Δ-Init ansonsten wiederum ein sehr großer potenzieller Suchraum ergibt.

Um die Grenzen von GIA auszutesten, wurden Benchmarks konstruiert, bei denen $c_{\Delta_{best}}$ größer als zwei ist. Die Eckdaten dieser Benchmarks sind in Tabelle 6.7 gezeigt ($\#dfg$). Weiterhin ist die Größe von $Cost(b)$ und $Cost_\Delta(b)$ gezeigt und die Zeiten für die Generierung der alternativen Überdeckungen (t_{cover}), sowie der Kostenconstraints (t_{cost}) aufgeführt.

In Tabelle 6.8 sind die Resultate für diese Benchmarks gezeigt. Die besten gefundenen Kosten sind in Spalte c_{best} wiedergegeben. Dahinter stehen in Klammern jeweils die Werte von $c_{\Delta_{best}}$. Die Laufzeiten für die Anwendung der vordefinierten Labelingstrategien $labeling/1$ (Spalte t_{seq}), $labeleff/1$ (Spalte t_{eff}) und $labeleffc/1$ (Spalte t_{effc}) gezeigt. Weiterhin sind in diesen Spalten die Zeitpunkte für das Finden der Zwischenresultate c_i angegeben. Die Versuche wurden nach maximal 24h Laufzeit mit dem bis dahin besten gefundenen Resultat abgebrochen. Die Werte, für die demnach nicht mit Garantie gesagt werden kann, ob sie auch das Optimum sind, sind in der Spalte c_{best} unterstrichen dargestellt.

Zunächst erkennt man an diesen Resultaten die Abhängigkeit der Laufzeit von $c_{\Delta_{best}}$ und n=$|Cost_\Delta(b)|$ sehr gut. Es ist nun auch ein merkbarer Unterschied in der Anwendung der vordefinierten Labelingstrategien zu erkennen. Für die Benchmarks l1-l5 ist die Strategie $labeleffc/1$ die beste, gefolgt von $labeleff/1$: Hier ergeben sich bis zu 30-fache Laufzeitverbesserungen (l4). Die Benchmarks l6-l8 zeigen die Grenzen von GIA, da diese nicht innerhalb von 24h terminierten. Aber auch hier werden Unterschiede in der Anwendung der Labelingstrategien deutlich, wenn man sich die Zeitpunkte der besten generierten Resultate anschaut. Bis auf l8 waren auch hier die Strategien $labeleffc/1$ und $labeleff/1$ die besseren Strategien. Für l7 konnte die Strategie $labeling/1$ das beste Resultat nicht finden. Für l8 konnte allerdings die Strategie $labeling/1$ innerhalb der 24h Grenze ein besseres Resultat finden als die anderen Stra-

	$\|Cost_\Delta(b)\|$	$c_{best}(c_{\Delta_{best}})$	t_{seq}	t_{eff}	t_{effc}
l1	32	16(3)	9.2 0.2(16)	3.5 0.3(16)	2.0 0.2(16)
l2	42	17(3)	13.6 8.4(17) 1.2(18) 0.3(19,20)	6.3 0.3(17)	3.9 0.3(17)
l3	50	21(6)	1635 350(21) 124(22) 7.3(23) 0.7(25,26)	627 0.3(21,23)	196 0.2(21,23)
l4	60	21(6)	6835 1578.2(21) 7.3(22) 1.7(23) 0.2(24)	694 0.4(21)	203 0.4(21)
l5	55	24(5)	1519 2.0(24) 0.5(26)	481 0.2(24,25)	339 0.4(24,25)
l6	67	33(9)	24h 33(173.4) 8.3(35) 0.6(37)	24h 0.3(33,35)	24h 0.4(33,35)
l7	106	41(13)	24h 1.5(43) 0.8(45)	24h 4684(41,42) 2.4(43) 0.9(44,45)	24h 10544(41,42) 2.1(43) 0.8(44,45)
l8	188	87(36)	24h 68449(87) 32535(88) 1278.(89) 223(90) 20.5(92) 1.4(93)	24h 29.8(89) 10.6(90) 1.4(93,94)	24h 30.4(89) 9.4(90,91) 1.5 (93)

Tabelle 6.8.: Resultate für die internen Benchmarks.

	$c_{best_{trad}}$	c_{best}	%
l1	26	16	38%
l2	34	17	50%
l3	39	21	46%
l4	46	21	54%
l5	47	24	49%
l6	62	33	47%
l7	91	41	55%
l8	166	88	47%

Tabelle 6.9.: Vergleich der Codegüte der größeren Benchmarks mit dem traditionellen baumbasierten Verfahren.

tegien.

Betrachtet man die Generierungszeiten der Zwischenresultate, ist auffällig, dass Resultate, die maximal 10% von den besten gefundenen Resultaten abweichen, immer innerhalb von 10 Sekunden gefunden werden. Die innerhalb von 1 Sekunde gefundenen Resultate weichen im Mittel um 7% von den Besten ab. Das schon relativ frühzeitig gute Resultate zur Verfügung stehen, kann genutzt werden, um die Suche durch Timeouts zeitlich einzuschränken. So kann z.B. ein Benutzer akzeptable Laufzeitschranken angeben, die die Optimierungsläufe maximal verbrauchen dürfen. Man kann die Abhängigkeit zwischen n, $c_{\Delta_{best}}$ und entsprechenden Laufzeitschranken auch durch Tabellen oder Funktionen in den Compiler integrieren, so dass dieser zu gegebenen Benchmarks die Laufzeitgrenzen selbst bestimmen kann. Da die Bestimmung von $c_{\Delta_{best}}$ nicht ohne einen Optimierungslauf möglich ist, wird die Optimierung für eine definierte Zeitschranke gestartet und das beste gefundene Resultat als obere Schätzung von $c_{\Delta_{best}}$ genommen. Eine gute Zuordnung in Form von Tabellen bzw. Funktionen muss letztendlich aus der Erfahrung resultieren, die man durch Anwendung von GIA auf eine angemessene Anzahl von Benchmarks gewinnt.

In Tabelle 6.9 ist noch ein Vergleich der Codegüte von GIA im Vergleich zum traditionellen Verfahren für die internen Benchmarks gezeigt. Die mittlere Verbesserung liegt hier bei 48.25%.

6.3.4. Partitionierung

Um auch das Laufzeitverhalten der Optimierung von großen Basisblöcken zu verbessern, wird in diesem Kapitel eine Strategie eingeführt, die auf der Partitionierung großer Basisblöcke in kleinere und handhabbare Basisblöcke basiert. Die Strategie garantiert zwar keine optimalen Resultate mehr; die Experimente zeigen aber, dass immer Werte gleich oder nah bei den durch die optimalen Techniken gefundenen Resultaten erzielt werden. Die Strategie zur Partitionierung besteht in der Zerlegung des Datenflussgraphen eines Basisblocks an den CSEs. Dies garantiert zwar nicht immer, dass dadurch hinreichend kleine Datenflussgraphen entstehen. Für die betrachteten Benchmarks ist diese Strategie allerdings ausreichend.

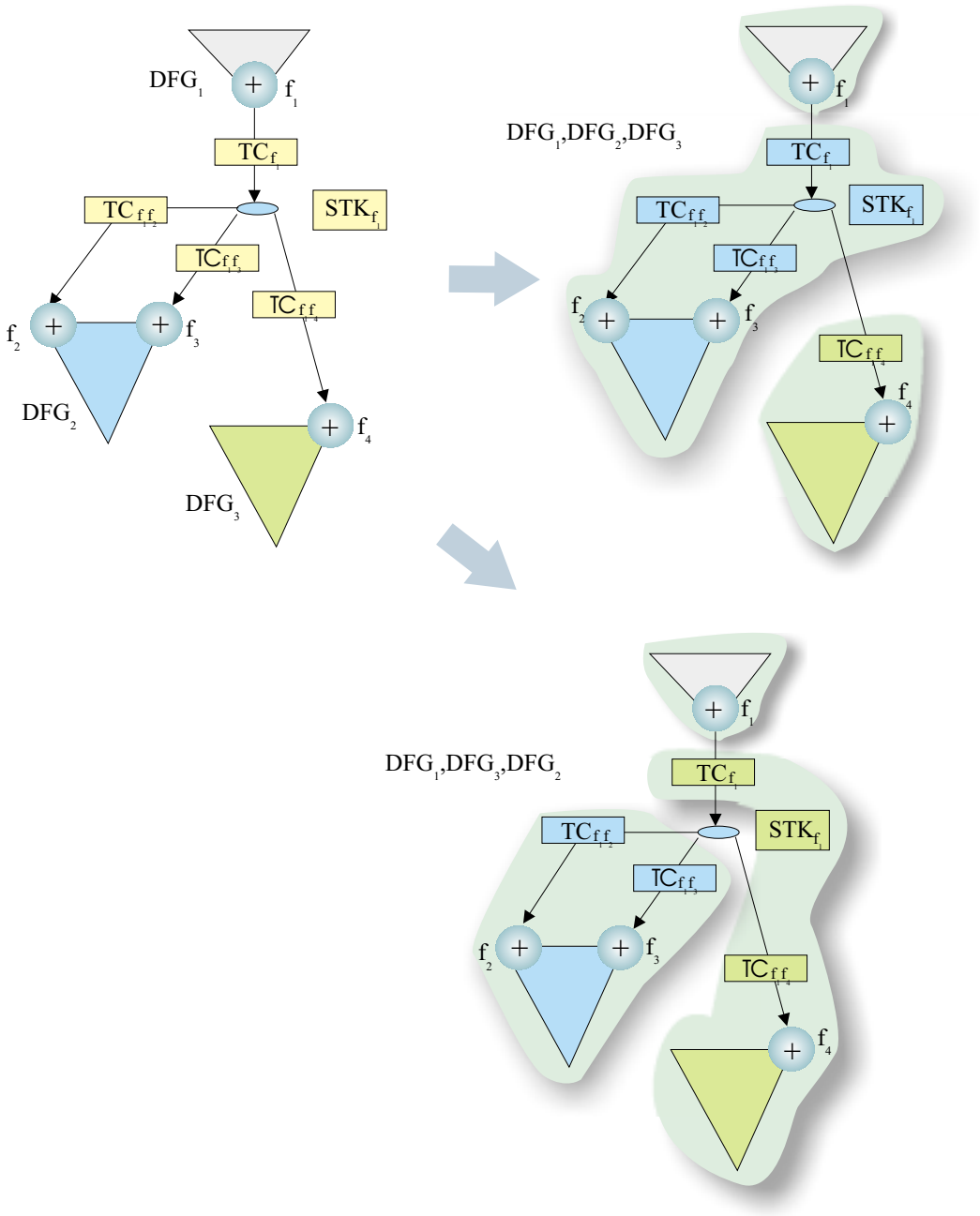


Abbildung 6.8.: Partitionierung von Datenflussgraphen.

	c_{best}	c_{part}	t_{part}	c_{inv}	t_{inv}
l1	16	16	0.4	16	2.3
l2	17	17	0.6	17	0.6
l3	21	21	0.8	23	6.9
l4	21	23	1.2	23	7.4
l5	24	26	0.9	24	1.2
l6	33	37	7.9	34	170.3
l7	41	43	2.5	49	2.2
l8	88	83	6.4	91	68.9

Tabelle 6.10.: Resultate für Partitionierung und Permutation für die Benchmarks l1-l8.

Die Zerteilung an den CSEs bewirkt im Unterschied zu den baumbasierten Verfahren keine Zerteilung in Teilbäume. Die Transferkosten zwischen Teilgraphen werden weiterhin bzgl. der Graphstruktur berechnet. Die Transferkosten TC_{fv} und die ST-Komponenten STK_{fv} eines CSEs fv werden erst innerhalb der Optimierung des ersten Teildatenflussgraphen gelabelt, der Benutzungen von fv enthält. Die Vorgehensweise ist in Abb. 6.8 verdeutlicht. Der gezeigte Datenflussgraph wird in die drei Teilgraphen DFG_1, DFG_2, DFG_3 zerlegt. Die Kostenvariablen der CSE f_1 sind ebenfalls dargestellt. In der Abbildung sind zwei Labelingvarianten gezeigt, die sich durch unterschiedliche Labelingreihenfolgen der Teildatenflussgraphen DFG_1, DFG_2, DFG_3 ergeben:

1. DFG_1, DFG_2, DFG_3
2. DFG_1, DFG_3, DFG_2

Zunächst wird DFG_1 optimiert. Das Labeling der Kostenvariablen TC_{f_2} und der zusätzlichen STK-Variablen STK_{f_1} wird aber erst während des Labelings von DFG_2 durchgeführt, und es werden die Kostenvariablen von STK_2 zu Benutzungen innerhalb von DFG_2 gelabelt: TC_{f_1,f_2} und TC_{f_1,f_3}. Beim der Optimierung von DFG_3 wird noch die Kostenvariable TC_{f_1,f_4} gelabelt. Unterschiedliche Permutationen der Reihenfolge, in der Teildatenflussgraphen optimiert werden, können zu unterschiedlichen resultierenden Kosten führen, wie im Folgenden noch gezeigt wird.

Mit der Partitionierung ist es möglich, selbst die größeren Datenflussgraphen in wenigen Sekunden zu optimieren. Die Resultate sind in Tabelle 6.10 gezeigt. In Spalte c_{part} sind die Kosten und unter t_{part} die entsprechenden Laufzeiten gezeigt. Bemerkenswert ist, dass für den größten Benchmark l8 ein wesentlich besseres Resultat gefunden wurde, als das nach 24h Laufzeit beste Resultat unter Betrachtung des gesamten Basisblocks.

Um die Effekte der Permutierung der Optimierungsreihenfolgen zu verdeutlichen, wurden die Teildatenflussgraphen noch einmal in inverser Reihenfolge optimiert. Die Resultate der inversen Optimierung sind unter den Spalten c_{inv} und t_{inv} in Tabelle 6.10 gezeigt. Man erkennt einen deutlichen Unterschied zwischen den beiden Varianten. Neben den verschiedenen Abweichungen von c_{best} sind stellenweise auch größere

	c_{opt}	c_{part}	t_{part}	c_{inv}	t_{inv}
ru	7	7	0.2	7	0.2
cm	13	13	1.1	13	1.1
cu	22	22	1.5	22	1.6
bi1	17	17	1.6	17	1.8
co	12	12	0.7	12	0.8
ruN	14	14	1.1	14	1.0
cuN	29	29	1.7	29	1.8
biN	27	27	1.8	27	1.8
fir	15	15	1.5	15	1.4

Tabelle 6.11.: Resultate für Partitionierung und Permutation für die DSPStone-Benchmarks .

Laufzeitunterschiede zu erkennen. Grund hierfür ist, dass die Benchmarks l1, l3, l4, l6 und l8 am Ende größere Teilgraphen besitzen. Diese verwenden viele CSEs, die auch schon in davor liegenden Teildatenflussgraphen verwendet werden. In der ersten Variante werden schon einige der Kostenvariablen dieser CSEs gebunden, so dass in den letzten Teildatenflussgraphen weniger Kostenvariablen zu labeln sind, was sich auf die Laufzeit auswirkt. Es wurde aber in allen Benchmarks das beste Resultat innerhalb einer CPU-Sekunde gefunden.

Da es schwer ist, Kriterien für eine grundsätzlich gute Permutation zu finden, und eine bestimmte Permutation i.d.R. nicht immer bessere Resultate für alle Benchmarks liefert, ist es denkbar, mehrere Permutationen durchzuführen. Bei den Laufzeiten der Partitionierung ist es durchaus möglich, mehrere Permutationen laufen zu lassen und das beste Resultat zu nehmen, besonders, wenn man sich auch noch Timeouts zunutze macht. Wird aus den beiden gezeigten Varianten immer das beste der beiden Resultate ausgewählt, ergibt sich lediglich eine mittlere Verschlechterung von 1.5% gegenüber den Resultaten, die durch die Techniken auf ganzen Datenflussgraphen erzielt werden.

Für die DSPStone-Benchmarks wurde ebenfalls die Partitionierung durchgeführt, um die Auswirkungen auf die Codegüte zu prüfen. Es wurden die gleichen Resultate gefunden, die auch schon ohne Anwendung der Partitionierung erreicht wurden (also in diesem Fall die optimalen Resultate - siehe Tabelle 6.11). Es zeigt sich, dass bei den meisten Benchmarks die Laufzeiten leicht erhöht sind: hier macht sich der erhöhte Aufwand der Partitionierung der Datenflussgraphen gegenüber den geringen Laufzeiten bemerkbar.

6.3.5. Beibehaltung alternativer Überdeckungen

In ECLiPSe kann spezifiziert werden, dass nach den Optimierungsläufen nur bestimmte Domainvariablen an die letztendlich besten gefundenen Werte gebunden werden. Dies wird ausgenutzt, indem nur die Bindungen der Kostenvariablen nach der Optimierung aufrecht erhalten bleiben. Die Bindungen, die sich durch den Check der

STK-Variablen ergeben, werden dadurch rückgängig gemacht. Durch die Kostenconstraints werden somit die Wertebereiche der weiteren Domainvariablen nur gemäß der Constraintpropagierung eingeschränkt. Hier verbleiben oft sehr große Freiheitsgrade bzgl. der STK-Variablen, so dass man also eine Menge alternativer Überdeckungen zu einer optimalen Kostenüberdeckung erhält.

Die Auswirkungen, die sich durch Beibehaltung bzw. Nichtbeibehaltung alternativer Überdeckungen auf die resultierende Codegüte ergeben, werden im Kapitel 7 noch herausgestellt. Hier zeigt sich, dass durch die erhöhten Freiheitsgrade wesentlich besserer Code generiert wird. Allerdings zahlt man dafür auch den Preis, dass die Existenz von Lösungen nicht mehr garantiert werden kann. Dies wird im nachfolgenden Kapitel beschrieben.

6.4. Existenz von Lösungen

Es wird zuerst auf die lokale Existenz von Lösungen zu einem CSP $GIA_{CSP}(b)$ eingegangen. Danach wird die globale Existenz von Lösungen bzgl. der Resultate von GIA untersucht. Die von GIA generierten alternativen Überdeckungen einer Funktion fun sollten im Idealfall Lösungen bzgl. des CSPs $COVER_{ffp}(fun)$ besitzen.

Existenz von Lösungen zu $GIA_{CSP}(b)$

Lösungen von $GIA_{CSP}(b)$ existieren immer genau dann, wenn $COVER(b)$ eine Lösung besitzt. In Kapitel 5.5.2 wurden Kriterien angegeben, wann die Existenz einer Lösung für $COVER(fun)$ bzw. für $COVER(b)$ gewährleistet ist. Die Grundlage für das in Kapitel 5.5.2 eingeführte Kriterium I war, dass es eine Menge M zusammenhängender sequenzieller ST-Komponenten gibt und alle auf einem Prozessor umgesetzten Operatoren mindestens eine bzgl. M erreichbare Maschinenoperation besitzen. Unter Kriterium II wurde weiterhin die Bedingung an ein CoLIR-Programm gestellt, dass jede FMO mindestens eine erreichbare Maschinenoperation umfasst, und alle Eingangs- und Ausgangssetzungen jedes Basisblocks nur Setzungsalternativen aus M haben. In diesem Fall ist es klar, dass es immer eine Maschinenoperation pro FMO gibt, so dass auch immer ein Datentransferpfad von jeder Setzung zu allen Benutzungen existiert. Dies folgt unmittelbar aus der Konstruktion der beiden Kriterien. Unter diesen Kriterien können Lösungen lokal generiert werden, wobei die Existenz einer globalen Lösung gewährleistet bleibt: Jede Lösung $COVER(b)$ zu einem Basisblock $b \in fun$ ist eine partielle Lösung von $COVER(fun)$. Dies ist die Grundlage dafür, dass Lösungen basisblockweise generiert werden können.

Für die ADSP2100-Familie ist das Kriterium I gegeben. Kriterium II kann ebenfalls eingehalten werden. Die Eingabe zu GIA ist in der derzeitigen Implementierung ein IR-reines CoLIR-Programm, dass zunächst in ein CoLIR-Programm mit frischen FMOs transformiert wird. Nach der Anwendung des Prädikates $fmo/5$ umfassen alle FMOs zunächst noch alle alternativen Maschinenoperationen bzgl. ihres Operators. Somit besitzt jede FMO mindestens eine erreichbare Maschinenoperation und somit existieren auch alle Datentransferpfade zwischen Setzungen und deren Benutzungen.

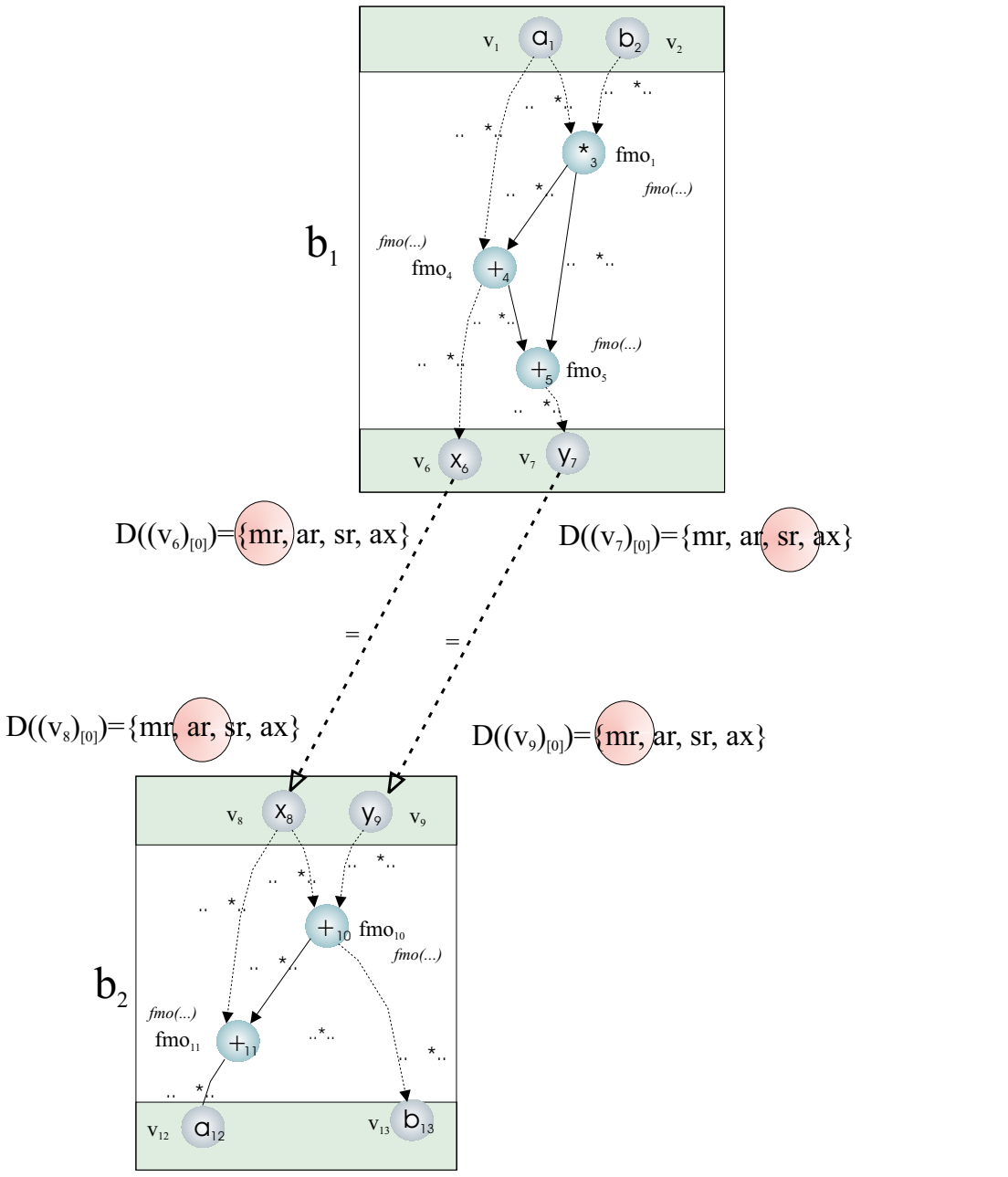


Abbildung 6.9.: Konflikte bzgl. der globalen Existenz von Lösungen.

Durch die Constraints der Konventionen werden die Bedingungen des Kriteriums II ebenfalls nicht verletzt. GIA kann auf allgemeine FMO-reine CoLIR-Programme angewendet werden. In diesem Fall ist zu prüfen, ob die CoLIR-Programme der Vorgängerphase dem Kriterium II genügen, um Schlüsse auf die Existenz einer Lösung ziehen zu können.

Globale Existenz von Lösungen der Resultate von GIA

Die Existenz von Lösungen kann für die Resultate von GIA nicht garantiert werden, wenn man pro Basisblock eine Menge von alternativen Lösungen beibehält. In diesem Fall ist die Existenz von Lösungen für die Funktionen bzgl. des CSPs $COVER_{ffp}(fun)$ nicht mehr gewährleistet. Das Problem entsteht dadurch, dass bzgl. der pro Basisblock besten gefundenen Kosten nur jeweils eine Lösung existieren muss. Danach bleiben nur die Bindungen der Kostenvariablen aufrecht erhalten. Die Bindungen der weiteren Domainvariablen werden rückgängig gemacht, wobei die Domains dieser Variablen aber gemäß der Constraintpropagierung, die aus den Bindungen der Kostenvariablen resultiert, noch eingeschränkt werden können. Da die Constraints nur lokale Konsistenz garantieren, können in den verbleibenden Domains durchaus Werte vorkommen, die global zu keiner Lösung gehören. Dieses Problem ist in Abb. 6.9 schemenhaft dargestellt. Angenommen, es wird zum Basisblock b_1 eine Lösung θ_1 generiert. In b_1 wird dadurch die Ausgangssetzung $(v_6)_{[0]}$ an die ST-Komponente mr gebunden. Die Bindungen von θ_1 werden jetzt bis auf die der Kostenvariablen rückgängig gemacht, und der Domain von $(v_6)_{[0]}$ ist jetzt durch $D((v_6)_{[0]}) = \{mr, ar, sr, ax\}$ gegeben. Im nächsten Schritt wird der Basisblock b_2 optimiert und eine Lösung θ_2 erzeugt, die die Eingangssetzung $(v_8)_{[0]}$ an ar bindet. Für v_6 und v_8 gibt es eine Kante $v_6 \rightarrow v_8 \in gDFG(fun)$ im globalen Datenflussgraphen (wobei $b_1, b_2 \in fun$ gilt). Hierdurch ist gefordert, dass $(v_6)_{[0]} = (v_8)_{[0]}$ gilt. Da die Bindungen in b_1 wieder rückgängig gemacht wurden, ist die Gleichheit jetzt zulässig, da der aktuelle Domain von v_6 mit $D((v_6)_{[0]}) = \{mr, ar, sr, ax\}$ die Gleichheit mit $\{ar\}$ erfüllen kann. Es ist aber leider nicht sichergestellt, dass $\{(v_6)_{[0]} \mapsto ar\}$ eine partielle Lösung von b_1 ist. Ist das nicht der Fall, wird dies durch die Constraints von $COVER(fun)$ und durch die im Constraintstore verbliebenen Constraints von $GIA_{CSP}(b_1)$ nicht unbedingt aufgedeckt[3].

Es sei hier bemerkt, dass es nicht gelungen ist, ein konkretes Beispiel zu konstruieren, das eine Instanz dieses Problems darstellt. Weiterhin ist auch in allen Beispielen, die getestet wurden, nie ein Konflikt aufgetreten. Sollte dies doch mal vorkommen kann die folgende Eigenschaft der resultierenden alternativen Überdeckungen von GIA ausgenutzt werden: alle FMOs sind bzgl. M_{adsp} erreichbare FMOs. Somit ist zumindest die Existenz einer Lösung gemäß $COVER$ garantiert. Sollte in einer nachfol-

[3] Auch wenn die CSPs nur Basisblockweise gelöst werden, so werden sie ja schon im Kontext von $COVER(fun)$ gelöst, das zuvor generiert wurde. Außerdem verbleiben auch die zu einem CSP $GIA_{CSP}(b_1)$ erzeugten Constraints gemäß der verbleibenden Bindungen der Kostenvariablen weiterhin im Constraintstore. So werden bei der Optimierung eines CSPs $GIA_{CSP}(b_2)$ (wobei angenommen wird, dass b_2 nach b_1 optimiert wird) zwar globale Aspekte berücksichtigt, wenn Constraints im Constraintstore Variablen aus b_2 und aus anderen Basisblöcken umfassen und durch Constraintpropagierung noch weitere Constraints anderer Basisblöcke reaktivieren. Da diese aber nur auf lokale Konsistenz prüfen, kann nicht auf globale Konsistenz geschlossen werden.

genden Codegenerierungsphase von GIA ein Konflikt auftreten, kann man GIA für den entsprechenden Basisblock erneut ausführen und somit eine neue lokale, gültige Lösung generieren. In der Regel sollte dies recht effizient möglich sein, da in den meisten Fällen nur wenige zusätzliche Datentransfers erzeugt werden müssen, um die Konflikte aufzulösen, und somit ist auch das Δ-Optimum sehr klein. Ein weiterer möglicher Ausweg aus dieser Situation ist, simultan zu den alternativen Überdeckungen eine einzelne globale Lösung zu erzeugen. Die weiteren Codegenerierungsphasen werden auf beiden Varianten durchgeführt, und man behält, sobald kein Konflikt mehr auftreten kann, die bessere von beiden. Es sei hier schon auf das nachfolgende Kapitel verwiesen, in dem demonstriert wird, dass es durchaus lohnenswert ist ein CoLIR-Programm mit GIA zu generieren, in denen Funktionen evtl. keine Lösung bzgl. $COVER_{ffp}$ besitzen.

6.5. Zusammenfassung

In diesem Kapitel wurden Techniken zur optimalen, graphbasierten Instruktionsauswahl beschrieben. Die Teilprobleme der Mustererkennung sowie der Repräsentation alternativer Überdeckungen von Maschinenoperationsmustern wurden schon durch das CSP-Modell $COVER$ realisiert. Die hier umzusetzende Aufgabe bestand in der Entwicklung eines Kostenmodells zur Auswahl der kostengünstigsten Überdeckung und in der Entwicklung von Labelingstrategien, die ein möglichst schnelles Finden guter Überdeckungen erlauben. Die beschriebenen Techniken sind unter dem Begriff GIA zusammengefasst worden, und sie sind alle Ausprägungen des gleichen CSP-Modells GIA_{CSP}, wobei sie sich nur durch die Anwendungen verschiedener Labelingstrategien unterscheiden.

Ausgehend von der reinen Anwendung vordefinierter Labelingstrategien der FD-Bibliothek von ECLiPSe, die zu absolut inakzeptablen Laufzeiten führten, wurden wesentlich effizientere Strategien entwickelt. Dabei wurden wichtige Konzepte eingeführt, die zu wesentlichen Laufzeitverbesserungen der Optimierung führten und auch in den nachfolgenden Kapiteln zum Einsatz kommen:

- *Beschneidung des Suchraums*: Die Suche in bestimmten Teilbereichen des Suchbaums kann auf die reine Prüfung der Existenz einer Lösung reduziert werden, wenn sich in diesem Teilbereich durch das Labeln der Domainvariablen keine Veränderungen der Kosten ergeben können. Somit muss in diesen Bereichen nur eine Suche bis zum Finden einer Lösung durchgeführt werden. Der Teilbaum kann dann unmittelbar verlassen werden, indem zu einem Backtrackingpunkt zurückgekehrt wird, an dem sich das Labeling von Variablen auch wieder auf die Kosten auswirkt. Diese Art der Suche wurde mit VSV-Labeling bezeichnet. Die hierauf aufbauende Labelingstrategie generierte zuerst eine Variablenbelegung für die Kostenvariablen aus $GIA_{cost}(b)$, wodurch die Kosten determiniert wurden. Nach anschließendem erfolgreichem Labeling der STK-, FU- und IT-Variablen wurde dann die Suche nach einer besseren Lösung wieder unmittelbar über den Kostenvariablen fortgesetzt. Die Resultate dieser Strategie lassen sich wie folgt zusammenfassen:

  - Alle DSPStone-Benchmarks ließen sich in weniger als 2 Sekunden exakt lösen. Eine Analyse zeigte, dass der Komplexitätsgrad des Suchraums für diese Technik nicht primär von der Anzahl der Kostenvariablen abhängt. Viel mehr hängt der Komplexitätsgrad vom Δ-Optimum bzw. vom Δ-Init-Wert ab, dessen Größe ausschlaggebend für die Komplexität der Suche ist. Für die betrachteten Benchmarks des DSPStone-Benchmarksets war dieser Wert nicht größer als zwei und begründete das sehr gute Laufzeitverhalten auch für die größeren Benchmarks. Gerade für Letztgenannte waren zunächst auf Grund des exponenziellen Charakters des Suchproblems, auch unter Einsatz von CLP, eher höhere Laufzeiten erwartet worden.
  - Um ein objektives Maß des Laufzeitverhaltens der Techniken aus GIA zu erhalten, wurden interne Benchmarks konstruiert, die größere Δ-Optima aufwiesen. Resultate:
    * Optimale graphbasierte Instruktionsauswahl ist selbst für große Basisblöcke noch in akzeptabler Zeit möglich, sofern sich das Δ-Optimum im Bereich kleiner gleich 8 bewegt (für die ADSP2100-Familie).
    * Lösungen mit Kosten nahe dem Optimum werden schon relativ frühzeitig generiert. Die Lösungen aller Benchmarks, die jeweils innerhalb der ersten CPU-Sekunde gefunden wurden, hatten bzgl. ihrer Kosten eine mittlere Abweichung von 7% im Vergleich zu den optimalen bzw. besten gefundenen Kosten. Diese Eigenschaft kann generell ausgenutzt werden, um Laufzeitschranken für die Optimierungsläufe zu definieren, unter denen immer noch sehr gute Resultate zu erwarten sind.

