Universität des Saarlandes
Fachrichtung 6.2 Informatik
Lehrstuhl für Programmiersprachen und Übersetzerbau
Prof. Dr. R. Wilhelm

Generisches Softwarepipelining auf Assemblerebene

Diplomarbeit
zur Erlangung des akademischen Grades
„Diplom-Informatiker“

Markus Pister
pister@cs.uni-sb.de

März 2005

Eidestattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Saarbrücken, 21. März 2005

Markus Pister

Danksagung

Mein Dank gilt Herrn Prof. Dr. R. Wilhelm für die Vergabe des interessanten und spannenden Themas und die Möglichkeit, die Räumlichkeiten seines Lehrstuhles zu nutzen. Die kollegiale und ungezwungene Atmosphäre innerhalb seiner Arbeitsgruppe ermöglichte mir ein zielgerichtetes und motiviertes Arbeiten.
Herrn Prof. Finkbeiner danke ich dafür, dass er sich bereit erklärt hat, diese Arbeit als Zweitgutachter zu bewerten. Besonders möchte ich Herrn Dr. Daniel Kästner für seine motivierte und engagierte Betreuung sowie für das Korrekturlesen der vorliegenden Arbeit danken. Seine Anregungen und Kommentare halfen sehr in allen Phasen der Entstehung.
Marc Langenbach und Stephan Wilhelm danke ich für die fachkundige Unterstützung bei allen implementierungstechnischen Problemen. Ebenso möchte ich meinem Freund und Kollegen Marc Schlickling für die fruchtvolle Zusammenarbeit während des Studiums und dieser Arbeit danken.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Unsichtbar für den Benutzer werden heutzutage moderne Mikroprozessoren zur automatischen Steuerung und Kontrolle von technischen Anlagen, Unterhaltungs- und Haushaltselektronik, Autos, Flugzeugen und vielem mehr eingesetzt, weshalb Prozessoren mittlerweile eigens für solche **eingebetteten Systeme** entwickelt werden. Die Anwendungsgebiete dieser Systeme sind sehr vielfältig und der Markt dafür ist in den letzten Jahren stark angewachsen[1]. So verrichten z.B. modernste Mikroprozessoren in über 700 Millionen Mobiltelefonen weltweit ihre Dienste. Ein Mittelklassefahrzeug besitzt mittlerweile bis zu 200 und mehr Mikrocontroller. Eingebettete Systeme finden insbesondere auch Anwendung in vielen sicherheitskritischen Bereichen, wie z.B. in Airbag- und Flugkontroll-Steuerungssystemen der Automobil- bzw. Luftfahrtindustrie.
Zusammen mit immer komplexeren Anwendungsszenarien steigt die Komplexität der Software, die in eingebetteten Systemen verwendet wird. So wird ein Mobiltelefon mittlerweile nicht mehr nur zum Telefonieren benutzt, sondern muss Aufgaben der Termin- und Kontaktverwaltung beherrschen bis hin zum Versenden einer email und dem Aufbau einer Internetverbindung. Fahrzeugstabilitätsprogramme müssen sehr viele Eingabedaten aus den verschiedensten Sensoren eines Wagens in kurzer Zeit auswerten und davon abhängig Aktionen wie das Abbremsen eines Rades veranlassen. Diese Komplexität zeigt sich auch in der Größe der Software. Betrachtet man z.B. die Entwicklung eines elektronischen Wählsystems von Siemens, so ist der Programmcode von ca. 10 Millionen Instruktionen im Jahre 1980 innerhalb von 15 Jahren auf ca. 50 Millionen Instruktionen angewachsen.
Die Software vieler eingebetteter Systeme unterliegt harten Realzeitbedingungen, die

[1] nach einer Pressemitteilung der *Venture Developement Corporation* (`http://www.vdc-corp.com`) im September 2004 soll der Markt für eingebettete Systeme im Zeitraum von 2003-2008 jährlich um ca. 6,27% anwachsen

sich in strikten Laufzeitschranken ausdrücken. Beispielsweise muss die Steuerungssoftware des Flugkontrollcomputers eines Flugzeuges eine strikte obere Schranke bezüglich ihrer Antwortzeit einhalten, um die Stabilität des Flugzeuges in der Luft zu gewährleisten. Dies unterstreicht die enorme Wichtigkeit solcher Zeitschranken in Bezug auf Sicherheitskriterien.
Eine weitere Einschränkung eingebetteter Systeme gegenüber Standardsystemen ist der stark limitierte Platzverbrauch. Sowohl der zur Verfügung stehende Platz auf dem Chip als auch der Speicherplatz für Daten ist sehr beschränkt, denn zum einen bewirkt ein höherer Platzverbrauch einen erhöhten Energiekonsum und zum anderen steigen damit auch die Kosten der Hardware. Bei Absatzraten von beispielsweise um fünf Mio. Automobile pro Jahr der Volkswagen AG (Quelle: Jahresbericht 2004), in denen jeweils bis zu 200 und mehr Mikroprozessoren arbeiten, ist es enorm wichtig, die Kosten der Hardware möglichst gering zu halten.
Aber auch die Masse des Systems spielt eine Rolle für den Energieverbrauch. Denn je leichter beispielsweise die Fahrzeugelektronik ist, desto weniger Energie wird zur Fortbewegung benötigt.
Eine direkte Folge dieser Realzeitbedingungen und der steigenden Komplexität sowohl von Hard- als auch von Software ist die Notwendigkeit schnellerer Programmausführung bei nur moderat anwachsendem Programmcode. Man versucht, dies durch mehr und mehr spezialisierte Hardware zu erreichen, wobei man zunehmend anwendungsspezifische Funktionalität in die Hardware integriert. Der TriMedia TM1000-Prozessor von Philips unterstützt beispielsweise Multimediaanwendungen dadurch, dass auf seinem Chip bereits Decoder und sog. Multimedia-Units vorhanden sind, die eigenständig bereits Videoströme dekodieren oder anderweitig vorverarbeiten können. Solche anwendungsspezifischen Prozessoren mit ihren speziellen Hardwareeigenschaften führen zu spezifischen Restriktionen und Besonderheiten, auf die bei der Codegenerierung Rücksicht genommen werden muss. Beispiele dafür sind unter anderem verteilte heterogene Registerbänke und irreguläre Restriktionen der Parallelität auf Instruktionsebene. Solche Einschränkungen gegenüber regulären Prozessoren bezeichnet man auch als **irreguläre Hardwareeigenschaften**.
Das Problem, eine schnelle Programmausführung zu erreichen, versucht man durch die Entwicklung von Prozessoren mit hohem Grad an verfügbarer Parallelität auf Instruktionsebene zu begegnen. Hier sind die VLIW- oder EPIC-Architekturen zu nennen. Als Beispiel sei der Philips TriMedia TM1000-Prozessor oder der Intel® Itanium™ genannt, die beide über massive Parallelität auf Instruktionsebene verfügen.
Um die volle Leistung solcher Prozessoren ausnutzen zu können, braucht man gute Übersetzer, die Code generieren, der das Leistungspotential der Maschine ausreizt. Bis vor einigen Jahren hat sich die Forschung auf dem Gebiet des Übersetzerbaus vorwiegend mit der Codegenerierung für reguläre Prozessor-Architekturen beschäftigt, sodass heutzutage Compiler existieren, die dafür qualitativ hochwertigen Code generieren. Für irreguläre Architekturen sind selbst die Ergebnisse moderner Übersetzer oft noch weit davon entfernt, optimalen Code zu erzeugen ([ŽMSM94, Leu98]). Man braucht

Codeerzeugungstechniken, die die speziellen Hardwareeigenschaften irregulärer Architekturen ausnutzen können. Irreguläre Prozessoren sind sehr unterschiedlich und um deshalb den Entwicklungsaufwand neuer Compiler zu minimieren, sind vor allem **retargierbare Codegenerierungstechniken** sehr interessant. Die Herausforderung liegt im Entwurf von Optimierungstechniken, die flexibel genug sind, auf verschiedene irreguläre Eigenschaften anpassbar zu sein und dabei eine hohe Codequalität erreichen zu können.
Retargierbare Techniken werden allerdings nur selten in der Industrie eingesetzt, da der Wechsel der Compilerinfrastruktur hohe Kosten zu Folge hätte, die die Unternehmen scheuen. Eine Lösung sind sog. **Postpass-Ansätze**, bei denen keine Änderungen an der Compilerinfrastruktur notwendig sind sondern der erzeugte Assemblercode als Eingabe dient. Ein Postpass-Optimierer kann einfach und leicht in bereits existierende Entwicklungssysteme integriert werden.
Auf Basis dieser Idee wurde das PROPAN-System ([Käs00a, Käs00, KL99]) entwickelt. Es basiert auf einem retargierbaren Framework zur Definition von hochwertigen Postpass-Optimierungen auf Assemblerebene. Auf Basis einer Prozessorspezifikation in TDL wird ein maschinenabhängiger Optimierer generiert, der effizientsteigernde Programmtransformationen auf Assemblerebene vornimmt.
Diese Vorgehensweise hat gegenüber einer Optimierung innerhalb einer Compilerinfrastruktur den Vorteil, über Bibliotheksgrenzen hinweg anwendbar zu sein. Außerdem kann durch den Postpass-Ansatz die Optimierung auch auf handgeschriebenem Assembler durchgeführt werden, was in der Softwareentwicklung für eingebettete Systeme leider immer noch üblich ist. Eine weiterer Vorteil ist die Flexibilität dieses Ansatzes, da er mit anderen Postpass-Techniken kombiniert werden kann.

Betrachtet man die Ausführung von Programmen näher, so stellt man fest, dass die meiste Zeit der Ausführung in Schleifen verbracht wird. Will man also die Ausführungszeiten von Programmen optimieren, so kann die Optimierung der Ausführungszeiten der im Programm enthaltenen Schleifen einen wesentlichen Beitrag dazu leisten. Azyklische Instruktionsanordnungsverfahren erzielen nur suboptimale Schedules für Schleifen, da sie nicht über Iterationsgrenzen hinweg optimieren können.
Softwarepipelining ist ein statisches, globales und zyklisches Instruktionsanordnungsverfahren, mit dessen Hilfe man schnellere Schedules für Schleifen erzeugen kann. Das Verfahren überlappt die Ausführung verschiedener Iterationen einer Schleife durch die parallele Ausführung von Operationen aus unterschiedlichen Iterationen. Dadurch kann eine erhebliche Beschleunigung bei moderatem Codezuwachs erzielt werden.

Im Rahmen dieser Diplomarbeit wird die Kombination von Softwarepipelining mit dem Ansatz der Postpass-Optimierung untersucht. Die Ausgangsbasis stellt das PROPAN-Framework dar, das die Voraussetzung bietet, Softwarepipelining retargierbar auf Assemblerebene einzusetzen.

1.2 Gliederung der Arbeit

In der vorliegenden Diplomarbeit werden zunächst einige Grundlagen der Codeerzeugung und Instruktionsanordnung erläutert und ausgehend davon der verwendete Softwarepipelining-Algorithmus samt einer prototypischen Implementierung und den damit erzielten experimentellen Ergebnissen dargestellt. Die Kapiteleinteilung sieht dabei wie folgt aus:

Kapitel 2 erläutert die grundlegende Problematik der Codeerzeugung und die zugrundeliegenden Programmdarstellungen.

Kapitel 3 gibt einen Überblick über VLIW-Architekturen, die das Haupt-Anwendungsgebiet für Softwarepipelining darstellen. Hier wird auch der für die experimentellen Ergebnisse verwendete Prozessor (TriMedia TM1000) vorgestellt.

Kapitel 4 beschreibt die Grundlagen der Instruktionsanordnung, kategorisiert die existierenden Techniken und zeigt, wie sich Softwarepipelining darin einordnet.

Kapitel 5 beschreibt Softwarepipelining als Instruktionanordnungsmethode und welche verschiedenen Ansätze zur Umsetzung existieren.

Kapitel 6 erläutert Aufbau und Struktur des PROPAN-Frameworks, in das der in Kapitel 7 beschriebene Algorithmus integriert wurde.

Kapitel 7 ist das Kernstück der Arbeit. Der Softwarepipelining-Algorithmus wird im Detail vorgestellt, zusammen mit den erforderlichen Anpassungen zum Einsatz des Verfahrens auf Assemblerebene.

Kapitel 8 beschreibt die prototypische Implementierung des im vorhergehenden Kapitel erläuterten Algorithmus innerhalb des PROPAN-Framework.

Kapitel 9 stellt die mit der Implementierung erzielten Ergebnisse vor und interpretiert sie.

Kapitel 10 schließt die Arbeit mit einer Zusammenfassung ab und gibt einen Ausblick auf weitere Verbesserungen.

Kapitel 2

Grundlagen der Codeerzeugung

Dieses Kapitel befasst sich mit den Grundlagen der Codeerzeugung, die als Basis für die folgenden Kapitel dienen. Zuerst wird das Problem der Codeerzeugung im Allgemeinen beschrieben. Darauf folgen Definitionen fundamentaler Programmdarstellungen zusammen mit verschiedenen Beziehungen zwischen Maschinensprachenoperationen und anschließend folgen noch ein paar kurze Bemerkungen zur Codeerzeugung für Prozessoren im eingebetteten Bereich.
Details zur Codeerzeugung im Allgemeinen finden sich in [WM97],[WM92]. Codeerzeugung vor dem Hintergrund des Phasenkopplungsproblems und seiner Bedeutung für Codeerzeugung im eingebetteten Bereich beschreibt [Käs00a].

2.1 Das Problem der Codeerzeugung

Während der Codeerzeugung soll eine optimale Sequenz von Maschinenoperationen erzeugt werden, die die Semantik des eingegebenen Hochspracheprogramms widerspiegelt. Der Begriff optimal weist hier mehrere Facetten auf: optimal im Sinne der Performanz bedeutet eine Codesequenz zu erzeugen, die unter Nutzung des Leistungspotentials der Maschine möglichst schnell abgearbeitet werden kann. Im Allgemeinen steht dies im Gegensatz dazu, möglichst speicherplatzsparende Codesequenzen zu erzeugen, wie sie vor allem im eingebetteten Bereich aufgrund des dort nur in geringer Menge vorhandenen Speicherplatzes sehr wichtig sind.
Die Codeerzeugung umfasst insgesamt drei Schritte:

- Codeselektion,
- Registerzuteilung und
- Instruktionsanordnung

Die Codeselektion wählt die zu verwendenden Maschinenoperationen für das gegebene Eingabeprogramm aus. Dabei bleiben oft mehrere Alternativen für eine Anweisung der Hochsprache zur Auswahl. Ihre Anzahl hängt dabei wesentlich davon ab, ob es sich bei dem Zielprozessor um eine CISC- oder RISC-Architektur handelt [WM97],[Bra91].Für den letzteren ist die Anzahl an Alternativen geringer.
Die Registerzuteilung entscheidet, welche Werte/Variablen in Registern des Prozessors gehalten werden. Im Allgemeinen werden zu gleicher Zeit mehr Werte benötigt als Register zur Verfügung stehen. Daher ist es wichtig, die Anzahl der Speicherzugriffe zu minimieren. Danach werden die zur Verfügung stehenden physischen Register den in Registern zu haltenden Werten/Variablen zugewiesen.
Die Instruktionsanordnung wiederum bestimmt die Reihenfolge, in der die Maschinenoperationen ausgeführt werden. Dabei entscheidet sich der Grad an Parallelität in der Ausführung der Operationen. Die Möglichkeiten der Anordnung sind hier durch

verschiedene Abhängigkeiten zwischen den Operationen im Eingabeprogramm selbst sowie durch die Eigenschaften des Zielprozessors begrenzt.
Die Codeerzeugung stellt eine sehr anspruchsvolle Aufgabe für einen Übersetzer dar, da alle ihre Phasen eine sehr hohe worst-case Komplexität besitzen: Codeselektion, Registerallokation und -zuteilung und Instruktionsanordnung sind $\mathcal{NP}$-vollständige Probleme [GJ79]. Durch diese hohe Berechnungskomplexität basieren viele Verfahren in der Codeerzeugung auf Heuristiken, obgleich auch exakte Algorithmen ([Käs00a],[Win04]) entwickelt wurden.
Die Abarbeitungsreihenfolge der einzelnen Phasen der Codeerzeugung kann variiert werden, wobei sich die Phasen gegenseitig beeinflussen. Dies hat immensen Einfluss auf die Eigenschaften und Qualität des erzeugten Maschinenprogramms. Z.B. kann in der Codeselektion eventuell eine kleinere Codesequenz ausgewählt werden, wenn die Registerzuteilung bereits stattgefunden hat und damit die Anzahl der benötigten Register bekannt ist. Umgekehrt kann es dazu kommen, dass bei vorheriger Registerzuteilung in der Codeselektion keine passende Codesequenz mehr gefunden werden kann. Diese gegenseitige Beeinflussung der Phasen der Codeerzeugung bezeichnet man auch als das **Phasenkopplungsproblem** und ist näher in [Käs00a] beschrieben.

Bemerkung. Da die Codeselektion bei RISC-Prozessoren einfacher ist, gewinnt hier die gegenseitige Beeinflussung zwischen Registerzuteilung und Instruktionsanordnung mehr an Bedeutung.

2.2 Fundamentale Programmdarstellungen

In dieser Arbeit wurde die Assemblerebene als Ausgangspunkt gewählt, daher beziehen sich die folgenden Definitionen auch auf die dort vorkommenden Operationen, wobei die meisten der Aussagen auch analog auf Anweisungen der Hochsprache angewendet werden können.
Es werden im folgenden zum einen wichtige fundamentale Programmdarstellungen und zum anderen Eigenschaften und Beziehungen zwischen Operationen definiert. Aus letzteren kann man Bedingungen ableiten, die die verfügbare Parallelität des Eingabeprogrammes bestimmen.
Die Terminologie und Notation lehnt sich dabei an die in [Käs00a, Käs97] an.

Definition 2.1 (Maschinenoperation). Eine **Maschinenoperation** eines Prozessors P ist eine Anweisung, die von P ausgeführt wird und eine Zustandsänderung in P bewirkt. ■

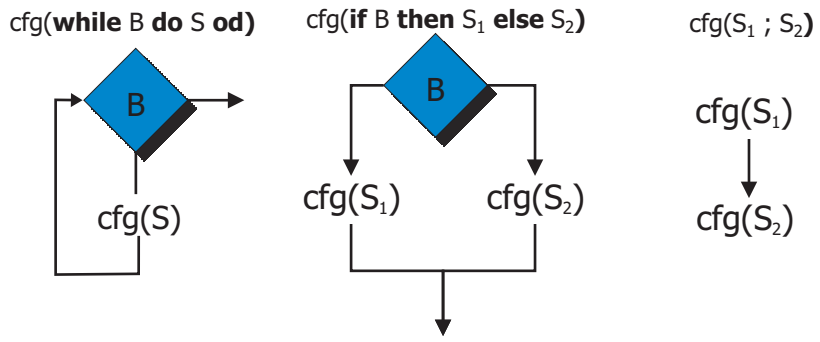
Definition 2.2 (VLIW-Instruktion). Eine **VLIW[1]-Instruktion** ist eine Maschineninstruktion, die mehrere Maschinenoperationen (vgl. Definition 2.1) enthält. Alle

[1] Very long instruction word

diese Operationen werden parallel ausgeführt. ■

In der Literatur wird der Begriff Instruktion und Operation häufig synonym verwendet. In dieser Arbeit bezieht sich der Begriff Instruktion immer auf eine echte VLIW-Instruktion.

Definition 2.3 (Kontrollflussgraph). Der **Kontrollflussgraph** einer Prozedur ist ein knotenmarkierter, kantengeordneter, gerichteter Graph $G_{cf} = (N, E_{cf}, n_{start}, n_{stop})$. Zu jeder primitiven Anweisung p der Prozedur gibt es einen Knoten $n_p \in N$, der mit dieser Anweisung markiert ist. Für zusammengesetzte Anweisungen ergeben sich die in Abbildung 2.1 dargestellten Teilgraphen. Kanten,die zu expliziten Sprüngen gehören, führen vom Knoten des Sprungs zu dem Sprungziel. n_{start} sei der eindeutig bestimmte Eintrittsknoten der Prozedur; er gehört zur ersten auszuführenden Anweisung der Prozedur. Ebenso sei n_{stop} ein Endknoten, der auf jedem Weg durch den Kontrollflussgraph erreicht wird.
Die Einsetzung eines Teilgraphen $G'_{cf} = (N'_{cf}, E'_{cf}, n'_{start}, n'_{stop})$ geschieht folgendermaßen: Alle eingehenden Kanten führen zu n'_{start}, alle ausgehenden Kanten zum eindeutig aus dem Kontext bestimmten Nachfolgeknoten. ■



(a) Schleife (b) Konditional (c) Sequenz

Abbildung 2.1: Kontrollflussgraph für zusammengesetzte Anweisungen

Bemerkung. Eine Kante im Kontrollflussgraph einer Schleife, die zurück zum Anfang des Schleifenkörpers führt, bezeichnet man als **Backedge**.

Da jeder Knoten im Kontrollflussgraph mit einer Operation beschriftet ist, wird im folgenden der Begriff des Knotens oft synonym mit dem der Operation verwendet. Die Pfade ausgehend vom Startknoten eines Kontrollflussgraphen beschreiben eine mögliche Reihenfolge, in der die Anweisungen eines Programmes ausgeführt werden. In Kapitel 4 wird erläutert, dass eine Änderung dieser Reihenfolge eine Leistungssteigerung erbringen kann. Natürlich darf die Semantik des Programmes dadurch nicht beeinflusst werden.