- *Partitionierung*: Ein Konzept, die Laufzeiten für große Basisblöcke zu reduzieren, ist die Partitionierung eines Basisblocks in eine Menge kleinerer handhabbarer Teilblöcke, die in noch akzeptabler Zeit optimiert werden können. Es hat sich gezeigt, dass sich hierdurch auch die großen Basisblöcke der Benchmarks l1-l8 in wenigen Sekunden optimieren ließen, wobei die Resultate gleich bzw. sehr nahe bei den besten gefundenen Resultaten lagen. Es ergab sich für l1-l8 eine mittlere Abweichung von 4% von den besten Resultaten. Im Fall von Benchmark l8 war das Resultat sogar besser, was zulässig ist, da für l8 die Optimierung nach 24 CPU-Stunden Laufzeit abgebrochen und die Optimalität des Resultats somit nicht verifiziert wurde. Für die DSPStone-Benchmarks ergaben sich durch Anwendung der Partitionierung keinerlei Abweichungen von den optimalen Resultaten.

- *Permutation:* Durch Permutation der Reihenfolge, in der die Partitionen eines Basisblocks optimiert werden, konnten noch Verbesserungen bei suboptimalen Resultaten erzielt werden. Es ergab sich jetzt für l1-l8 eine mittlere Abweichung von 1.5% von den besten Resultaten. Die Effizienz der auf Partitionierung basierten Optimierung lässt hier durchaus eine Menge unterschiedlicher Permutationen zu.

Welche Strategie durchzuführen ist, kann in $COCOON$ z.Z. durch den Benutzer bestimmt werden, der auch Zeitschranken für die Optimierungsläufe auferlegen kann.

Durch das Δ-Optimum ist aber auch ein Kriterium gegeben, durch dass der Compiler bestimmen könnte, welche Strategie zu wählen ist. So könnte zunächst ein Testlauf stattfinden, der einen Δ-Init-Wert bestimmt und aus diesem und den Eckdaten eines Programms die Strategie und die Timeouts festlegt. Es könnten aber auch parallel alternative Optimierungstechniken gestartet und das beste Resultat, das innerhalb einer bestimmten Zeitschranke erzielt wird, an die nachfolgenden Phasen weitergegeben werden. Solche Konzepte beschränken sich nicht nur auf die Instruktionsauswahl, sondern sind für alle Phasen sinnvoll. Konzepte, Compilerphasen durch eine Menge alternativer Techniken umzusetzen, existieren schon in Compilersystemen. Im Cosy-Compilersystem [111] können z.B. ganze Sequenzen alternativer Phasen ausgeführt werden. Dabei hat der Benutzer die Möglichkeit, die Anordnung, in der Techniken ausgeführt werden sollen, selbst zu konfigurieren.

Zur Adaption an andere Prozessoren kann GIA retargiert werden. Die Formulierung der FMOs, auf der Basis der generalisierten Propagierung, erlaubt eine relativ intuitive Spezifikation des Befehlssatzes. Weiterhin sind die Techniken zum Labeling und auch die der Partitionierung unabhängig von spezifischen Prozessoreigenschaften. Allerdings muss derzeit beim Retargieren noch die ECLiPSe- bzw. die Prolog-Syntax verwendet werden. Eine Adaption an ein allgemeineres Format ist aber realisierbar. Auch wenn an einigen Stellen im Text explizit Bezug auf Ressourcen der ADSP2100-Familie genommen wurde, sind diese Referenzen in den Implementierungen durch generische Prädikate umgesetzt worden.

Bei der Entwicklung von GIA hat sich gezeigt, dass der Constraintansatz eine saubere Trennung zwischen dem CSP-Modell, dem Kostenmodell, der Lösungssuche (Labeling-Strategie) und der Optimierung bietet. Weiterhin können Teilprobleme in modularer Weise durch Teil-CSP-Modelle spezifiziert und durch Vereinigung dieser CSP-Modelle zusammengefügt werden. So konnte $COVER$ ohne Modifikation wiederverwendet und das Kostenmodell sauber und einfach ergänzt werden. In der gleichen Form ist auch eine einfache Adaption weiterer Kostenmodelle möglich. Durch die Trennung der Spezifikation von CSP-Modell und Suche war es weiterhin möglich, schnell und einfach eine Vielfalt von Labelingstrategien zu testen.

GIA hält immer noch die Integration der Instruktionsauswahl in andere Phasen offen, da das Resultat aus einer Menge alternativer Überdeckungen besteht. Die eigentliche Integration der Instruktionsauswahl in andere Phasen wird durch das CSP-Modell $COVER$ erreicht, das dann um die Constraints dieser Phasen ergänzt wird. Die im nachfolgenden Kapitel beschriebenen Techniken (I^3R) haben gezeigt, dass aus den von GIA generierten alternativen Überdeckungen hochqualitativer Code generiert werden kann. Weitere Anwendungen von $COVER$ und GIA finden sich in [10, 12] und im Anhang A.

7. Phasenkopplung - Das CSP-Modell I^3R_{CSP}

In diesem Kapitel werden integrierte Verfahren zur Instruktionsauswahl, Instruktionsanordnung und Registerallokation betrachtet. Das Optimierungsziel dieser Techniken ist, die Instruktionszyklen jedes Basisblocks zu minimieren. Die Eingabe der Optimierungstechniken sind Basisblöcke mit alternativen Überdeckungen. Dabei wird angenommen, dass diese alternativen Überdeckungen den Anforderungen des CSP-Modells $COVER_{ffp}$ genügen: notwendige Datentransferpfade (abgesehen von Spillcode) sollten also schon vollständig integriert sein und partielle FMOs nur noch in Form von Sub-FMOs vorkommen. Weiterhin sollten nur noch FMOs mit LIR-Operatoren vorkommen. Dies entspricht der Klasse alternativer Überdeckungen, die durch GIA generiert werden und hier auch als Eingabe dienen.

Die Integration von Instruktionsauswahl, Instruktionsanordnung und Registerallokation wird durch das Hinzufügen von Constraints für die Instruktionsanordnung und die Registerallokation zu den bestehenden Constraints des CSP-Modells $COVER_{ffp}$ realisiert. Die Integration der Phasen umfasst die folgende simultane Betrachtung der einzelnen Phasen:

- *Instruktionsauswahl*: Durch $COVER_{ffp}$ wird die korrekte Auswahl von Maschinenoperationen gewährleistet. Weiterhin wird die zulässige Auswahl von Maschinenoperationen bei Parallelisierung mit anderen FMOs bzgl. der Instruktionstypen überwacht.

- *Instruktionsanordnung*: Diese umfasst Constraints, die die zulässige Zuordnung von FMOs zu Instruktionszyklen bzgl. der Datenabhängigkeiten gewährleisten. Weiterhin müssen Ressourcenkonflikte bei der Parallelisierung von FMOs vermieden werden. Die Instruktionstypen der FMOs einer Maschineninstruktion müssen gleich sein. Darüber hinaus wird hier die korrekte Parallelisierung von AGU-Operationen gesteuert.

- *Registerallokation*: Registerfiles werden so ausgewählt, dass gültige Maschinenoperationen gebildet und die instruktionstypbedingten Restriktionen bzgl. der Verwendung von Registerfiles in FMOs eingehalten werden (diese Constraints sind schon durch das CSP-Modell $COVER_{ffp}$ gegeben). Die Registerallokation umfasst nicht die Generierung von Spillcode (dies wird erst in einer nachgeschalteten Phase durchgeführt, da ein exaktes CSP-Modell, das die Integration von Spillcode für die ADSP2100-Familie umfasst, nicht mehr mit akzeptablen

Optimierungszeiten realisierbar ist). Es wird aber ein Kostenmodell für den Registerdruck einer bestimmten Anordnung von Maschinenoperationen definiert, mit der Intention, den notwendigen Spillcode möglichst gering zu halten.

Die im Folgenden beschriebenen Optimierungstechniken werden mit I^3R ($I^3R = IIIR$ = integrierte Instruktionsauswahl, Instruktionsanordnung und Registerallokation) bezeichnet und bauen alle auf demselben CSP-Grundmodell auf, das mit I^3R_{CSP} bezeichnet wird. Die unterschiedlichen Ausprägungen von Optimierungstechniken aus I^3R ergeben sich durch die Adaption unterschiedlicher Labelingstrategien und Kostenmodelle des CSP-Modells:

- *Labelingstrategien*: Es werden unterschiedliche Labelingstrategien adaptiert, um die Auswirkungen auf die Laufzeit der Optimierung zu untersuchen. Ziel ist die Entwicklung einer möglichst effizienten Strategie. Die entsprechende Ausprägung von I^3R wird dann mit I^3R_l bezeichnet, gemäß der eingesetzten Labelingstrategie l.
- *Kostenmodelle*: Es werden zwei Kostenmodelle untersucht. Im ersten Kostenmodell wird nur die Minimierung der Instruktionszyklen betrachtet. Das zweite erweitert das erste Kostenmodell um ein Maß des Registerdrucks, um durch dessen Minimierung auch die Generierung von Spillcode der nachgeschalteten Registerallokation gering zu halten. Das Ziel ist hier also die Verbesserung der Codegüte. I^3R wird unter der Anwendung des ersten Kostenmodells mit I^3R_l und unter Anwendung des zweiten Kostenmodells mit I^3R_{l+ra} notiert (je nach eingesetzter Labelingstrategie l).
- *Partitionierung*: Die Anwendung der Partitionierung von Basisblöcken führte in GIA schon dazu, dass die Laufzeiten für große Basisblöcke drastisch reduziert wurden. Resultate waren immer gleich bzw. sehr nahe dem Optimum. Es wird auch bei I^3R ein guter Kompromiss zwischen Laufzeit und Codegüte erwartet. Techniken, die die Partitionierung nutzen, werden in der Form I^3R_{l+part} bzw. durch $I^3R_{l+ra+part}$ notiert.

Die Resultate von I^3R werden im Gesamtkontext der Codegenerierung von COCOON für die ADSP2100-Familie betrachtet, die mit $COCOON_{adsp2100}$ bezeichnet wird. Das Resultat von $COCOON_{adsp2100}$ ist ein CoLIR-Programm, das sich unmittelbar in lauffähigen Assemblercode transformieren läßt. Die erzielte Codegüte wird mit der Codegüte von *handgeneriertem* und *compilergeneriertem* Code der DSPStone-Benchmarks verglichen.

Das Kapitel ist wie folgt gegliedert:

- In Kapitel 7.1 wird ein Überblick über die Teilphasen von I^3R und deren Einbettung in den gesamten Codegenerierungsprozess von $COCOON_{adsp2100}$ gegeben.
- Kapitel 7.2 umfasst eine Einführung des CSP-Modells I^3R_{CSP}.

- Die Beschreibung der Entwicklungsstufen von effizienten Suchstrategien ist Gegenstand des Kapitels 7.3. Dies umfasst die schrittweise Entwicklung von intuitiven, aber ineffizienten Strategien, bis hin zu komplexeren, effizienteren Labelingstrategien. Die in Kapitel 7.3 beschriebenen Strategien machen noch keinen Gebrauch vom Registerdruckmodell. Die nach I^3R eingesetzten Optimierungstechniken werden beschrieben und die resultierende Codegüte untersucht.
- In Kapitel 7.4 wird I^3R_{CSP} um ein Registerdruckmodell erweitert und dessen Auswirkung bzgl. der Verbesserung der Codegüte untersucht. Es wird gezeigt, wie sich das erweiterte Modell auf die Laufzeiten auswirkt.
- Das Konzept der Partitionierung wird in Kapitel 7.5 bzgl. der Reduzierung der Laufzeiten und der erhaltenden Codegüte untersucht.
- In Kapitel 7.6 wird die Auswirkung auf die Codegüte gezeigt, wenn durch GIA nur eine einzige optimale Lösung an I^3R übergeben wird.
- Kapitel 7.7 enthält Bemerkungen zur Existenz von Lösungen zu I^3R_{CSP}.

7.1. Überblick über I^3R und dessen Integration in COCOON

Es werden die Teilphasen von I^3R beschrieben und die Einbettung von I^3R in den gesamten Codegenerierungsprozess von $COCOON_{adsp2100}$ erläutert (siehe auch Abb. 7.1):

1. Ein Programm der LANCE-IR wird in ein IR-reines CoLIR-Programm CoLIR1 transformiert. Dieses wird von GIA auf eine Menge optimaler alternativer Überdeckungen abgebildet, die durch CoLIR2 gegeben sind.
2. CoLIR2 ist die Eingabe von I^3R. Die Optimierungstechniken von I^3R werden pro Basisblock angewendet und umfassen die folgenden Teilphasen:
   a) Generierung des CSP-Modells $COVER$ (man erinnere sich, dass in der letzten Teilphase von GIA die Constraints entkoppelt werden).
   b) Die Erweiterung auf das CSP-Modell I^3R_{CSP} und die Optimierung werden für jeden einzelnen Basisblock b durchgeführt:
      i. Pro Basisblock ist schon das CSP $COVER(b)$ durch die globale Generierung von $COVER(fun)$ gegeben. Es erfolgt jeweils die lokale Erweiterung auf das CSP $COVER_{ffp}(b)$ und dann die lokale Erweiterung auf das gesamte CSP $I^3R_{CSP}(b)$. Für die zweite Variante des Kostenmodells werden hier zusätzliche Constraints für das Modell des Registerdrucks generiert.
      ii. Anwendung der Optimierungsprozedur $minimize/2$ auf Basisblock b. Minimiert wird die Anzahl der Instruktionszyklen I_{max}. In der zweiten betrachteten Variante des Kostenmodells wird $I_{max} + RD$ minimiert, wobei RD der Wert des Registerdrucks ist.

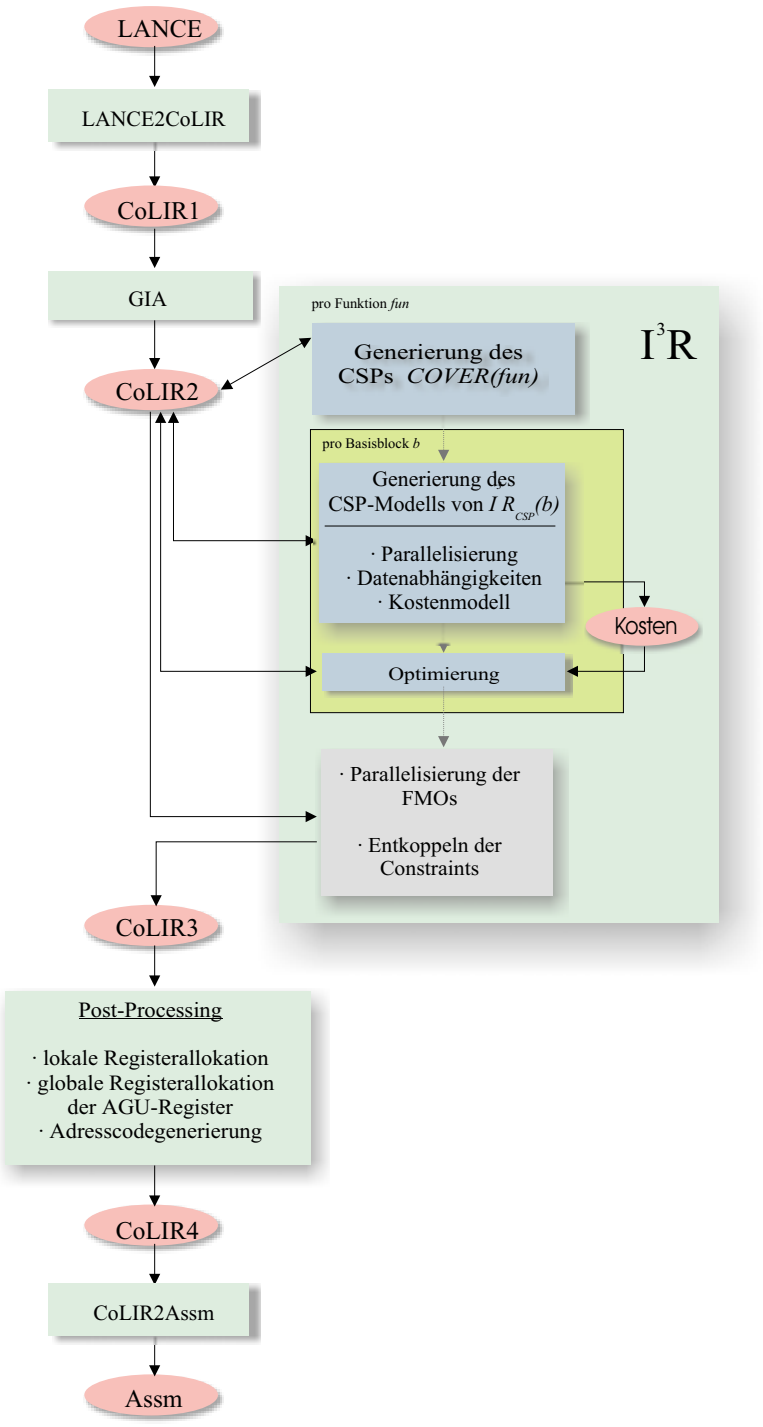


Abbildung 7.1.: Überblick über die Integration von I^3R in $COCOON_{adsp2100}$ und über die Teilphasen von I^3R.

c) Zu jedem Basisblock b wird ein Basisblock b' generiert, in dem die FMOs gemäß ihrer Zuordnung zu Instruktionszyklen parallelisiert sind. Das Resultat ist ein CoLIR-Programm CoLIR3. In diesem CoLIR-Programm sind alle Domainvariablen gebunden.

3. Postprocessingphasen:

   a) Lokale Registerallokation über den Nicht-AGU-Registern.

   b) Globale Registerallokation der AGU-Register.

   c) Adresscodegenerierung.

   Resultat ist ein CoLIR-Programm CoLIR4, das zu vollständigem und lauffähigem Assemblercode korrespondiert.

7.2. Das CSP-Modell I^3R_{CSP}

In diesem Kapitel wird das CSP-Modell I^3R_{CSP} spezifiziert, das allen Ausprägungen von I^3R zugrunde liegt. Dieses Modell umfasst das Teil-CSP-Modell $COVER_{ffp}$ und spezifiziert zusätzlich pro Basisblock b die folgenden Teil-CSP-Modelle:

- *Bildung legaler Maschineninstruktionen*: Dies umfasst die restriktive Verwendung von Ressourcen bei der Parallelausführung, die sich durch die Instruktionstypen ergeben, Ressourcenkonflikte bzgl. der Benutzung von Funktionseinheiten bei Parallelausführung, sowie spezifische Bedingungen für die Parallelausführung von AGU-Operationen.

- *Zeitliche Abhängigkeiten zwischen den FMOs bzgl. der Datenabhängigkeiten*: FMOs, die bestimmte Datenabhängigkeiten besitzen, müssen in einer bestimmten zeitlichen Reihenfolge ausgeführt werden. Diese müssen auch die Ausführungszeiten von Maschinenoperationen berücksichtigen.

- *Kostenmodell:* Constraints zur Realisierung des Kostenmodells, das in der ersten Variante nur die Minimierung der Instruktionszyklen eines Basisblocks umfasst und noch keinen Registerdruck betrachtet.

Jede FMO $f \in FMO(b)$ eines Basisblocks b wird mit einer Domainvariablen I_f assoziiert, deren Domain die Menge der möglichen Instruktionszyklen (Ausführungszeitpunkte) umfasst, denen f potenziell zugeordnet werden kann. Weiterhin wird jedem Basisblock eine Domainvariable $I_{b_{max}}$ zugeordnet, die der zu minimierenden Anzahl an Instruktionszyklen entspricht. Es wird davon ausgegangen, dass alle Instruktionszyklen Werte größer oder gleich Null zugeordnet werden: $I_f \geq 0$ für alle $f \in FMO(b)$. Diese Domainvariablen werden im Folgenden auch als *I-Variablen* bezeichnet. Die Menge aller I-Variablen eines Basisblocks b wird mit $I(b) = \{I_f | f \in FMO(b)\}$ bezeichnet.

I-Variablen

7.2.1. CSP-Modelle $COVER$ und $COVER_{ffp}$

Durch das CSP-Modell $COVER$ wird die Auswahl legaler Maschinenoperationen und die Existenz von Datentransfers gewährleistet. Weiterhin wird für die Techniken aus I^3R die explizite Existenz von allen Datentransfer-Operationen zwischen Setzungen und Ihren Benutzungen gefordert, so dass also keine Lösungen existieren, in denen noch Datentransfer-Operationen eingefügt werden müssen. Außerdem müssen auch alle komplexen Operationen bereits gebildet worden sein: Es gibt also keine FMO, die Nicht-Sub-FMO ist und auf eine partielle Maschinenoperation abgebildet werden kann. Diese Anforderungen entsprechen der Erweiterung von $COVER$ auf das CSP-Modell $COVER_{ffp}$ (siehe Kapitel 5.5.1).

7.2.2. Constraints zur Bildung legaler Maschineninstruktionen

Die Constraints zur Bildung legaler Maschineninstruktionen lassen sich in die folgenden Gruppen untergliedern:

- Allgemeine Grundvoraussetzungen zur Parallelisierung von FMOs.
- Wechselseitige Beziehungen zwischen verschiedenen FMOs, die derselben Maschineninstruktion zugeordnet werden.
- Spezifische Parallelisierungsbedingungen von AGU-Operationen.

Instruktionstypbedingte Restriktionen bzgl. der Ressourcenverwendung in einer FMO sind schon auch durch das CSP-Modell $COVER$ gegeben (sie brauchen daher nicht erneut sprezifiziert zu werden).

Grundvoraussetzungen zur Parallelisierung

Wenn zwei FMOs f_1 und f_2 parallelisiert werden, was durch die Erfüllung des Constraints $I_{f_1} = I_{f_2}$ gegeben ist, dann müssen beide FMOs den gleichen Instruktionstyp ($IT_{f_1} = IT_{f_2}$) besitzen und die FE-Variablen ungleich sein ($FE_{f_1} \neq FE_{f_2}$), da es ansonsten einen Ressourcenkonflikt gibt. Für jedes Paar f_i, $f_j \in FMO(b)$ wird daher ein Constraint $I_i = I_j \Rightarrow IT_i = IT_j \wedge FE_i \neq FE_j$ generiert.

Weiterhin gibt es für bestimmte Prozessoren Restriktionen, wieviele Maschinenoperationen gleichzeitig schreibend bzw. lesend auf Registerfiles zugreifen können. Die lesenden Zugiffe werden bei der ADSP2100-Familie durch die Instruktionstypen korrekt gehandhabt. Bei den schreibenden Zugriffen kann es zu Konflikten kommen, da durch die Instruktionstypen nicht das gleichzeitige Schreiben in ein Registerfile verhindert wird: So können z.B. beide Setzungen einer parallel ausgeführten ALU- und LOAD-Operation dem Registerfile ar zugeordnet werden. Um diesen Ressourcenkonflikt zu vermeiden, wird für jedes Paar f_i, $f_j \in FMO(b)$ ein Constraint $I_i = I_j \Rightarrow RF_i \neq RF_j$ erzeugt.

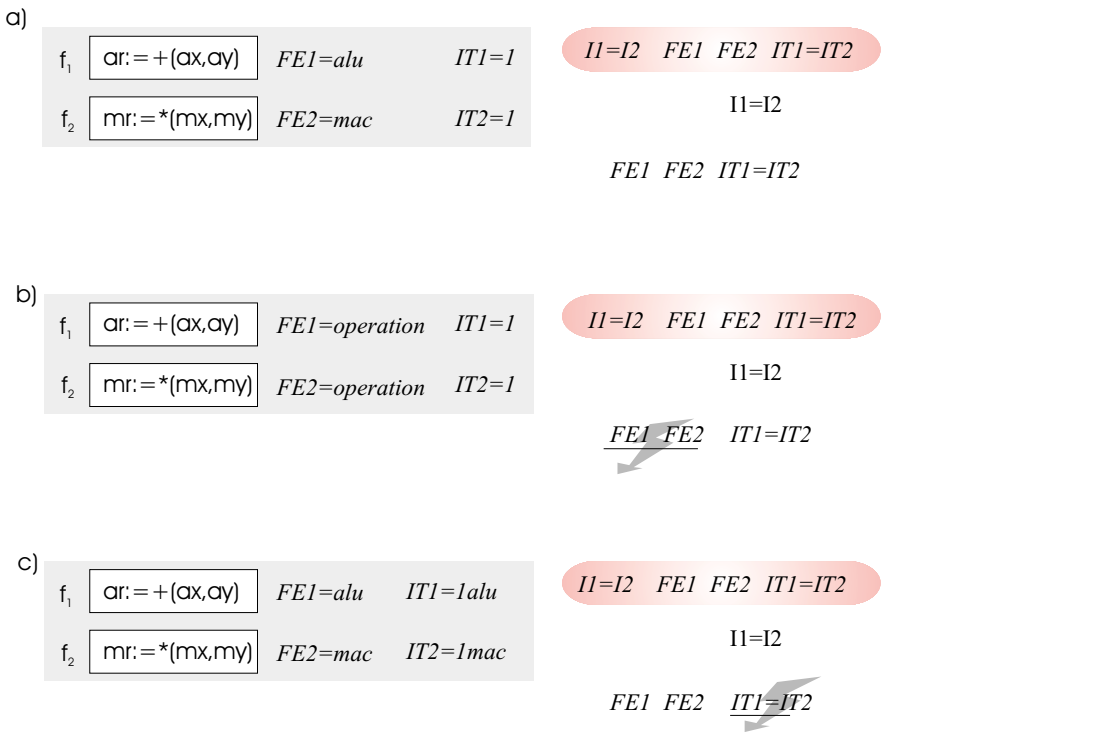
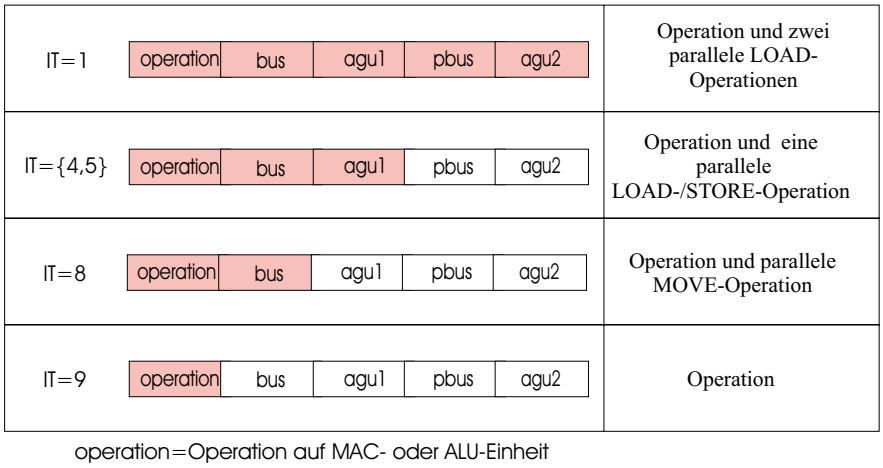


Abbildung 7.2.: Noch nicht erkannte Konflikte bei der Parallelisierung von FMOs.

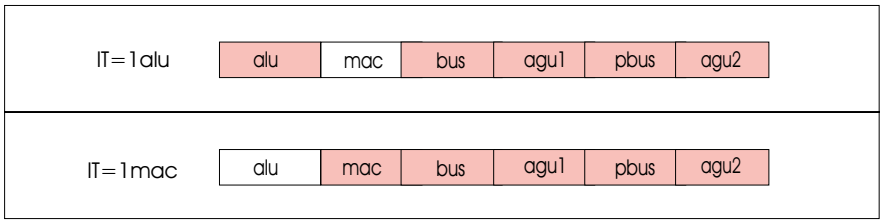
Instruktionstypbedingte Restriktionen zwischen FMOs

Die Ressourcen von FMOs müssen den Restriktionen genügen, die sich durch ihren Instruktionstyp ergeben. Für jede einzelne FMO sind diese Restriktionen schon durch die Definition von $fmo/5$ spezifiziert worden. Es gibt aber weitere Restriktionen, die sich zwischen unterschiedlichen FMOs einer Maschineninstruktion ergeben. Instruktionstyp 1 der ADSP2100-Familie läßt es z.B. zu, dass eine Operation auf der ALU- oder der MAC-Einheit parallel zu zwei LOAD-Operationen und zu zwei AGU-Operationen ausgeführt werden kann. Es ist aber nicht möglich, innerhalb des Instruktionstypen 1 Operationen auf der ALU- und der MAC-Einheit parallel auszuführen. Die bisher spezifizierten Constraints können diesen Konflikt leider nicht erkennen. Dies ist in Abb. 7.2 a) verdeutlicht: Die FMOs f_1 und f_2 werden beide dem gleichen Instruktionszyklus zugeordnet und sind beide vom Instruktionstyp 1. Da die Funktionseinheiten verschieden sind, wird kein Konflikt bzgl. der Parallelausführung durch die bestehenden Constraints erkannt.

Notwendig ist ein Mechanismus, der die Exklusivität der Benutzung der ALU-Einheit oder der MAC-Einheit im Instruktionstyp 1 spezifiziert. Betrachtet man einmal die Instruktionstypen in Abb. 7.3 a), so gibt es Regionen im korrespondierenden Instruktionswort, denen bestimmte Operationen einer bestimmten Funktionseinheit zugeordnet werden (auch wenn diese Regionen im realen Instruktionswort nicht so sauber getrennt vorliegen, wie es in Abb. 7.3 dargestellt ist, und je nach Instruktionstyp auch eine eigene Struktur im Instruktionswort aufweisen). Diese Regionen werden im Fol-



(a) Teilmenge der Instruktionstypen der ADSP2100-Familie und deren erlaubte Slots (grau dargestelt).



(b) Neue Instruktionstypen mit eindeutiger Zuordnung von Funktionseinheiten zu Slots, die den Instruktionstyp 1 ersetzen.

Abbildung 7.3.: Modell der Instruktionstypen.

genden mit dem Begriff *Slots* bezeichnet. Je nach Instruktionstyp können bestimmte Slots benutzt werden und andere nicht. Slots, die benutzt werden dürfen, sind in der Abbildung grau unterlegt (Slots, die nicht benutzt werden, sind in den realen Instruktionswörtern natürlich nicht vorhanden). Zu parallelisierende FMOs können gemäß ihres Typen nur bestimmten Slots zugeordnet werden. Die arithmetische bzw. logische Operation z.B. kann im Instruktionstypen 1 nur dem Slot 'operation' zugeordnet werden.

Slots

Um nun die Beziehung zwischen Funktionseinheiten, Instruktionstypen und Slots herzustellen, werden drei alternative Lösungsmöglichkeiten vorgeschlagen:

1. *Variante*: Man erkennt in Abb. 7.3 a), dass für die ALU- und die MAC-Einheit nur ein Slot existiert, dem sie zugeordnet werden können. Alle weiteren Slots können eindeutig Funktionseinheiten zugeordnet werden. Man kann nun anstelle der Funktionseinheiten auch unmittelbar die möglichen Slots in der Definition von $fmo/5$ angeben. Somit werden die FMOs der ALU- und MAC-Einheit einfach einer virtuellen Funktionseinheit $operation$ zugeordnet. In diesem Fall greifen die bestehenden Constraints, und der Konflikt wird erkannt (siehe Abb. 7.2 b)). Diese Variante ist sicherlich für die ADSP2100-Familie mit dem geringsten Aufwand umzusetzen.

2. *Variante*: Die Slots für die ALU- und MAC-Einheit werden explizit in den Instruktionstypen modelliert, und man führt für die unterschiedlichen erlaubten Mengen von Slots einen eigenen Instruktionstypen ein. In Abb. 7.3 b) wurden die neuen Instruktionstypen '1mac' und '1alu' eingeführt, die nun den Instruktionstyp 1 ersetzen. Dies muss auch für alle weiteren Instruktionstypen vollzogen werden, in denen mehrere Funktionseinheiten einem Slot zugeordnet werden. Für die neu eingeführten Instruktionstypen greifen die bestehenden Constraints für das Beispiel: da die Gleichheit der IT-Variablen nicht mehr gegeben ist, wird dieses Constraint jetzt verletzt (siehe Abb. 7.2 c)). Hier entsteht zwar der Aufwand, neue Typen einzuführen, ansonsten bleibt die Modellierung hier aber transparent, da die Zuordnung der FMOs zu den originalen Funktionseinheiten erhalten bleibt. Es geht lediglich ein Abstraktionsmittel verloren, das im Zusammenlegen bestimmter Instruktionstypen besteht.

3. *Variante*: In der dritten Variante werden die Slots explizit durch Domainvariablen modelliert und die Beziehung zwischen Slots, Funktionseinheiten und Instruktionseinheiten muss über entsprechende Constraints spezifiziert werden. Diese Variante ist mit dem höchsten Modellierungsaufwand verbunden und würde neben zusätzlichen Domainvariablen für die Slots auch zusätzliche Constraints erfordern.

Für die ADSP2100-Familie ist die erste Variante gewählt worden, da dies mit dem geringsten Aufwand zur Änderung des Modells verbunden ist und das Modell auch weiterhin verständlich bleibt. Welche Variante zu wählen ist, hängt sicherlich von dem zu modellierenden Instruktionssatz ab. Berücksichtigt man, dass das Prozessormodell gegebenenfalls automatisch generiert werden soll, ist die zweite Variante vorzuziehen, da hier die Korrespondenz zu den realen Ressourcen der Architektur eher gegeben ist.

```
load_fmo(ld,D,[A,d],bus,IT) :-    Regel 1
   A::[i1,i2],
   dreg(D),
   IT::[4,12].
load_fmo(ld,D,[i2,p],bus,IT) :-   Regel 2
   dreg(D),
   IT::[5,13].
load_fmo(ld,D,[A,d],bus,1) :-     Regel 3
   A::[i1,i2],
   D::[ax,mx]
load_fmo(ld,D,[i2,p],pbus,1) :-   Regel 4
   D::[ay,my].
load_fmo(ld,D,[addr,d],bus,3) :-  Regel 5
   reg(D).
```

Abbildung 7.4.: Definition der LOAD-Operationen der ADSP2100-Familie.

In Abb. 7.5 ist dargestellt, wie die Constraints im Zusammenspiel eine Parallelausführung erkennen und die Domainvariablen gemäß der Restriktionen einschränken. Dazu wird die Parallelisierung zweier FMOs der Klasse $load$ betrachtet (in Abb. 7.4 ist nochmal die Definition der LOAD-Operationen gezeigt). Es wird angenommen, dass der aktuelle Constraintstore aus dem Constraint $I1 = I2 \Rightarrow FE1 \neq FE2 \wedge IT1 = IT2$ (die Domainvariablen FE_{f_i}, IT_{f_i} und I_{f_i} werden durch die Variablen FEi, ITi und Ii dargestellt) und den Constraints $load\_fmo(f_1)$ bzw. $load\_fmo(f_2)$ besteht. Angenommen, dass das Constraint $I1 = I2 \wedge FE1 = bus$ zum Constraintstore hinzugefügt wird, welches die Parallelausführung der FMOs f_1 und f_2 fordert und die Funktionseinheit für das Laden von f_1 an bus bindet (in der Modellierung werden Busse als die ausführenden Funktionseinheiten von Datentransfers betrachtet). Dies führt zu Reaktivierungen der Constraints (im Constraintstore), und die einzelnen Reaktivierungsschritte werden nun im einzelnen nachvollzogen:

1. *Schritt*: Durch die Gleichsetzung von $I1$ und $I2$ ($I1 = I2$) und dem Constraint $I1 = I2 \Rightarrow FE1 \neq FE2 \wedge IT1 = IT2$ folgt unmittelbar $FE1 \neq FE2 \wedge IT1 = IT2$.

2. *Schritt*: Aus $FE1 = bus$ und $FE1 \neq FE2$ folgt unter der Beachtung des aktuellen Domains $D(FE2) = \{bus, pbus\}$, dass $D(FE2)$ jetzt auf $\{pbus\}$ gesetzt wird, also $FE2 = pbus$ gilt.

3. *Schritt:* Die Bindungen der Domainvariablen $FE1$ und $FE2$ führen zur Reaktivierung der Constraints $load\_fmo(f_1)$ und $load\_fmo(f_2)$. Gemäß der Bindungen von $FE1$ und $FE2$ führt dies zu $IT1 \in \{1, 4, 5, 12, 13\}$ (Regeln 1,2,3 in Abb. 7.4) und $IT2 = 1$ (Regel 4 in Abb. 7.4).

4. *Schritt*: Es kommt zur Reaktivierung des Constraints $IT1 = IT2$ durch die Bindung der Variablen $IT1$ und $IT2$, was zur Bindung von $IT1$ an den Instruktionstyp 1 führt ($IT1 = 1$).

5. *Schritt*: Durch die Bindung von $IT1$ wird $load\_fmo(f_1)$ reaktiviert, und die Ressourcen werden gemäß Regel 3 gebunden.

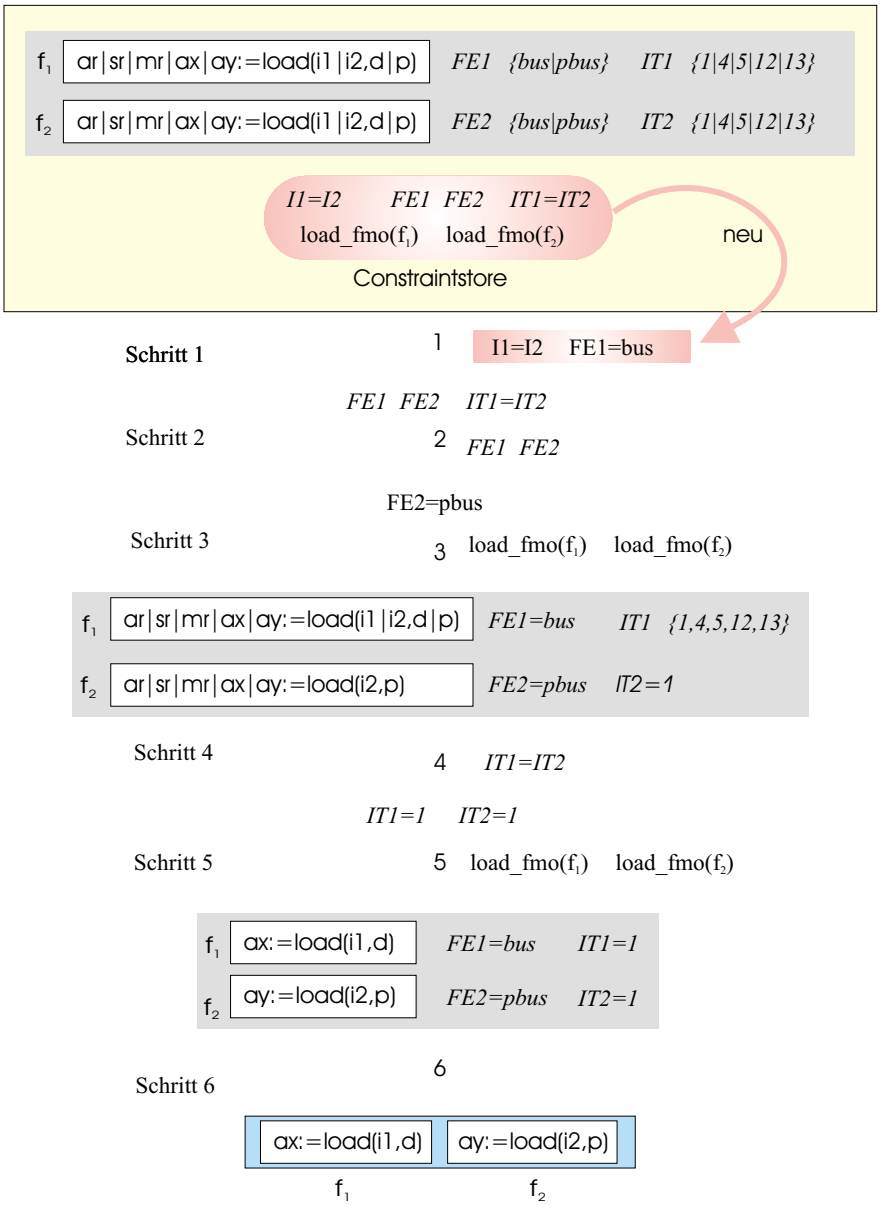


Abbildung 7.5.: Wirkungsweise der Constraints bei Parallelisierung von FMOs.

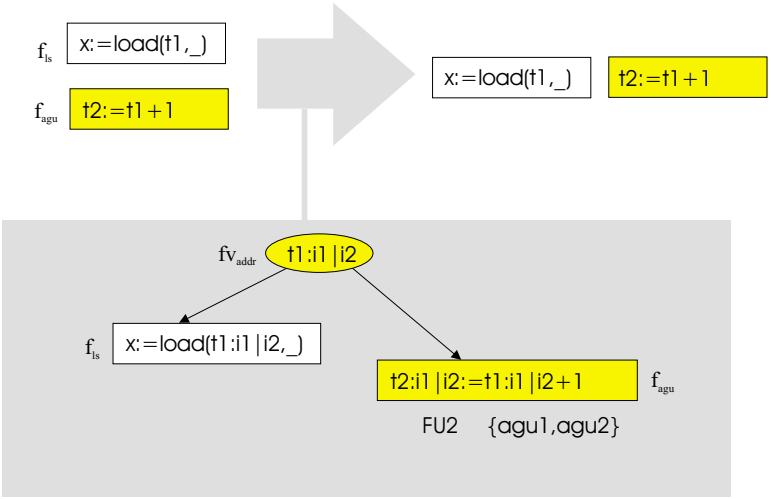


Abbildung 7.6.: Voraussetzung zur Parallelisierung von AGU-Operationen.

6. *Schritt*: Die Ressourcen beider FMOs sind nun fest gebunden und die Bedingungen für eine Parallelisierung sind erfüllt.

Spezifische Parallelisierungsbedingungen von AGU-Operationen

Eine Autoinkrement- bzw. Autodekrementoperation f_{agu}, die potenziell auf einer der AGU-Einheiten ausgeführt werden kann (d.h. $D(FE_{f_{agu}}) \cap \{agu1, agu2\} \neq \emptyset$), kann parallel zu einer LOAD- bzw. STORE-Operation f_{ls} ausgeführt werden, wenn f_{agu} den Inhalt $addr$ eines Adressregisters inkrementiert, den f_{ls} zur Referenzierung des Speichers benutzt (siehe auch Abb. 7.6, in der $addr$ in der Programmvariablen $t1$ abgelegt wird). Adresse $addr$ ist das erste Argument von f_{ls} ($fv_{addr} \rightarrow^{[1]} f_{ls} \in DFG(b)$) und das erste Argument von f_{agu} ($fv_{addr} \rightarrow^{[1]} f_{agu} \in DFG(b)$). Der Wert $addr$ liegt in einem der Adressregister $i1$ oder $i2$ vor und wird dort auch von den FMOs f_{ls} und f_{agu} benutzt.

Es muss sichergestellt werden, dass eine AGU-Operation nur unter der gezeigten Konstellation mit einer LOAD- oder STORE-Operation parallelisiert wird. Diese Konstellation kann statisch geprüft werden. Liegt sie vor, so wird f_{agu} den AGU-Einheiten zugeordnet, indem ein Constraint $FE \in \{agu1, agu2\}$ generiert wird. Zum einen kann f_{agu} mit f_{ls} parallelisiert werden ($I_{f_{agu}} = I_{f_{ls}}$), zum anderen kann f_{agu} separat ausgeführt werden und wird in diesem Fall dem Instruktionstyp 21 zugeordnet (Maschineninstruktionen dieses Typs nehmen auf der ADSP2100-Familie nur eine einzige AGU-Operation auf). Weiterhin wird es durch die implizite Antiabhängigkeit von f_{agu} und f_{ls} notwendig, dass f_{agu} bei Nichtparallelausführung mit f_{ls} nach f_{ls} ausgeführt wird.

Die genannten Bedingungen werden durch folgendes Constraint für f_{agu} und f_{ls} realisiert:

$$I_{f_{agu}} \neq I_{f_{ls}} \Leftrightarrow (IT_{f_{agu}} = 21 \wedge I_{f_{agu}} > I_{f_{ls}})$$

Liegt die Konstellation für f_{agu} nicht vor, kann f_{agu} nicht parallelisiert werden und es wird das Constraint $FE_{f_{agu}} \in \{agu1, agu2\} \Rightarrow IT_{f_{agu}} = 21$ generiert. Die Prämisse ($FE_{f_{agu}} \in \{agu1, agu2\}$) ist notwendig, da Operationen zur Adressberechnung nicht zwingend den AGU-Einheiten zugeordnet werden müssen.

Es gibt nun zwei Sonderfälle:

1. *Fall*: Es gibt mehr als eine FMO der Klasse $load$ oder $store$, die $addr$ als Adresse benutzen. In diesem Fall darf das Inkrementieren bzw. Dekrementieren erst mit der LOAD-/STORE-Operation ausgeführt werden, die dem höchsten Instruktionszyklus zugeordnet wird. Die Instruktionszyklen der LOAD-/STORE-Operationen $f_1, ..., f_n$, für die $fv_{addr} \rightarrow^{[1]} f_i \in DFG(b)$ und $f_i \in \{f_1, ..., f_n\}$ gilt, sei durch die Liste $Is = \{I_{f_1}, ..., I_{f_n}\}$ gegeben. Es wird das folgende Constraint generiert:

   $$maxlist(Is, I_{max}) \wedge (I_{f_{agu}} \neq I_{max} \Leftrightarrow (IT_{f_{agu}} = 21 \wedge I_{f_{agu}} > I_{max}))$$

   Darin ist $maxlist(Is, I_{max})$ ein Constraint der FD-Bibliothek von ECLiPSe, das zusichert, dass I_{max} der maximale Wert der Liste Is zugeordnet wird.

2. *Fall:* Es gibt mehr als eine AGU-Operation, die $addr$ als Operanden benutzen und dessen Wert modifizieren. Dies ist durch Autoinkrement- bzw. Autodekrementoperationen nicht mehr sinnvoll zu unterstützen, da durch das erste Inkrementieren bzw. Dekrementieren der Wert von $addr$ modifiziert wird und der ursprüngliche Wert für alle anderen Benutzungen vorher in andere Register gerettet werden muss. Sinnvoller ist es, solche CSEs schon im Vorfeld aufzuspalten und jede Benutzung einem eigenen AGU-Register zuzuordnen. In den DSPStone-Benchmarks kommen solche Fälle nicht vor. Kommt im Allgemeinen so ein Fall vor, werden alle AGU-Operationen dem Instruktionstyp 21 zugeordnet (es wird also keine der Operationen parallelisiert). Die notwendigen Kopien von $addr$ werden durch die in der Postprocessingphase durchgeführten Registerallokation eingefügt.

7.2.3. Constraints der Datenabhängigkeiten

Wie schon in Kapitel 4.3.2 Seite 95 erwähnt, spiegeln die Datenabhängigkeitsgraphen zeitliche Abhängigkeiten zwischen FMOs (bzw. Maschinenoperationen) eines Basisblocks wider. Im Folgenden werden nicht die unterschiedlichen Klassen von Setzungen und Benutzungen unterschieden (z.B. DDGs über Bezeichnerzuordnungen oder über den Speicherzugriffen), sondern die entsprechenden Beziehungen werden in einem gemeinsamen Datenabhängigkeitsgraphen $DDG(b)$ verschmolzen. Die drei Arten von Datenabhängigkeiten (true dependency, false dependency, output dependency) implizieren die folgenden zeitlichen Abhängigkeiten zwischen FMOs[1]:

[1] Dabei wird davon ausgegangen, dass eine Maschinenoperation innerhalb eines Instruktionszyklus abgearbeitet werden kann. Es ist kein Problem, das Modell auch auf Ausführungszeiten, die größer als ein Instruktionszyklus sind, umzustellen.

1. *true dependency*: Für eine Kante $f \rightarrow^{true,pos} f' \in DDG(b)$ benutzt f' den Wert, den f generiert, als Argument an Position pos. Daraus ergibt sich die folgende zeitliche Abhängigkeit zwischen den Instruktionszyklen für f und f': $I_f < I_{f'}$.
2. *false dependency*: Für eine Kante $f \rightarrow^{false,pos} f' \in DDG(b)$ schreibt f' einen Wert, den f benutzt. Daraus ergibt sich, dass f' nicht früher als f ausgeführt werden darf, da f ansonsten den falschen Wert lesen würde: $I_f \leq I_{f'}$.
3. *output dependency*: Für eine Kante $f \rightarrow^{output,pos} f' \in DDG(b)$ setzt f' eine Ressource, die auch von f gesetzt wird. Daraus ergibt sich, dass f' nicht früher zu f ausgeführt werden darf, da ansonsten die Benutzungen von f' den falschen Wert lesen würden, und nicht zeitgleich, da sich ansonsten ein Ressourcenkonflikt ergibt: $I_f < I_{f'}$.

7.2.4. Kostenmodell von I^3R ohne Registerdruck

Das Kostenmodell ohne Beachtung des Registerdrucks ist relativ einfach. Es besteht in der Minimierung der Domainvariablen $I_{b_{max}}$ und muss lediglich garantieren, dass alle Instruktionszyklen I_f von FMOs $f \in FMO(b)$ kleiner gleich $I_{b_{max}}$ sind:

$$I_f \leq I_{b_{max}}$$

Das um den Registerdruck erweiterte Kostenmodell wird in Kapitel 7.4.1 beschrieben.

7.2.5. Das CSP-Modell I^3R_{CSP} im Überblick

Zur Generierung des CSPs von $I^3R_{CSP}(b)$ zu einer Funktion fun mit $b \in fun$ wird zuerst $COVER(fun)$ global für die Funktion generiert. Die weiteren Constraints für $I^3R_{CSP}(b)$ werden jeweils lokal pro Basisblock generiert (so auch die Erweiterungen auf $COVER_{\mathit{fffp}}(b)$).

Die Domainvariablen von $I^3R_{CSP}(b)$

Die Domainvariablen von $I^3R_{CSP}(b)$ ergeben sich aus der Menge der Domainvariablen aus $COVER_{\mathit{fffp}}(b)$ und der Menge der Instruktionszyklen $I(b) \cup \{I_{b_{max}}\}$.

Die Constraints von $I^3R_{CSP}(b)$

Zu den Constraints aus $COVER_{\mathit{fffp}}(b)$ werden noch die folgenden Constraints generiert:

- *Bildung legaler Maschineninstruktionen:*
  - Ressourcenkonflikte bei Parallelausführung:
    - $I_i = I_j \Rightarrow IT_1 = IT_2 \wedge FE_i \neq FE_j$

$* \; I_i = I_j \Rightarrow RF_i \neq RF_j$

- Die Restriktionen der Ressourcenverwendung in einer FMO, die sich aus den Instruktionstypen ergeben, werden durch das Prädikat $fmo/5$ abgedeckt. Noch nicht erfasste Ressourcenkonflikte, die sich durch instruktionstypbedingte Restriktionen zwischen verschiedenen FMOs ergeben, werden durch eine alternative Modellierungsart der Instruktionstypen erfasst.
- Die Parallelisierung von AGU-Operationen setzt voraus, dass die spezifizierten Vorbedingungen erfüllt sind. Falls nicht, wird ein Constraint

  $$FE_{f_{agu}} \in \{agu1, agu2\} \;\Rightarrow\; IT_{f_{agu}} = 21$$

  generiert, was die separate Ausführung der Operation bewirkt, wenn sie einer der AGU-Einheiten zugeordnet wird. Sonst werden - je nachdem, ob nur eine LOAD-/STORE-Operation die von f_{agu} modifizierte Adresse benutzt (1. Fall), oder eine Menge von LOAD-/STORE-Operationen die Adresse benutzen (2. Fall) - die folgenden Constraints generiert:

  1. Fall:
     $$I_{f_{agu}} \neq I_{f_{ls}} \;\Leftrightarrow\; (IT_{f_{agu}} = 21 \wedge I_{f_{agu}} > I_{f_{ls}})$$
  2. Fall:
     $$maxlist(Is, I_{lsmax}) \wedge \; (I_{f_{agu}} \neq I_{lsmax} \;\Leftrightarrow\; (IT_{f_{agu}} = 21 \wedge I_{f_{agu}} > I_{lsmax}))$$

- *Datenabhängigkeiten*:
  - *true dependency*: Aus $f \rightarrow^{true,pos} f' \in DDG(b)$ folgt $I_f < I_{f'}$.
  - *false dependency*: Aus $f \rightarrow^{false,pos} f' \in DDG(b)$ folgt $I_f \leq I_{f'}$.
  - *output dependency*: Aus $f \rightarrow^{output,pos} f' \in DDG(b)$ folgt $I_f < I_{f'}$.
- *Kostenmodell*: Für jede FMO $f \in FMO(b)$ wird ein Constraints $I_f \leq I_{b_{max}}$ generiert. Die Optimierungsfunktion minimiert den Wert der Domainvariablen $I_{b_{max}}$.

7.3. Minimierung der Instruktuktionszyklen (I^3R_l)

Unter dem Begriff I^3R_l wird die Menge von Optimierungstechniken erfasst, die als Kostenmodell die Minimierung der Instruktionszyklen $I_{b_{max}}$ zu gegebenem Basisblock b verwenden. Sie unterscheiden sich in der Anwendung der Labelingstrategie l, die während der optimierenden Suche benutzt wird.

In Abb. 7.7 ist das Optimierungsprädikat zur Realisierung von I^3R_l gezeigt. Darin gibt $cover\_vars/4$ zum Basisblock b die Domainvariablen der Mengen $STK(b)$, $FE(b)$ und $IT(b)$ zurück. Die Constraints des CSP-Modells I^3R_{CSP} werden lokal zu jedem Basisblock durch das Prädikat $iiir\_csp/3$ generiert, welches die Domainvariablen der Menge $I(b)$ und die Domainvariable $I_{b_{max}}$ zurückliefert. Die Optimierung wird (wie schon in GIA) durch das Prädikat $minimize/2$ durchgeführt. $l/4$ repräsentiert die jeweils eingesetzte Labelingstrategie über den Domainvariablen der Mengen $I(b)$, $STK(b)$, $FE(b)$ und $IT(b)$. Die in diesem Kapitel referenzierten Variablenbelegungen θ beziehen sich alle auf die Menge $I(b) \cup STK(b) \cup FE(b) \cup IT(b)$.

```
iiir(BB):-
  cover_vars(BB,STKs,FEs,ITs),
  iiir_csp(BB,Is,Imax),
  minimize(l(Is,STKs,FEs,ITs),Imax).
```

Abbildung 7.7.: Optimierungsprädikat zur Umsetzung von I^3R_l.

```
ls(Is,STKs,FEs,ITs):-
   labeling(Is),
   cut((labeling(FEs),
        labeling(STKs),
        labeling(ITs))).
```

Abbildung 7.8.: Intuitive Labelingstrategie $ls/4$.

7.3.1. Erste intuitive Labelingstrategien

Die erste betrachtete Labelingstrategie entspricht dem Konzept des VSV-Labeling, das in GIA eingeführt wurde (siehe Kapitel 6.3.2 S. 162): Es wird zunächst eine Menge S von Domainvariablen gelabelt, die die Kosten determinieren. Durch das Labeling der verbleibenden Domainvariablen bleibt zu zeigen, dass θ_S eine partielle Lösung ist, und bei Erfolg kann die Suche nach einer besseren Lösung unmittelbar über der Menge S fortgesetzt werden. Die erste Labelingstrategie nimmt entsprechend dazu die Aufteilung von $I(b)$, $STK(b)$, $FE(b)$ und $IT(b)$ in die folgenden beiden Mengen von Domainvariablen vor:

1. Es werden zuerst die I-Variablen aus $I(b)$ gelabelt, und man erhält eine partielle Variablenbelegung $\theta_{I(b)}$. Hierdurch erhält man eine Zuordnung aller FMOs aus b zu Instruktionszyklen, woraus sich unmittelbar der minimal mögliche Wert für $I_{b_{max}}$ ergibt.
2. Es bleibt zu zeigen, dass sich $\theta_{I(b)}$ zu einer Variablenbelegung θ fortsetzen lässt, die auch Lösung ist. Dazu müssen noch die Domainvariablen aus $STK(b)$, $FE(b)$ und $IT(b)$ erfolgreich gelabelt werden.

Der ECLiPSe-Code dieser Labelingstrategie, gegeben durch das Prädikat $ls/4$, ist in Abb. 7.8 dargestellt. Leider stellt sich hier heraus, dass das Labeling der I-Variablen aus $I(b)$ relativ viele ungültige partielle Variablenbelegungen $\theta_{I(b)}$ hervorbringt, die keine partiellen Lösungen repräsentieren. Der Grund, warum viele ungültige partielle Variablenbelegungen für $I(b)$ generiert werden, ist, dass Konflikte bzgl. der Parallelausführung oft erst durch das Labeln der FE-Variablen aus $FE(b)$ erkannt werden, da die Gültigkeit bzw. das Fehlschlagen des Constraints $FE_1 \neq FE_2$ erst definitiv feststeht, wenn eine der FE-Variablen FE_1 oder FE_2 instanziiert ist. Dies führt dazu, dass relativ viel Zeit für die Suche über $STK(b)$, $FE(b)$ und $IT(b)$ aufgebracht werden muss, da es hier zu vielen Fehlschlägen und Backtrackingschritten kommt. Mit dieser

```
lif(Is,STKs,FEs,ITs):-
   lif(Is,FEs),
   cut((labeling(STKs),labeling(ITs))).

lif([],[]).
lif([I|Is],[FE|FEs]):-
   indomain(I),
   indomain(FE),
   lif(Is,FEs).
```

Abbildung 7.9.: Verzahnte Labelingstrategie $lif/4$ (lif = label I- und F-Variable).

Strategie ist allein das Finden einer gültigen Lösung für kleine Benchmarks sehr zeitaufwendig (z.B. sind schon zum Finden von Lösungen für die DSPStone-Benchmarks cm und cu mehrere Minuten notwendig).

Die nächste Strategie, die angewandt wurde, generiert daher zuerst eine partielle Lösung zur Menge $STK(b) \cup FE(b) \cup IT(b)$ und optimiert dazu die Anordnung durch das Labeling von $I(b)$. Die optimierende Suche muss jetzt allerdings über dem Suchraum von $STK(b) \cup FE(b) \cup IT(b)$ und von $I(b)$ durchgeführt werden. Es werden zwar schnell Lösungen gefunden, doch deren Kosten können entsprechend der partiellen Variablenbelegung $\theta_{STK(b) \cup FE(b) \cup IT(b)}$ sehr weit entfernt vom Optimum liegen. Versuche haben gezeigt, dass eine Konvergierung gegen das Optimum auch für die kleinen Benchmarks sehr zeitintensiv ist (hier lag die Laufzeit für die Optimierung des DSPStone-Benchmarks cu bei über 15.000 CPU-Sekunden).

Aufgrund der obigen Beobachtung liegt es nahe, eine verzahnte Labelingstrategie über den Domainvariablen aus $STK(b) \cup FE(b) \cup IT(b)$ und $I(b)$ zu entwickeln, die eine Anordnung generiert und frühzeitig das Scheitern einer partiellen Anordnung sicherstellt.

7.3.2. Simultanes Labeling der I- und FE-Variablen

Die erste Idee einer verzahnten Strategie ist, dass I_f und FE_f einer FMO f immer gemeinsam gelabelt werden. Werden zwei FMOs demselben Instruktionszyklus zugeordnet, kann unmittelbar entschieden werden, ob ein Ressourcenkonflikt bzgl. der Verwendung von Funktionseinheiten vorliegt.

Der entsprechende ECLiPSe-Code dieser Strategie ist in Abb. 7.9 zu sehen. Dem Prädikat $lif/4$ wird die Liste der I-Variablen aus $I(b)$ und die Listen der FE-, STK- und IT-Variable übergeben. Das Prädikat $lif/2$ realisiert die verzahnte Labelingstrategie über den I- und FE-Variablen. Für die Listen $Is = [I_1, ..., I_n]$ und $FEs = [FE_1, ..., FE_n]$ gilt, dass für alle $i \in \{1, ..., n\}$ die I_i und FE_i mit derselben FMO f assoziiert sind ($I_i = I_f \wedge FE_i = FE_f$). Die Reihenfolge der Domainvariablen entspricht der aktuell gegebenen Anordnung der FMOs im Basisblock. In $lif/2$ wird in jedem Schritt i durch das Prädikat $indomain/1$ I_i und FE_i gelabelt. Durch das Binden der Funktionseinheit wird erreicht, dass nach der Ausführung von $lif/2$ die Existenz einer Lösung

bm	$\#fmos$	$\#mis$	$\#t_{lif}$	$\#t_{lif_i}$
ru	5	4	0.1	0.1(5)
cm	10	6	1.1	0.7(6) 0.2(7)
cu	15	8	80.3	0.6(8) 0.3(10)
bi1	17	9	0.8	0.3(9) 0.1(10)
co	5	2	0.1	0.1(2,3)
ruN	9	4	0.2	0.1(4,5)
cuN	27	9	726.8	7.1(9) 0.4(10) 0.3(11)
biN	25	11	5528	0.4(11) 0.2(12)
fir	8	3	0.3	0.1(3,4)

Tabelle 7.1.: Resultate von I^3R_{lif}.

wahrscheinlicher wird, da hierdurch die allgemeinen Restriktionen bzgl. der Parallelausführung geprüft werden können. Die Existenz einer Lösung kann aber letztendlich nur durch das Labeln der Domainvariablen aus $STK(b)$ und $IT(b)$ sichergestellt werden. Allerdings verbleiben hier durch das Labeln der FE-Variablen nicht mehr so große Freiheitsgrade. STK-Variablen liegen schon gebunden vor, so dass auch hier die Konflikte schon im Vorfeld aufgedeckt werden konnten.

Die mit dieser Strategie erreichten Laufzeiten und Resultate sind in Tabelle 7.1 dargestellt. In Spalte $\#fmos$ ist die Anzahl der FMOs des betrachteten Basisblocks gegeben, und Spalte $\#mis$ zeigt die Anzahl der minimal notwendigen Instruktionszyklen. Die Gesamtzeiten der Optimierung sind in Spalte t_{lif} wiedergegeben. In Spalte t_{lif_i} sind zusätzlich noch die Generierungszeitpunkte der optimalen Resultate und der Zwischenresultate dargestellt. Die kleineren Benchmarks können innerhalb weniger Sekungen optimiert werden. Für die größeren Beispiele (cuN und biN) sind immer noch sehr hohe Laufzeiten notwendig. Es zeigt sich aber, dass im Vergleich zur gesamten Laufzeit, die optimalen Resultate sehr schnell gefunden werden. Die hohen Gesamtlaufzeiten lassen sich durch die Verbesserungen der Strategie $lif/4$ im nächsten Unterkapitel noch weiter reduzieren.

7.3.3. Weitere Verbesserung der Strategie $lif/4$

Betrachtet man den Suchbaum, der durch $lif/4$ aufgebaut wird, so wird durch das Labeln der $I-$ und FE-Variablen jeweils ein Backtrackingpunkt für I und für FE angelegt. Der Zweck des Bindens der FE-Variablen ist es, vorzeitig Parallelisierungskonflikte aufzudecken. Für bestimmte Variablenbelegungen $\theta_{I(b)}$ kommt es aber vor, dass eine FMO f zu keiner anderen FMO parallel ausgeführt wird. Trotzdem wird für f die Domainvariable FE_f gelabelt und somit der Suchraum unnötig vergrößert. Um

dieses zu vermeiden, wird das Labeling über die Variablen der Liste Is durchgeführt, bis die Instruktionszyklen zweier FMOs f_1 und f_2 gleich sind. An dieser Stelle wird nun auch FE_{f_1} gelabelt. Der Mechanismus zum Labeling der FE-Variablen basiert dabei auf Constraints, die die Instruktionszyklen von jeweils zwei FMOs überwachen. Diese Constraints führen dann bei Gleichheit der Instruktionszyklen das Labeling einer der FE-Variablen durch und generieren dazu einen Backtrackingpunkt[2].

Die Strategie ist in Abb. 7.10 durch das Prädikat $dlf/4$ umgesetzt worden. Es werden zuerst durch $watch\_is/2$ die Constraints erzeugt, die die Gleichheit zweier I-Variablen zu je zwei FMOs überwachen und gegebenenfalls die FE-Variable labeln. Durch das anschließende Labeling der I-Variablen werden diese Constraints reaktiviert und streuen dynamisch das Labeln der FE-Variablen ein. Dies wird durch das Constraint $cwatch\_is(FE_i, I_j, I_i)$ realisiert (durch $watch\_is/2$ wurden alle nichtdoppelten Paarbildungen $cwatch\_is(FE_i, I_i, I_j)$ für $I_i, I_j \in [I_1, ..., I_n]$ gebildet). Dieses Constraint wird reaktiviert, sobald eine der beiden I-Variablen instanziiert wird:

1. Sind I_{f_1} und I_{f_2} beide instanziiert, wird geprüft, ob I_{f_1} und I_{f_2} gleich sind (Regel 1). Wenn ja, wird FE_i gelabelt. Durch $indomain(FE_i)$ wird ein Backtrackingpunkt im Suchbaum generiert, der bewirkt, dass auch über dem Domain von FE_i Backtracking durchgeführt wird. Da die Prämisse von $I_i = I_j \Rightarrow FE_i \neq FE_j$ erfüllt ist, kann durch die Instanziierung von FE_i ein Ressourcenkonflikt durch das Constraint $FE_i \neq FE_j$ frühzeitig erkannt werden. Sind I_{f_1} und I_{f_2} ungleich, geschieht nichts, und das Constraint wird aus dem Constraintstore eliminiert.

2. Ist eine der beiden Domainvariablen noch nicht instanziiert (Regel 2), so wird durch den Aufruf des Prädikates $suspend/2$ eine neue Instanz des Constraints $cwatch\_is(FE_i, I_i, I_j)$ im Constraintstore angelegt. Das zweite Argument von $suspend/2$ definiert die Reaktivierungsbedingung, die hier besagt, dass das Constraint $cwatch\_is(FE_i, I_i, I_j)$ reaktiviert werden soll, sobald eine der Variablen der Liste $[I_i, I_j]$ instanziert wird[3].

Existiert zu einer FMO f_1 keine weitere FMO f_2 mit $I_{f_1} = I_{f_2}$, wird FE_{f_1} auch nicht gelabelt und es wird kein Backtrackingpunkt angelegt.

Zum Labeling der I-Variablen werden unterschiedliche Strategien angewandt, die zu unterschiedlichen Ausprägungen von $dlf/4$ führen:

- $dlf/4$: Anwendung der vordefinierten Strategie $labeling/1$.
- $dlfeff/4$: Anwendung der vordefinierten Strategie $labeleff/1$.
- $dlfeffc/4$: Anwendung der vordefinierten Strategie $labeleffc/1$.

[2] Siehe auch Kapitel 2.4.3 S. 46.