Beispiel 2.1. Man betrachte die folgende Codesequenz:

```
        r0 = Mem[r3]
2       r1 = r0 + 5
        cmp r1, #0
4       beq _T
        r2 = r2 * r1
6       b _END
_T:     r2 = 0
8   _END: Mem[r4] = r2
        r2 = 0
10      r2 = 1
```

Listing 2.1: Beispiel Maschinenprogramm

Abbildung 2.2 zeigt den Kontrollflussgraphen für Listing 2.1 gemäß Definition 2.3. ■

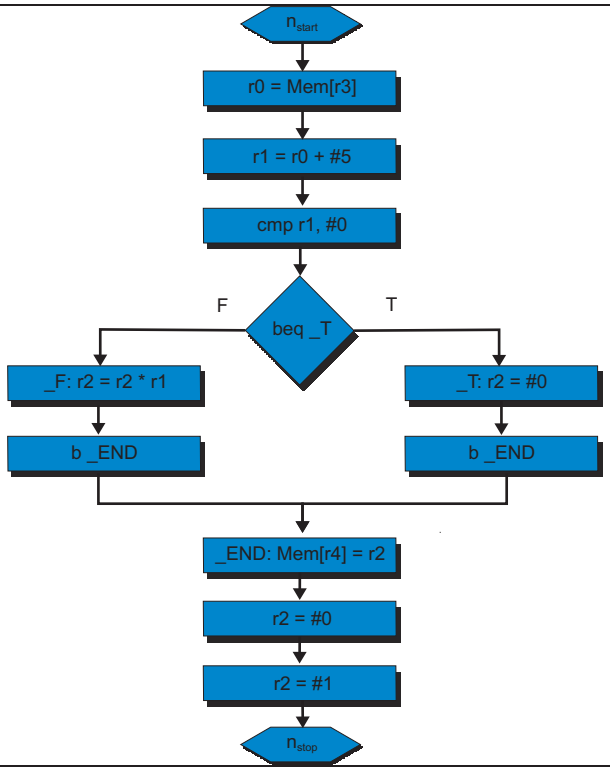


Abbildung 2.2: Kontrollflussgraph

Definition 2.4 (Pfad). Ein **Pfad** π von einem Knoten n_1 zu einem Knoten n_k in einem gerichteten Graph $G = (N, E)$ ist eine Sequenz von Kanten, beginnend mit dem Knoten $n_1 \in N$ und endet mit dem Knoten $n_k \in N$ mit $\pi =$

$(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$ und $(n_i, n_{i+1}) \in E$ für $i = 1, \ldots, k-1$. Die Länge $l(\pi)$ von π ist definiert als die Anzahl der Kanten in π, d.h. $l(\pi) = k - 1$. ■

Definition 2.5 (Basisblock). Ein **Basisblock** in einem Kontrollflussgraphen ist ein maximal langer Pfad, der höchstens am Anfang eine Verschmelzung und höchstens am Ende eine Verzweigung hat. Dabei bezeichnet man Knoten mit mehr als einem Vorgänger als **Verschmelzungen** und Knoten mit mehr als einem Nachfolger als **Verzweigungen**. ■

Für jeden Basisblock gilt: wird seine erste Anweisung ausgeführt, dann werden bei fehlerfreier Verarbeitung (keine Laufzeitfehler, Exceptions, o.ä.) auch alle anderen Anweisungen ausgeführt.

Definition 2.6 (Basisblockgraph). Der **Basisblockgraph** G_B eines Kontrollflussgraphen G_{cf} entsteht aus G_{cf}, indem Basisblöcke zu einem Knoten zusammengefasst werden. Kanten in G_{cf}, die zum ersten Knoten eines Basisblockes führen, führen in G_B in den Knoten des Basisblockes. Kanten, die in G_{cf} aus dem letzten Knoten eines Basisblockes herausführen, führen in G_B aus dem Knoten des Basisblockes heraus. ■

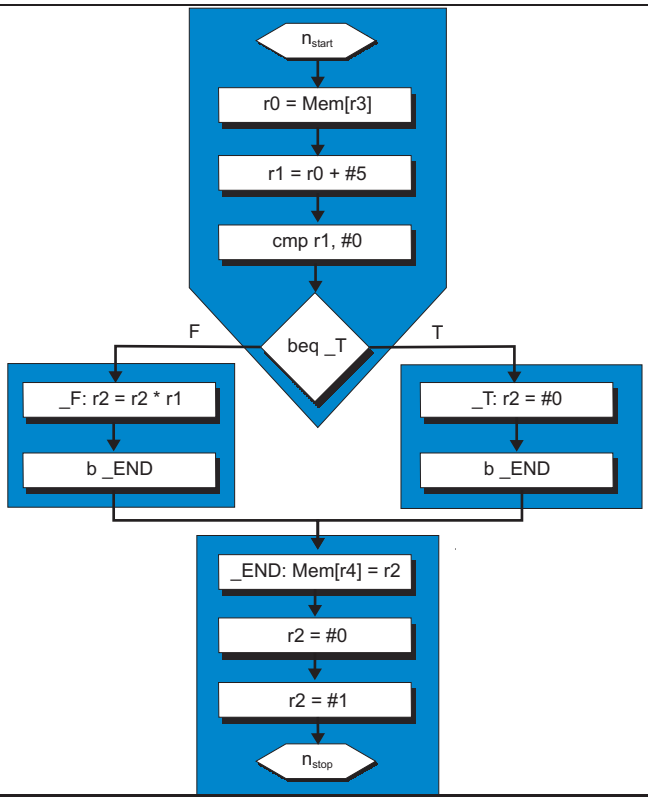


Abbildung 2.3: Basisblockgraph

Beispiel 2.2. Abbildung 2.3 zeigt den Basisblockgraphen für den Kontrollflussgraphen aus Abbildung 2.2. Wie man sieht teilt sich der Kontrollflussgraph in vier Basisblöcke auf: der erste Basisblock beginnt mit dem Anfang der Codesequenz und endet mit der bedingungsabhängigen Verzweigung, da sich der Kontrollfluss dort aufspaltet. Demnach ergeben sich die zwei nächsten Basisblöcke aus den beiden möglichen Pfaden nach der Verzweigung. Da sich danach der Kontrollfluss wieder vereinigt, beginnt der vierte Basisblock mit der ersten Operation nach der Vereinigung des Kontrollflusses und endet mit dem Ende der Codesequenz. ■

Jede Operation greift auf Speicher-Komponenten der Maschine wie Register oder Speicher zu; diese bezeichnet man als **Ressourcen**. Durch Ressourcenzugriffe verändert sich der Zustand der Maschine während der Ausführung eines Programmes. Man unterscheidet hierbei lesende Zugriffe, **Benutzungen** genannt, und schreibende Zugriffe, **Setzungen** genannt. Benutzungen sind z.B.:

- Benutzung von Registerinhalten in Operationen oder zur Adressierung,
- Speichern von Registerinhalten,
- Laden des Inhaltes von Speicherzellen und
- Abfragen der Teile des Bedingungscodes in bedingten Sprüngen.

Von Setzungen spricht man z.B. bei:

- Modifikation von Registerinhalten durch Lade-Befehle, durch Resultatsablage nach Operationen und durch automatische In- bzw. Dekrementierung von Adressregistern,
- Setzen von Überlauf-, Unterlauf- oder Übertragungsbits und Vergleichsergebnissen im Bedingungscode,
- Abspeichern von Werten in Speicherzellen und
- Modifikation eines Kellerzeigers (Stackpointer) durch Kellern oder Entkellern von Werten und durch Befehle zur Prozedurorganisation.

Das Aufeinanderfolgen von Setzungen und Benutzungen in einer Codesequenz bestimmt die Datenabhängigkeiten zwischen den Operationen der Sequenz.

Definition 2.7 (Datenabhängigkeitsgraph). Sei $G_{cf} = (N_{cf}, E_{cf}, n_{start}, n_{stop})$ ein Kontrollflussgraph. Sein Datenabhängigkeitsgraph ist ein gerichteter Graph $G_{dd} = (N_{dd}, E_{dd})$ mit Knoten- und Kantenbeschriftungen, dessen Knoten mit den Operationen einer Prozedur beschriftet sind. Die Menge der Kanten ist definiert als

$E_{dd} \subseteq N_{dd} \times N_{dd} \times \mathcal{R} \times \mathcal{T}$, wobei $\mathcal{R}$ die Ressource der Maschine und $\mathcal{T} = \{t, a, o\}$ den Typ der Datenabhängigkeit darstellt. Eine Kante geht vom Knoten i zum Knoten j, wenn die Operation von Knoten i im Kontrollflussgraph vor der Operation von Knoten j ausgeführt werden muss, d.h. wenn ein Pfad von i nach j im Kontrollflussgraph existiert, und wenn

- i eine Ressource setzt, j sie benutzt und der Pfad von i nach j setzungsfrei ist (**Setzung-Benutzungabhängigkeit, true dependence**)
- i eine Ressource benutzt, j sie setzt und der Weg von i nach j setzungsfrei ist (**Benutzung-Setzungabhängigkeit, anti dependence**)
- i und j die gleiche Ressource setzen und der Weg von i nach j setzungs- und benutzungsfrei ist (**Setzung-Setzungabhängigkeit, output dependence**).

Bezeichnet E_{dd}^{true} die Menge der Setzung-Benutzungabhängigkeiten, E_{dd}^{anti} die Menge der Benutzung-Setzungabhängigkeiten und E_{dd}^{output} die Menge der Setzung-Setzungabhängigkeiten, dann kann man die Kantenmenge des Datenabhängigkeitsgraphen schreiben als $E_{dd} = E_{dd}^{true} \cup E_{dd}^{anti} \cup E_{dd}^{output}$. ■

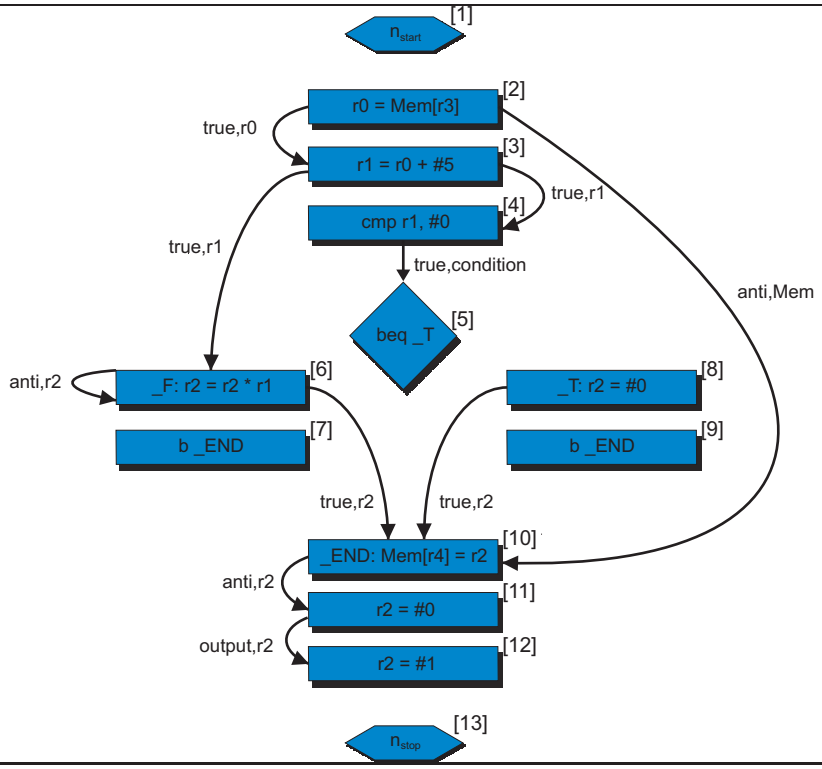


Abbildung 2.4: Datenabhängigkeitsgraph

Beispiel 2.3. Gegeben sei der Kontrollflussgraph zum Listing 2.1 aus Beispiel 2.1. Abbildung 2.4 zeigt den Datenabhängigkeitsgraphen dazu. Die Kanten sind hier sowohl mit dem Typ der Datenabhängigkeit als auch mit der beteiligten Ressource beschriftet.
Triviale Setzung-Benutzung-Abhängigkeiten bestehen zwischen den Knoten 2 und 3, 3 und 4, 3 und 6, 6 und 10 sowie 8 und 10. Eine Benutzung-Setzung-Abhängigkeit besteht zwischen den Knoten 10 und 11, und eine Setzung-Setzung-Abhängigkeit besteht zwischen 11 und 12.
Man beachte, dass es keine Setzung-Setzung-Abhängigkeit zwischen den Knoten 6 und 11 bzw. 8 und 11 gibt, da dazwischen in Knoten 10 noch eine Benutzung vorkommt und damit dort kein benutzungsfreier Pfad existiert. Ebenso gibt es zwischen diesen Knoten keine Benutzung-Setzung-Abhängigkeit, da in einer Maschinenoperation immer zuerst die Operanden gelesen werden und dann erst das Ergebnis geschrieben werden kann. Daher folgt der Benutzung von r2 in Knoten 6 und 8 als Operand eine Setzung. Dann ist der Pfad von 6 nach 11 bzw. 8 nach 11 nicht mehr setzungsfrei. ■