[3] Es ist hier nicht möglich zu spezifizieren, dass beide Variablen instanziiert sein müssen, da dies durch den internen Reaktivierungsmechanismus von ECLiPSe nicht erfasst werden kann.

```
dlf(Is,STKs,FEs,ITs):-
   watch_is(FEs,Is),
   labeling(Is),
   cut((labeling(STKs),labeling(ITs))).

watch_is([],[]):-!.
watch_is([FE|FEs],[I|Is]):-
  watch_is(FE,I,Is),
  watch_is(FEs,Is).

watch_is(_,_,[]):-!.
watch_is(FE1,I1,[I2|Is]):-
   cwatch_is(FE1,I1,I2),
   watch_is(FE1,I1,Is).

cwatch_is(FE1,I1,I2):-          Regel 1
   nonvar(I1),
   nonvar(I2),
   !,
   (I1=I2->indomain(FE1);true).
cwatch_is(I1,I2,FE1):-          Regel 2
   suspend(cwatch_is(I1,I2,FE1),
           [I1,I2]->inst).
```

Abbildung 7.10.: Labelingstrategie $dlf/4$ (dlf = dynamisches Labeling der FE-Variablen).

bm	$\#fmos$	$\#mis$	t_{lif}	t_{dlf}	t_{dlfeff}	$t_{dlfeffc}$	$t_{dlfsmin}$
ru	5	4	0.1	0.1 0.1(4)	0.1 0.1(4)	0.1 0.1(4)	0.1 0.1(4)
cm	10	6	1.1	0.9 0.6(6) 0.1(7)	0.2 0.2(6)	0.1 0.1(6)	0.3 0.1(6)
cu	15	8	80.3	71.6 0.5(8) 0.3(9)	2.6 0.3(8)	30.5 7.3(8) 0.3(9)	216.8 2.7(8) 0.3(9)
bi1	17	9	0.8	0.6 0.1(9)	0.5 0.1(9)	13.3 0.2(9)	0.6 0.2(9)
co	5	2	0.1	0.1 0.1(2)	0.1 0.1(2)	0.1 0.1(2)	0.1 0.1(2)
ruN	9	4	0.2	0.1 0.1(4,5)	0.1 0.1(4)	0.1 0.1(4)	0.2 0.2(4)
cuN	27	9	726.8	420.7 0.5(9) 0.4(10,11)	76.0 0.3(9)	289.5 64.2(9) 8.1(10)	1415.8 32.4(9) 0.3(10)
biN	25	11	5528.3	5037.3 0.3(11,12)	502.1 0.2(11)	1558.9 1427(11) 8.3(12)	20492.9 1.2(11) 0.3(12)
fir	8	3	0.3	0.3 0.1(3,4)	0.3 0.1(3)	0.5 0.2(3)	0.3 0.2(3)

Tabelle 7.2.: Resultate der Varianten der Labelingstrategie $dlf/4$.

- $dlfls/4$: Hier wird eine Labelingstrategie zum Labeln der I-Variablen eingesetzt, die einer häufig angewendeten Heuristik innerhalb des List-Scheduling entspricht. Es wird die I-Variable aus der Liste Is ausgewählt, deren FMO den längsten kritischen Pfad besitzt. Bei zwei Variablen mit gleicher Pfadlänge wird die I-Variable mit dem kleinsten Domain gewählt.

Die Resultate unter Anwendung dieser Labelingstrategien sind in Tabelle 7.2 dargestellt. Es zeigt sich, dass durch das dynamische Labeling der FE-Variablen Verbesserungen gegenüber $lif/4$ erzielt werden. Drastische Verbesserungen der Laufzeit werden unter dem Einsatz der Labelingstrategie $dlfeff/4$ (Spalte t_{dlfeff}) erzielt. Unter dieser Strategie werden auch die besten Generierungszeitpunkte der optimalen Resultate erzielt, die alle unter einer CPU-Sekunde liegen. Die Zeiten der Strategie $dlfeffc/4$ liegen zwischen den Zeiten $dlf/4$ und $dlfeff/4$. Der Einsatz der List-Scheduling-Heuristik $dlfls/4$ liefert schlechtere Laufzeiten als die Labelingstrategie $lif/4$, und die Generierungszeitpunkte der optimalen Resultate in $dlfls/4$ waren viel schlechter als bei $dlfeff/4$.

Als allgemeines Resultat ist zu sagen, dass die Strategie $dlfeff/4$ sowohl bzgl. der Laufzeit als auch in Hinblick auf den Einsatz von Timeouts mit Abstand die beste Strategie ist. Es stellt sich die Frage, ob der Einsatz des dynamischen Labelings der FE-Variablen überhaupt so lohnenswert war, da die offensichtlich drastische Reduk-

tion der Laufzeiten ja letztendlich durch die modifizierte Suchreihenfolge über den I-Variablen erreicht wird. Der Einsatz des dynamischen Labelings läßt sich aber durchaus rechtfertigen:

- Es war immerhin eine bis zu 50 prozentige Verbesserung der Laufzeiten gegenüber $lif/4$ zu vermerken.
- Die Strategie war einfach zu implementieren und verdeutlicht die Möglichkeit, Backtrackingpunkte durch Constraints dynamisch in die Evaluierung benutzerdefinierter Prädikate einzustreuen.

Es ist im voraus nicht immer offensichtlich, ob eine Strategie zu guten Resultaten führt. Die Wirkungsweisen der Constraints, der Constraintpropagierung und der Suchraumbeschneidungen sind so komplex, dass häufig nur Vermutungen angestellt werden können, ob eine bestimmte Strategie den Suchraum effektiv reduziert. So ist es während der Entwicklung von Strategien auch schon häufig zu Überraschungen gekommen: Wo man mit Sicherheit eine Verbesserung der Laufzeiten erwartet hat, haben sich die Laufzeiten auch schon mal drastisch verschlechtert. Das heißt nicht, dass man nicht trotzdem analysieren kann, warum es letztendlich zu den überraschenden Effekten gekommen ist. Die Fälle, in denen eine als gut vermutete Strategie auch zum Erfolg führt, waren nach unseren Erfahrungen auch in der Mehrzahl. Trotzdem bedarf es immer zunächst einer Implementierung dieser Strategien, um auch letztendlich sicherzugehen, dass nicht doch unerwünschte Nebeneffekte auftreten. Die Eleganz und Einfachheit, die der constraintbasierte Ansatz gibt, liefert einen großen Beitrag, dass eine Menge an Strategien schnell entwickelt und untersucht und somit auch eher nicht ganz so offensichtliche Strategien umgesetzt werden können.

7.3.4. Postprocessing: Registerallokation und Adresscodegenerierung

Zur Generierung eines vollständigen Maschinenprogramms, das man mit handgeneriertem oder compilergenerierten Assemblercode vergleichen kann, muss noch eine Registerallokation und die Generierung von Adresscode durchgeführt werden. In diesem Abschnitt werden diese Phasen nur kurz skizziert und danach der resultierende Code mit dem handgenerierten Assemblercode der DSPStone-Benchmarks und mit dem compilergenerierten Code des GNU-Compilers der ADSP2100-Familie verglichen.

Registerallokation

Es wird eine lokale Registerallokation über den Nicht-AGU-Registerfiles und dann eine globale Registerallokation über den AGU-Registerfiles durchgeführt:

- *Lokale Registerallokation für die Nicht-AGU-Registerfiles*: Diese wird pro Basisblock ausgeführt. Hier wird eine ähnliche constraintbasierte Technik eingesetzt, wie sie in [11] beschrieben wird. Dabei wird die Sequenz der Maschineninstruktionen durchlaufen und on-the-fly Spillcode eingefügt, wo es erforderlich ist. Dabei

wird nicht unbedingt in den Speicher gespillt, sondern es werden auch freie Register ausgenutzt. Hierdurch können zum einen Speicherreferenzen, aber auch Maschinenoperationen zum Zurückladen der Werte eingespart werden. Letzteres ist der Fall, wenn die Benutzung eines in ein Register gespillten Wertes auf das neue Register realloziert werden kann.

- *Globale Registerallokation für die AGU-Registerfiles*: Für die AGU-Register wird eine globale Registerallokation auf der Basis von Standardgraphfärbungsalgorithmen durchgeführt. Zuvor werden Autoinkrement- und Autodekrementoperationen, die sukzessiv eine Adresse modifizieren, jeweils dem gleichen AGU-Register zugeordnet.

Die Anzahl der gleichzeitig lebendigen Werte, die in den betrachteten DSPStone-Benchmarks den AGU-Registern zugeordnet waren, übersteigt in keinem Fall die Kapazität der AGU-Register, so dass hier in den betrachteten Benchmarks niemals Spillcode eingefügt werden musste. Es wurde nur Spillcode für die Nicht-AGU-Registerfiles generiert. Die gespillten Werte konnten aber immer in freien Registern untergebracht werden, so dass hier keine zusätzlichen Speicherreferenzen entstanden.

Adresscodegenerierung

Der Grund, die Adresscodegenerierung zum Ende der Codegenerierung durchzuführen, ist, dass zur effektiven Ausnutzung der AGUs und dessen Autoinkrement- und Autodekrementoperationen die Reihenfolge der Speicherzugriffe bekannt sein muss. Diese steht aber letztendlich erst nach der Instruktionsanordnung und der Generierung von Spillcode fest.

Für die Adresscodegenerierung werden Techniken aus [69] eingesetzt. Außer Initialisierungscode für die Adressregisterfiles wurden durch den erzeugten Adresscode keine zusätzlichen Maschineninstruktionen eingefügt. Die zusätzlich generierten FMOs von AGU-Operationen konnten alle durch parallelisierbare Autoinkrement- und Autodekrementoperationen realisiert werden. Es ist allerdings zu bemerken, dass die meisten Speicherreferenzen in den DSPStone-Benchmarks schon explizit im C-Code durch Autoinkrement- und Autodekrementoperationen dargestellt werden. Diese Operationen werden schon in GIA den AGU-Operationen zugeordnet und während I^3R parallelisiert. Da für die DSPStone-Benchmarks ansonsten nur in nicht benutzte Register gespillt werden kann, entstehen auch hier keine zusätzlichen Speicherreferenzen. Die verbleibenden Speicherreferenzen umfassen sehr wenige Zugriffe auf statische skalare Variablen, für die durch die eingesetzten Adressoptimierungstechniken ein Adresslayout generiert wird, welches durch AGU-Operationen ideal ausgenutzt werden kann.

7.3.5. Resultierende Codegüte

Der resultierende Code von $COCOON_{adsp2100}$ wird im Folgenden mit dem handgenerierten Assemblercode der DSPStone-Benchmarks und dem Code des GNU-Compilers[4] für die ADSP2100-Familie verglichen. Bei dem Vergleich des Codes wird sich auf die in den DSPStone-Benchmarks markierten Profilingregionen beschränkt. Bei den kontrollflussumfassenden Benchmarks umfassen diese Regionen die zentralen Schleifen. Verglichen wird die Anzahl der Maschineninstruktionen, so dass die Codegüte hier zunächst an der Codegröße gemessen wird. Bei den datenflussdominierten Benchmarks kann daraus auch unmittelbar auf die Performance des Codes geschlossen werden. Für die Schleifen ist es aber auch zulässig auf die Performance zu schließen, wenn man sich auf die Betrachtung der Schleifenrümpfe beschränkt:

- Sowohl unser Compiler, der handgenerierte Code als auch der GNU-Compiler setzen die Schleifen durch Zero-Overhead-Loops um. Der Overhead für das Abfragen der Schleifenbedingungen und Modifizieren der Schleifenzähler ist in allen drei Fallen gleich hoch (so auch der Initialisierungscode für die Zero-Overhead-Loops).
- Der handgenerierte Code setzt zwar iterationsübergreifende Optimierungen ein, die aber keine Techniken umfassen, welche die Performance durch Vergrößerung des Schleifencodes erzielen (z.B. Abrollen des Schleifenrumpfes). Auch die Anzahl der Schleifeniterationen ist bei unserem Code, sowie dem handgeneriertem und dem compilergeneriertem Code gleich.

Somit kann auch hier von der Anzahl der Maschineninstruktionen einer Schleife auf deren Performance geschlossen werden. Daher werden in den Resultaten primär die Schleifenrümpfe betrachtet. Die iterationsübergreifenden Techniken des handgenerierten Codes erfordern allerdings, dass den Schleifen zusätzlich zum Initialisierungscode der Zero-Overhead-Loops auch weiterer Code vor dem Eintritt in die Schleife (Schleifenprolog) und nach dem Austritt aus der Schleife (Schleifenepilog) hinzugefügt wird. Da dies die Codegröße erhöht und auch die durchschnittliche Performance des Schleifenrumpfes beeinflusst, werden die Auswirkungen ebenfalls in den Resultaten explizit gezeigt.

In Tabelle 7.3 ist in der Spalte $\#mis_{iiir}$ die Anzahl der generierten Instruktionszyklen (der relevanten Basisblöcke) nach Anwendung von Techniken aus I^3R_l dargestellt. In Spalte $\#mis_{cocoon}$ ist die generierte Anzahl von Maschineninstruktionen nach der Registerallokation und Adressgenerierung angegeben (die Differenz der Werte aus den Spalten $\#mis_{cocoon}$ und $\#mis_{iiir}$ ist die Anzahl der zusätzlich durch das Postprocessing generierten Maschineninstruktionen). Spalte $\#mis_{hand}$ zeigt die Anzahl der Maschineninstruktionen des handgenerierten Referenzcodes, und in $\#mis_{gnu}$ ist die Anzahl der generierten Maschineninstruktionen des GNU-Compilers gezeigt. In den Spalten $\%cocoon$ und $\%gnu$ ist der Overhead von $COCOON_{adsp2100}$ und vom GNU-Compiler, verglichen zum handgenerierten Code, dargestellt.

[4] Dies ist bislang der einzige bekannte für diese Prozessorfamilie verfügbare Compiler. Er wird über Analog Devices distributiert.

bm	$\#mis_{iiir}$	$\#mis_{cocoon}$	$\#mis_{hand}$	$\#mis_{gnu}$	$\%cocoon$	$\%gnu$
ru	4	4	4	6	0%	50%
cm	6	8	6	16	33%	166%
cu	9	13	9	23	44%	155%
bi1	9	12	12	29	0%	141%
co	2	2	1	6	100%	500%
ruN	4	4	3	8	33%	166%
cuN	9	11	9	40	22%	344%
biN	11	14	8	37	75%	362%
fir	3	3	2	11	50%	450%

Tabelle 7.3.: Vergleich der Codegüte bzgl. der Performance der Schleifenrümpfe.

bm	$\#mis_{cocoon}$	$\#mis_{hand}$	$\#mis_{gnu}$	$\%cocoon$	$\%gnu$
co	4	5(2)	8	-20%	60%
ruN	6	7(2)	10	-14%	42%
cuN	13	11(0)	42	18%	281%
biN	16	14(4)	39	14%	178%
fir	5	7(0)	13	-29%	86%

Tabelle 7.4.: Vergleich der Codegröße des kompletten Schleifencodes.

Für die kontrollflussumfassenden Benchmarks werden in dieser Tabelle zunächst nur die Schleifenrümpfe verglichen. Unser Code hat für diese Benchmarks einen mittleren Overhead von 56%, wogegen der GNU-Compiler es auf einen mittleren Overhead von 364% bringt. Für die datenflussdominierten Benchmarks weist unser Code im Mittel sogar nur einen Overhead von 20% auf, wogegen der Code des GNU-Compilers im Mittel um 128% schlechter ist.

Der Grund, dass der mittlere Overhead von $COCOON_{adsp2100}$ und beim GNU-Compiler bei den kontrollflussumfassenden Benchmarks höher ist als bei den datenflussdominierten Benchmarks, liegt in der erwähnten Ausnutzung von iterationsübergreifenden Optimierungen im handgenerierten Code. Diese werden weder in unserem noch im GNU-Compiler eingesetzt, könnten aber in Zukunft noch integriert werden. Wie erwähnt, werden durch diese Techniken zusätzliche Instruktionen vor Eintritt in die Schleife (Schleifenprolog) und nach Austritt aus der Schleife (Schleifenepilog) notwendig, wodurch die eigentliche Codegröße der Schleife ansteigt (allerdings nicht die der Schleifenrümpfe). In Tabelle 7.4 ist ein Vergleich der Codegröße durchgeführt, wenn man den Code zur Initialisierung der Zero-Overhead-Loops, den Schleifenrumpf sowie Schleifenprolog und Schleifenepilog mit einbezieht. Für $COCOON_{adsp2100}$ und den GNU-Compiler ist die Anzahl der zusätzlichen Maschineninstruktionen hier konstant gleich 2, der jeweils durch den Initialisierungscode der Schleifen hinzukommt. Beim handgenerierten Code kommen durch den Schleifenprolog und Schleifenepilog weitere Maschineninstruktionen hinzu. Diese zusätzliche Anzahl an Maschineninstruktionen ist in Spalte $\#mis_{hand}$ in den Klammern dargestellt (sie ist aber in der davor dargestellten Anzahl der Maschineninstruktionen für den gesamten Schleifencode des handgenerierten Codes schon enthalten). Es ist zu erken-

bm	$\#mis_{cocoon}$	$\#mis_{hand}$	$\#mis_{gnu}$	$\%cocoon$	$\%gnu$
co	2	1+0.12	6	77%	436%
ruN	4	3+0.12	8	28%	156%
cuN	11	11+0	40	22%	344%
biN	14	8+1	37	56%	311%
fir	3	7+0	11	50%	450%

Tabelle 7.5.: Vergleich der Performance des kompletten Schleifencodes.

nen, dass der von uns generierte Code bzgl. der Codegröße im Mittel 6% besser abschneidet als der handgenerierte Code, der GNU-Compiler aber immer noch im Mittel einen Overhead von 129% aufweist. Zieht man den zusätzlichen Code für Schleifenprolog und Schleifenepilog bei der Performance der Schleifen des handgenerierten Codes mit ein, ergeben sich noch Verbesserungen bzgl. des mittleren Overheads bei $COCOON_{adsp2100}$ (jetzt 46%) und beim $GNU - Compiler$ (jetzt 349%) (siehe Tabelle 7.5: Die sich durch den Schleifenprolog und Schleifenepilog zusätzlich ergebenden Instruktionen wurden anteilmäßig auf die Schleifenrümpfe umgerechnet; co und ruN haben jeweils 16 Iterationen und biN hat 4 Iterationen).

Durch Anwendung der Registerallokation in der Postprocessingphase wurde bis zu 44% Spillcode, gemessen an $\#mis_{iir}$, generiert. Dies ergibt sich durch Differenzbildung der beiden Spalten $\#mis_{cocoon}$ und $\#mis_{iiir}$ in Tabelle 7.3 (durch die Adresscodegenerierung wurden keine zusätzlichen Maschineninstruktionen eingefügt, die in die Profilingregionen fallen). Im Folgenden wird untersucht, ob dieser Spillcode - durch Integration eines Registerdruckmodells in I^3R_{CSP} - reduziert werden kann.

7.4. Integration des Registerdrucks (I^3R_{l+ra})

In den bislang betrachteten Optimierungstechniken aus I^3R_l ist kein Registerdruck im Kostenmodell berücksichtigt worden. So kann eine ungünstige Anordnung von FMOs einen hohen Registerdruck erzeugen, was die Generierung von Spillcode erfordert (bis zu 44% in den DSPStone-Benchmarks). Untersuchungen mehrerer Maschineninstruktionsanordnungen haben gezeigt, dass sich durch Umordnung der Maschinenoperationen durchaus weniger Spillcode ergeben kann, ohne dass sich die Länge der Maschineninstruktionssequenz erhöht. Der Aspekt des Registerdrucks wird daher mit in das Kostenmodell von I^3R_{CSP} integriert und die korrespondierenden Techniken unter dem Begriff I^3R_{l+ra} erfasst. Das Kostenmodell des Registerdrucks wird im Folgenden beschrieben und die Auswirkung auf die resultierende Codegüte und auf die Laufzeiten gezeigt.

7.4.1. Modell des Registerdrucks

Registerdruck

Unter *Registerdruck* wird hier verstanden, dass zu einem bestimmten Zeitpunkt der Programmausführung die Anzahl der lebendigen Werte die Registerfilekapazität übersteigt. Um ein Maß für den Registerdruck zu bekommen, sagen wir, dass der Registerdruck zum Zeitpunkt I gleich dem Maximum aus der Anzahl der lebendigen Werte

minus der Registerfilekapazität und 0 ist. Unter *Registerdruckerhöhung* wird verstanden, dass zu einem bestimmten Zeitpunkt I der Registerdruck durch die Ausführung einer FMO erhöht wird und somit einen Wert größer 0 annimmt. *Registerdruckerhöhung*

Im Folgenden wird von drei Hilfsfunktionen Gebrauch gemacht:

- $mindomain(V)$ liefert zu einer Domainvariablen V den kleinsten Wert des Domains $D(V)$.
- $maxdomain(V)$ liefert zu einer Domainvariablen V den größten Wert des Domains $D(V)$.
- $regsize(STK)$ liefert zur STK-Variablen STK die Summe der Registerfilekapazitäten $\sum_{rf \in D(STK)} size(rf)$, wobei $size(rf)$ die Registerfilekapazität des Registerfiles rf liefert.

Die Menge der Domainvariablen des CSP-Modells I^3R_{CSP} werden erweitert, indem jeder FMO f eine Domainvariable E_f zugeordnet wird, die die *Endzeit der Lebensdauer des Wertes von* f bezeichnet. Die Endzeit E_f einer FMO f ist der Instruktionszyklus, an dem der Wert von f das letzte Mal im Basisblock benutzt wird. Dies ist demnach das Maximum der Startzyklen aller Benutzungen von f oder $I_{b_{max}} + 1$, falls der Wert auch noch in einem nachfolgenden Basisblock benutzt wird. Es werden für jedes E_f die folgenden Constraints generiert: *Endzeit der Lebensdauer des Wertes von f: E_f*

- Jede Endzeit einer FMO ist größer als die Startzeit: $E_f > I_f$. Dabei wird von Maschinenoperationen ausgegangen, die innerhalb eines Instruktionszyklus abgearbeitet werden. Sonst muss das obige Constraint zu $E_f \geq I_f + D_f$ modifiziert werden, wobei D_f die Ausführungszeit von f bezeichnet.
- Existiert eine Kante $f \rightarrow^{pos} v \in DFG(b)$ mit $v \in VAR_{out}(b)$, ist der Wert, den f erzeugt, auch bei Austritt aus dem Basisblock noch lebendig, was durch das Constraint $E_f = I_{b_{max}} + 1$ sichergestellt wird.
- Hat f keine Kante $f \rightarrow^{pos} v \in DFG(b)$ mit $v \in VAR_{out}(b)$ und ist $Is = [I_{f_1}, ..., I_{f_n}]$ die Liste der Ausführungszeiten von FMOs $f_1, ..., f_n$, die f benutzen, so wird ein Constraint $maxlist(Is, E_f)$ generiert.

Das hier verwendete Kostenmodell des Registerdrucks besteht aus der Summe der Registerdruckerhöhungen, wobei angenommen wird, dass jede FMO maximal zu einer Registerdruckerhöhung um 1 führen kann. Das Modell umfasst kein exaktes Kostenmaß für den potenziell entstehenden Spillcode, da dies ein zu komplexes un zeitaufwendiges Constraintmodell erfordern würde.

Die Constraints des Registerdruckmodells müssen für eine FMO f prüfen, ob die Menge der Register, in die der Wert von f potenziell geschrieben werden kann, nicht schon zeitgleich durch Werte anderer FMOs belegt sind. Dazu wird der Begriff der *potenziellen Interferenz* eingeführt. Es wird gesagt, dass *FMO f potenziell mit f' interferiert*, wenn *potenzielle Interferenz*

- f potenziell in die Lebensdauer von f' fallen kann, was durch die Bedingung

$$mindomain(I_{f'}) \leq maxdomain(I_f) \wedge mindomain(I_f) < maxdomain(E_{f'})$$

  gegeben ist, und

- $f_{[0]}$ und $f'_{[0]}$ potenziell dem gleichen Registerfile zugeordnet werden können: $D(f'_{[0]}) \cap D(f_{[0]}) \neq \emptyset$.

In diesem Fall besteht die Möglichkeit, dass der Wert von f' zum Startzeitpunkt von f noch lebendig ist und f das gleiche Registerfile wie f' setzt. Es besteht potenziell die Möglichkeit einer Registerdruckerhöhung, solange es zu einer FMO f eine Menge potenziell interferierender FMOs gibt, deren Anzahl größer oder gleich der kleinsten Registerfilekapazität eines der Registerfiles aus $D(f_{[0]})$ ist. Ist die Anzahl der potenziellen Kandidaten kleiner als die kleinste Registerfilekapazität eines Registerfiles aus $D(f_{[0]})$, kann durch f kein Registerdruck entstehen, da, selbst wenn $f_{[0]}$ diesem Registerfile zugeordnet wird, dessen Kapazität nicht überschritten wird.

Es wird gesagt, dass FMO f sicher in die Lebensdauer von f' fällt, wenn

$$maxdomain(I_{f'}) \leq mindomain(I_f) \wedge maxdomain(I_f) < mindomain(E_f).$$

Sei nun PI die Menge der FMOs, mit denen f potenziell interferiert, und $RFs = D(f_{[0]})$. Eine FMO f *erhöht mit Sicherheit* den Registerdruck, wenn es eine Menge $F \subseteq PI$ gibt, die folgende Voraussetzungen erfüllt:

- Die Setzungen von $f_i \in F$ können nur Registerfiles aus RFs zugeordnet werden, d.h. $(\bigcup_{f_i \in F} D((f_i)_{[0]})) \subseteq RFs$. Somit wird versucht, $|F|$ Werte auf maximal $regsize(RFs)$ Register zu verteilen.

- f setzt kein Registerfile, das nicht in $\bigcup_{f_i \in F} D((f_i)_{[0]})$ enhalten ist, d.h. $RFs \subseteq (\bigcup_{f_i \in F} D((f_i)_{[0]})$. Ansonsten könnte die Setzung von f einem Registerfile zugeordnet werden, dem die Setzungen der FMOs von F nicht zugeordnet werden können. Dadurch besteht eine Möglichkeit, der Registerdruckerhöhung auszuweichen. Es muss also, unter Hinzunahme der ersten Voraussetzung, $D(f_{[0]}) = (\bigcup_{f_i \in F} D((f_i)_{[0]}))$ gelten.

- Und es muss gelten: $|F| \geq regsize(RFs)$.

Für die Menge F ist durch die Bedingung $D(f_{[0]}) = (\bigcup_{f_i \in F} D((f_i)_{[0]}))$ sichergestellt, dass kein $f' \in F$ ein Registerfile setzt, das nicht in $D(f_{[0]})$ enthalten ist, und umgekehrt setzt f kein Registerfile, das nicht in $(\bigcup_{f_i \in F} D((f_i)_{[0]})$ enhalten ist. Somit wird versucht, $|F| + 1$ Werte in den Registerfiles aus $D(f_{[0]})$ unterzubringen. Da schon $|F|$ mindestens gleich $regsize(D(f_{[0]}))$ ist, wird demnach die vorhandene Kapazität durch die Setzung von f mindestens überstiegen, und es findet eine Registerdruckerhöhung statt.

Die Registerdruckerhöhung einer FMO f wird durch eine Domainvariable $RD_f \in \{0, 1\}$ wiedergegeben, die bei einer sicheren Registerdruckerhöhung auf eins gesetzt

```
iiir(BB):-
  cover_vars(BB,STKs,FEs,ITs),
  iiir_csp(BB,Is,Imax),
  ra_druck(BB,Imax,RDs),
  sumlist(RDs,RD),
  K#=Imax+RD,
  minimize(l(Is,FEs,STKs,ITs),K).
```

Abbildung 7.11.: Optimierungsprädikat zu I^3R_{l+ra}.

wird. Der Registerdruck eines Basisblocks b ergibt sich zu $RD(b) = \sum_{f \in FMO(b)} RD_f$. Der Registerdruck wird für eine FMO f durch das Constraint $watch\_rd(f, PI_f)$ kontrolliert, wobei PI_f die Liste der potenziell interferierenden FMOs von f ist. Das Constraint $watch\_rd(f, PI_f)$ prüft zunächst, ob die oben beschriebene Bedingung der sicheren Registerdruckerhöhung für eine Teilmenge von PI_f gegeben ist. Ist dies der Fall, wird RD_f auf 1 gesetzt. Ist dies nicht der Fall und es existieren noch potenzielle Kandidaten in $PI'_f \subseteq PI_f$, deaktiviert sich $watch\_rd(f, PI_f)$ und generiert ein neues Constraint $watch\_rd(f, PI'_f)$. Dabei muss $|PI'_f| \geq min\{size(rf) | \, rf \in D(f_{[0]})\}$ sein, so dass die Anzahl der potenziell interferierenden FMOs mindestens so groß ist, wie die kleinste Registerfilekapazität der Setzungsalternativen von f (s.o.). Die Reaktivierungsbedingung für $watch\_rd/2$ ist entweder die Instanziierung eines der $I_{f'}$ mit $f' \in PI'_f$ oder die Modifikation des Domains von $f_{[0]}$ oder die Modifikation eines Domains $D(f'_{[0]})$ mit $f' \in PI'_f$. Ist $|PI'_f| < min\{size(rf) | \, rf \in D(f_{[0]})\}$, kann keine Registerdruckerhöhung durch f entstehen und RD_f wird auf 0 gesetzt. In diesem Fall wird kein neues Constraint $watch\_rd/2$ erzeugt.

Das um Registerdruck erweiterte Prädikat $iiir/1$ ist in Abb. 7.11 zu sehen. Das Prädikat $ra\_druck/3$ generiert die Constraints des Registerdruckmodells. $I_{b_{max}}$ wird als zweites Argument von $ra\_druck/3$ übergeben, um die Constraints der Endzeiten evtl. in Beziehung mit $I_{b_{max}}$ setzen zu können (falls es Kanten $f \rightarrow^{pos} v \in DFG(b)$ mit $v \in VAR_{out}(b)$ gibt). Die Kostenfunktion besteht jetzt in der Minimierung der Summe $C = I_{b_{max}} + RD$.

7.4.2. Laufzeiten und Resultate

Die Laufzeiten und Kosten von I^3R_{l+ra} werden unter Anwendung der Labelingstrategie $dlfeff/4$ angegeben. Die Resultate sind in der Tabelle 7.6 gezeigt. In der Spalte $\#mis_{dlfeff+ra}$ sind die resultierenden Kosten gezeigt, die in der Form "*Instruktionszyklen + Registerdruck*" dargestellt sind. Die Spalte $t_{dlfeff+ra}$ enthält die Gesamtlaufzeiten und in der Spalte $t_{(lsdfeff+ra)_i}$ sind die Generierungszeitpunkte der optimalen Resultate und der Zwischenresultate aufgeführt. Die Spalte t_{dlfeff} zeigt zum Vergleich nochmal die Laufzeiten der Labelingstrategie $dlfeff/4$ ohne Registerdruck.

Vergleicht man die Laufzeiten $t_{dlfeff+ra}$ mit denen von t_{dlfeff}, zeigt sich im schlimmsten Fall eine Verdopplung der Laufzeiten. Es haben sich allerdings auch die Generierungszeiten der optimalen Resultate stark erhöht, die unter der Variante ohne Registerdruck alle innerhalb von einer Sekunde erreicht wurden. Hier sind jetzt bis zu 123

bm	$\#misl_{dlfeff+ra}$	$t_{dlfeff+ra}$	$t_{(dlfeff+ra)_i}$	t_{dlfeff}
ru	4+0	0.1	0.1(4+0)	0.1
cm	6+0	0.3	0.3(6+0) 0.1(6+1)	0.2
cu	8+1	4.7	24.9(8+1) 12.8(8+2) 0.7(8+3) 0.3(10+2)	2.6
bi1	9+2	0.8	0.5(9+2) 0.1(9+3)	0.5
co	2+0	0.1	0.1(2+0,3+0)	0.1
ruN	4+0	0.2	0.1(4+0)	0.1
cuN	9+1	97.9	51.2(9+1) 40.3(9+2) 0.4(10+2)	76.0
biN	11+1	886.3	123.5(11+1) 0.6(11+2) 0.3(11+3)	502.1
fir	3+0	0.4	0.2(3+0,4+0)	0.3

Tabelle 7.6.: Resultate von $I^3R_{dlfeff+ra}$.

bm	$\#misl_{dlfeff+ra}$	$\#mis_{cocoon}$	$\#mis_{hand}$	$\%cocoon$
ru	4+0	4 (4)	4	0%
cm	6+0	6 (8)	6	0%
cu	9+0	9 (13)	9	0%
bi1	9+2	11 (12)	12	-8%
co	2+0	2 (2)	1+0.12	77%
ruN	4+0	4 (4)	3+0.12	28%
cuN	9+1	10 (11)	9+0	11%
biN	11+1	12 (14)	8+1	33%
fir	3+0	3 (3)	2+0	50%

Tabelle 7.7.: Vergleich der resultierenden Codegüte von $COCOON_{adsp2100}$ unter Verwendung von $I^3R_{dlfeff+ra}$.

CPU-Sekunden (bei biN) notwendig. Alles in allem ergeben sich aber immer noch akzeptable Laufzeiten.

Es stellt sich nun die Frage, ob die Integration des Registerdrucks außer höheren Laufzeiten auch zu einer Reduzierung des Spillcodes geführt hat. In Tabelle 7.7 ist der resultierende Code nach der Registerallokation und der Adresscodegenerierung in Spalte $\#mis_{cocoon}$ gezeigt (in Klammern stehen die Resultate ohne Registerdruck). Diese werden wiederum mit dem handgenerierten Assemblercode verglichen (Spalten $\#mis_{hand}$). Der jetzige Overhead von $COCOON_{adsp2100}$ ist in der Spalte $\%cocoon$ gezeigt. Gerade bei den datenflussdominierten Benchmarks erkennt man eine drastische Verbesserung. Hier wurde die Qualität des handgenerierten Codes erreicht, und unser Code war in einem Beispiel (bi1) sogar um eine Instruktion besser. Bei den kontrollflussumfassenden Benchmarks ergaben sich ebenfalls Verbesserungen. Hier könnten die Ergebnisse des handgenerierten Codes nur noch durch Integration iterationsübergreifender Optimierungen in $COCOON_{adsp2100}$ erreicht werden. Durch die Integration des Registerdrucks ergibt sich für die datenflussdominierten Benchmarks jetzt ein mittlerer Overhead von -2% (vorher 20%; 128% beim GNU-Compiler) und bei den kontrollflussumfassenden Benchmarks ein Overhead von 40% (vorher 46%; 349% beim GNU-Compiler). Insgesamt liegt der mittlere Overhead bei 21% (vorher 34%; 245% beim GNU-Compiler).