Um globale Optimierungen auf Assemblerebene durchführen zu können, reicht die obige Definition des Datenabhängigkeitsgraphen nicht aus und muss daher erweitert werden zum **verallgemeinerten Datenabhängigkeitsgraphen**. Man benötigt für Schleifen und Verzweigungen Informationen über Datenabhängigkeiten auf allen möglichen Programmpfaden. Bei einer bedingten Anweisung enthält der verallgemeinerte Datenabhängigkeitsgraph alle Beziehungen zwischen Setzungen und Benutzungen auf beiden möglichen Pfaden. Bei einer Schleife ergeben sich die zwei möglichen Pfade durch die Schleifenabbruchbedingung, also den Pfad zu einer neuen Iteration und den zum Schleifenausgang. Auch hier müssen die Datenabhängigkeiten auf beiden Pfaden betrachtet werden.
Die Datenabhängigkeiten eines Eingabeprogrammes stellen eine wichtige Komponente für die Instruktionsanordnung dar, da sich durch eine Datenabhängigkeit je nach ihrem Typ Präzedenzbedingungen berechnen lassen (vgl. Kapitel 4). Jedoch reichen die Informationen über Datenabhängigkeiten alleine nicht aus. Wie Beispiel 2.4 zeigt, benötigt man zusätzlich Informationen über Kontrollabhängigkeiten, um ausreichende Bedingungen über das Verschieben von Operationen aufstellen zu können.

Beispiel 2.4. Abbildung 2.5 zeigt einen Basisblockgraphen. Betrachtet man nur die Datenabhängigkeiten, so könnte man die Operation aus dem letzten Basisblock in seinen Vorgängerbasisblock verschieben, wie durch den dicken Pfeil markiert. Dadurch würde sich aber die Semantik des Programmes ändern und man muss zusätzlich Kontrollabhängigkeiten in die Analyse mit aufnehmen. ■

Definition 2.8 (Dominator). Sei $G_{cf} = (N_{cf}, E_{cf}, n_{start}, n_{stop})$ ein Kontrollflussgraph. Ein Knoten $n \in N_{cf}$ **dominiert** einen Knoten $m \in N_{cf}$, $n \; \Delta_d \; m$, genau dann, wenn jeder Pfad vom Startknoten n_{start} der Prozedur zum Knoten m den Knoten n enthält. ■

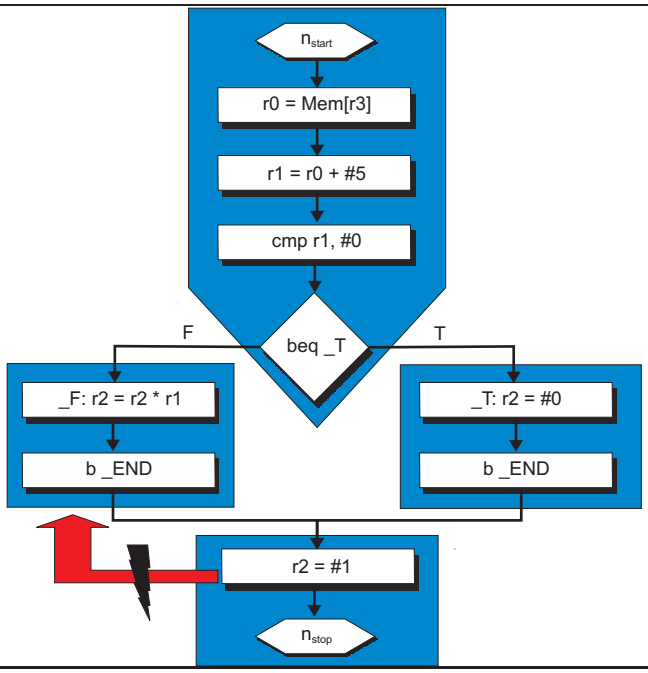


Abbildung 2.5: Problematik globaler Transformationen

Bemerkung. Jeder Knoten dominiert sich selbst.

Definition 2.9 (unmittelbarer Dominator). Sei $G_{cf} = (N_{cf}, E_{cf})$ ein Kontrollflussgraph. Ein Knoten $n \in N_{cf}$ ist ein **unmittelbarer Dominator** eines Knoten $m \in N_{cf}$, genau dann, wenn

- $n \ \Delta_d \ m$
- $\nexists z : x \ \Delta_d \ z \wedge z \ \Delta_d \ y \wedge z \neq x.$

■

Definition 2.10 (Dominatorbaum). Der **Dominatorbaum** T_d eines Kontrollflussgraphen G_{cf} ist ein Baum, der alle Knoten von G_{cf} enthält. Seine Wurzel ist der Startknoten n_{start} der Prozedur. Eine Kante zwischen einem Knoten n und einem Knoten m besteht genau dann, wenn m ein unmittelbarer Dominator von n ist. ■

Definition 2.11 (Postdominator). Sei $G_{cf} = (N_{cf}, E_{cf}, n_{start}, n_{stop})$ ein Kontrollflussgraph. Ein Knoten $n \in N_{cf}$ **postdominiert** einen Knoten $m \in N_{cf}$, $n \ \Delta_p \ m$ genau dann, wenn jeder Pfad von m zu n_{stop} den Knoten n enthält. ■

Bemerkung. Ein Knoten kann sich nie selbst postdominieren.

Definition 2.12 (unmittelbarer Postdominator). Sei ein Kontrollflussgraph $G_{cf} = (N_{cf}, E_{cf}, n_{start}, n_{stop})$ gegeben. Ein Knoten $n \in N_{cf}$ ist ein **unmittelbarer Postdominator** eines Knotens $m \in N_{cf}$, genau dann, wenn

- $n \ \Delta_p \ m$
- $\nexists z : n \ \Delta_p \ z \wedge z \ \Delta_p \ m \wedge z \neq n.$

■

Definition 2.13 (Postdominatorbaum). Der **Postdominatorbaum** T_p eines Kontrollflussgraphen ist ein Baum, der alle Knoten des Kontrollflussgraphen G_{cf} enthält. Seine Wurzel ist der Endknoten n_{stop} der Prozedur. Eine Kante zwischen zwei Knoten n und m des Postdominatorbaums existiert genau dann, wenn n der unmittelbare Postdominator von m ist. ■

Der Postdominatorbaum kann durch die Berechnung des Dominatorbaums auf dem invertierten Kontrollflussgraph berechnet werden; ein Algorithmus findet sich in [Käs97, ASU86].

Definition 2.14 (Kontrollabhängigkeit). Sei $G_{cf} = (N_{cf}, E_{cf}, n_{start}, n_{stop})$ ein Kontrollflussgraph. Ein Knoten $n \in N_{cf}$ ist **kontrollabhängig** von einem Knoten $m \in N_{cf}$, $n \ \delta^a_{cf} \ m$, wenn folgende Bedingungen erfüllt sind:

1. $(n, a, \lambda) \in E_{cf}$, mit $\lambda \in \{T, F, \epsilon\}$,
2. m postdominiert nicht n, $\neg(m \ \Delta_p \ n)$, und
3. $\exists \ path \ \pi = n, a, \ldots, m : \forall z \in \pi, z \neq n \wedge z \neq m : m \ \Delta_p \ z.$

Ein Knoten m ist von einem Knoten n genau dann kontrollabhängig, wenn $n \delta^a_c m$ gilt. ■

Definition 2.15 (Kontrollabhängigkeitsgraph). Sei G_{cf} ein Kontrollflussgraph. Sein **Kontrollabhängigkeitsgraph** $G_{cd} = (N_{cf}, E_{cf}, n_{start}, n_{stop})$ ist ein gerichteter azyklischer Graph $G_{cd} = (N_{cd}, E_{cd})$ mit Kantenbeschriftungen, sodass

$$(n, m, \lambda) \in E_{cd} \Leftrightarrow n \ \delta^a_c \ m \wedge (n, a, \lambda) \in E_{cf} \wedge \lambda \in \{T, F\}$$

■

Definition 2.16 (Kontrolläquivalenz). Sei $G_{cd} = (N_{cd}, E_{cd})$ ein Kontrollabhängigkeitsgraph. Zwei Knoten $n_1, n_2 \in N_{cd}$ sind **kontrolläquivalent**, wenn sie denselben Vorgänger m im Kontrollabhängigkeitsgraph besitzen und wenn die Kanten (m, n_1, λ_1) und (m, n_2, λ_2) die gleichen Beschriftungen haben, d.h. $\lambda_1 = \lambda_2$. ■

Programme mit Schleifen führen zu einer Besonderheit in den Datenabhängigkeitsinformationen. Hier entstehen Abhängigkeiten durch die erneute Ausführung des Schleifenkörpers, welche von den übrigen Datenabhängigkeiten unterschieden werden müssen. Dazu wird der Datenabhängigkeitsgraph erneut erweitert.

Definition 2.17 (Schleife). Sei $G_{bb} = (N_{bb}, E_{bb})$ ein Basisblockgraph und T_d sein Dominatorbaum. Ein **Schleife** $G_l = (N_l, E_l, h_l)$ ist ein Teilgraph von G_{bb} mit $N_l \subseteq N_{bb}$ und $E_l \subseteq N_l \times N_l$, sodass $E_l \subseteq E_{bb}$ gilt. G_l muss folgende Bedingungen erfüllen:

- Es muss einen eindeutigen Startpunkt, den Schleifenkopf h_l geben, der alle Blöcke der Schleife dominiert, d.h. $\forall b \in N_l : h_l \ \Delta_d \ b$.
- Es muss mindestens einen Pfad vom Anfang des Schleifenkopfes $h_l \in E_l$, der zu sich selbst führt, geben.