7.4.3. Unterabschätzung im Registerdruckmodell

Das bislang beschriebene Registerdruckmodell würde eigentlich wieder eine vollständige Suche über die Menge $STK(b)$ erforden, da nach dem Labeling der $I-$ und FE-Variablen durchaus noch für bestimmte FMOs f das Constraint $watch\_rd/2$ mit potenziell interferierende FMOs existieren kann. Je nachdem, wie die Setzungen dieser FMOs gelabelt werden, können sich noch weitere Registerdruckerhöhungen ergeben. Es hat sich in Experimenten gezeigt, dass die Suche über diesen Setzungen die gezeigten Laufzeiten bis um das 10-fache erhöhen, ohne dadurch eine Verbesserung der Resultate zu bewirken. Daher wurden die angegebenen Zeiten nur unter einer Lösung für die Setzungen erzeugt. Zuvor wurden alle weiteren, noch nicht instanziierten RD-Variablen auf Null gesetzt, was u.a. die Deaktivierung der noch im Costraintstore existierenden $watch\_rd(...)$ Constraints zur Folge hat. Somit wird der Registerdruck im Kostenmodell zugunsten der Laufzeit eher unterschätzt. Die Folgen einer Überschätzung wurden hier nicht weiter untersucht. Es sei noch bemerkt, dass in den betrachteten Benchmarks der Registerdruck immer der Anzahl der eingefügten Spill-Operationen entsprach, was ein Anzeichen dafür ist, dass es hinreichend genau ist.

7.5. Partitionierung

Da die Methode der Partitionierung schon im Kontext von GIA drastische Laufzeitreduktionen unter Beibehaltung sehr guter Resultate erzielt hat, wird der Einsatz auch unter I^{3}R getestet. Die Strategie zur Partitionierung von Datenflussgraphen wurde von GIA übernommen: Der $DFG(b)$ zu Basisblock b wird an den CSEs in eine Sequenz

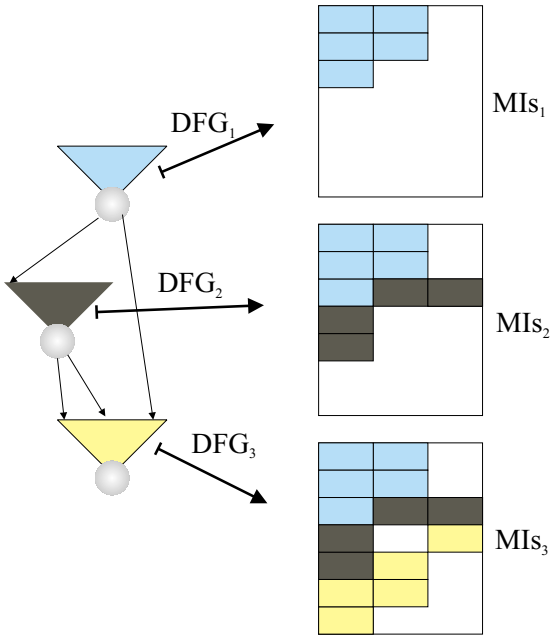


Abbildung 7.12.: I^3R über partitionierten Datenflussgraphen.

$[DFG_1, ..., DFG_n]$ von Teildatenflussgraphen zerlegt. Die Optimierung aus I^3R wird jeweils auf jeden Teildatenflussgraph angewandt und berücksichtigt bei Optimierung von DFG_i die schon generierte Anordnung von FMOs, die durch die Optimierung der Teildatenflussgraphen DFG_j (mit $1 < j < i$) generiert wurde: FMOs aus DFG_i werden wenn möglich mit den schon angeordneten FMOs der Teildatenflussgraphen DFG_j parallelisiert.

Dies ist in Abb. 7.12 verdeutlicht. Der gezeigte Datenflussgraph besteht aus den drei Teildatenflussgraphen DFG_1, DFG_2 und DFG_3. Die Anwendung von I^3R führt zu einer ersten partiellen Sequenz von Maschineninstruktionen MIs_1. Die Anwendung von I^3R auf DFG_2 fügt die FMOs von DFG_2 zu MIs_1 hinzu und es entsteht die partielle Sequenz MIs_2. Es wird entsprechend mit DFG_3 verfahren, was in der nun vollständigen Sequenz MIs_3 resultiert.

Die Partitionierung wurde mit der Labelingstrategie $dlfeff/4$ einmal ohne und einmal mit Berücksichtigung des Registerdrucks durchgeführt. Die Resultate der Partitionierung ohne Registerdruck sind in Tabelle 7.8 zu sehen. In Spalte $\#mis_{dlfeff+part}$ ist die Anzahl der benötigten Instruktionszyklen dargestellt. Im Vergleich zu den Resultaten ohne Partitionierung zeigen sich nur für die Benchmarks bi1 und cuN Abweichungen (die ursprünglichen erzielten Resultate ohne Partitionierung sind in den Klammern in der Spalte $\#mis_{dlfeff+part}$ gezeigt). Es ergeben sich drastische Reduktionen in den Laufzeiten, die nun alle unter einer CPU-Sekunde liegen.

Unerwartet gute Resultate haben sich nach der Anwendung der Registerallokation und der Adresscodegenerierung ergeben. Hier ist der resultierende Code fast so gut wie bei $dlfeff/4$ mit Berücksichtigung des Registerdrucks! Der Grund dafür ist allerdings einleuchtend, da die FMOs der Teildatenflussgraphen nicht mehr so stark

bm	$\#mis_{dlfeff+part}$	$t_{dlfeff+part}$	$\#mis_{cocoon}$	$\#mis_{hand}$	$\%cocoon$
ru	4(4)	0.1	4	4	0%
cm	6(6)	0.1	6	6	0%
cu	9(8)	0.3	9	9	0%
bi1	11(9)	0.5	13	12	8%
co	2(2)	0.1	2	1+0.12	77%
ruN	4(4)	0.2	4	3+0.12	28%
cuN	10(9)	0.8	10	9+0	11%
biN	11(11)	0.7	13	8+1	44%
fir	3(3)	0.1	3	2+0	50%

Tabelle 7.8.: Laufzeiten und Vergleich der Codegüte für die Labelingstrategie $dlfeff/4$ unter Einsatz der Partitionierung.

bm	$\#mis_{dlfeff+ra+part}$	$t_{dlfeff+ra+part}$	$\#mis_{cocoon}$	$\#mis_{hand}$	$\%cocoon$
ru	4+0	0.1	4	4	0%
cm	6+0	0.2	6	6	0%
cu	9+0	0.6	9	9	0%
bi1	11+1	0.75	12	12	0%
co	2+0	0.1	2	1+0.12	77%
ruN	4+0	0.2	4	3+0.12	28%
cuN	10+0	1.0	10	9+0	11%
biN	11+1	0.8	12	8+1	33%
fir	3+0	0.2	3	2+0	50%

Tabelle 7.9.: Laufzeiten und Vergleich der Codegüte für die Labelingstrategie $dlfeff/4$ mit Partionierung und unter Berücksichtigung des Registerdrucks.

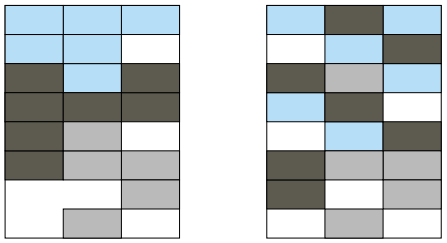


Abbildung 7.13.: Jeweilige Streuung von FMOs bei Optimierung über den partitionierten Teildatenflussgraphen (a) und über dem vollständigen Datenflussgraphen eines Basisblocks (b).

	$dlfeff$	$dlfeff + ra$	$dlfeff + part$	$dlfeff + ra + part$	gnu
df	20%	-2%	2%	0%	128%
cf	46%	40%	42%	40%	364%
all	34%	21%	24%	22%	259%

Tabelle 7.10.: Mittlere prozentuale Abweichungen vom handgeneriertem Referenzcode (df = nur datenflussdominiert; cf = nur kontrollflussumfassend; all = alles).

verstreut werden, wie unter der Optimierung des gesamten Datenflussgraphen (siehe Abb. 7.13). Hierdurch werden zum einen die Lebensdauern von Werten verkürzt und somit auch der Registerdruck verringert. Allerdings können auch unter der Partitionierung ohne Registerdruck noch ungünstige Kombinationen entstehen. Daher wird die Partitionierung auch nochmal unter der Betrachtung des Registerdrucks durchgeführt. Die Resultate sind in Tabelle 7.9 gezeigt. Hier ergeben sich durch die Berücksichtigung des Registerdrucks noch leichte Verbesserungen (Benchmarks bi1 und biN).

In Tabelle 7.10 ist der mittlere prozentuale Overhead, verglichen zum handgenerierten Referenzcode, einmal ohne Partitionierung (jeweils mit und ohne Berücksichtigung des Registerdrucks) und einmal mit Partitionierung gezeigt (wiederum jeweils mit und ohne Berücksichtigung des Registerdrucks).

7.6. Konsequenzen bei determinierten Überdeckungen von GIA

In diesem Kapitel werden die Vorteile der Beibehaltung alternativer Überdeckungen durch GIA demonstriert. Dazu ist GIA darauf eingeschränkt worden, pro Basisblock die zuletzt beste gefundene Lösung als Resultat zu liefern, ohne die Variablenbindungen rückgängig zu machen. Die Auswirkungen auf die Codegüte sind in Tabelle 7.11 zu sehen. Der mittlere Overhead steigt von 21% auf 79%, da gerade durch die festen Bindungen der STK-Variablen viele der Restriktionen zur Parallelausführung von Maschinenoperationen verletzt werden. Weiterhin ist eine drastische Verschlechterung

bm	$\#mis_{dlfeff+ra}$	$t_{dlfeff+ra}$	$\#mis_{cocoon}$	$\#mis_{hand}$	$\%cocoon$
ru	5+0	0.1	5	4	25%
cm	9+0	0.35 0.2 (9+1,9+2,9+3)	10	6	66%
cu	14+1	1437.5 63.8(14+1) 46.9(14+2) 15.0(14+3) 0.3(16+2,16+3) 0.1(18+3)	15	9	66%
bi1	15+2	38.7 1.4(15+2) 1.0(15+3) 0.2(15+4)	17	12	42%
co	3+0	0.2	3	1+0.12	168%
ruN	5+0	0.2	5	3+0.12	60%
cuN	13+2	7794.3 149.6(13+2) 124.5(13+3) 0.9(13+4) 0.2(16+2)	15	9+0	66%
biN	12+3	3999.8 0.2(12+3)	15	8+1	66%
fir	5+0	89.2 0.1(5+0)	5	2+0	150%

Tabelle 7.11.: Laufzeiten und Vergleich der Codegüte für die Labelingstrategie $dlfeff/4$ unter Berücksichtigung des Registerdrucks und determinierter Eingabe durch GIA.

der Laufzeiten sowie der Generierungszeitpunkte der besten Resultate zu vermerken. Da weniger FMOs parallelisiert werden können, erhöhen sich die Wertebereiche der I-Variablen und dadurch der potenzielle Suchraum wesentlich.

7.7. Existenz von Lösungen

Die Existenz einer Lösung zu $I^3R_{CSP}(b)$ kann im Allgemeinen nicht garantiert werden. Auch bei der Eingabe von CoLIR-Programmen, die von GIA generiert werden, läßt sich noch nicht auf das grundsätzliche Vorhandensein einer Lösung schließen (siehe Kapitel 6.4). Eine Lösung existiert mit Sicherheit, wenn von GIA keine alternativen Überdeckungen erzeugt werden, sondern jeweils nur eine einzige Lösung pro Funktion. In diesem Fall gibt es für jedes $I^3R_{CSP}(b)$ zumindest eine sequenzielle Lösung[5] pro Basisblock b, die gerade aus der sequentiellen Maschineninstruktionssequenz von b besteht, die von GIA generiert wurde.

Die Probleme bzgl. der Existenz von Lösungen treten erst durch die Beibehaltung von alternativen Überdeckungen durch GIA auf. Wie schon in Kapitel 6.4 gezeigt, kann nach der Generierung einer Lösung für einen Basisblock nicht garantiert werden, dass auch für die weiteren Basisblöcke noch Lösungen bzgl. des CSP-Modells $COVER_{fffp}$ existieren. Die Codegüte von I^3R, die durch die Beibehaltung alternativer Überdeckungen durch GIA erzielt werden kann, spricht auf jeden Fall dafür auch mal keine Lösung für ein CSP $I^3R_{CSP}(b)$ in Kauf zu nehmen, sofern die Möglichkeit besteht, den Code von b effizient zu reparieren. Wie in Kapitel 6.4 erläutert, besteht dazu die Möglichkeit. Außerdem ist in allen durchgeführten Experimenten nie ein solcher Fall aufgetreten. Bezüglich der Nicht-Existenz einer Lösung sei noch bemerkt, dass gerade dies einen exponenziellen Zeitaufwand bei der Suche mit sich bringen kann. In diesem Fall können Timeouts definiert werden, die spezifizieren, nach wieviel erfolglosem Zeitaufwand die Suche nach einer ersten Lösung abgebrochen wird.

7.8. Zusammenfassung

In diesem Kapitel wurden Techniken zur integrierten Instruktionsauswahl, Instruktionsanordnung und Registerallokation vorgestellt. Die Techniken, die unter dem Begriff I^3R zusammengefasst wurden, bauen alle auf demselben CSP-Modell auf. Die Unterschiede zwischen den Techniken ergaben sich durch den Einsatz verschiedener Labelingstrategien und der Betrachtung zweier Kostenmodelle. Unter dem ersten Kostenmodell wurde die reine Minimierung der Instruktionszyklen eines Basisblocks durchgeführt. Das zweite Kostenmodell erweiterte das erste Modell um die Berücksichtigung des Registerdrucks eines Basisblocks. Ziel der Erweiterung war es, eine Verbesserung der Codegüte zu erreichen, indem durch die Minimierung des Registerdrucks auch eine reduzierte Generierung von Spillcode erhofft wurde (der eigentliche Spillcode wird erst in einer nachgeschalteten Phase von I^3R generiert). Unter Anwendung der Partitionierung wurden (wie schon in GIA) die Effekte auf die Laufzeitverbesserung und den Grad der Verschlechterung bzgl. der Codegüte untersucht.

[5] (D.h., dass jede Maschineninstruktion des Basisblocks maximal eine Maschinenoperation enthält)

Die Techniken aus I^3R wurden im Kontext der gesamten Codegenerierung betrachtet. Die Eingabe zu I^3R wurde durch GIA generiert und es wurden Phasen zur Generierung von Spillcode und zur Erzeugung von Adresscode nachgeschaltet. Die erzeugten Resultate wurden mit handgeneriertem und compilergeneriertem[6] Assemblercode der DSPStone-Benchmarks verglichen. Die wesentlichen Resultate können wie folgt zusammengefasst werden:

- Unter Einsatz der besten betrachteten Labelingstrategie und ohne Betrachtung des Registerdrucks konnte schon eine hohe Codegüte erreicht werden. Hier lag der Overhead unseres Codes im Vergleich zum handgenerierten Assemblercode im Mittel bei nur 34%. Der GNU-Compiler wies einen mittleren Overhead von 259% auf. Es wurden akzeptable Laufzeiten erreicht: Bis zu 27 FMOs[7] konnten innerhalb von 500 CPU-Sekunden optimiert werden. Die Generierungszeitpunkte der optimalen Resultate lagen alle unter einer CPU-Sekunde.

- Unter Berücksichtigung des Registerdrucks konnte die Codegüte verbessert werden, da weniger Spillcode generiert wurde. Der Overhead lag jetzt bei 21%, wobei für die datenflussdominierten Benchmarks sogar ein Overhead von -2% im Mittel (vorher 20% und beim GNU-Compiler 128%) erreicht wurde. Die Laufzeiten lagen jetzt bei maximal 900 CPU-Sekunden.

- Die schon unter GIA erfolgreich eingesetzte Partitionierung führte auch unter I^3R wiederum zu guten Resultaten. Es wurden nur sehr geringe Abweichungen von der Codegüte im Vergleich mit der Optimierung ohne Partitionierung erzielt. Der mittlere Overhead lag jetzt ohne Betrachtung des Registerdrucks bei nur 24%. Grund hierfür ist, dass FMOs durch die lokale Optimierung der Teilblöcke nicht mehr so stark über den resultierenden Basisblock gestreut werden und sich hierdurch die Lebensdauern von FMOs verkürzen. Unter der Partitionierung mit Einbeziehung des Registerdrucks konnte der mittlere Overhead noch auf 22% reduziert werden. Die Laufzeiten der Partitionierung lagen mit und ohne Registerdruck alle unter einer CPU-Sekunde. Sie konnten somit drastisch reduziert werden, und die Codegüte wich dabei im Mittel nur 5% von den besten Resultaten ab.

- Die Betrachtung alternativer Überdeckungen hat sich als äußerst vorteilhaft erwiesen. Durch die Enschränkung, dass GIA nur eine einzige Überdeckung pro Funktion generiert, verschlechterte sich der mittlere Overhead des generierten Codes von 21% auf 79% (bei Anwendung der besten Lösungstrategie von I^3R).

I^3R umfasst mehrere Methoden zur Integration von Codegenerierungstechniken, die durch die folgenden Konzepte charakterisiert sind:

- *Verzögerung des Bindens von Ressourcen:* Durch die Beibehaltung von alternativen Überdeckungen durch GIA wurde die Instruktionsauswahl in I^3R integriert.

[6] Dieser Code wurde vom GNU-Compiler der ADSP2100-Familie generiert, der auch der einzige für diese Prozessorfamilie verfügbare Compiler ist. Er wird über Analog Devices distributiert.

[7] Dies war die größte Anzahl von FMOs, die in den Basisblöcken der betrachteten DSPStone-Benchmarks vorkam.

Generell wird durch die Beibehaltung von Freiheitsgraden den nachfolgenden Phasen mehr Flexibilität bei deren Entscheidungen ermöglicht.

- *Vereinigung der CSP-Modelle unterschiedlicher Phasen:* Hierdurch werden beim Lösen der Constraints, sowie bei der Optimierung alle Aspekte der verschiedenen Phasen simultan berücksichtigt. Auch die Integration der Instruktionsauswahl in I^3R beruht nicht allein auf der Beibehaltung der Alternativen, sondern auf der Verschmelzung des CSP-Modells $COVER_{ffp}$ mit den Constraints der Instruktionsanordnung und der Registerallokation.

- *Feinkörnige Integration von Teilaspekten anderer Phasen durch Teil-CSP-Modelle:* Es ist natürlich möglich, nur Teilaspekte einer Phase mit anderen Phasen zu verschmelzen.

- *Integration von Kostenmodellen zur Abschätzung der Auswirkungen von Entscheidungen auf nachgeschaltete Phasen:* Anstatt die Aspekte einer Phase exakt zu modellieren und auf diese Weise in andere Phasen zu integrieren, können nachgeschaltete Phasen auch durch ein entsprechendes Kostenmodell berücksichtigt werden. Dies umfasst i.d.R. zwar nur eine Abschätzung der Effekte auf die nachfolgenden Phasen, ist aber oft schon ausreichend, um ihnen Vorteile zu verschaffen. In I^3R wurde dies in Form des Kostenmodells umgesetzt, das den Registerdruck berücksichtigte. Zwar lag hier kein exaktes Modell des zu erwartenden Spillcodes vor und doch konnte ein grosser Teil des Spillcodes, der durch die I^3R nachgeschaltete Registerallokation erzeugt wurde, reduziert werden.

In Hinblick auf die Entwicklung generischer Techniken konnten die meisten Constraints von I^3R_{CSP} ohne maschinenspezifische Informationen der ADSP2100-Familie formuliert werden. Bei der Formulierung der Constraints, die die Bedingungen zur Parallelausführung der AGU-Operationen spezifizieren, wurden allerdings explizit Ressourcen der ADSP2100-Familie verwendet. Hier ist es jedoch möglich, die entsprechenden Stellen durch generische Prädikate zu ersetzen.

Der größte Vorteil des constraintbasierten Ansatzes war wiederum die Möglichkeit der modularen Spezifikation des CSP-Modells, der Labelingstrategien und der Kostenmodelle. Hier konnte gezeigt werden, dass die Spezifikation der CSP-Modelle von Teilphasen der Codegenerierung elegant und modular durchgeführt werden konnte. Die Möglichkeit, ohne Modifikation des CSP-Modells Labelingstrategien zu entwickeln und verschiedene Kostenmodelle zu adaptieren, erlaubte es, eine Vielzahl von varianten Labelingstrategien zu erproben.

8. Zusammenfassung

8.1. Problemstellung

Der Einsatz von Prozessoren in eingebetteten Systemen ermöglicht schnellere Designzyklen und Fertigungszeiten der Systeme gegenüber dem Einsatz dedizierter Hardware. Darüber hinaus lassen sich wesentlich flexiblere Systeme entwerfen, in denen späte Designentschlüsse einfliessen und noch spät Fehler beseitigt werden können. Bei der Entwicklung von Software für eingebettete Prozessoren besteht der Wunsch, moderne Hochsprachen für die Programmierung einzusetzen. Ein großes Problem ist der Mangel an guten Compilern, die den hohen Anforderungen an die benötigte Codegüte gerecht werden können. Dies ist besonders für DSPs und ASIPs der Fall: Die erforderliche Qualität des generierten Codes genügt gerade bei Realzeitanforderungen bei weitem nicht den gestellten Anforderungen, und der Code ist oft um ein Vielfaches größer als der vergleichbare handgeschriebene Code. Ein weiteres Problem, das der Verfügbarkeit guter Compiler im Wege steht, ist, dass deren Entwicklung zu teuer ist. Durch den Einsatz in sehr wenigen Applikationen stehen die Entwicklungskosten der Compiler in keiner Relation zu deren Verwendung. Um trotzdem den Einsatz von Prozessoren zu ermöglichen, wird in der Entwicklung von Software häufig auf die Assemblerprogrammierung zurückgegriffen, was zu großen Nachteilen führt: Entwicklungszeiten und Phasen zum Testen und zur Fehlerkorrektur verlängern sich i.d.R. wesentlich. Weiterhin ist auch die Einarbeitung in die nicht einfache Assemblerprogrammierung der jeweiligen Prozessoren ein Faktor. Die Wiederverwendung von Assemblerprogrammen ist bei einem Prozessorwechsel kaum noch möglich. Die potenzielle Kette eines automatischen Fertigungsprozesses vom Design hin zum Produkt ist nicht mehr realisierbar. Bei der zunehmenden Komplexität der Systeme ist der fehleranfällige Einsatz der Assemblerprogrammierung nicht mehr akzeptabel.

8.2. Anforderungen und Ziele

Grundlegendes Ziel dieser Arbeit war die Entwicklung neuer Compilertechniken für Festkomma-DSPs und ASIPs, wobei der Schwerpunkt auf Prozessoren mit hochgradig irregulären Datenpfaden mit eingeschränkter Parallelausführung auf Instruktionsebene liegt. Dabei mussten die folgenden Anforderungen an Compilertechniken und an die Compilerstruktur beachtet werden:

- *Anforderungen an Techniken*: Um Eigenschaften von DSPs effektiv auszunutzen sind Techniken notwendig, die eine graphbasierte Instruktionsauswahl unterstützen, neben der Zuordnung von Programmvariablen zu Registerfiles auch

die notwendigen Datentransfers berücksichtigen und die restriktive Parallelität ausnutzen. Die wechselseitigen Abhängigkeiten zwischen den oben genannten Punkten müssen berücksichtigt werden, woraus sich die Forderung nach phasengekoppelten Techniken ergibt, die die wichtigsten Phasen der Codegenerierung - die Instruktionsauswahl, die Registerallokation und die Instruktionsanordnung - sinnvoll integrieren.

- *Anforderungen an die Compilerstruktur*: Eine schnelle Adaption von Compilern an neue Prozessoren erfordert möglichst retargierbare Techniken und eine einfache Integration neuer Techniken. Zur Einhaltung der Codequalität ist es notwendig, klassenspezifische retargierbare Techniken zu entwickeln. Die Integration neuer Techniken erfordert eine transparente Zwischenrepräsentation eines Programms, auf das möglichst alle Phasen der Codegenerierung aufsetzen können.

- *Neue Implementierungsmethoden*: Die vielen Randbedingungen von irregulären Prozessorarchitekturen, sowie die Forderung nach der Phasenkopplung von Techniken, machen das Finden guter heuristischer Methoden sehr schwierig. Da es sich um kombinatorische Optimierungsprobleme handelt, besteht ein Bedarf an hocheffizienten suchbasierten Verfahren. Es stellt sich die Frage nach Methoden, die die Spezifikation und die Implementierung der Phasenkopplung handhabbar machen, da der Einsatz konventioneller Programmiermethoden hier an seine Grenzen stößt und die Entwicklung adäquater Techniken oft nicht mehr möglich macht.

Aus den Anforderungen ergaben sich die in dieser Arbeit verfolgten und umgesetzten Ziele:

- *Entwicklung neuer Techniken zur Gewährleistung einer hohen Codequalität*: Dies erfordert Techniken zur graphbasierten Instruktionsauswahl, Registerallokation mit Handhabung von irregulären Registerfiles sowie der Ausnutzung der stark eingeschränkten Parallelität in der Instruktionsanordnung. Starke wechselseitige Abhängigkeiten zwischen diesen Phasen erfordern den Einsatz phasengekoppelter Techniken.

- *Schnelle Adaption an Prozessoren und Erweiterbarkeit*: Zur schnellen Adaption von Compilern an neue Prozessoren sollte ein transparentes Datenmodell für Zwischenrepräsentationen entwickelt werden, das eine einfache Erweiterung um neue Techniken ermöglicht und alle Phasen effektiv unterstützt. Die Codequalität hat bei der Entwicklung von Techniken zu Lasten der Generalisierung Priorität. Daher war das Ziel, zunächst Techniken für dedizierte Prozessoren zu entwickeln. Generische Aspekte der Techniken sollten aber schon bei der Entwicklung herausgestellt werden.

- *Einsatz neuer Optimierungsmethoden*: Die Entwicklung der Techniken basierte auf dem Einsatz der Constraint-Logikprogrammierung unter dem Einsatz des Systems ECLiPSe. Ziel war es, die Eignung von CLP im Problembereich zu zeigen.

8.3. Beiträge der Arbeit in Hinblick auf die Zielsetzungen

8.3.1. Entwicklung neuer Techniken zur Generierung einer hohen Codegüte

Gemeinsame zugrundeliegende Zwischenrepräsentation CoLIR

Die constraintbasierte Zwischenrepräsentation CoLIR wurde als Grundlage für phasenintegrierte Techniken geschaffen, um die primären Phasen der Codegenerierung simultan auf einer gemeinsamen Datenbasis ausführen zu können. Auf der Basis faktorisierter Maschinenoperationen ist ein intuitives und handhabbares Konzept zur Repräsentation alternativer Maschinenprogramme eingeführt worden, das es auf einfache Weise ermöglicht, das Binden von Ressourcen in einer bestimmten Phase zu verzögern, und möglichst viele Freiheitsgrade für nachfolgende Phasen beizubehalten. Der constraintbasierte Ansatz erlaubt wechselseitige Abhängigkeiten über aufeinanderfolgende Codegenerierungsphasen zu propagieren.

CSP-Modell zur graphbasierten Instruktionsauswahl

Das CSP-Modell $COVER$ wurde als generische Grundlage zur Implementierung von Techniken zur Instruktionsauswahl sowie zur Integration der Instruktionsauswahl in andere Codegenerierungsphasen entwickelt. Die eingeführten Modellierungskonzepte erlauben eine einfache und intuitive Modellierung und Repräsentation des Befehlssatzes eines Prozessors. Die Generierung der CSPs zu $COVER$ ist darüber hinaus vollkommen unabhängig von der Spezifikation der FMOs eines bestimmten Prozessors. Hierdurch ist die Basis zur Retargierung des CSP-Modells $COVER$ gegeben.

Das CSP-Modell $COVER$ integriert die folgenden Aufgaben der Instruktionsauswahl in einem einheitlichen, constraintbasierten Konzept:

- *Mustererkennung von Maschinenoperationen und Datentransferpfaden*: Die Mustererkennung von Maschinenoperationen und die Darstellung alternativer Maschinenoperationsmuster basieren jeweils auf denselben Modellierungskonzepten von FMOs. Dies umfasst auch die Erkennung und Darstellung komplexer und graphbasierter Maschinenoperationsmuster. Domains von Setzungen und Benutzungen enthalten nur ST-Komponenten, für die mindestens ein Datentransferpfad von der Setzung zur korrespondierenden Benutzung existiert. Alternative Datentransferpfade können simultan durch eine einzige Sequenz von FMOs dargestellt werden.
- *Auswahl von Maschinenoperationen*: Die Auswahl einer Überdeckung mit legalen Maschinenoperationen unter der Berücksichtigung der Existenz von Datentransferpfaden zu einem Basisblock b ist potentiell durch das Labeling der Domainvariablen des CSPs $COVER(b)$ gegeben.

Weiterhin werden durch die Constraints der FMOs die Wechselbeziehungen zwischen STK-, FE-Variablen zu IT-Variablen hergestellt. Hier fließen neben den üblichen

Aufgaben der Instruktionsauswahl auch schon Aufgaben der Instruktionsanordnung mit ein, die Ressourcen gemäß der Parallelisierung von FMOs korrekt auszuwählen.

Die benutzerdefinierten Constraints können so formuliert werden, dass alle Lösungen zu $fmo(Class, Op, RF_o := [RF_1, ..., RF_n], FE, IT)$ ohne Backtracking generiert werden können. Für das Goal

$$fmo(..),\ \ldots,\ fmo(...),\ cg(...).$$

sind somit in $cg(...)$ alle Lösungen für die FMOs simultan verfügbar. Diese Eigenschaft ist elementar für die Effizienz der entwickelten Optimierungsverfahren.

Exakte Methoden zur graphbasierten Instruktionsauswahl

Es wurden Techniken zur optimalen, graphbasierten Instruktionsauswahl beschrieben, die unter dem Begriff GIA zusammengefasst wurden. Diese Techniken sind durch Ergänzung des CSP-Modells $COVER$ um ein Kostenmodell, sowie der Adaption verschiedener Suchstrategien implementiert worden. Die Resultate von GIA lassen sich wie folgt zusammenfassen:

- Es konnten effiziente Labelingstrategien zur exakten Instruktionsauswahl entwickelt werden, womit alle DSPStone-Benchmarks in weniger als 2 Sekunden exakt zu lösen waren. Eine Analyse der Laufzeiten zeigte aber, dass der Komplexitätsgrad des Suchraums für die Techniken aus GIA nicht primär von der Anzahl der Domainvariablen, über die gesucht wird, sondern auch wesentlich vom Wert des Δ-Optimums und dem Δ-Init-Wert abhängt[1]. Für die DSPStone-Benchmarks war das Δ-Optimum nicht größer als zwei und der Δ-Init-Wert lag i.d.R. sehr nahe beim Δ-Optimum. Dies führte zu einem sehr guten Laufzeitverhalten.

- Um ein objektiveres Maß des Laufzeitverhaltens der Techniken aus GIA zu erhalten, wurden interne Benchmarks konstruiert, die größere Δ-Optima aufwiesen. Optimale graphbasierte Instruktionsauswahl war für kleine, mittelgroße und große Datenflussgraphen noch in akzeptablen Laufzeiten möglich, sofern sich das Δ-Optimum im Bereich kleiner gleich 8 bewegte (für die ADSP2100-Familie).

- Zu beobachten war weiterhin, dass gute Resultate schon relativ frühzeitig generiert werden. Resultate, die innerhalb der ersten Sekunde der Optimierungsläufe generiert wurden, hatten eine mittlere Abweichung von 7% von den optimalen bzw. besten gefundenen Resultaten. Diese Beobachtung kann ausgenutzt werden, um Laufzeitschranken für die Optimierungsläufe zu definieren.

[1] Zur Erinnerung: Bei der Assoziation eines Basisblocks mit Kostenvariablen ist eine Teilmenge dieser Kostenvariablen schon an feste Werte gebunden, dessen Summe S eine untere Schranke der Kosten darstellt. Die Differenz aus bestem Resultat und S ist gerade das Δ-Optimum. Die Differenz aus den Kosten der zuerst gefundenen Lösung und S ist der Δ-Init-Wert.

- Durch die Partitionierung großer Basisblöcke in eine Menge kleinerer noch handhabbarer Basisblöcke hat sich gezeigt, dass sich auch die grossen Basisblöcke innerhalb weniger Sekunden optimieren ließen, wobei sich eine mittlere Abweichung von 4% von den besten gefundenen Resultaten ergab. Durch Permutation über der Reihenfolge, in der die Partitionen eines Basisblocks optimiert werden, konnte die mittlere Abweichung noch auf 1.5% reduziert werden.

Die einzusetzenden Strategien und Timeouts für GIA können durch den Benutzer eingestellt werden. Diese könnten aber auch durch den Compiler selbst bestimmt werden, da der Δ-Init-Wert ein Kriterium liefert, ob zu einem gegebenem Basisblock noch in akzeptabler Zeit mit einem exakten Resultat gerechnet werden kann.

Die Techniken aus GIA generieren nicht nur eine Lösung, sondern liefern als Resultat eine Menge von alternativen Überdeckungen, wobei jede dieser Alternativen eine optimale Überdeckung gemäß der gefundenen optimalen Kosten darstellt. Hierdurch werden für die nachfolgenden Phasen genügend Freiheitsgrade aufrechterhalten, um guten Code zu generieren, wie die Resultate der nachgeschalteten Techniken aus I^3R gezeigt haben. Durch die Beibehaltung alternativer Überdeckungen wird die Instruktionsauswahl in die nachfolgenden Phasen integriert.

Integrierte Instruktionsauswahl, Instruktionsanordnung und Registerallokation

Es wurden Techniken zur integrierten Instruktionsauswahl, Instruktionsanordnung und Registerallokation vorgestellt, die alle auf demselben CSP-Modell I^3R_{CSP} aufbauen. Die Unterschiede der Techniken ergaben sich durch den Einsatz verschiedener Labelingstrategien und zweier Kostenmodelle. Das erste Kostenmodell umfasste die reine Minimierung der Instruktionszyklen eines Basisblocks. Das zweite Kostenmodell erweiterte das erste Modell um die Berücksichtigung des Registerdrucks eines Basisblocks. Der erzeugte Code wurde mit handgeneriertem und compilergeneriertem Assemblercode der DSPStone-Benchmarks verglichen und die Resultate können wie folgt zusammengefasst werden:

- Unter Einsatz der besten betrachteten Labelingstrategien und ohne Betrachtung des Registerdrucks lag der mittlere Overhead des Codes im Vergleich zum handgeneriertem Assemblercode bei nur 34%. Der GNU-Compiler[2] wies einen mittleren Overhead von 259% auf. Bis zu 27 FMOs[3] konnten noch in einer Zeit bis zu 500 CPU-Sekunden durch die beste Lösungsstrategie optimiert werden. Die Generierungszeitpunkte der optimalen Resultate lagen alle im Bereich einer CPU-Sekunde.

- Unter Berücksichtigung des Registerdrucks lag der mittlere Overhead bei 21%. Für die datenflussdominierten Benchmarks konnte sogar ein mittlerer Overhead von -2% erreicht werden. Die Laufzeiten lagen bei maximal 900 CPU-Sekunden.

[2] Zur Erinnerung: Der compilergenerierte Code wurde vom GNU-Compiler der ADSP2100-Familie generiert, der auch der einzige für diese Prozessorfamilie verfügbare Compiler ist.

[3] Zur Erinnerung: Dies war die größte Anzahl von FMOs, die in jeweils einem Basisblock der betrachteten DSPStone-Benchmarks vorkam.

- Die Anwendung der Partitionierung resultierte in nur sehr geringen Abweichungen der Codegüte. Der mittlere Overhead lag ohne Betrachtung des Registerdrucks bei nur 24% und konnte mit Berücksichtigung des Registerdrucks auf 22% reduziert werden. Die Laufzeiten der Partitionierung lagen mit und ohne Registerdruck alle unter einer CPU-Sekunde.

Allgemeine Konzepte zur Phasenkopplung

Die in der Arbeit umgesetzten Techniken umfassen mehrere allgemeine Konzepte zur Integration von Codegenerierungstechniken:

- *Verzögerung des Bindens von Ressourcen:* Durch die Beibehaltung von Freiheitsgraden in den Ressourcen, haben die nachfolgenden Phasen mehr Flexibilität bei deren Entscheidungen. Die bestehenden wechselseitigen Beziehungen können durch die Constraints im Constraintstore in nachgeschaltete Phasen propagiert werden.
- *Vereinigung der CSP-Modelle unterschiedlicher Phasen:* Hierdurch werden beim Lösen der Constraints sowie bei der Optimierung alle Aspekte der verschiedenen Phasen simultan berücksichtigt. Durch die Integration von Teil-CSP-Modellen ist es möglich, unterschiedliche feinkörnige Kopplungsgrade von Phasen zu erreichen.
- *Integration von Kostenmodellen zur Abschätzung der Auswirkungen von Entscheidungen einer Phase auf nachgeschaltete Phasen:* Anstatt die Aspekte einer Phase exakt zu modellieren und sie in andere Phasen zu integrieren, können nachgeschaltete Phasen auch durch ein entsprechendes Kostenmodell berücksichtigt werden. Dies umfasst i.d.R. zwar nur eine Abschätzung der möglichen Effekte, ist aber oft schon ausreichend, um Vorteile für die nachfolgenden Phasen zu verschaffen.

8.3.2. Adaption an neue Prozessoren

Die Codequalität hatte bei der Entwicklung von Techniken zu Lasten der Generalisierung Priorität. Daher wurden die Techniken zunächst für dedizierte Prozessoren entwickelt. Allerdings konnten die Techniken schon zum größten Teil durch generische und modulare CSP-Modelle umgesetzt werden, so dass hier eine Basis zur schnellen Adaption der Techniken an neue Prozessoren geschaffen wurde. Hier bietet gerade die intuitive Spezifikation von FMOs eine elegante Methode zur schnellen Adaption neuer Befehlssätze. Weiterhin wurde die Zwischenrepräsentation CoLIR entwickelt, die als Grundlage zur Entwicklung neuer phasengekoppelter Techniken dient und eine einfache Integration neuer bzw. die Neukombination bestehender Techniken ermöglicht. Der intuitive Ansatz von CoLIR zur Darstellung alternativer Überdeckungen ermöglicht eine schnelle Entwicklung von neuen CSP-Modellen.

Auch der constraintbasierte Ansatz liefert ein hohes Potenzial, Techniken schnell an die Eigenschaften neuer Prozessoren anzupassen:

- Constraints können auf einer sehr hohen Sprachebene formuliert werden, was eine schnelle Entwicklung und Implementierung neuer Techniken ermöglicht.
- Bestehende CSP-Modelle können sehr einfach um neue - gegebenenfalls prozessorspezifische - Aspekte erweitert werden, ohne diese modifizieren zu müssen.
- Die schnelle Adaption neuer Kostenmodelle und neuer Labelingstrategien erlaubt es, Optimierungsstrategien prozessorspezifisch anzupassen.
- Die einfache Neukombination von CSP-Modellen erlaubt darüber hinaus eine schnelle Entwicklung neuer phasengekoppelter Techniken unter Verwendung bereits existierender Modelle, deren Kopplungsgrad an die Bedürfnisse des Prozessors angepasst werden kann.

8.3.3. Einsatz von CLP

Es werden zuerst die Vorteile, die sich schon nur durch die reine Constraint-Programmierung ergaben, aufgezeigt. Danach werden die Vorteile, die sich zusätzlich noch durch die Anwendung der Constraint-Logikprogrammierung ergaben, erläutert. Hierdurch soll klar herausgestellt werden, dass zur Umsetzung der in dieser Arbeit vorgestellten Techniken auch andere Systeme zur Constraint-Programmierung in Erwägung gezogen werden können, da sich die Vorteile der beschriebenen Techniken nicht allein auf den Einsatz von CLP beschränken und sich auch unabhängig von der Logikprogrammierung realisieren lassen.

Constraint-Programmierung

Der Constraintansatz bietet eine saubere Trennung zwischen dem CSP-Modell, dem Kostenmodell, der Lösungssuche (Labelingstrategie) und der Optimierung. Durch die Trennung ist es möglich, zu einem gegebenen CSP-Modell schnell und einfach eine Vielfalt von Labelingstrategien zu testen, ohne das CSP-Modell dabei ändern zu müssen. Vordefinierte Such- und Optimierungsmethoden befreien den Entwickler von Compilern zunächst von der Arbeit, diese selber zu implementieren. Der Benutzer ist aber auch in der Lage, hier eigene Methoden zu entwickeln. Eine der größten Stärken des constraintbasierten Ansatzes ist es, dass Teilaspekte der Probleme modular und intuitiv spezifiziert und dann einfach zu einem Ganzen vereinigt werden können. Hierdurch wird ein klares und handhabbares Konzept zur Phasenkopplung gegeben. Weiterhin ist es hierdurch auch möglich, Erweiterungen vorzunehmen, ohne das bestehende Modell ändern zu müssen.

Constraint-Logikprogrammierung

Neben den schon erwähnten Vorteilen der Constraint-Programmierung bietet die Constraint-Logikprogrammierung durch die Einbettung von Constraintlösen in die Logikprogrammierung einen äußerst intuitiven und natürlichen Sprachrahmen zur Spezifikation von Constraints und zur Definition neuer benutzerdefinierter Constraints. Weiterhin bietet sie die Möglichkeit, disjunktive Regeln zu spezifizieren. Der sich daraus

ergebende Nichtdeterminismus bei der Ausführung logischer Programme wird implizit durch das System umgesetzt. Hierdurch ergibt sich eine relativ einfache Formulierung und Implementierung von Suchalgorithmen. Die implizite Handhabung von Nichtdeterminismus umfasst auch einen Backtrackingmechanismus, der den ursprünglichen Zustand an einem Verzweigungspunkt automatisch wiederherstellt. Daher ist der Entwickler davon befreit, Veränderungen des Zustands mitprotokollieren zu müssen.

8.4. Konklusion

Es hat sich gezeigt, dass der Einsatz exakter phasengekoppelter Methoden, unter Einsatz der Constraint-Logikprogrammierung, durchaus realistisch und bis zu einer bestimmten Basisblockgröße einsetzbar ist. Zur Handhabung großer Blöcke wurden Methoden eingeführt, die zwar keine optimalen Ergebnisse mehr garantieren, aber immer noch Resultate nahe dem Optimum liefern. So können bis zu einer bestimmten Basisblockgröße noch die optimalen Techniken und ab einer bestimmten Größe die suboptimalen Techniken eingesetzt werden.

Zur schnellen Adaption von Compilertechniken an neue Prozessoren wurde die gemeinsame Zwischenrepräsentation CoLIR entwickelt, die als Grundlage zur Entwicklung neuer phasengekoppelter Techniken dient. Die generische Struktur ermöglicht die Entwicklung generischer Techniken. Generische CSP-Modelle können zur Implementierung neuer Techniken wiederverwendet und auf einfache Weise erweitert werden. Eine schnelle Adaption neuer Kostenmodelle und Labelingstrategien erlaubt es, effiziente Optimierungsstrategien prozessorspezifisch anzupassen. Die einfache Neukombination von CSP-Modellen erlaubt darüber hinaus eine schnelle Entwicklung neuer phasengekoppelter Techniken, deren Kopplungsgrad an die Bedürfnisse des Prozessors angepasst ist. Weiterhin ermöglicht auch die abstrakte Spezifikationsmethode eine schnelle Entwicklung und Implementierung neuer Techniken.

Der Einsatz der Constraint-Logikprogrammierung lieferte einen idealen Sprachrahmen zur Spezifikation von integrierten Techniken. Teilphasen können modular und intuitiv spezifiziert und einfach zu integrierten Techniken vereinigt werden, wodurch ein klares und handhabbares Konzept zur Phasenkopplung gegeben wird. Die Spezifikation effizienter suchbasierter Optimierungstechniken wird durch eine saubere Trennung zwischen dem CSP-Modell, dem Kostenmodell und der Lösungssuche (Labeling-Strategie) unterstützt, wodurch schnell und einfach eine Vielfalt von Strategien umgesetzt werden konnte, ohne das CSP-Modell ändern zu müssen.

Die hier vorgestellte Arbeit stellt erst die Grundlagen zum Einsatz der Constraint-Programmierung und der Constraint-Logikprogrammierung im Bereich der Codegenerierung für eingebettete Prozessoren dar. Es sind folgende weitere Arbeiten anzustreben:

- Weiterentwicklung der in dieser Arbeit vorgestellten Ansätze.
- Adaption weiterer Prozessoren (u.a. VLIW-artige Prozessoren). Da sich hier wesentlich größere Suchräume ergeben, wird hier die Anwendung von Timeouts,

Partitionierung und Permutation sicherlich eine sehr große Rolle spielen. Dazu zählt auch die Entwicklung fortgeschrittener Strategien zur Partitionierung von Basisblöcken und zur Permutation der Partitionen.

- Einsatz der Constraint-Programmierung und der Constraint-Logikprogrammierung in weiteren Phasen der Codegenerierung.
- Adaption anderer Constraint-Systeme wie z.B. CPLEX [27], das neben einem IP- und ILP-Solver, auch ein System zur allgemeinen Constraint-Programmierung bietet. Dies wird in Form von C++ Bibliotheken zur Verfügung gestellt.

Die Constraint-Logikprogrammierung hat sich im Einsatzgebiet der Codegenerierung für eingebettete Prozessoren sehr gut bewährt. Von daher sollte die Integration dieses Ansatzes in reale Compiler, zumindest auf der Basis der Constraint-Programmierung, auch der Gegenstand zukünftiger Arbeiten sein.

A. Phasenkopplung-Die Modelle IDB und I^3RS

In diesem Kapitel werden zwei weitere Ansätze zur Phasenkopplung beschrieben, die auf CoLIR und dem CSP-Modell $COVER$ aufbauen. Eine umfassende Erläuterung dieser Techniken, wie es für GIA und I^3R durchgeführt wurde, ist im Rahmen dieser Arbeit nicht mehr möglich, da sie diesen sprengen würde. Die Techniken werden hier nur noch in ihren wichtigen konzeptionellen Eigenschaften beschrieben. Dieser Anhang wurde in die Arbeit aufgenommen, um die Vielseitigkeit und die Flexibilität der in der Arbeit eingeführten Modellierungsmethoden zu demonstrieren. Dazu wird zusätzlich eine weitere Prozessorfamilie - die TMS320C1x / C2x / C5x-Familie von Texas Instruments - betrachtet, die sich wesentlich von der Architektur der ADSP2100-Familie unterscheidet (s.u.). Im Folgenden wird zuerst ein weiterer Phasenkopplungsansatz für die ADSP2100-Familie vorgestellt und dann eine Integration der Instruktionsauswahl, Instruktionsanordnung und Registerallokation für die TMS320C1x / C2x / C5x-Familie beschrieben, die auch die Generierung von Spillcode umfasst.

A.1. ADSP2100-Familie

Der nachfolgend beschriebene Ansatz setzt die Phasenkopplung von Instruktionsauswahl, Instruktionsanordnung und Registerallokation durch verzögertes Binden von Ressourcen um. Dieses Konzept wird auch schon in GIA eingesetzt, wo eine Menge optimaler Überdeckungen generiert und an die nachfolgenden Phasen weitergegeben wird. Somit wird die Auswahl einer konkreten optimalen Überdeckung in die nachfolgenden Phasen verlagert, die durch eine größere Auswahl an Ressourcen mehr Flexibilität in ihren Entscheidungen erhalten. Dieses Konzept wird hier weiterverfolgt, indem Instruktionsauswahl, Registerallokation und Instruktionsanordnung in separaten Phasen hintereinandergeschaltet werden, und jede Phase eine bzgl. ihres Optimierungskriteriums möglichst große Menge an sinnvollen Alternativen beibehält und an die nachfolgende Phase weitergibt. Dabei werden die verbleibenden Constraints nach jeder Phase im Constraintstore beibehalten, so dass sie in den nachfolgenden Phasen mit berücksichtigt werden. Dieses Verfahren wird mit IDB (IDB = *Integration durch Delayed-Binding*) bezeichnet. Es ist im Detail in [11] beschrieben und wird im Folgenden nur grob skizziert.

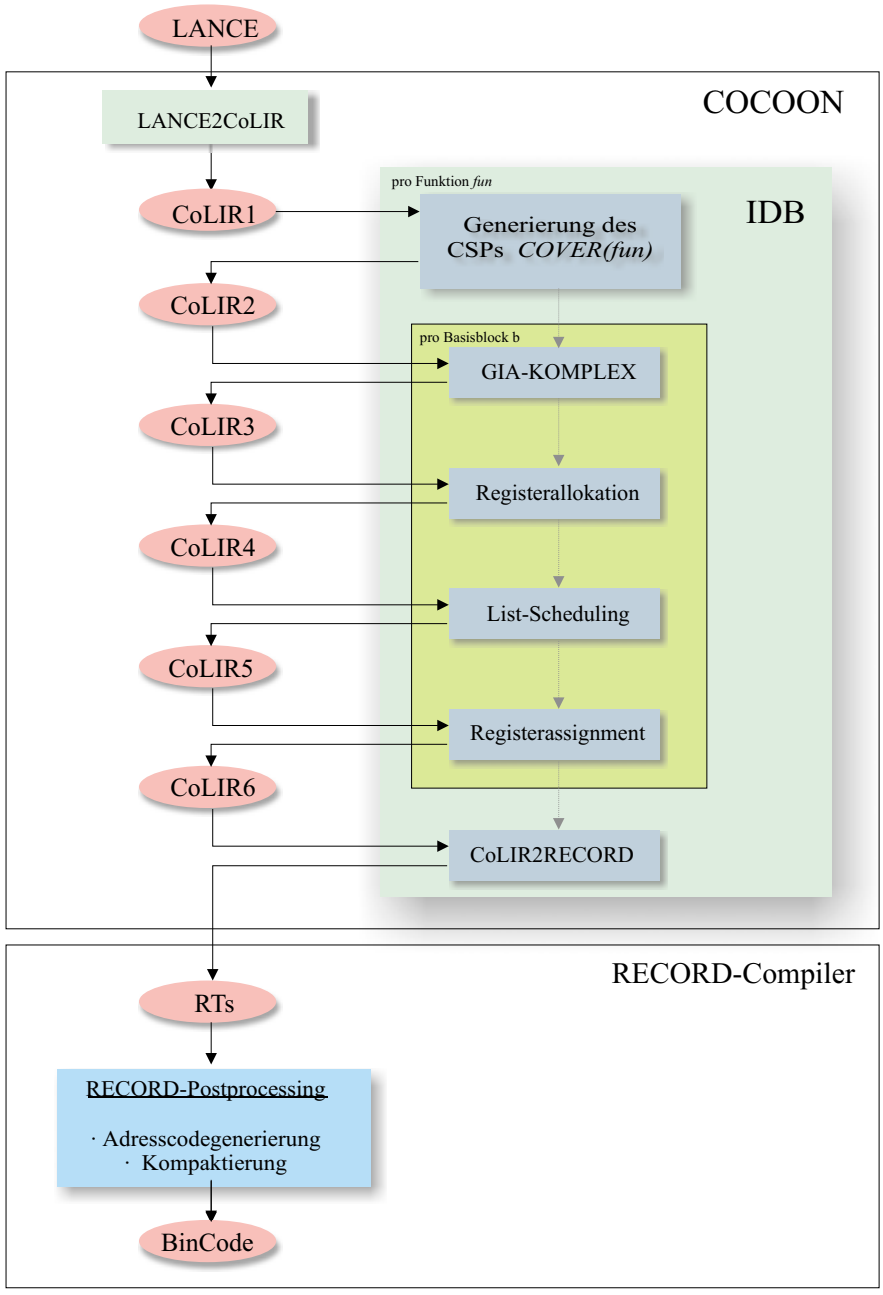


Abbildung A.1.: Überblick über die Teilphasen von IDB.

A.1.1. Konzept

In Abb. A.1 ist der Überblick über die Teilphasen von IDB gegeben:

- Zunächst wird die LANCE-IR in das CoLIR-Programm CoLIR1 transformiert.
- Hieraus werden die alternativen Überdeckungen gemäß $COVER$ generiert, was zu dem CoLIR-Programm CoLIR2 führt.
- In der Teilphase GIA-KOMPLEX wird eine Instruktionsauswahl ähnlich der zu GIA ausgeführt. Der einzige Unterschied ist, dass hier gemäß der optimalen Überdeckungen nur komplexe FMOs erhalten bleiben. Dies wird dadurch erreicht, indem nur die Kostenvariablen der FMOs, nicht aber die Transferkosten gebunden bleiben. Somit werden Sub-FMOs gebunden aber die Datentransferpfade bleiben offen, und es werden auch keine Datentransfer-Operationen eingefügt. Das Resultat ist das CoLIR-Programm CoLIR3.
- Die nächste Phase besteht aus der Registerallokation. Neben dem Einfügen von Spillcode umfasst diese auch das Einfügen notwendiger Datentransfer-Operationen. Diese Phase verfolgt wiederum das Ziel, nur bei Bedarf Ressourcen zu binden. Das Resultat CoLIR4 umfasst eine Menge alternativer Überdeckungen.
- Der nächste Schritt besteht in der Instruktionsanordnung, der aus der Anwendung eines erweiterten List-Scheduling-Ansatzes besteht. Die Erweiterung besteht in der Handhabung von FMOs und der Beibehaltung alternativer Überdeckungen. Die Ausgabe ist ein CoLIR-Programm CoLIR5.
- Aufgabe der Registerassignmentphase ist es, zunächst eine Lösung bzgl. der verbliebenen Constraints im Constraintstore zu generieren. Dadurch werden den Setzungen und Benutzungen der FMOs von CoLIR5 gemäß der Registerfilezuordungen konkrete Register zugeordnet. Wie schon bei GIA und bei I^3R kann es passieren, dass keine Lösung existiert (was bislang aber nicht vorgekommen ist). Hier kann als Ausweg Korrekturcode generiert werden, bei dem dann alle Ressourcen gebunden werden, und es muss eine erneute Registerallokation durchgeführt werden.
- Die Postprocessingphase wird bei IDB durch den RECORD-Compiler [69] durchgeführt, der die Generierung von Adresscode und eine abschließende Kompaktierung durchführt. Dazu wird CoLIR in das Format der Adresscodegenerierung des RECORD-Compilers transformiert. Die Ausgabe von RECORD ist ein Maschinenprogramm in Form von Binärcode und einem “lesbaren” Format, das die Maschineninstruktionen in Form der parallel ausgeführten RTs darstellt.

Es werden nachfolgend nur noch die elementaren Resulte des Ansatzes erläutert.

A.1.2. Resultate

Mit IDB kann wiederum Code generiert werden, der mit dem handgenerierten Code der DSPStone-Benchmarks gleichwertig ist. Die Laufzeiten von IDB liegen im Bereich von wenigen Minuten (für Details siehe [11]).

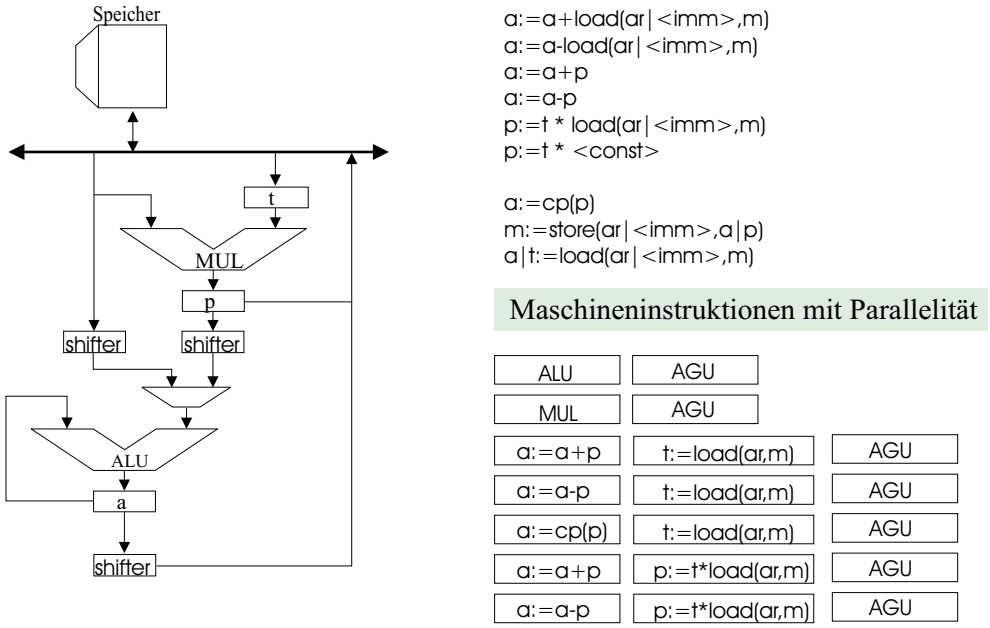


Abbildung A.2.: Prozessorarchitektur der TMS320C1x / C2x / C5x-Familie und ein Ausschnitt aus dessen Befehlssatz.

A.2. Texas Instruments TMS320C1x/C2x/C5x-Familie

Im Folgenden wird eine Technik zur integrierten Instruktionsauswahl, Instruktionsanordnung und Registerallokation beschrieben, die die Generierung von Spillcode umfasst. Die Betrachtung der Spillcodegenerierung war in I^3R nicht mehr mit vertretbarem Aufwand und akzeptablen Laufzeiten zu realisieren. Für die hier betrachtete TMS320C1x / C2x / C5x-Familie kann dies zumindest für die Nicht-AGU-Registerfiles umgesetzt werden, da es sich hierbei um Registerfiles mit jeweils nur einem Register handelt. Die Handhabung von Spillcode läßt sich somit durch Constraints modellieren, die ähnlich der Behandlung von Ressourcenkonflikten bei Funktionseinheiten sind. Weiterhin steht bei einem Registerfilekonflikt der Spillkandidat fest, wodurch sich ein Suchraum ergibt, für den noch in akzeptabler Zeit Lösungen gefunden werden können. Das hier beschriebene Modell wird mit I^3RS (I^3RS = I^3R und Spilling) bezeichnet. Es wird zuerst kurz die Prozessorarchitektur beschrieben und dann die wesentlichen Konzepte von I^3RS erläutert. Danach wird das CSP-Modell skizziert und die Resultate beschrieben.

A.2.1. Prozessorarchitektur und Befehlssatz

In Abb. A.2 ist ein Teil des Datenpfades der TMS320C1x / C2x / C5x-Familie dargestellt, der bei allen Prozessoren dieser Familie gleich ist. Desweiteren ist eine Teilmenge der Maschinenoperationen sowie der Maschineninstruktionen zu sehen. Die Prozes-

soren der TMS320C1x / C2x / C5x-Familie besitzen eine AGU (nicht in der Abbildung gezeigt), eine ALU, eine Multiplikationseinheit (MUL) und mehrere Shifter, die der ALU vor- und nachgeschaltet sind. Das Resultat einer ALU-Operation wird in den Akkumlator a geschrieben, der erste Operand kommt wiederum aus a und der zweite Operand kommt entweder aus dem Akkumulator p oder aus dem Speicher. Das Resultat der MUL-Operationen wird in den Akkumulator p geschrieben, der erste Operand kommt aus dem Speicher und der zweite aus dem t-Register. Die Registerfiles a, p und t bestehen jeweils aus nur einem Register.

Die Multiplikationseinheit unterstützt nur die reine Multiplikation. Eine MAC-Operation, wie sie auf der ADSP2100-Familie existiert, wird nicht unterstützt. Dafür werden komplexe Operationsmuster bestehend aus Shift-Operationen und ALU-Operationen unterstützt. ALU- und MUL-Operationen können direkt auf einen Operanden aus dem Speicher zugreifen. Wie schon in Kapitel 5.3.3 bzgl. der Richtlinien zur Verwendung von Speicherzugriffen erwähnt, werden diese unmittelbaren Speicherzugriffe durch Sub-FMOs der Klasse $load$ realisiert. Somit werden diese Operationen auch durch komplexe Operationen modelliert.

ALU- und MUL-Operationen können jeweils mit einer AGU-Operation parallelisiert werden, sofern diese die Adresse des in der Operation verwendeten Speicherzugriffs modifiziert. Diese Adresse muss dann in einem der Adressregister des Adressregisterfiles ar stehen. Weiterhin kann eine ALU-Operation oder ein Datentransfer von p nach a ($a := p$) parallel zum Laden des t-Registers und zu einer AGU-Operation durchgeführt werden. In diesem Fall ist der einzige Speicherzugriff durch $t := load(ar, m)$ gegeben, wobei die Adresse des Speicherzugriffs in einem Register ar_i von ar steht und die AGU-Operation diese Adresse in ar_i modifiziert. Es besteht die Möglichkeit eine ALU-, MUL- und AGU-Operation parallel auszuführen, wobei dann nur die MUL-Operation einen Speicherzugiff umfassen darf. Es sei bemerkt, dass die parallelen AGU-Operationen jeweils optional ausgeführt werden können.

A.2.2. Das Konzept

Überblick

In Abb. A.3 wird ein Überblick über die Teilphasen von I^3RS gegeben:

- Zunächst wird die LANCE-IR nach CoLIR1 transformiert. Es werden daraus die alternativen Überdeckungen und das CSP $COVER(fun)$ für jede Funktion fun generiert, die dann durch CoLIR2 gegeben sind.
- In CoLIR2 werden die dynamischen Datentransferpfade eingefügt, wodurch das CoLIR-Programm CoLIR3 entsteht.
- Zu jedem Basisblock von CoLIR3 wird das CSP $I^3RS_{CSP}(b)$ generiert. In der Optimierung werden den FMOs aus CoLIR3 feste Instruktionszyklen zugeordnet, mit dem Optimierungsziel, die Anzahl der Instruktionszyklen pro Basisblock zu minimieren.

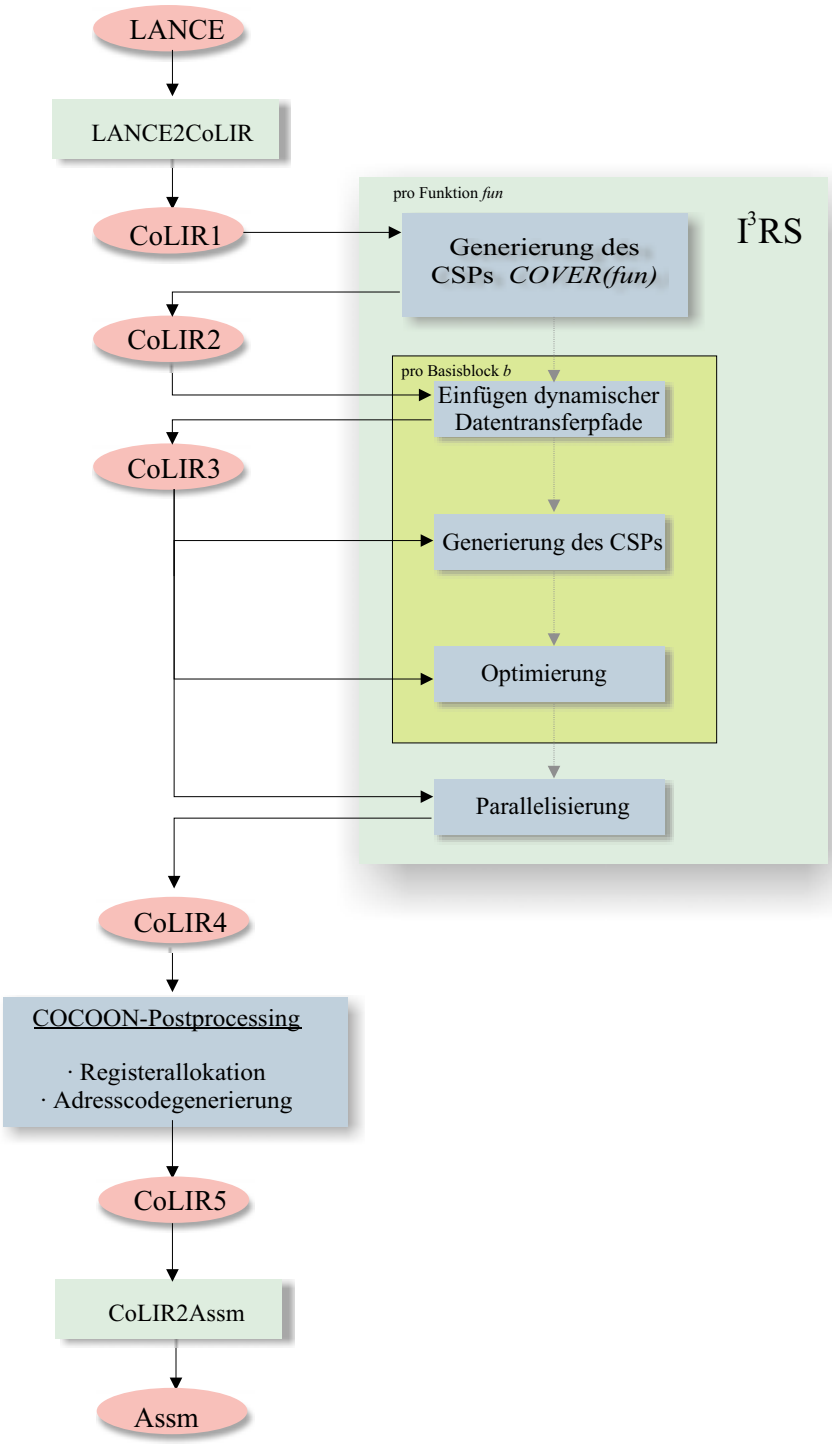


Abbildung A.3.: I^3RS für die TMS320C1x / C2x / C5x-Familie im Überblick.

- Die FMOs werden gemäß der Zuordnungen zu Instruktionszyklen (CoLIR4), entsprechenden Maschineninstruktionen zugeordnet und parallelisiert. Dabei werden Dummy-Copy-Operationen eliminiert und partielle FMOs zu komplexen FMOs zusammengefügt.
- Es wird eine Postprocessingphase durchgeführt, die eine globale Registerallokation und die Adresscodegenerierung umfasst. Das Resultat ist Pseudo-Assemblercode.

Nachfolgend werden die zentralen Konzepte zur Modellierung der dynamischen Datentransferpfade und der komplexen Operationen in I^3RS gezeigt.

Modell der dynamischen Datentransferpfade

Dynamische Datentransferpfade sind bei einer vollen Integration wichtig, da sie explizit die potenziellen Möglichkeiten repräsentieren, Datentransferpfade zwischen Setzungen und Benutzungen zu bilden. Ohne die explizite Darstellung der notwendigen Datentransfer-Operationen ist es nur bedingt möglich, die benötigten Registerfiles für Setzungen und die benötigten Slots für die Datentransfer-Operationen hinreichend genau zu erfassen. Zum einen erfordert die Einbeziehung des Spillcodes Informationen darüber, welche Registerfiles benutzt werden, was ohne die Darstellung der Datentransfer-Operationen nur sehr lückenhaft möglich ist, da in diesem Fall nicht alle Setzungen bekannt sind. Zum anderen ist es ohne Darstellung der Datentransfer-Operationen (besonders im Kontext restriktiver Parallelität) nicht möglich, genau zu sagen, welche Datentransfer-Operationen parallelisiert werden und welche nicht. In diesem Fall hätte man auch kein genaues Modell der benötigten Instruktionsanzahl.