Die Kanten $(b, h_l) \in E_l$ werden als Rückwärtskanten (backward edges) von G_l bezeichnet; alle anderen Kanten heißen Vorwärtskanten (forward edges). Der Schleifenkörper von G_l ist definiert als $N_l - \{h_l\}$. ■

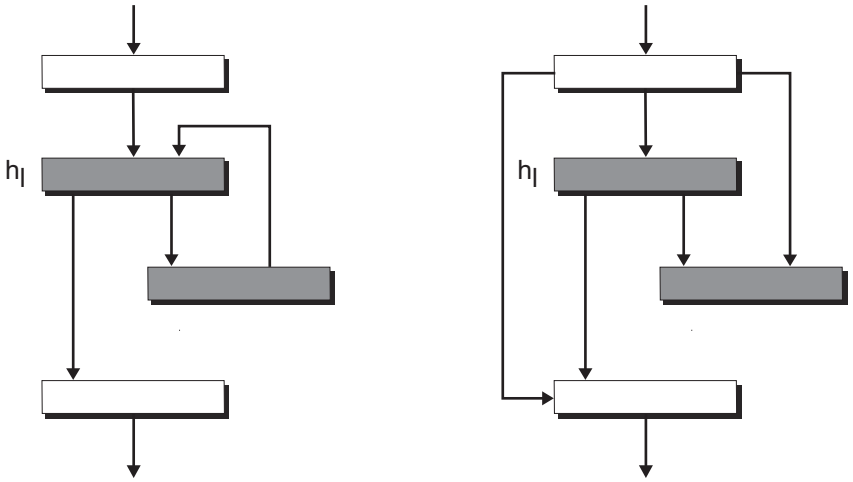
Man spricht davon, dass eine Schleife $G_{l'}$ eine andere Schleife G_l beinhaltet, wenn $N_l \subseteq N_{l'}$. Die Schachtelungstiefe $depth_l$ der Schleife G_l ist definiert als die Anzahl der Schleifen $G_{l'} \neq G_l$, die $G_{l'}$ beinhaltet. Schleifenabhängige und schleifenunabhängige Datenabhängigkeiten werden nun mithilfe der reduzierten transitiven Hülle des Basisblockgraphen definiert.

Definition 2.18 (reduzierte transitive Hülle des Basisblockgraphen). Sei der Basisblockgraph $G_{bb} = (N_{bb}, E_{bb}, b_{start}, b_{stop})$ einer Prozedur gegeben und sei E_{bb}^- die Menge aller Vorwärtskanten in E_{bb}. Dann ist die **reduzierte transitive Hülle des Basisblockgraphen** definiert als $G_{bb}^+ = (N_{bb}, E_{bb}^+, b_{start}, b_{stop})$. Es existiert eine Kante (b, g) in E_{bb}^+ genau dann, wenn die folgenden Bedingungen erfüllt sind:

- es existiert ein Pfad von b nach g in E_{bb}^-, oder
- es existiert eine Schleife $G_l = (N_l, E_l, h_l)$ mit $b \in N_l, g \notin N_l$, sodass ein Pfad von h_l nach g in E_{bb}^- existiert.

■

Beispiel 2.5. Abbildung 2.6 zeigt einen Basisblockgraphen und seine reduzierte transitive Hülle. Die schattierten Blöcke bilden eine Schleife L, dessen Schleifenkopf der Block h_l darstellt. ■



(a) Basisblockgraph (b) reduzierte transitive Hülle

Abbildung 2.6: Reduzierte transitive Hülle eines Basisblockgraphen

Ausgehend von einem Kontrollflussgraphen G_{cf} und der reduzierten transitiven Hülle des Basisblockgraphen kann die *reduzierte transitive Hülle des Kontrollflussgraphen* G_{cf}^{+} berechnet werden. Alle eingehenden Kanten eines Blockes $b \in G_{bb}$ sind repräsentiert durch Kanten, die auf den ersten Knoten von E_{cf} zeigen, der zu b gehört; alle ausgehenden Kanten eines Blockes b werden durch die ausgehenden Kanten des Knotens der letzten Instruktion von b repräsentiert. Die Kanten zwischen Instruktionen desselben Baissblockes sind dieselben wie in G_{cf}.

Definition 2.19 (Schleifenabhängige Datenabhängigkeiten). Sei ein Datenabhängigkeitsgraph $G_{dd} = (N_{dd}, E_{dd})$ des Kontrollflussgraphen G_{cf} einer Prozedur gegeben und sei G_{cf}^{+} die reduzierte transitive Hülle des Kontrollflussgraphen G_{cf}.
Eine Datenabhängigkeit $(i, j, r, \tau) \in E_{dd}$ zwischen einer Operation i und einer anderen Operation j nennt man eine **schleifenabhängige Datenabhängigkeit (loop-carried dependence)**, genau dann, wenn in G_{cf}^{+} die Instruktion, die die Operation i enthält, kein Nachfolger der Instruktion, die Operation j enthält, ist.
Alle anderen Datenabhängigkeiten nennt man **schleifenunabhängige Datenabhängigkeiten (loop-independent dependences)**. ■

Betrachtet man den Datenabhängigkeitsgraphen auf Ebene einer Hochsprache, kann man die schleifenabhängigen Datenabhängigkeiten weiter unterscheiden. Nämlich in der Anzahl der Iterationen, die sie umspannen. Zur Formalisierung dieser Unterscheidung wird der Datenabhängigkeitsgraph für Schleifen erweitert:

Definition 2.20 (Distanz einer Datenabhängigkeit). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph. Die **Distanz** $dist_{e_{dd}}$ einer Datenabhängigkeit $e_{dd} \in E_{dd}$ mit $e_{dd} = (i, j, r, \tau)$ bezeichnet die Anzahl der Iterationen einer Schleife, die e_{dd} umspannt, d.h. wieviele Iterationswechsel zwischen Ausführung der Operation i und der Ausführung der Operation j stattfinden. ■

Satz 2.1. *Betrachtet man den Datenabhängigkeitsgraphen auf Assemblerebene, so gilt*

$$0 \leq dist(e_{dd}) \leq 1.$$

■

Dies begründet sich darin, dass Abhängigkeiten, die mehr als eine Iteration umspannen, nur dann auftreten können, wenn man höherwertige Datenstrukturen wie Felder o.ä. betrachtet. Diese sind auf Assemblerebene, wo man eindimensionale Register betrachtet, nicht vorhanden.
Beispiel 2.6 illustriert die Distanzen von Datenabhängigkeiten innerhalb von Schleifen.

Beispiel 2.6 (Distanzen von Datenabhängigkeiten). Listing 2.2 (a) zeigt eine Schleife, die eine true dependence ausgehend von der Anweisung in Zeile 6 zu sich selbst beinhaltet, allerdings eine Iteration später, da a[i-1] eine Iteration vor der aktuellen Iteration berechnet wurde. Die Distanz dieser Abhängigkeit ist also 1.
In Listing 2.2 (b) ist eine Schleife gezeigt, die eine true dependence ausgehend von der Anweisung in Zeile 7 zu sich selbst beinhaltet. Hier wird auf a[i-2] zugegriffen, was zwei Iterationen zuvor definiert wurde. Die Distanz dieser Abhängigkeit ist also 2. ■

```
1 int i;
2 int a[10];
3
4 a[0] = 0;
5
6 for ( i=1 ; i<10 ; ++i )
7     {
8   a[i] = a[i-1] + 1;
9
10 }
```

(a)

```
1 int i;
2 int a[10];
3
4 a[0] = 0;
5 a[1] = 1;
6 for ( i=2 ; i<10 ; i+=2
7    ) {
8   a[ i ] = a[i-2] + 2;
9   a[i+1] = a[i] + 1;
10 }
```

(b)

Listing 2.2: Distanzen von Datenabhängigkeiten in Schleifen

2.3 Codeerzeugung für eingebettete Prozessoren

Im eingebetteten Bereich hat man es oft mit Realzeitbedingungen zu tun, weshalb hier härtere Forderungen an die Codequalität gesetzt werden als im Desktopbereich. Die Wichtigkeit dieser Realzeitbedingungen wird dadurch unterstrichen, dass viele eingebettete Systeme sicherheitsrelevante und -kritische Aufgaben übernehmen, wie z.B. Airbag-Steuerungssysteme oder Steuerungssoftware in der Luftfahrt.
Hohe Anforderungen an das System und den Prozessor bedeuten, dass man eine hohe Leistung bei gleichzeitig möglichst geringem Speicher- und Energieverbrauch benötigt. Es muss also ein Tradeoff zwischen zwei konfliktierenden Forderungen gefunden werden. Dies wird im eingebetteten Bereich durch spezielle Prozessoren gelöst, die sog. *Digitalen Signalprozessoren (DSP)*. Sie zeichnen sich durch irreguläre Hardwareeigenschaften aus: z.B. besitzen sie verteilte, heterogene Registerbänke oder irregulär beschränkte Parallelität auf Instruktionsebene. Gezielte Evaluationen ([SWS95],[VDL+98]) haben gezeigt, dass die klassischen (auf Heuristiken basierenden) Codeerzeugungsverfahren für solche Architekturen keinen qualitativ ausreichenden Code erzeugen, weshalb im Bereich der digitalen Signalbearbeitung häufig noch direkt auf Assemblerebene programmiert wird. Der Grund dafür ist, dass die in den klassischen Verfahren vorgenommenen Transformationen in erster Linie Wert auf Leistungssteigerung legen und dabei die Minimierung der Codegröße eher vernachlässigen. Dadurch erhöhen sich auch die Auswirkungen des Phasenkopplungsproblems (vgl. Abschnitt 2.1) der Codeerzeugung.
Diesem Problem kann man mit speziellen Übersetzern entgegnen, die die vorhandenen Potentiale voll ausschöpfen. Nachteilig daran ist, dass man sehr viel Wissen über die jeweilige Zielarchitektur in den Übersetzer einbringen muss, was sowohl sehr kostenintensiv als auch schwierig angesichts der immer schnelleren Entwicklungszyklen ist. Eine weitere Lösung stellt die Entwicklung von retargierbaren Übersetzersystemen dar, bei denen die Architekturinformationen über Spezifikationen modular zum Übersetzer selbst gehalten werden.
Die Industrie reagiert auf das Problem aber auch mit der Entwicklung von Übersetzern, die es dem Benutzer zumindest erlauben, zusätzlich zu den üblichen Optimierungsoptionen zwischen einer Leistungsoptimierung und einer Platzoptimierung zu wählen.
Weitere Details über die Besonderheiten der Codeerzeugung für eingebettete Systeme finden sich in [Käs00a].

Kapitel 3

VLIW-Architekturen

Dieses Kapitel beschreibt Prozessoren, die zur Klasse der sog. **VLIW-Architekturen** gehören. Der folgende Abschnitt befasst sich dabei mit allgemeinen Eigenschaften dieser Architektur. Abschnitt 3.2 zeigt die vorher beschriebenen Eigenschaften am Beispiel des Philips TriMedia TM1000-Prozessors, der zu dieser Klasse von Prozessoren gehört.

3.1 Allgemeine Eigenschaften

VLIW-Prozessoren gehören zu einer modernen Rechnerarchitektur mit vielen funktionalen Einheiten Diese können parallel zueinander arbeiten, sodass zu einem Zeitpunkt mehrere Operationen gleichzeitig ausgeführt werden können. Deshalb bestehen die Instruktionen (siehe Definition 2.2) aus mehreren Operationen, die parallel zueinander ausgeführt werden. Hierbei ist die Anzahl der Operationen pro Instruktion für den jeweiligen Prozessor fix, d.h. eine Instruktion teilt sich in eine feste Anzahl sog. *Issue-Slots* auf, von denen jeder eine Operation aufnehmen kann. Dadurch entsteht Parallelität auf Instruktionsebene. Abbildung 3.1 zeigt die Aufteilung einer Instruktion in mehrere *Issue-Slots*. Hierbei sind die einzelnen *Issue-Slots* einer Instruktion aber nicht immer völlig gleichwertig. Je nach der spezifischen Implementierung des Prozessors, können bestimmte Operationen seines Instruktionssatzes nur auf bestimmten *Issue-Slots* ausgeführt werden. Diese Einschränkungen limitieren die verfügbare Parallelität und werden als **irreguläre Hardwareeigenschaften** bezeichnet.
Bei VLIW-Prozessoren handelt es sich zumeist um sog. **RISC-Prozessoren**, d.h.