Die dynamischen Datentransferpfade werden pro Basisblock zwischen den Setzungen und Benutzungen eingefügt. Dabei kann zunächst davon ausgegangen werden, dass eine feste Länge k bekannt ist, so dass durch eine Sequenz von k FMOs jeder notwendige Datentransferpfad zwischen beliebigen Setzungen und Benutzungen dargestellt werden kann. Dabei muss berücksichtigt werden, dass durch die Datentransfer-Operationen eines Pfades auch die Möglichkeit zum Spilling enthalten sein muss. Bei der TMS320C1x / C2x / C5x-Familie ist $k = 3$. Hierdurch kann sich die Knotenanzahl eines Datenflussgraphen potenziell um $3 * |E|$ erhöhen, wobei E die Menge der Kanten ist. Da man hierdurch sehr schnell an die Grenzen noch möglicher exakter Suche stößt, werden die Längen der dynamischen Datentransferpfade in Abhängigkeit von den Operatoren der FMOs und deren Setzungen und ihren Benutzungen gemacht, da hier nicht in jedem Fall die maximale Pfadlänge von 3 notwendig ist.

Die Modellierung dynamischer Datentransferpfade wurde schon in Kapitel 5.2.2 S. 122 beschrieben. Einer der wesentlichen Modellierungstricks besteht in der Einführung von Dummy-COPY-Operationen, deren Zuordnung zur ausgezeichneten Funktionseinheit $none$ dazu führt, dass die COPY-Operation durch einen identischen Datentransfer umgesetzt wird (X:=X). In I^3RS führen solche Datentransfer-Operationen zu keinen Ressourcenkonflikten und benötigen auch keine Ausführungszeit. Sie werden somit quasi ignoriert. Trotzdem werden durch sie einige Sonderbehandlungen in der Definition der Constraints zu Formulierung der Datenabhängigkeiten und der

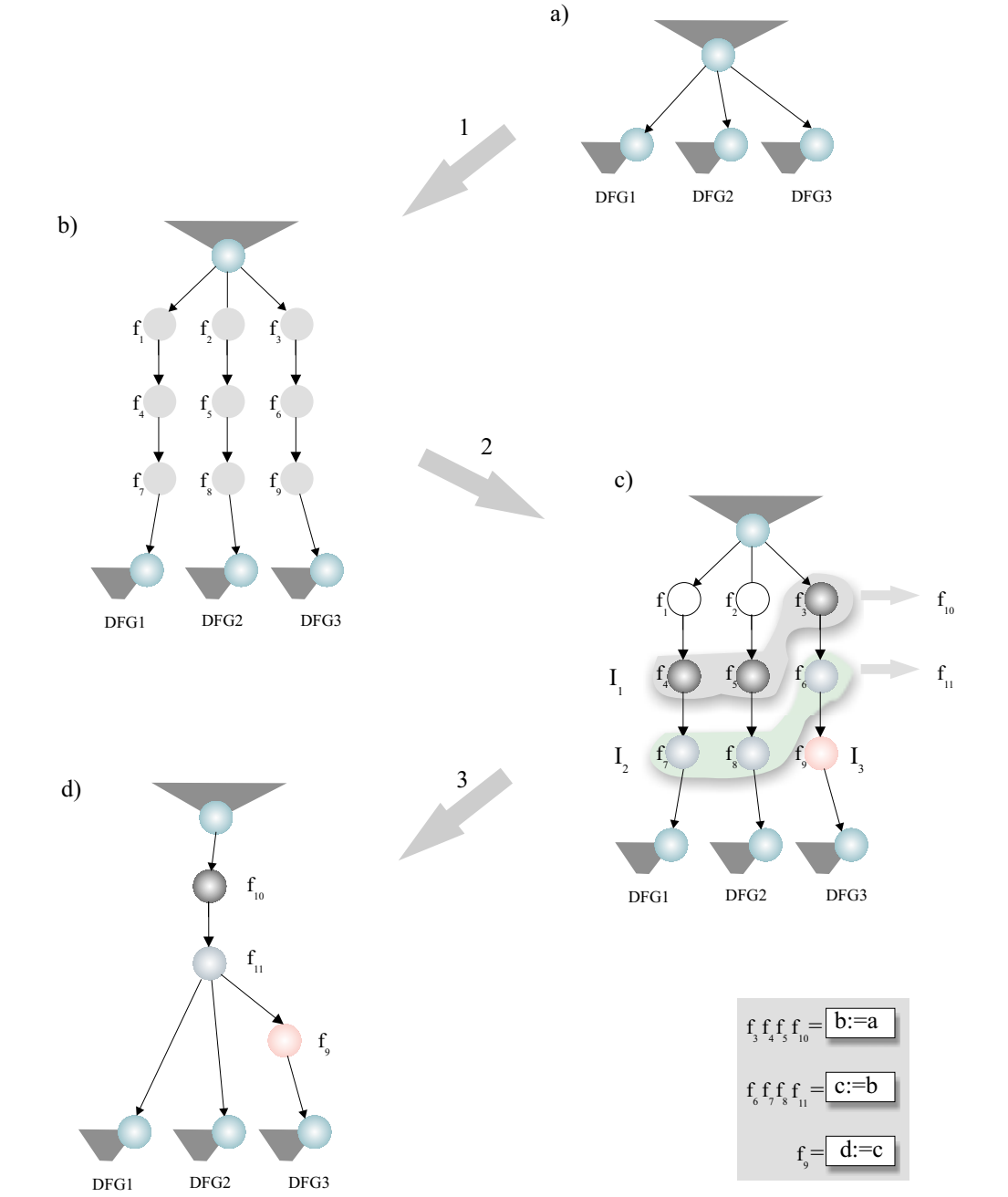


Abbildung A.4.: Überlagerung von dynamischen Datentransferpfaden

Ressourcenkonflikte notwendig (s.u.). Um das Spilling durch die dynamischen Datentransferpfade zu erfassen, wird von dem in Kapitel 5.2.2 beschriebenen Konzept Gebrauch gemacht, dass simultan LOAD-, STORE- und COPY-Operationen durch eine einzige FMO eines dynamischen Datentransferpfades repräsentiert werden können.

Überlagerung von Datentransferpfaden

Besonderes Augenmerk gilt hier der Handhabung der Datentransferpfade von CSEs. Von der Setzung einer CSE zu jeder Benutzung gibt es jeweils einen eigenen dynamischen Datentransferpfad (siehe Abb. A.4 b)). Um hier die Generierung redundanter Datentransfer-Operationen zu vermeiden, können redundante FMOs derselben CSE parallelisiert werden, ohne dass hierdurch ein Ressourcenkonflikt entsteht. Das heißt, dass zwei Datentransfer-Operationen, die auf zwei unterschiedlichen Pfaden derselben CSE liegen, zum gleichen Zeitpunkt auf derselben Funktionseinheit ausgeführt werden dürfen. Es wird einfach angenommen, dass solche FMOs zu einer FMO verschmolzen werden. Es wird hier von der *Überlagerung* von FMOs gesprochen. Dabei ist es wichtig, dass sich nur solche FMOs überlagern können, die bzgl. der Ressourcen absolut identische Datentransfer-Operationen repräsentieren.

Überlagerung von FMOs

Das Beispiel in Abb. A.4 skizziert im Schritt 1 zunächst das Einfügen der dynamischen Datentransferpfade in einen Datenflussgraphen. Jeder der drei eingefügten Pfade besteht jeweils aus drei FMOs (durch die grauen Kreise angedeutet). In Schritt 2 wird die Optimierung durchgeführt, wonach die FMOs bestimmten Instruktionszyklen zugeordnet worden sind. In Abb. A.4 c) sind f_1 und f_2 auf Dummy-COPY-Operationen abgebildet worden ($FE_{f_1} = none$ und $FE_{f_2} = none$). f_3, f_4 und f_5 werden demselben Instruktionszyklus I_1 zugeordnet, und wir nehmen an, dass sich alle drei überlagern. Es sei nochmal bemerkt, dass dies impliziert, dass alle Ressourcen von f_3, f_4 und f_5 jetzt identisch sind. Sie werden zu einer einzigen neuen Maschinenoperation f_{10} verschmolzen. f_6, f_7 und f_8 werden ebenfalls überlagert und der neuen Maschinenoperation f_{11} zugeordnet. Die Datentransfer-Operation f_9 bleibt erhalten. Die daraus resultierenden Datentransferpfade (jetzt nicht mehr dynamisch) sind in Abb. A.4 d) gezeigt. Hier ist zu sehen, dass die ersten beiden Datentransferpfade zu einem Pfad $[f_{10}, f_{11}]$ zusammengefallen sind und der Dritte aus der Verlängerung dieses Pfades besteht.

Im Folgenden werden die FMOs der dynamischen Datentransferpfade eines Basisblocks b durch die Menge $DDP(b)$ gegeben, wobei $DDP(b) \subset FMO(b)$ gilt. Weiterhin ist $ddp(f)$ die Menge der Datentransfer-Operationen, die durch die eingefügten Datentransferpfade zwischen einer Setzung $f_{[0]}$ und deren Benutzungen gegeben sind.

Komplexe FMOs

Die Modellierung von komplexen FMOs basiert nach wie vor auf der Darstellung partieller FMOs auf der Basis flüchtiger Komponenten (siehe Kapitel 5.3). Auch Sub-FMOs werden der Funktionseinheit $none$ zugeordnet, und sie führen zu keinen Ressourcenkonflikten bei der Parallelisierung und werden in “Nullzeit” ausgeführt. Die eigentlichen Ressourcenkonflikte und die Ausführungszeiten komplexer FMOs werden durch die Wurzeln der komplexen FMOs gehandhabt.

A.2.3. Das CSP-Modell I^3RS_{CSP}

I^3RS_{CSP} umfasst das Teil-CSP-Modell $COVER_{dp}$. Wie schon in GIA und in I^3R wird zunächst pro CoLIR-Funktion fun das CSP $COVER(fun)$ generiert. Die Erweiterung auf $COVER_{dp}$ und auf die restlichen Domainvariablen und Constraints von I^3RS_{CSP} wird wiederum lokal pro Basisblock durchgeführt.

Domainvariablen

- Wie schon in I^3R ist jeder Basisblock b mit einer Domainvariablen $I_{b_{max}}$ assoziiert, welche die Anzahl notwendiger Instruktionszyklen, die minimiert werden soll, repräsentiert.

- Jeder FMO $f \in FMO(b)$ wird eine Domainvariable I_f (Instruktionszyklus) zugeordnet, welche den Ausführungszeitpunkt von f widerspiegelt, und es gilt $0 \leq I_f \leq I_{b_{max}}$. Weiterhin ist $fv \in FMO(b) \cup VAR_{in}(b)$ mit einer Endzeit E_f assoziiert, die die maximal notwendige Lebensdauer des von fv generierten Wertes wiedergibt, und es gilt $0 \leq E_{fv} \leq I_{b_{max}} + 1$. Durch $I_{b_{max}} + 1$ werden Lebensdauern modelliert, die über die Basisblockgrenzen hinausgehen. Dabei werden keine Lebensdauern von Programmvariablen berücksichtigt, die zwar über die Lebensdauer des Basisblocks hinweg lebendig sind, aber im Basisblock weder gesetzt noch benutzt werden (Registerdruck, der durch solche Programmvariablen entsteht, wird durch die globale Registerallokation in der Postprocessingphase gehandhabt).

- Zu jedem $f \in FMO(b)$ gibt es eine Domainvariable $D_f \in \{0, 1\}$, welche die Abarbeitungsdauer von f unter der potenziellen Zuordung einer FMO zu $none$ widerspiegelt. Es werden nur Maschinenoperationen der TMS320C1x / C2x / C5x-Familie modelliert, die innerhalb eines Instruktionszyklus abgearbeitet werden können. Wird eine FMO der Funktionseinheit $none$ zugeordnet, wird durch $D_f = 0$ die "Nullzeit" von Sub-FMOs bzw. von Dummy-COPY-Operationen modelliert.

Mindestlebensdauer

- Jede FMO $f \in FMO(b)$ ist mit der *Mindestlebensdauer* der Operanden assoziiert, die durch die Domainvariable ML_f gegeben ist, und es gilt $0 \leq ML_f \leq I_{b_{max}}+1$. Wird f nicht der Funktionseinheit $none$ zugeordnet, so ist die Mindestlebensdauer der Operanden von f gleich I_f. Wenn f der Funktionseinheit $none$ zugeordnet wird, müssen die Operanden von f mindestens bis zu dem Instruktionszyklus lebendig sein, an dem der letzte datenflussabhängige Nachfolger von f (mit $f \rightarrow^{pos} f' \in DFG(b)$) ausgeführt wird. ML_f wird in diesem Fall mit der maximalen Mindestlebensdauer aller Nachfolger von f gleichgesetzt. Somit wird auch erfasst, dass sich durch die Zuordnung der Nachfolger von f zu $none$ die Lebensdauern der Operanden von f auf die Mindestlebensdauer der Nachfolger der Nachfolger (kein Schreibfehler!) usw. verlängern können.

Pfadnummern

- Zur Modellierung der Überlagerungen der Datentransferpfade von CSEs werden die Mengen $dtp(f)$ mit eindeutigen *Pfadnummern* assoziiert. Alle FMOs von

dynamischen Datentransferpfaden mit $f_1, f_2 \in ddp(f)$ besitzen dieselbe Pfadnummer $pn_{f_1} = pn_{f_2}$ und für $f_1 \in ddp(f)$ und $f_2 \in ddp(f')$ mit $f \neq f'$ gilt $pn_{f_1} \neq pn_{f_2}$. Es können sich nur die Datentransfer-Operationen von dynamischen Datentransferpfade überlagern, welche die gleiche Pfadnummer besitzen. Es sei bemerkt, dass pn_f eine feste Zahl und keine Domainvariable ist.

Datenabhängigkeiten von Programmvariablen

Die Möglichkeit, FMOs der Funktionseinheit $none$ zuzuordnen, erfordert eine differenzierte Handhabung der Datenabhängigkeiten von Programmvariablen und Speicherreferenzen. Wir werden hier zuerst auf die Datenabhängigkeiten von Programmvariablen eingehen. Da CoLIR-Programme in lokaler SSA-Form vorliegen, reicht es, für die Programmvariablen die Datenflussabhängigkeiten zu betrachten, die auch durch den lokalen Datenflussgraphen $DFG(b)$ gegeben sind.

Im Unterschied zu I^3R, wo für jede Kante $f \rightarrow^{pos} f' \in DFG(b)$ ein Constraint $I_f < I_{f'}$ eingefügt wird, wird hier ein Constraint

$$I_f + D_f \leq I_{f'}$$

erzeugt. D_f dient der korrekten Handhabung der zeitlichen Abhängigkeiten zwischen f und seinen Vorgängern und Nachfolgern für den Fall, dass f der Funktionseinheit $none$ zugeordnet wird. Wird f nicht der Funktionseinheit $none$ zugeordnet, so wird D_f auf 1 gesetzt, wodurch die gewohnte Datenflussrelation $I_f < I_{f'}$ ($I_f + 1 \leq I_{f'} \equiv I_f < I_{f'}$) realisiert wird. Um diesen Fall zu handhaben, wird das Constraint

$$FE_f \neq none \;\Rightarrow\; D_f = 1$$

generiert. Die Zuordnung von FMOs zu $none$ wird zunächst an einem Beispiel betrachtet. In Abb. A.5 a) ist ein partieller Datenflussgraph mit den Knoten f_1, f_2 und f_3 dargestellt, in dem f_2 der Funktionseinheit $none$ zugeordnet wird und $f_1 \rightarrow^{pos} f_2 \in DFG(b)$ und $f_2 \rightarrow^{pos'} f_3 \in DFG(b)$ gilt. Rechts neben dem Datenflussgraphen ist eine partielle Instruktionssequenz gegeben, in der skizziert ist, wie f_2 in Form einer Sub-FMO in f_3 eingebettet wird. Durch die Constraints der Datenflussabhängigkeiten muss gewährleistet sein, dass $I_{f_1} < I_{f_3}$ ist und dass I_{f_1} potenziell auch den Wert $I_{f_3} - 1$ annehmen kann. Dies wird durch das Setzen von D_{f_2} auf 0 erreicht. Die allgemeine Realisierung dieser Bedingungen wird durch das folgende Constraint realisiert

$$FE_f = none \;\Rightarrow\; D_f = 0 \wedge I_f = I_{f'},$$

sofern f nur einen Nachfolger $f \rightarrow^{pos} f' \in DFG(b)$ hat. Das zusätzliche Constraint $I_f = I_{f'}$ ist dabei nicht zwingend notwendig. Es verhindert aber, dass z.B. I_{f_2} beliebige Werte zwischen $I_{f_1} < I_{f_2} \leq I_{f_3}$ annimmt, was letztendlich nur zu redundanten Lösungen führt und den Suchraum unnötig erhöht.

Man kann sich schnell klarmachen, dass diese Modellierung auch für FMOs in einer FMO-Sequenz gilt, in der aufeinanderfolgenden FMOs $none$ zugeordnet wird (siehe Abb. A.5b)). In Abb. A.5c) ist ein Beispiel gezeigt, in der eine CSE f_2 mit der Funktionseinheit $none$ assoziiert wird. In Kapitel 5.3 S. 124 wurde beschrieben, dass CSEs, die zu

Sub-FMOs werden, durch jeweils eine Kopie einer Sub-FMO in den jeweilen Nachfolgern realisiert werden (dies entspricht auch der realen Umsetzung). Dies ist ebenfalls in den partiellen Instruktionssequenzen in Abb. A.5c) wiedergegeben, in denen die beiden Kopien $f_{2'}$ und $f_{2''}$ als Sub-FMOs in f_3 und f_4 eingebettet sind (die beiden alternativen Instruktionssequenzen stellen zwei mögliche Anordnungen für f_3 und f_4 dar). Im Modell wird diese Aufspaltung aber erst in der nachfolgenden Phase der Parallelisierung realisiert. Um die zeitlichen Abhängigkeiten gemäß des Datenflusses im Beispiel einzuhalten, reicht es, dass D_{f_2} auf 0 gesetzt wird. Somit ist gewährleistet, dass f_1 auf jeden Fall sowohl vor f_3 als auch vor f_4 ausgeführt wird.

Um redundante Lösungen und unsinnige Wertzuordnungen an I_{f_2} zu vermeiden, wird I_{f_2} auf den kleinsten Instruktionszyklus der Nachfolger gesetzt, was durch das Constraint $minlist([I_{f_3}, I_{f_4}], I_{f_2})$ realisiert wird. Für den allgemeinen Fall wird das Constraint

$$FE_f = none \;\Rightarrow\; D_f = 0 \wedge minlist(Is_f,\, I_f)$$

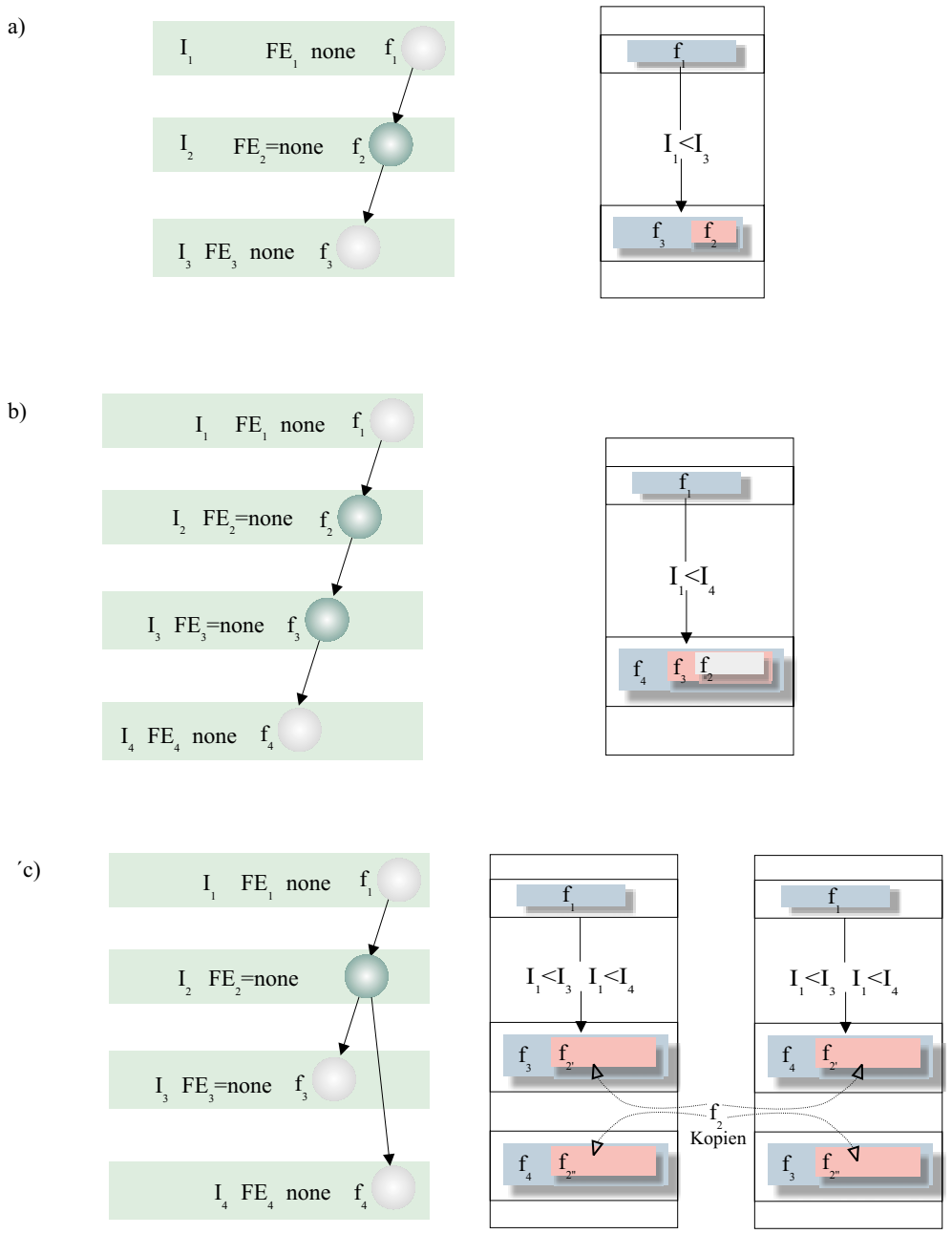
generiert, wobei durch Is die Instruktionszyklen der Nachfolger von f gegeben sind.

Die Datenabhängigkeiten von LOAD- und STORE-Operationen

Es bestehen die folgenden Besonderheiten bei der Handhabung der Datenabhängigkeiten von Speicherreferenzen im Unterschied zur Umsetzung in I^3R:

- LOAD-Operationen können Sub-Operationen komplexer Maschinenoperationen werden, wodurch eine besondere Handhabung der Antiabhängigkeiten erforderlich wird.
- LOAD- und STORE-Operationen, die in den dynamischen Datentransferpfaden vorkommen, werden nicht bei den Datenabhängigkeiten der Speicherzugriffe berücksichtigt. Dies ist zulässig, da Speicherzugriffe innerhalb von dynamischen Datentransferpfaden gemäß der Pfadnummern eindeutigen Speicherzellen zugeordnet werden, die zu keinen Konflikten mit den Speicherzugriffen der schon existierenden LOAD- und STORE-Operationen mit $f \notin DDP(b)$ führen können. Die zeitlichen Abhängigkeiten zwischen einer STORE-Operation f_{st} und einer LOAD-Operation f_{ld} , die in einem dynamischen Datentranferpfad in einer Datenflussbeziehung stehen, werden durch die Bezeichnerzuordnungen hergestellt, indem $(f_{st})^{bs}_{[0]}$ (Bezeichner der Setzung der STORE-Operation) und $(f_{ld})^{bs}_{[2]}$ (Bezeichner des zweiten Operanden der LOAD-Operation) gleiche Bezeichner sind. Die Abhängigkeiten werden somit durch die lokalen Datenflussgraphen wiedergegeben.

Es wird jetzt zunächst die Handhabung der Datenflussabhängigkeit (true dependency) zwischen STORE- und LOAD-Operationen gezeigt. In Abb. A.6 ist ein Beispiel für eine STORE-Operation f_1 und einer LOAD-Operationen f_2 dargestellt, wobei f_2 der Funktionseinheit $none$ zugeordnet wird (in der partiellen Instruktionssequenz ist wiederum die Einbettung der Sub-FMO f_2 in Form der beiden Kopien $f_{2'}$ und $f_{2''}$ in die



(a) Ohne CSEs

Abbildung A.5.: Beispiele zur Handhabung von *none*.

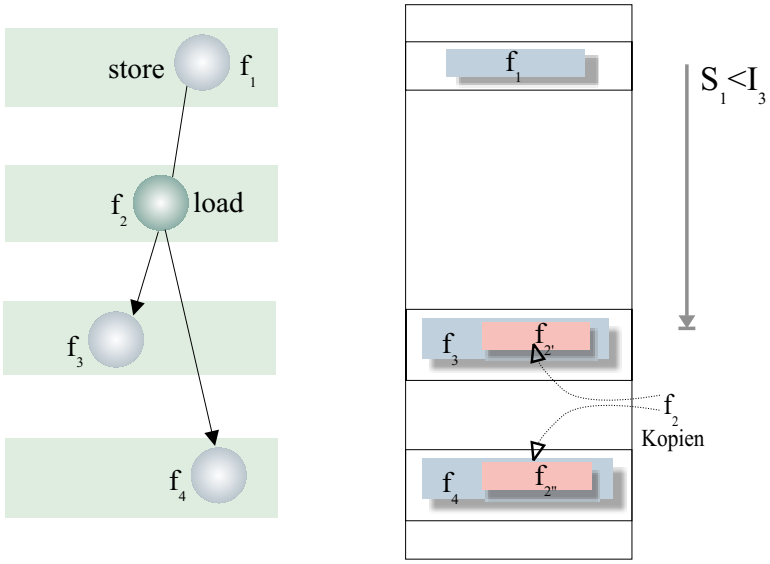


Abbildung A.6.: Handhabung von *true dependencies* bei LOAD- und STORE-Operationen.

FMOs f_3 und f_4 wiedergegeben). Für die STORE-Operation muss gewährleistet sein, dass sie vor f_3 und f_4 ausgeführt wird. Daher ist es hinreichend, für f_1 und f_2 das Constraint $I_{f_1} < I_{f_2}$ zu erzeugen. Allgemein wird für $f \rightarrow^{true,pos} f' \in DDG_{mem}(b)$ das Constraint $I_f < I_{f'}$ generiert. Es ist also keine Sonderbehandlung der Datenabhängigkeiten vor.

Die Umsetzung der Antiabhängigkeiten erfordert allerdings gegenüber I^3R eine modifizierte Handhabung. In Abb. A.7 ist die Antiabhängigkeit zwischen einer LOAD-Operation f_2 und einer STORE-Operation f_5 dargestellt. Darin wird f_2 der Funktionseinheit $none$ zugeordnet (und wird somit Sub-Operation von f_3 und von f_4). Der Unterschied zur üblichen Handhabung der Antiabhängigkeit ist jetzt, dass sich die Lebensdauer des Speicherzugriffs von f_2 auf den maximalen Ausführungszeitpunkt der Nachfolger von f_2 erstreckt. Da der Speicherzugriff ein Operand von f_2 ist, wird dieser Aspekt nun durch die Domainvariable ML_{f_2} wiedergegeben. Somit muss für f_2 und f_5 das Constraint $ML_{f_2} \leq I_{f_5}$ generiert werden. Allgemein wird für $f \rightarrow^{false,pos} f' \in DDG_{mem}(b)$ das Constraint $ML_f \leq I_{f'}$ erzeugt. Wird eine LOAD-Operation nicht der Funktionseinheit $none$ zugeordnet, so ist $ML_f = I_f$ (per Definition von ML_f), wodurch $I_f \leq I_{f'}$ gilt und somit wiederum die Bedingungen der Antiabhängigkeit erfüllt sind.

Die Ausgabeabhängigkeit erfordert keine besondere Handhabung und somit wird für $f \rightarrow^{output,pos} f' \in DDG(b)$ das Constraint $I_f < I_{f'}$ generiert.

Lebensdauern und Spilling

Die maximale Lebensdauer einer FMO wird durch die Endzeit der FMO wiedergegeben und wird zur Realisierung des Spillings benötigt. Wir gehen zuerst auf die Cons-

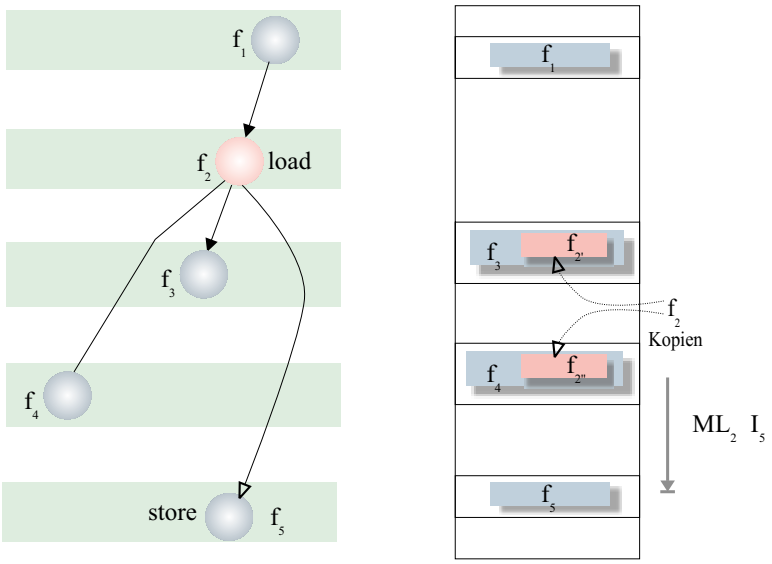


Abbildung A.7.: Handhabung von *false dependencies* bei LOAD- und STORE-Operationen.

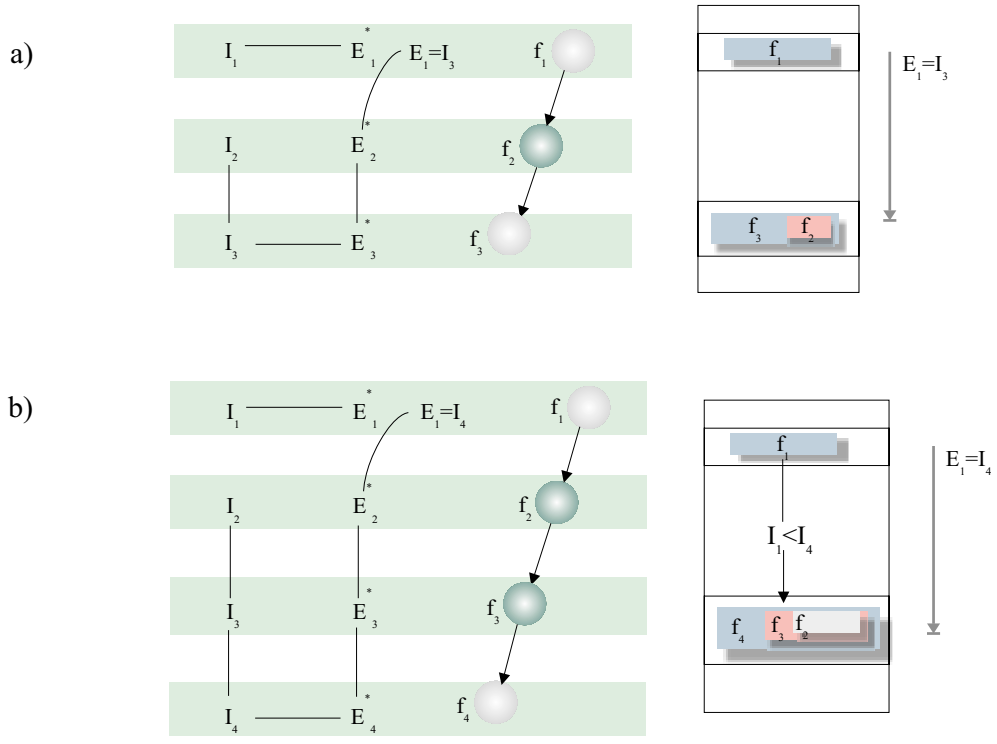


Abbildung A.8.: Handhabung von *none* bei CSEs

traints zur Handhabung der Lebensdauern ein und werden dann die Constraints zum Spilling einführen. Wie beim Registerdruckmodell von I^3R müssen auch hier zwei Fälle bzgl. der Endzeiten einer FMO unterschieden werden:

1. *Fall*: Für eine FMO f gibt es eine Kante $f \rightarrow^{pos} f' \in DFG(b)$, wobei f' eine Ausgangsvariable ist (also $f' \in VAR_{out}(b)$ gilt). In diesem Fall ist der Wert von f auch noch nach Austritt aus dem Basisblock b lebendig, was durch das Constraint
$$E_f = I_{b_{max}} + 1$$
modelliert wird.

2. *Fall*: Es gibt keine Kante $f \rightarrow^{pos} f' \in DFG(b)$ mit $f' \in VAR_{out}(b)$. Dann entspricht die Endzeit E_f dem Maximum der Mindestlebenszeiten von Operanden, die den Wert von f benutzen. Für die Liste $MLs_f = [ML_{f_1}, ..., ML_{f_n}]$ mit $f \rightarrow^{pos} f_i \in DFG(b)$ und $1 \leq i \leq n$ wird das Constraint
$$maxlist(MLs_f,\ E_f)$$
erzeugt.