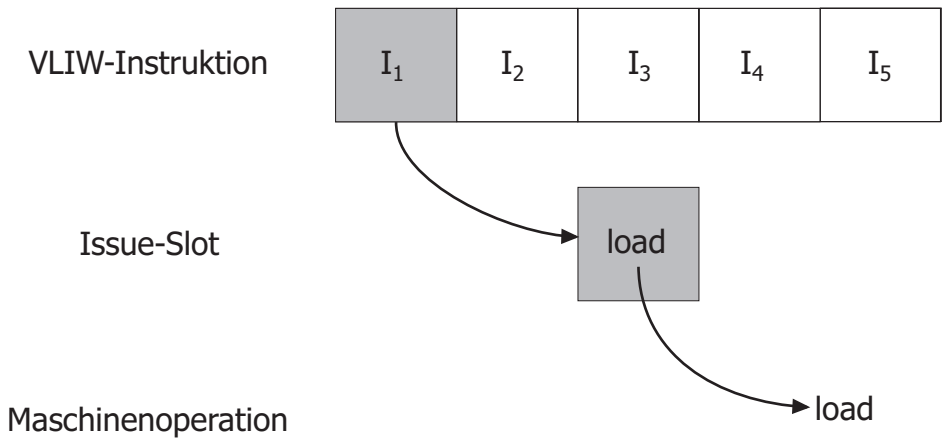


Abbildung 3.1: VLIW-Instruktion

ihr Instruktionssatz ist relativ einfach gehalten und nicht so komplex wie bei **CISC-Prozessoren**. Dadurch wird die Komplexität des Chips reduziert, wodurch mehr Platz

für weitere funktionale Einheiten vorhanden ist. Dies wiederum erhöht die Parallelität. Im Allgemeinen verfügen VLIW-Prozessoren über eine große Anzahl an Registern, um die verfügbare Parallelität besser ausnutzen zu können.
Um die Komplexität von VLIW-Prozessoren weiter gering zu halten, findet kein **dynamisches Scheduling** statt. Die CPU überprüft also zur Laufzeit nicht, ob die Ausführungsreihenfolge der Operationen leistungssteigernd geändert werden kann (**out-of-order Execution**). Der Übersetzer hat hier die Aufgabe, einen möglichst guten Schedule zu erzeugen, was man als **statisches Scheduling** bezeichnet.
Bei solchen Prozessoren tritt ein Problem bei der Ausführung von Verzweigungsoperationen, deren Ausführung von Bedingungscode abhängt (vgl. Abschnitt 5.5.1), auf. Erst nachdem die Bedingung ausgewertet wurde, ist überhaupt bekannt, ob der Sprung genommen wird oder nicht. Es ist also unbekannt, welche Operation als nächste ausgeführt wird, was zur Folge hat, dass keine weiteren Operationen mehr gestartet werden können. Dadurch bricht die verzahnte (parallele) Ausführung der Operationen in der Instruktionspipeline ab. Man spricht in dem Fall auch von einem **Pipeline-Stall**.
Um solche Pipeline-Stalls zu verhindern, gibt es das Prinzip der **Delay Slots**; nach der Ausführung einer Verzweigungs-Operation wird der Sprung wenn überhaupt erst nach einer bestimmen Anzahl von Zyklen (den Delay Slots) genommen, d.h. erst dann wird die Ausführung der Operationen des Sprungzieles begonnen. In diesen Delay Slots können dann andere von der Verzweigungs-Operation unabhängige Operationen ausgeführt werden.

3.2 Philips TriMedia TM1000

3.2.1 Übersicht

Der Philips TriMedia TM1000-Prozessor ist ein digitaler Signalprozessor (DSP), der für Hochleistungsmultimediaanwendungen entwickelt wurde. Zu seinen Anwendungsbereichen gehören die Echtzeitbearbeitung von Audio- und Videodaten sowie die Verarbeitung von Grafik- und Telekommunikationströmen.
Der Kern des TriMedia TM1000 besteht aus einer 32 Bit VLIW-CPU, die mit 100 MHz getaktet ist. Auf ihr läuft ein Realzeitbetriebssystem, das den gesamten DSP steuert. Der TriMedia TM1000 verfügt über 128 homogene Allzweckregister mit einer jeweiligen Breite von 32 Bit und einen Multimedia-Coprozessor, der unabhängig vom Kern arbeiten kann. Verbunden mit einem Bussystem und einer Hauptspeicherschnittstelle befinden sich diese Komponenten zusammen auf einem Chip.
Der beschriebene Aufbau ist in Abbildung 3.2 schematisch dargestellt.

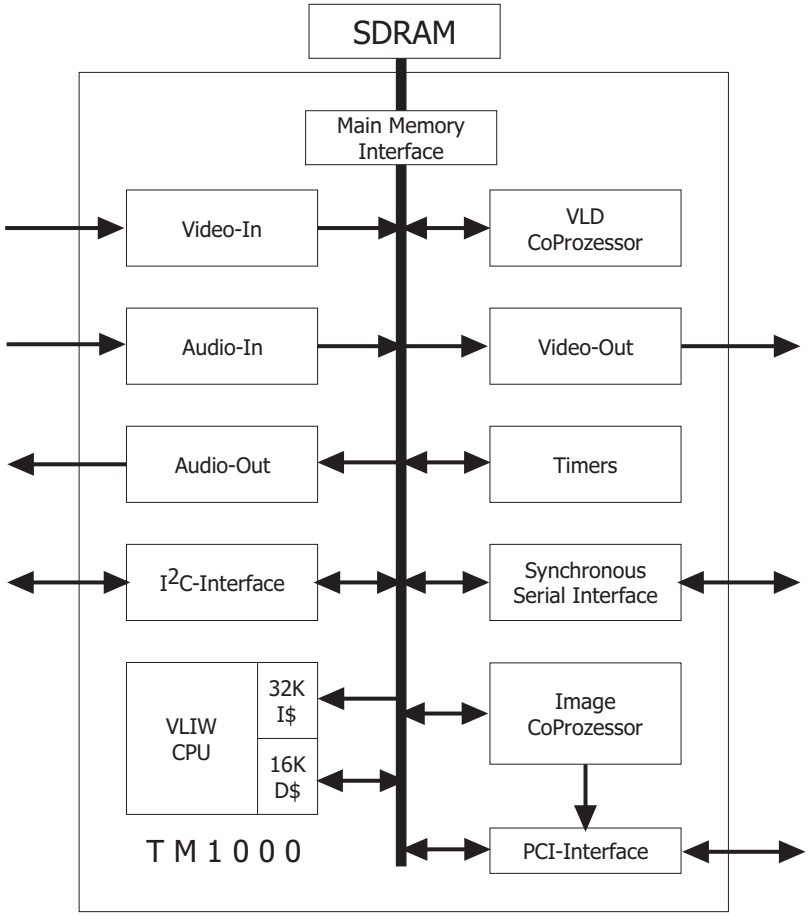


Abbildung 3.2: Blockschaltbild des TriMedia TM1000-Prozessors

3.2.2 Funktionale Einheiten

Der TriMedia TM1000-Prozessor verfügt über 27 parallel arbeitende funktionale Einheiten innerhalb des Prozessorkerns. Darunter die sog. **Integer-Units**, **Floating Point-Units** und die sog. **Multimedia-Units**. Von den beiden erstgenannten gibt es jeweils mehrere Instanzen.
Zu den Multimedia-Units gehören Video-Eingabe-Unit und die Video-Ausgabe-Unit, die beide direkt an digitale Kamerasysteme angebunden werden können. Neben den Video-Eingabe-Units und Video-Ausgabe-Units gibt es die Audio-Eingabe-Units und Audio-Ausgabe-Units. Sie können direkt an digitale Audiosysteme angeschlossen werden. Durch diese direkte Anbindung digitaler Multimediasysteme können Eingangs- bzw. Ausgangsströme bereits vorverarbeitet werden, ohne den Prozessorkern selbst zu belasten. Der TriMedia TM1000 verfügt außerdem über einen **Image-Coprozessor**,

der den Prozessorkern ebenfalls entlastet, indem er z.B. das Kopieren von Bildern in den Framebuffer von Video-Ausgabegeräten oder Bildfilteraufgaben übernimmt. Eine weitere Entlastung der CPU wird durch den vorhandenen Decoder erreicht, der selbstständig MPEG-1 und MPEG-2 Ströme dekodieren kann.
Außerdem verfügt der TriMedia TM1000 über ein **synchrones serielles Interfaces**, mittels dessen eine direkte Anbindung an Modems und ISDN-Leitungen möglich ist. Mittels der sog. **I^2C-Schnittstelle** können externe Videogeräte, wie Decoder und einige Kameratypen, kontrolliert und überwacht werden.
Insgesamt wurden für den Multimediabereich viele relativ einfache Aufgaben, die sonst durch die CPU erledigt werden, in Hardware-Units ausgelagert, um die CPU für komplexere Berechnungen frei zu halten. Eine typische Vorgehensweise im Bereich eingebetteter Systeme.
Der TriMedia TM1000 unterstützt folgende Datentypen:

- vorzeichenbehaftete bzw. -lose ganze Zahlen (*signed/unsigned integer*) in Zweierkomplementdarstellung,
- IEEE-konforme Gleitkommazahlen mit einfacher Genauigkeit (*floats*)

Tabelle 3.1 zeigt die funktionalen Einheiten mit den Operationsklassen, die auf ihnen ausgeführt werden. Die Operationsklassen und ihre Ausführungszeiten sind in Abschnitt 3.2.6 näher beschrieben.

Funktionale Einheit	**Operationsklassen**
CONST	Immediate-Operationen
ALU	ganzzahl-arithmetische Operationen, Logik-Operationen sowie (De-)Kompressions-Operationen
DSPALU	DSP-arithmetische Operationen
DSPMUL	DSP-Multiplikationen
DMEM	Speicherzugriffe
DMEMSPEC	Cache-Operationen
SHIFTER	Shift-Operationen
BRANCH	Kontrollfluss-Operationen
FALU	gleitkomma-arithmetische Operationen, Gleitkomma-Konvertierungen
IFMUL	Ganzzahl- und Gleitkomma-Multiplikationen
FCOMP	Gleitkomma-Vergleiche
FTOUGH	Gleitkomma-Divisionen und Gleitkomma-Wurzelberechnungen

Tabelle 3.1: Funktionale Einheiten

3.2.3 Register

Der TriMedia TM1000 verfügt über 128 jeweils 32 Bit breite homogene Allzweckregister. Hierbei besitzt Register 0 immer den Wert Null und Register 1 immer den Wert 1. Die restlichen Register können frei mit Werten belegt werden. Außerdem gibt es folgende Register mit speziellen Funktionen:

PC enthält die Adresse der aktuell auszuführenden Instruktion.

PCSW Kontrollregister, das gewisse Zustandangaben über den Prozessor enthält, so z.B., ob sich die Ausführung gerade in einer Interrupt-Service-Routine befindet, oder ob ein Write-Back-Bus-Error vorliegt. Dieses Register ist 16 Bit breit.

DPC,SPC *Destination Program Counter* und *Source Program Counter*, die die Zieladressen von Verzweigungen enthalten, falls die Verzweigungsoperation durch einen Interrupt unterbrochen wurde.

CCCOUNT *Clock Cycle Counter*, der die Anzahl der Zyklen seit dem letzten Reboot enthält. Dieses Register ist nicht 32 sondern 64 Bit breit.

3.2.4 Speicheradressierung

Die Adressierungsarten der Operationen lassen drei Zugriffsbreiten auf den Speicher zu: Bytezugriff, 16-Bitzugriff (half-word) und 32-Bitzugriff (word). Hierbei ist zu beachten, dass die Adressierung des Speichers immer registerbasiert ist, d.h. es gibt keine absolute Adressierung. Darüber hinaus gibt es sog. Register-Register-Operationen, deren Operanden Register sind und die auch ihr Ergebnis in ein Register schreiben.

3.2.5 Cache

Der VLIW-Kern des TriMedia TM1000-Prozessors enthält einen 32 KByte großen Instruktionscache zum Puffern der nächsten auszuführenden Instruktionen, und einen 16 KByte großen Datencache zum Puffern von Datenzugriffen auf den Speicher.
Der Datencache ist so entwickelt, dass er zwei gleichzeitige Datenzugriffe (lesend oder schreibend) erlaubt, was durch zwei parallel arbeitende Memory-Units erledigt wird. Der Instruktionscache enthält drei parallele Branch-Units, die das Laden von Instruktionen durchführen.
Beide Caches sind jeweils 8fach assoziativ mit einer Blockgröße von 64 Byte und über ein 32 Bit breites Bussystem an die Hauptspeicherschnittstelle angebunden.

3.2.6 Instruktionssatz

Der Instruktionssatz unterstützt das Konzept der sog. **prädikativen Ausführung**, d.h. die Ausführung der meisten Operationen ist abhängig von einem Prädikat. Die Prädikatierung beim TriMedia TM1000 ist registerbasiert; man kann für fast alle Operationen ein Register als speziellen Operand spezifizieren, sodass der Inhalt dieses Registers über die Ausführung der gesamten Operation entscheidet. Dies hat den Vorteil, dass man ein solches registerbasiertes Prädikat an einer beliebigen Stelle im Programm berechnen kann und an einer ganz anderen Stelle diese Berechnung Einfluss auf die Ausführung einer Operation nehmen kann. Dadurch werden aber auch Kontrollabhängigkeiten in Form von Datenabhängigkeiten definiert, die relativ große Lebensspannen produzieren können, was die Instruktionsanordnung erschwert (vgl. Abschnitt 5.3.2).
Je nach der Bindung einer Operation zum Typ einer funktionalen Einheit, auf der die Operation ausgeführt wird, ergibt sich folgende Klassifikation der Operationen:

- Speicherzugriffe (load, store),
- Shift-Operationen (arithmetische, logische Shifts, ...),
- Logik-Operationen (UND-, ODER-Operationen, ...),
- ganzzahl-arithmetische Operationen,
- Ganzzahl-Multiplikationen,
- Ganzzahl-Vergleiche (z.B. <-Relation, ...),
- gleitkomma-arithmetische Operationen,
- Gleitkomma-Konvertierungen (z.B. von ganzzahligen Werten zu Gleitkommawerten, ...),
- Gleitkomma-Vergleiche (z.B. <-Relation, ...),
- Gleitkomma-Multiplikationen,
- Immediate-Operationen, die z.B. Konstanten in Register schreiben,
- Sign-/Null-Extension-Operationen,
- Kontrollflussoperationen (Verzweigungen bzw. Aufrufe),
- Kompressions- und Dekompressionsoperationen, die Matrizen packen bzw. entpacken können.

- Spezialregister-Operationen, die die Inhalte spezieller Register, wie z.B. den des DPC-Register, verändern,
- DSP-Operationen, die spezielle abgewandelte arithmetische Operationen durchführen,
- Cache-Operationen, die die Allokation bzw. Verändern von Cacheinhalten ermöglichen und

Der TriMedia TM1000-Prozessor verfügt also nicht nur über den üblichen Satz von Operationen, wie arithmetische, logische und vergleichende Operationen, u.s.w., sondern zudem über spezielle Operationen, die für Multimediaanwendungen oft benötigt werden.
Tabelle 3.1 in Abschnitt 3.2.2 listet auf, welche der hier genannten Operationsklassen auf welchen funktionalen Einheiten ausgeführt werden.
Die Ausführungszeiten der Operationen reichen dabei von einem Zyklus für einfache Additionen bis hin zu 17 Zyklen für spezielle Operationen, die Gleitkommadivisionen oder Wurzelberechnungen durchführen. Tabelle 3.2 zeigt die Ausführungszeiten auf den jeweiligen funktionalen Einheiten. Die einzelnen Operationen können hierbei nicht

Funktionale Einheit	**Zyklen**
const	1
alu	1
dspalu	2
dspmul	3
dmem	3
dmemspec	3
shifter	1
branch	3
falu	3
ifmul	3
fcomp	1
ftough	17

Tabelle 3.2: Ausführungszeiten der Operationen

in beliebigen Issue Slots ausgeführt werden sondern bestimmte Bedingungen müssen vom Codeerzeuger sichergestellt werden. Dies ist im folgenden Abschnitt erläutert.
Der TriMedia TM1000 gehört zu der Klasse der `load/store`-Prozessoren, d.h. ein Zugriff auf den Speicher ist nur über spezielle load und store-Operationen möglich. Hierbei ergeben sich einige Besonderheiten aufgrund der Parallelität auf Instruktionsebene. Sei T_{load} der Startzeitpunkt einer load-Operation und T_{store} der Startzeitpunkt

einer store-Operation, bei denen sich die Speicherbereiche, auf die zugegriffen werden soll, überschneiden. Dann gelten folgende Bedingungen:

- $T_{load} < T_{store}$: der load-Befehl lädt die alten Speicherinhalte.
- $T_{store} < T_{load}$: der load-Befehl lädt die neuen Speicherinhalte.
- $T_{load} = T_{store}$: dies führt zu undefiniertem Verhalten.
- zwei Speicheroperationen (store) zur gleichen Zeit führen zu undefiniertem Verhalten.

Außerdem stellt der TriMedia TM1000 spezielle Befehle zur Unterstützung von Multimediaanwendungen bereit, z.B. zur Anwendung von Clipping.

3.2.7 Ausführungsbedingungen

Wie in Abschnitt 3.1 erwähnt, teilt sich eine VLIW-Instruktion in mehrere *Issue-Slots* auf. Da diese die Bindung von Operationen auf die internen funktionalen Einheiten, auf denen sie ausgeführt werden, repräsentieren, gibt es einige Regeln dafür, welche Operationen auf welchen *Issue-Slots* einer Instruktion ausgeführt werden können.
So ist jeder *Issue-Slot* an bestimmte funktionale Einheiten gebunden, d.h. eine Operation kann nur auf einem *Issue-Slot* ausgeführt werden, dem von jeder benötigten funktionalen Einheit mindestens eine zugeordnet ist. Außerdem sind insgesamt fünf Write-Back-Busse vorhanden, sodass zu jedem Zeitpunkt nicht mehr als fünf Resultate in ihre Ziele zurückgeschrieben werden können. Diese Begrenzung korrespondiert mit der maximalen Anzahl an Operationen pro Instruktion, die ebenfalls bei fünf liegt. Genauer gesagt enthält eine Instruktion immer exakt fünf Operationen. Ist eine oder mehrere dieser Operationen allerdings ein `nop`, so spricht man häufig davon, die Instruktion enthielte weniger als fünf Operationen.
Da die Ausführungszeiten der Operationen nicht einheitlich sind (vgl. Tabelle 3.2), müssen während der Codeerzeugung die fünf Instanzen des Write-Back-Bus gesondert behandelt werden. Andernfalls würde die Beschränkung einer Instruktion auf fünf Operationen auch automatisch die Beschränkung auf fünf Benutzungen des Write-Back-Bus zu einem Zeitpunkt implizieren.
Abbildung 3.3 zeigt die funktionalen Einheiten samt ihrer Zuordnung zu *Issue-Slots* (vgl. Abschnitt 3.2.6).

Weitere Informationen über den TriMedia TM1000-Prozessor können [Phi97] entnommen werden.