Wird einer FMO f die Funktionseinheit $none$ zugeordnet, müssen die Lebensdauern der Operanden von f auf das Maximum der ML-Variablen der Nachfolger von f gesetzt werden. Dies wird durch das Constraint $FE_f = none \Rightarrow maxlist(MLs_f, ML_f)$ realisiert, wobei die Liste $MLs = [ML_{f_1}, ..., ML_{f_n}]$ die ML-Variablen aller Nachfolger von f mit $f \rightarrow^{pos} f_i \in DFG(b)$ und $1 \leq i \leq n$ umfasst. Dies stellt sicher, dass sich die Lebenszeiten aller Operanden von f bis zu der letzten Benutzung von f erstrecken.

Wird eine FMO nicht der Funktionseinheit $none$ zugeordnet, so entspricht ML_f dem Instruktionszyklus von f:
$$FE_f \neq none \Rightarrow ML_f = I_f.$$

Spilling wird nur für die Registerfiles a, p und t betrachtet. Das Spilling der AGU-Register des Registerfiles ar wird nicht betrachtet. Da die Registerfiles a, p und t nur aus jeweils einem Register bestehen, müssen Setzungen, die a, p oder t zugeordnet werden, bei überlappenden Lebensbereichen ungleich sein. Bei Gleichheit der Setzungen von f und f' aus $FV(b)$ dürfen diese entweder nur dem Speicher m oder dem AGU-Registerfile ar zugeordnet werden, die gleiche Pfadnummer besitzen oder f bzw. f' sollte der Funktionseinheit $none$ zugeordnet sein. Dies kann durch das folgende Constraint für alle Paarbildungen $f,\ f' \in FV(b)$ realisiert werden:

$$I_{f'} \leq I_f \wedge I_f \leq E_{f'} \Rightarrow f_{[0]} \neq f'_{[0]} \vee f_{[0]} \in \{m, ar\} \vee pn(f) = pn(f') \vee FE_f = none \vee FE_{f'} = none.$$

Dabei können natürlich unnötige Constraintbildungen für FMO-Paare vermieden werden, bei denen klar ist, dass sich entweder die Lebensbereiche oder die Setzungen nicht überlappen können. Der Fall, dass $pn(f) = pn(f')$ ist kann z.B. auch statisch abgefragt werden. In diesem Fall muss das Constraint ebenfalls nicht generiert werden. Weiterhin darf nur für die $f \in FMO(b)$ die Gleichung $FE_f = none$ geprüft werden. Für Eingangs- und Ausgangssetzungen ist sie nicht in dem Constraint enthalten.

Parallelität

- Instruktionstypbedingte Restriktionen in der Verwendung von Ressourcenbenutzungen werden, wie schon in I^3R, durch die Constraints aus $COVER(b)$ abgedeckt.
- Restriktionen zwischen verschiedenen FMOs werden bei der TI-Prozessorfamilie vollständig durch die Instruktionstypen gehandhabt und werden somit ebenfalls durch $COVER(b)$ abgedeckt.
- Bei der Parallelisierung zweier FMOs müssen die Instruktionstypen gleich sein, und es darf kein Konflikt bei der Benutzung der Funktionseinheiten geben:

  $$I_f = I_{f'} \Rightarrow (FE_f \neq FE_{f'} \wedge IT_f = IT_{f'}) \vee FE_f = none \vee FE_{f'} = none \vee pn(f) = pn(f').$$

  Ausnahme ist, dass eine der FMOs der Funktionseinheit $none$ zugeordnet wird, oder beide FMOs dieselbe Pfadnummer besitzen. Zwei FMOs, welche die gleiche Pfadnummer haben und redundante Datentransfer-Operationen darstellen, dürfen überlagert werden, und können somit den gleichen Funktionseinheiten zugeordnet werden. In diesem Fall müssen alle anderen Ressourcen gleich sein, d.h. der Instruktionstyp sowie die Setzung und die Benutzungen dürfen jeweils nur den gleichen Ressourcen zugeordnet werden. Für alle f , $f' \in DDP(b)$ wird daher das Constraint

  $$I_{f'} = I_f \wedge FE_f = FE_{f'} \Rightarrow p(f) = pn(f') \wedge (\bigwedge_{pos \in f} f_{pos} = f'_{pos}) \wedge IT_f = IT_{f'}$$

  eingefügt. Das gezeigte Constraint hat auch bei Dummy-COPY-Operationen korrekte Wirkung.
- Die Bedingungen für AGU-Operationen müssen an die Möglichkeit angepasst werden, dass eine LOAD-Operation f_{ld} der Funktionseinheit $none$ zugeordnet wird, wodurch die LOAD-Operation zur Sub-FMO wird. In diesem Fall kann die AGU-Operation zum Zeitpunkt $ML_{f_{ld}}$ ausgeführt werden.

Kostenmodell

Minimiert wird die Anzahl der notwendigen Instruktionszyklen eines Basisblocks, gegeben durch die Domainvariable $I_{b_{max}}$.

A.2.4. Resultate

Zur Minimierung der Instruktionszyklen sind die Labelingstrategien des dynamischen Labelings der Funktionseinheiten aus I^3R verwendet worden. In den ersten Experimenten haben sich sehr hohe Laufzeiten ergeben, so dass schon für ru, ruN, cm und fir bis zu 25000 CPU-Sekunden notwendig waren und die Optimierungsläufe der weiteren Benchmarks nicht innerhalb von 24h terminierten. Ein wesentlicher Faktor ist es, dass es relativ lange dauert, bis überhaupt eine erste Lösung gefunden wird, wobei der Grund in der Einbeziehung des Spillcodes liegt. So kann es vorkommen, dass

während der Suche eine partielle Anordnung generiert wird, die kein Spilling für eine bestimmte FMO erlaubt, da es keine freien Slots in der partiellen Anordnung gibt, in welche die notwendigen Spill-Operation eingefügt werden können. Daher muss solange über den Suchraum dieser partiellen Anordnung Backtracking durchgeführt werden, bis entweder Slots frei sind, so dass gespillt werden kann, oder das Spilling durch die Umordnung nicht mehr notwendig ist. Das grundsätzliche Problem ist, dass STORE-Operationen nicht parallelisierbar sind, und daher eine freie Maschineninstruktion vorliegen muss, damit eine STORE-Operation eingefügt werden kann.

Tunneln

Ein Ausweg ist das *Tunneln* von STORE-Operationen: Dabei wird bei der Modellierung des Befehlssatzes erlaubt, dass maximal eine STORE-Operation in jeder Maschineninstruktion parallel zu den anderen FMOs ausgeführt werden darf. Da hierdurch die Kosten verfälscht werden, wird jede STORE-Operation, die parallelisiert wird, mit einem Strafzyklus versehen und dadurch der Kostenfehler wieder kompensiert. Die Kostenfunktion wird so abgeändert, dass jetzt die Summe $I_{b_{max}} + SZs$ minimiert wird, wobei SZs die Summenbildung über den Domainvariablen zur Umsetzung der Strafzyklen ist.

Mit dieser Verbesserung wurden die in Tabelle A.1 gezeigten Laufzeiten und Resultate erzielt. Man sieht, dass die Laufzeiten immer noch sehr hoch sind. Für die Benchmarks bi1, biN und cuN terminierten die Optimierungsläufe immer noch nicht innerhalb von 24 CPU-Stunden. Auch die Generierungszeitpunkte liegen wesentlich höher als bei I^3R (bis zu 1078 CPU-Sekunden). Man muss hier aber auch berücksichtigen, dass selbst nach der Reduktion der Pfadlängen die Anzahl der zu optimierenden FMOs immer noch mehr als das doppelte als bei I^3R beträgt. Weiterhin sind die zu handhabenden Constraints wesentlich komplexer, bedingt durch die möglichen Zuordnungen von FMOs zu $none$. Diese Zuordnung war in I^3R nicht mehr notwendig, da in GIA schon die Sub-FMOs und auch die Datentransferpfade fest zugeordnet wurden.

Die Resultate der Codegüte wurden mit handgeneriertem Code der DSPStone-Benchmarks und mit dem Code des kommerziellen Compilers von Texas Instruments (TI-Compiler) der TMS320C1x / C2x / C5x-Familie verglichen. Betrachtet man die Resultate, ergibt sich im Vergleich zum handgenerierten Code für die datenflussdominierten Benchmarks ein Overhead von -1% (40% beim TI-Compiler) und für die kontrollflussumfassenden Benchmarks ein Overhead von 95% (342% beim TI-Compiler). Auch hier wurden beim handgeneriertem Code iterationsübergreifende Optimierungen bei den Schleifen durchgeführt.

Zur Reduktion der hohen Laufzeiten wurde wiederum die Partitionierung eingesetzt, deren Resultate in Tabelle A.2 gezeigt sind. Man hat jetzt insgesamt akzeptable Laufzeiten, die alle innerhalb von 300 CPU-Sekunden liegen. Dies ist aber verglichen zu I^3R immer noch verhältnismäßig hoch. Dort lagen alle Laufzeiten unter Anwendung der Partitionierung bei maximal einer CPU-Sekunde. Betrachtet man die Resultate, ergeben sich nur bei den Benmarks cuN und biN geringe Verschlechterungen (<10%).

Man könnte annehmen, dass hier die Grenzen exakter Techniken und vollständiger Integration erreicht sind, wobei bei der Integration ja schon Abstriche gemacht wurden (z.B. kein Spilling von Werten, die AGU-Registerfiles zugeordnet werden). Trotzdem ist es noch lohnenswert, weitere laufzeitreduzierenden Quellen zu analysieren, die durch den beschriebenen Ansatz sicher noch nicht alle erschöpft wurden.

BM	$t\ (t_{best})$	$\#mis_{cocoon}$	$\#mis_{hand}$	$\#mis_{ti}$	$\%cocoon$	$\%ti$
ru	15(4)	5	5	5	0%	0%
cm	977(64)	10	12	15	-17%	25%
cu	47389(503)	17	16	25	6%	56%
bi1	24h(1078)	16	15	27	7%	80%
ruN	286(18)	5	5	11	0%	120%
cuN	24h(327)	18	17	31	6%	82%
biN	24h(935)	21	12	44	75%	267%
fir	776(23)	4	1	10	300%	900%

Tabelle A.1.: Resultate für I^3RS.

BM	$t\ (t_{best})$	$\#mis_{cocoon}$	$\#mis_{hand}$
ru	3(<1)	5	5
cm	46(3)	10	12
cu	124(5)	17	16
bi1	184(10)	16	15
ruN	2(<1)	5	5
cuN	135(8)	19(18)	17
biN	226(17)	23(21)	12
fir	2(<1)	4	1

Tabelle A.2.: Resultate für I^3RS unter Anwendung der Partitionierung.

B. Abkürzungen und Notationen

AGU address generation unit = Adressgenerierungseinheit.

BB basic block = Basisblock.

$b, b_1, ..., b_n$ sind Bezeichner für Basisblöcke.

CFG control flow graph = Kontrollflussgraph.

CLP Constraint-Logikprogrammierung.

CP Constraint-Programmierung.

CSE common subexpression = gemeinsamer Teilausdruck.

$CSE(b)$ ist die Menge der CSEs des Basisblocks b mit

$$CSE(b) = \{fv | fv \in FMO(b) \cup VAR_{in}(b) \; und \; |edges(fv, b)| > 1\}.$$

CSP constraint satisfaction problem = Problem über endlichen Wertebereichen.

$COVER$ ist das generische CSP-Modell, das die primären Aufgaben der Instruktionsauswahl umfasst.

$COVER(fun)$ ist das CSP von $COVER$ zu einer Funktion fun.

$COVER(b)$ ist das CSP von $COVER$ zu einem Basisblock b.

$COVER_{dp}$ ist eine Erweiterung von $COVER$, bei der dynamische Datentransferpfade explizit repräsentiert sein müssen.

$COVER_{dp}(fun)$ ist das CSP von $COVER_{dp}$ zu einer Funktion fun.

$COVER_{dp}(b)$ ist das CSP von $COVER_{dp}$ zu einem Basisblock b.

$COVER_{\not dp}$ ist eine Erweiterung von $COVER$, bei der Datentransferpfade explizit repräsentiert sein müssen und die Pfadlängen feststehen. Weiterhin können keine FMOs zu Sub-FMOs werden, die es nicht schon sind.

$COVER_{\not dp}(fun)$ ist das CSP von $COVER_{\not dp}$ zu einer Funktion fun.

$COVER_{\not dp}(b)$ ist das CSP von $COVER_{\not dp}$ zu einem Basisblock b.

$Cost(b)$ ist die Menge der Kostenvariablen des CSPs $GIA_{CSP}(b)$ zur graphbasierten Instruktionsauswahl über einem Basisblock b.

$Cost_{\Delta}(b)$ ist die Menge der ungebundenen Kostenvariablen des CSPs $GIA_{CSP}(b)$ zur graphbasierten Instruktionsauswahl über einem Basisblock b.

$\Delta - Optimum$ Bei der Assoziation eines Basisblocks mit Kostenvariablen des CSP-Modells GIA_{CSP}, ist eine Teilmenge dieser Kostenvariablen schon an feste Werte gebunden, dessen Summe S eine untere Schranke der Kosten darstellt. Die Differenz aus bestem Resultat und S ist gerade das Δ-Optimum. Die Differenz aus den Kosten der zuerst gefundenen Lösung und S ist der Δ-Init-Wert.

DFG data flow graph = Datenflussgraph.

$DFG(b)$ ist der lokale Datenflussgraph zu Basisblock b.

DDG data dependency graph = Datenabhängigkeitsgraph.

$DDG(b)$ ist der lokale Datenabhängigkeitsgraph zu Basisblock b.

DSP digitaler Signalprozessor.

$f, f_1, ..., f_n$ sind Bezeichner für FMOs eines Basisblocks b.

FD finite domain = endlicher Wertebereich.

FE Funktionseinheit.

FE_f ist Funktionseinheit einer FMO f.

$FE(b)$ ist die *Menge der Domainvariablen,* die die alternativen Zuordnungen von FMOs zu Funktionseinheiten spezifizieren:

$$FE(b) = \{FE|\ fmo\, with\, [fe : FE] \in FMO(b)\}$$

Elemente dieser Menge werden auch als *FE-Variablen* bezeichnet.

$FMO(b)$ ist die Menge aller FMOs innerhalb eines Basisblocks b.

$FV(b)$ ist die Menge der FMOs ($FMO(b)$), der Eingangs- ($VAR_{in}(b)$) und der Ausgangsvariablen ($VAR_{out}(b)$) eines Basisblock b mit

$$FV(b) = FMO(b) \cup VAR_{in}(b) \cup VAR_{out}(b)$$

$fv, fv_1, ..., fv_n$ sind Bezeichner für Elemente aus $FV(b)$ eines Basisblocks b.

fv_{pos} Zur Referenzierung von Setzungen und Benutzungen der Registerfilezuordnungen wird anstelle von fv_{pos}^{rfs} einfach kurz fv_{pos} geschrieben. Zugriffe auf Bezeichnerzuordnungen oder Sub-FMOs werden weiterhin mit fv_{pos}^{bs} bzw. fv_{pos}^{sos} notiert.

$gDFG(fun)$ ist der globale Datenflussgraph zur Funktion fun.

GIA graphbasierte Instruktionsauswahl.

GIA_{cost} ist das Kostenmodell der graphbasierten Instruktionsauswahl.

$GIA_{cost}(b)$ ist das zum CSP-Modell GIA_{cost} zugehörige CSP zu Basisblock b.

GIA_{CSP} ist das CSP-Modell der graphbasierten Instruktionsauswahl.

$GIA_{CSP}(b)$ ist das zum CSP-Modell GIA_{CSP} zugehörige CSP zu Basisblock b.

GPP general purpose processor = allgemeiner Prozessor.

HLO High-Level-Optimierungen des Middle-Ends.

$I(b)$ ist die Menge Domainvariablen, die jede FMO mit den Instruktionszyklen assoziiert. Domainvariablen dieser Menge werden auch als I-Variablen bezeichnet.

$IIIR$ (I^3R) integrierte Instruktionsauswahl, Instruktionsanordnung und Registerallokation.

I^3R umfasst die Mengen der integrierten Techniken zur Phasenkopplung der ADSP2100-Familie.

I^3R_{CSP} ist das CSP-Modell von I^3R.

$I^3R_{CSP}(b)$ ist das CSP von I^3R_{CSP} zum Basisblock b.

I^3R_l bezeichnet eine Technik aus I^3R unter Anwendung der Labelingstrategie l.

I^3R_{l+ra} bezeichnet eine Technik aus I^3R unter Anwendung der Labelingstrategie l und des Kostenmodells, das den Registerdruck umfasst.

I^3R_{l+part} bezeichnet eine Technik aus I^3R unter Anwendung der Labelingstrategie l und der Partitionierung.

$I^3R_{l+ra+part}$ bezeichnet eine Technik aus I^3R unter Anwendung der Labelingstrategie l und des Kostenmodells, dass den Registerdruck umfasst, unter der Anwendung der Partitionierung.

ILP Integer-Lineare-Programmierung.

IP Integer-Programmierung.

IT Instruktionstyp.

IT_f ist der Instruktionstyp einer FMO f.

$IT(b)$ ist die *Menge der Domainvariablen,* die die alternativen Zuordnungen von FMOs zu Instruktionstypen spezifizieren:

$$IT(b) = \{IT | \, fmo \, with \, [it : IT] \in FMO(b)\}$$

Elemente dieser Menge werden auch als *IT-Variablen* bezeichnet.

LLO Low-Level-Optimierungen des Back-Ends.

LP Lineare-Programmierung.

$\mathcal{M}$ ist die Menge der Speicherbänke eines Prozessors.

B. Abkürzungen und Notationen

$n \in G$ für einen Graphen G besagt, dass n ein Knoten von G ist.

$n \rightarrow^{attr} n' \in G$ für einen Graphen G besagt, dass $n \rightarrow^{attr} n'$ eine Kante von G mit Kantenmarkierung $attr$ ist.

$\mathcal{OC}$ Menge der Klassen von CoLIR-Operationen.

$\mathcal{R}$ ist die Menge der Register eines Prozessors.

$\mathcal{RF}$ ist die Menge der Registerfiles eines Prozessors.

RT Register-Transfer-Operation.

SSA static single assignment.

$\mathcal{ST}$ ist die Menge der ST-Komponenten eines Prozessors mit $\mathcal{ST} = \mathcal{RF} \cup \mathcal{M} \cup \mathcal{T}$.

$STK_{def}(b)$ ist die *Menge der Setzungen* (bzgl. der Registerfilezuordnungen) eines Basisblocks b:

$$STK_{def}(b) = \{fv_{[0]} \mid fv \in FV(b)\}$$

Elemente dieser Menge werden einfach nur als Setzung bezeichnet.

$STK_{use}(b)$ ist die *Menge der Benutzungen* Basisblocks b:

$$STK_{use}(b) = \{RF_i | fmo\ with\ [rfs : \_ := (..., RF_i, ..)] \in FMO(b)\}$$

Elemente dieser Menge werden einfach nur als Benutzung bezeichnet.

$STK(b)$ ist die Menge aller Setzungen und Benutzungen eines Basisblocks b mit

$$STK(b) = STK_{def}(b) \cup STK_{use}(b)$$

Elemente dieser Menge werden auch als *STK-Variablen* bezeichnet.

$\mathcal{T}$ ist die Menge der flüchtigen Komponenten eines Prozessors.

$VLIW$ very large instruction word.

$v, v_1, ..., v_n$ sind die Bezeichner für Elemente aus $VAR_{in}(b)$ und $VAR_{out}(b)$ eines Basisblocks b.

$VAR_{in}(b)$ ist die Menge der Eingangsvariablen V_{in} eines Basisblocks b.

$VAR_{out}(b)$ ist die Menge Ausgangsvariablen V_{out} eines Basisblocks b.

Literaturverzeichnis

[1] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] A.V. Aho and S.C. Johnson. Optimal Code Generation for Expression Trees. *Journal of the ACM*, 23(3):488–501, 1976.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, New York, 1986.

[4] V.H. Allan, J. Janardhan, R.M. Lee, and M. Srinivas. Enhanced Region Scheduling on a Program Dependence Graph. In *IEEE MICRO–25*, pages 72–80, 1992.

[5] W. Ambrosch, A. Ertl, F. Beer, and A. Krall. Dependence Conscious Register Allocation. In Juergen Gutknecht, editor, *Programming Languages and System Architectures*, volume 782, pages 125–136. LNCS Series, Springer–Verlag, Zurich, Switzerland, March 1994.

[6] A.W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[7] A. Balachandran, D.M. Dhamdere, and S. Biswas. Efficient Retargetable Code Generation Using Bottom-Up Tree Pattern Matching. *Comput. Lang. vol. 15, no. 3*, 1990.

[8] S. Bashford. Code Generation Techniques for Irregular Architectures. Technical Report 596, Lehrstuhl Informatik XII, University of Dortmund, November 1995.

[9] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language Version 4.1. Internal Report, University of Dortmund, September 1994.

[10] S. Bashford and R. Leupers. Constraint Driven Code Selection for Fixed-Point DSPs. In *36th Design Automation Conference (DAC)*, 1999.

[11] S. Bashford and R. Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems*, 4(2/3), 1999. Kluwer Academic Publisher.

[12] S. Bashford and R. Leupers. Graph based Code Selection Techniques for Embedded Processors. *ACM TODAES*, 5(4), 2000.

[13] A. Basu, R. Leupers, and P. Marwedel. Register-Constrained Address Computation in DSP Programs. In *Design Automation and Test in Europe (DATE)*, 1998.

[14] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 241–255, Toronto, Ontario, Canada, June 1991.

[15] D.A. Berson, R.Gupta, and M.L. Soffa. Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers. *Working Conf. on Parallel Architectures and Compilation Techniques*, August 1994.

[16] D.G. Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors*. PhD thesis, Dept. of Computer Science, Univ. of Washington, 1991.

[17] D.G. Bradlee, S.J. Eggers, and R.R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, 1991.

[18] D.G. Bradlee, R.R. Henry, and S.J. Eggers. The Marion System for Retargetable Instruction Scheduling. *SIGPLAN Notices*, 26(6):229–240, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[19] T.S. Brasier and Phillip H. Sweany. CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment. In *PACT'95*, Limassol, Cypros, 1995.

[20] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, April 1992.

[21] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[22] D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. *SIGPLAN Notices*, 26(6):192–203, 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

[23] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markenstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47–57, January 1981.

[24] F.C. Chow and J.L. Hennessy. Register Allocation by Priority–Based Coloring. *In* Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, pages 222–232, 1984.

[25] F.C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[26] W.F. Clocksin. *Clause and Effect*. Springer, 1997.

[27] CPLEX. ILOG Homepage. http://www.ilog.com.

[28] S. Davidson, D. Landskov, B.D. Shriver, and P.W. Mallet. Some Experiments in Local Microcode Compaction for horizontal Machines. *IEEE Transactions on Computers*, C-30(7):460–477, July 1981.

[29] Analog Devices. Homepage of Analog Devices. http://www.analog.com.

[30] Analog Devices. *ADSP-2101 User's Manual*. Analog Devices, 1991.

[31] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1986.

[32] H. Emmelmann, F.W. Schroer, and R. Landwehr. BEG – A Generator for Efficient Back Ends. *SIGPLAN Notices*, 24(7):227–237, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[33] A. Fauth. Beyond Tool-Specific Machine Descriptions. In Marwedel and Goossens [88], chapter 8, pages 138–152.

[34] A. Fauth, G. Hommel, A. Knoll, and C Mueller. Global Code Selection for Directed Acyclic Graphs. In Peter A. Fritzson, editor, *Compiler Construction*, volume 786 of *LNCS*, pages 128–141. Springer–Verlag, Eddinburgh, U.K., April 1994. 5'th International Conference, CC'94.

[35] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction-Set Processors in nML. In *ED&TC*, pages 503–507, 1995.

[36] C. Ferdinand, H. Seidl, and R. Wilhelm. Tree Automata for Code Selection. *Acta Informatica, Springer-Verlag*, pages 741–760, 1994.

[37] G. Fettweis, M. Weiss, W. Drescher, U. Walther, F. Engel, and S. Kobayashi. Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proc. of ICSPAT'98*, pages 1547–1551, 1998.

[38] A.J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[39] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C–30(7):478–490, July 1981.

[40] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., 1995.

[41] C. Fraser, R. Henry, and T.A. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[42] C. Fraser, R. Henry, and T.A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[43] S.M. Freudenberger and J.C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In Robert Giegerich and Susan L. Graham, editors, *"Code Generation — Concepts, Tools, Techniques" Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20-24 May 1991*, Workshops in Computing, pages 146–172. Springer-Verlag, 1991. ISBN 3-540-19757-5 and 3-387-19757-5.

[44] T. Frühwirt and S. Abdenadher. *Constraint-Programmierung*. Springer, 1997.

[45] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completness*. W.H. Freemann & Co, 1979.

[46] C.H. Gebotys. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *10th International Symposium on System Synthesis (ISSS)*. 1997.

[47] P.B. Gibbons and S.S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. *ACM SIGPLAN Notices*, 21(7):11–16, July 1986.

[48] J. Goodman and W. Hsu. Code Scheduling and Register Allocation. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[49] The Stanford SUIF Compiler Group. SUIF Compiler System Homepage. http://www-suif.stanford.edu/.

[50] R. Gupta and M.L. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[51] R. Gupta, M.L. Soffa, and T. Steele. Register Allocation via Clique Separators. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pages 264–274, 1989.

[52] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *DATE*, 1999.

[53] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the Aviv Retargetable Code Generator. In *Proceedings of the 35th DAC'98*, 1998.

[54] S. Hanono, G. Hadjiyiannis, and S. Devadas. Aviv: A Retargetable Code Generator Using ISDL. In *Proceedings of the 34th DAC'97*, 1997.

[55] R. Hartmann. Combined Scheduling and Data Routing for Programmable ASIC Systems. In *Proceedings of EDAC'92*, pages 486–490, March 1992.

[56] J.P. Hayes. *Computer Architecture and Organization*. McGraw-Hill, 1998.

[57] W. Heinrich. *Formal Desciption of Parallel Computer Architectures as a Basis of Optimizing Code Generation*. PhD thesis, TU Munich, 1993.

[58] J.L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers Inc., 1990.

[59] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[60] Texas Insruments. *TMS320C1x/C2x/C2xx/C5x Assembly Language Tools: Users Guide*. Custom Printing Company, 1998.

[61] N. Karmakar. A new Polynomial Algorithm for Linear Programming. In *Combinatorica*, volume 4, pages 373–395, 1984.

[62] A. Kifli, G. Goossens, and H. De Man. A unified scheduling model for high-level synthesis and code generation. In *DATE*, pages 234–238, 1995.

[63] D. Kästner. PROPAN: A Retargetable System for Postpass Optimizations and Analysis. In *LCTES*, 2000.

[64] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical Report Technical Report A / 01 / 98., Universität des Saarlandes, 1998.

[65] D. Kästner and M. Langenbach. Code Optimization by Integer Linear Programming. In *8th International Conference on Compiler Construction*, pages 122–136, 1999.

[66] D. Kästner and M. Langenbach. TDL - Eine Architekturbeschreibungssprache für Postpassoptimierungen und -analysen. In *DSP-Deutschland*, 1999.

[67] D. Lanneer, J. Praet, K. Schoofs, A. Kifli, W. Geuts, F. Toen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In Marwedel and Goossens [88], chapter 5, pages 85–102.

[68] D. Lanner, M. Cornero, G. Goossens, and H. De Man. Data Routing: a Paradigm for Efficient Data–Path Synthesis and Code Generation. In *Proc. 7th IEEE/ACM Int. Symp. on High–Level Synthesis*, May 1994.

[69] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.

[70] R. Leupers. Exploiting Conditional Instructions in Code Generation for Embedded VLIW Processors. In *Design Automation and Test in Europe (DATE)*, 1999.

[71] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *Design Automation and Test in Europe (DATE)*, 2000.

[72] R. Leupers. Compilertechniken für VLIW DSPs. In *DSP Deutschland*, 2000.

[73] R. Leupers. Register Allocation for Common Subexpressions in DSP Data Paths. In *ASP-DAC*, 2000.

[74] R. Leupers and P. Marwedel. Instruction Set Extraction From Programmable Structures. In *Proc. EURO-DAC 1994*. 1994.

[75] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. *ICCAD*, 1996.

[76] R. Leupers and P. Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. *Journal on Design Automation for Embedded Systems*, 1997.

[77] R. Leupers and P. Marwedel. Time-Constrained Code Compaction for DSPs. *IEEE Transactions on VLSI Systems, vol. 5, no. 1*, 1997.

[78] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. In *Proceedings of International Conference on ComputerAided Design*, 1995.

[79] S. Liao, S. Devadas, K. Keutzer, and S Tjiang. Storage Assignment to Decrease Code Size. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, 1995.

[80] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code Optimization Techniques in Embedded DSP Microprocessors. In *Proceedings of 1995 ACM/IEEE Design Automation Conference*, 1995.

[81] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection for DSP and ASIP code generation. In *Proc. European Design and Test Conf. (ED&TC)*, 1994.

[82] C. Liem, T. May, and P. Paulin. Register assignment through resource classification for ASIP microcode generation. In *Proc. ACM/IEEE Int. Conf.Computer-Aided Design*, 1994.

[83] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[84] M. Lorenz, D. Kottmann, S. Bashford, R. Leupers, and P. Marwedel. Optimized Address Assignment for DSPs with SIMD Memory Accesses. In *ASP-DAC*, 2001.

[85] M.M. Mano. *Computer System Architecture*. Prentice Hall International Editions, 1993.

[86] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[87] P. Marwedel. *Ein Software-System zur Synthese von Rechnerstrukturen und zur Erzeugung von Mikrocode*. Habilitationsschrift, Universität Kiel, 1985. (unveränderter Nachdruck in: Bericht Nr. 356 des Fachbereichs Informatik der Universität Dortmund, 1990).

[88] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[89] P. Marwedel and W. Schenk. Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System. In *EDAC93*, pages 63–69, 1993.

[90] B. Mesmann, A.H. Timmer, J.L. van Meerbergen, and J. Jess. Constraint Analysis for DSP Code Geberation. In *Proc ISSS'97*, 1997.

[91] S. Moon and K. Ebcioglu. An Efficient Resource Constraint Global Scheduling Technique for Superscalar and VLIW Processors. *IEEE MICRO-25*, 1992.

[92] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[93] S.S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[94] S.S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[95] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and their Impact on Parallelization. In *Languages and Compilers for Parallel Computing*, volume 589. LNCS Series, Springer–Verlag, 1991.

[96] R. Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. PhD thesis, University of Dortmund, 1998. Kluwer Academic Publishers.

[97] C. Norris and L. Pollok. A Scheduler–Sensitive Global Register Allocator. In *Proceedings of Supercomputing'93*, 1993.

[98] C. Norris and L. Pollok. Register Allocation over the Program Dependence Graph. *SIGPLAN Notices*, 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[99] C. Norris and L. Pollok. Register Allocation Sensitive Region Scheduling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, 1995.

[100] S. Novack and A. Nicolau. Trailblazing: A Hierarchical Approach to Percolation Scheduling. Technical Report TR–92–56, Irvine University, August 1993.

[101] S. Novack and A. Nicolau. Mutation Scheduling: A Unified Approach to Compiling for Fine–Grain Parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 16–30. Springer–Verlag, Ithaca,NY,USA, August 1994.

[102] L. Nowak. Graph based Retargetable Micrcode Compilation in the MIMOLA Design System. In *IEEE MICRO-20*, pages 126–132, 1987.

[103] IC Parc. Homepage of ECLiPSe. http://www.icparc.ic.ac.uk/eclipse/.

[104] P. Paulin, C. Liem, T. May, and S. Sutarwala. Flexware: A Flexible Firmware Developement Environment for Embedded Systems. In Marwedel and Goossens [88], chapter 4, pages 65–84.

[105] S.S. Pinter. Register Allocation with Instruction Scheduling. Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, pages 248–257, 1993.

[106] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction Set Definition and Instruction Selection for ASIPs. In *Proc. 7th ACM/IEEE Int. Symp. on High Level Synthesis*, pages 11–16, 1994.

[107] K. Rimey and P.N. Hilfinger. Lazy Data Routing and Greedy Scheduling. In *IEEE MICRO–21*, pages 111–115. 1988.

[108] SPAM. Homepage of the SPAM project. http://www.ee.princeton.edu/spam/.

[109] L. Sterling and E. Shapiro. *Prolog*. Addison-Wesley, 1988.

[110] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*. PhD thesis, Princeton University Department of EE, may 1998.

[111] Cosy Compilation System. ACE Companies Homepage. http://www.ace.nl.

[112] A. Timmer, M. Strik, J. van Meerbergen, and J. Jess. Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores. In *Design Automation Conference (DAC)*, 1995.

[113] Trimaran. Homepage of the Trimaran project. http://www.trimaran.org.

[114] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A Platform for Constraint Logic Programming. Contact address: IC-Parc, William Penney Laboratory Imperial College, London SW7 2AZ, email:mgw@doc.ic.ac.uk, aug 1997. http://www.icparc.ic.ac.uk/.

[115] D.H.D. Warren. Logic Programming and Compiler Writing. In *Software-Practice and Expirience*, volume 10, pages 97–125, 1980.

[116] M. Weiss, G. Fettweis, M. Lorenz, R. Leupers, and P. Marwedel. Toolumgebung für plattformbasierte DSPs der nächsten Generation. In *DSP Deutschland*, 1999.

[117] B. Wess. Code Generation based on Trellis Diagrams. In Marwedel and Goossens [88], chapter 6, pages 188–202.

[118] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

[119] T. Wilson, G. Grewal, S. Henshall, and D. Banerjii. An ILP-Based Approach to Code Generation. In Marwedel and Goossens [88], chapter 6, pages 103–118.

[120] V. Zivojnovic, J.M. Velarde, C. Schlaeger, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In *ICSPAT*. 1994.

Index