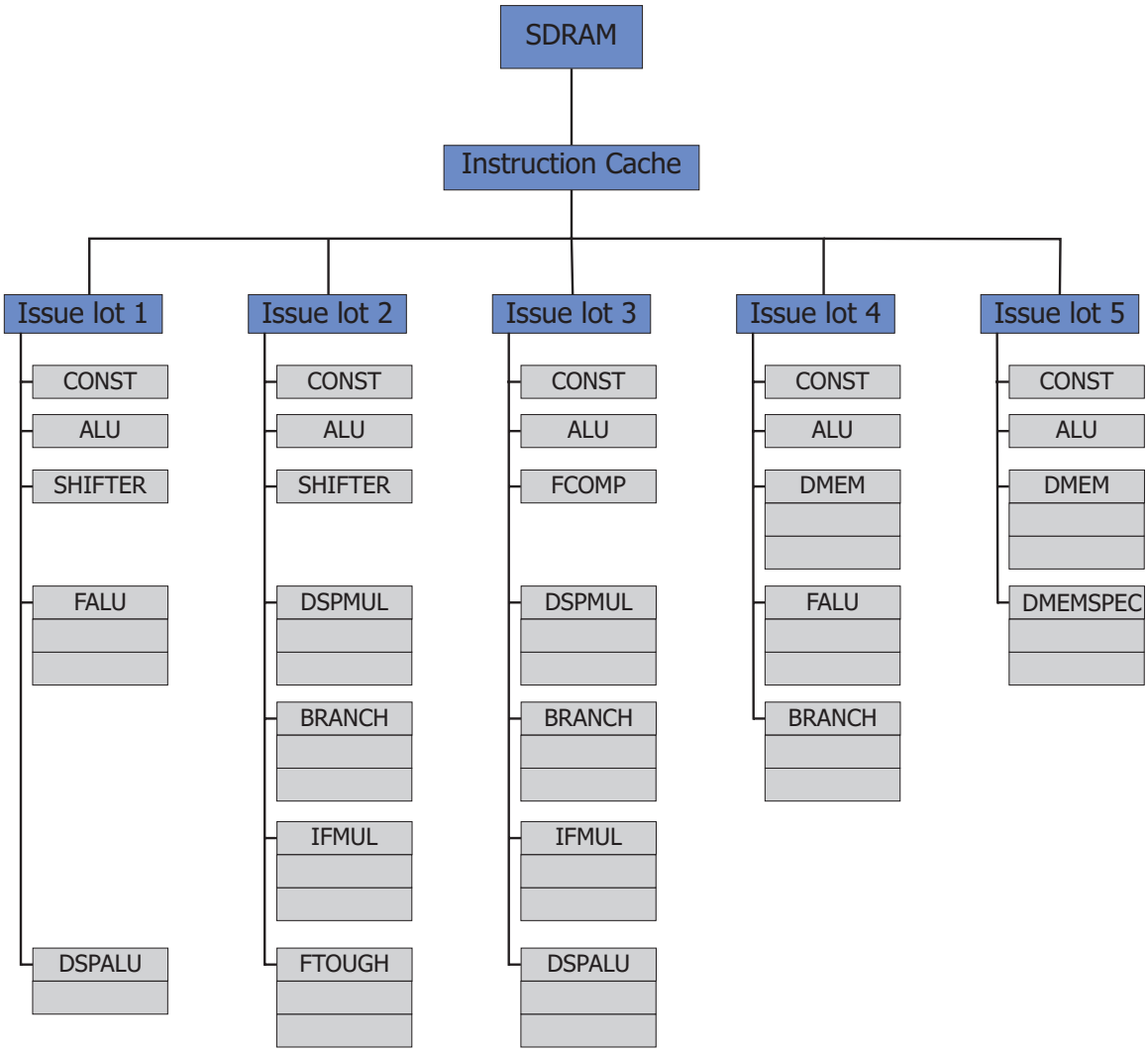


Abbildung 3.3: Zuordnung der funktionalen Einheiten zu Issue-Slots

Kapitel 4

Instruktionsanordnung

4.1 Einführung

Die Instruktionsanordnung bestimmt die Reihenfolge, in der eine Sequenz von Maschinenoperationen durch den Prozessor ausgeführt werden. Man geht dabei von einer initialen Ausführungsreihenfolge aus, wie sie durch den Kontrollflussgraph vorgegeben ist. Durch Umordnen der Operationen wird versucht, das verfügbare Potential an *Intraprozessorparallelität* möglichst voll auszuschöpfen mit dem Ziel, die Gesamtausführungszeit des Programmes zu minimieren. Potential für diese Parallelisierung ergibt sich aus den in modernen Prozessoren vorhandenen parallelen funktionalen Einheiten und/oder Instruktionspipelines.
Der verfügbare Grad an Parallelität, d.h. inwieweit Operationen in einer Maschinencodesequenz semantikerhaltend umsortiert werden können, ist hierbei durch verschiedene Abhängigkeiten zwischen den Operationen limitiert. Die Instruktionsanordnung lässt sich wie folgt definieren:

Definition 4.1 (Instruktionsanordnung). Sei der initiale Ablaufplan (**Schedule**) eines Eingabeprogrammes durch seinen Kontrollflussgraph gegeben. Die **Instruktionsanordnung** bezeichnet eine Sequenz von semantikerhaltenden Transformationen, die die initiale Ausführungsreihenfolge der Operationen ändert, mit dem Ziel, den Grad an paralleler Ausführung zu maximieren und damit die Gesamtausführungszeit des Programmes zu minimieren.
Der durch die Instruktionsanordnung erzielte Schedule muss dabei folgende Bedingungen erfüllen:

- Erfüllung aller Ressourcenbedingungen,
- Wahrung aller Datenabhängigkeiten und
- Wahrung aller Kontrollabhängigkeiten.

■

Die Einhaltung der Ressourcenbedingungen bedeutet, dass man höchstens soviele Operationen zur gleichen Zeit ausführt, wie auf dem zugrundeliegende Zielprozessor möglich ist. Diese Bedingung ist also nicht allgemein gültig berechenbar sondern spezifisch für den Zielprozessor.

Bemerkung. Das Problem der Instruktionsanordnung beschränkt auf die Einhaltung der Ressourcenbedingungen ist bekannt als das sog. *Resource-constrained Scheduling Problem*. Beschränkt man sich auf die Wahrung der Datenabhängigkeiten, nennt man das Problem *Precedence-constrained Scheduling Problem* ([GJ79]).

Betrachten wir das *Precedence-constrained Scheduling Problem*:
Der Datenabhängigkeitsgraph definiert eine partielle Ordnung auf den Maschinenoperationen des Eingabeprogrammes. Eine Präzedenzrelation ist dabei wie folgt definiert:

Definition 4.2 (Präzedenzrelation). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph. Ein Knoten i hat Vorrang über einen Knoten j bezüglich einer **Präzedenzrelation** $\prec \subset N_{dd} \times N_{dd}$, genau dann, wenn es einen Pfad von i nach j im Datenabhängigkeitsgraphen gibt.

$$i \prec j \Leftrightarrow i \xrightarrow[G_{dd}]{*} j$$

$\prec$ setzt also alle Operationen in Relation zueinander, die direkt oder indirekt voneinander abhängig sind. ■

Mit dieser Relation lässt sich das *Precedence-constrained Scheduling Problem* wie folgt beschreiben: man versucht, die Ausführungsreihenfolge der Operationen so zu verändern, dass die Gesamtausführungszeit möglichst gering wird, ohne dabei eine Vorrangbedingung zu verletzen.

Definition 4.3 (Precedence-constrained Scheduling Problem). Sei A eine Menge von Aufgaben der Ausführungszeit 1, auf denen eine partielle Ordnung $\prec$ definiert ist. Man hat m Maschinen und eine Obergrenze T für die Gesamtausführungszeit. Gesucht ist ein Schedule $\sigma : M \to \{1, \ldots, T\}$, so dass für alle $i \in \{1, \ldots, T\}$ mit $|\{t \in A : \sigma(t) = i\}| \leq m$ gilt:

$$t \prec t' \Rightarrow \sigma(t) < \sigma(t')$$

Für das Instruktionanordnungsproblem entsprechen die Maschinen den Prozessoren, die Aufgaben parallel arbeitenden funktionalen Einheiten. ■

Bemerkung. Das Problem des *Precedence-constrained Scheduling* ist bereits für $m = 2$ Maschinen $\mathcal{NP}$-vollständig ([GJ79]). Das gleiche gilt, wenn man als Längen der Aufgaben 1 und 2 zulässt.

In der Realität ist die Instruktionsanordnung noch komplexer, da zum einen nicht alle Operationen eines Prozessors die gleiche Ausführungszeit benötigen und zum anderen moderne Prozessoren Irregularitäten aufweisen. Letzteres kann z.B. bedeuten, dass bestimmte Operationen aufgrund von Hardwareanforderungen nicht parallelisiert werden können.

Beschränkt man die Instruktionsanordnung auf die Basisblöcke, d.h. für jeden Basisblock wird jeweils unabhängig voneinander ein Ablaufplan gesucht, bezeichnet man dies als **lokale Instruktionsanordnung**. Studien ([RF93]) haben gezeigt, dass die verfügbare Parallelität innerhalb eines Basisblockes typischerweise nicht größer als 2-3[1] ist. Um bessere Ergebnisse zu erzielen, muss man die Instruktionsanordnung über

[1] erzielt für numerische Programme, für nichtnumerische Programme ist der mögliche Parallelität meist geringer

die Grenzen von Basisblöcken hinweg anwenden, was man als **globale Instruktionsanordnung** bezeichnet. Dafür benötigt man zusätzlich die Kontrollabhängigkeiten des Eingabeprogrammes, um weiterhin die Erhaltung der Programmsemantik zu gewährleisten. Die Notwendigkeit der Kontrollabhängigkeiten für die globale Instruktionsanordnung wurde in Kapitel 2 erläutert.
Die Instruktionsanordnung nimmt einen wichtigen Stellenwert innerhalb der Codeerzeugung ein, da sie wesentlich die erzielte Performanz des Eingabeprogrammes auf dem zugrundeliegenden Prozessor beeinflusst. Im Laufe der Zeit wurden viele Verfahren entwickelt und verfeinert, die man grob in zwei Klassen einteilen kann. Man unterscheidet zum einen die **azyklische Instruktionsanordnung**, die, schleifenfreie Codesequenzen anordnet, und zum anderen die **zyklische Instruktionsanordnung**, die sich auf Schleifen konzentriert. Die Unterscheidung in lokale und globale Instruktionsanordnung ist hierbei orthogonal zu der Unterscheidung in azyklische und zyklische Verfahren bis auf die Ausnahme, dass es keine lokalen zyklischen Verfahren gibt.
Aufgrund der hohen Berechnungskomplexität sind heuristische Verfahren weit verbreitet und werden meist auch in aktuellen Übersetzern eingesetzt. Detailliertere Analysen heuristischer Verfahren sind in [Lan97] zu finden. Dennoch ist die Entwicklung von exakten Lösungsverfahren weiterhin Bestandteil aktueller Forschungsarbeiten ([Win04, Käs00a]), in denen meist die Methode der ganzzahligen Programmierung verwendet wird. Im folgenden werden die Eigenschaften beider Klassen an bekannten Heuristiken kurz vorgestellt.

4.2 Azyklische Instruktionsanordnung

Wie es der Name schon sagt, zeichnet sich die azyklische Instruktionsanordnung durch ihre Anwendbarkeit auf azyklische Codesequenzen aus. Da die Schranken, in denen Operationen umsortiert werden können, unter anderem von den Daten- und Kontrollabhängigkeiten bestimmt werden, ist das Ergebnis der Instruktionsanordnung eine topologische Sortierung, wenn man die in Abschnitt 4.1 angegebene Präzedenzrelation als Ordnung auf den Operationen betrachtet. Als Beispiel werden nun zwei azyklische Verfahren, *List Scheduling* ([Bas95]) und *Trace Scheduling* ([WM97, Fis81]) welche beide heuristik-basiert sind, vorgestellt. Weitere azyklische Instruktionsanordnungsverfahren sind in [Win04] aufgeführt.

4.2.1 List Scheduling

List Scheduling berechnet eine Menge von Operationen (Data Ready Set, DRS), die in die aktuelle Instruktion platziert werden können, berechnet. Dies hängt von den folgenden drei Kriterien ab:

1. alle Präzedenzbedingungen sind erfüllt, d.h. es gibt keine Operation, die noch nicht angeordnet ist, aber in Vorrangrelation zu einer Operation aus DRS steht.

2. alle Operationen aus DRS verletzen keine Ressourcenbedingungen des Prozessors, d.h. sie können parallel zu allen bereits angeordneten Operationen der aktuellen Instruktion ausgeführt werden.

Die letzte Bedingung entfällt, falls es sich bei dem Zielprozessor nicht um eine VLIW-Architektur (vgl. Kapitel 3) handelt.
Um nun zu entscheiden, welche der Operationen aus DRS zum Anordnen ausgewählt wird, berechnet man Prioritäten anhand von Gewichtungsheuristiken. Eine solche Heuristik, die beim List Scheduling verwendet wird, ist beispielsweise die sog. **highest-level-first priority**. Diese betrachtet die maximale Distanz bis zum Endknoten im Datenabhängigkeitsgraphen. Je nach Typ der Datenabhängigkeit zwischen zwei Operationen kann man eine minimale zeitliche Distanz in Zyklen angeben, die zwischen der Ausführung beider liegen muss. Die höchste Priorität hat dabei die Operation mit der geringsten Distanz. Dann wird die ausgewählte Operation in die aktuelle Instruktion eingefügt und schließlich die Operationsmenge DRS neu berechnet.
Ist die berechnete Menge DRS für eine Instruktion leer, so kann man keine Operation mehr zu der aktuellen Instruktion hinzufügen. In dem Fall wird eine neue Instruktion erzeugt und DRS neu berechnet. List Scheduling ist wohl eines der am weitesten verbreiteten lokalen Instruktionsanordnungsverfahren und besitzt eine Zeitkomplexität von $\mathcal{O}(n^2)$.
Aufgrund der Beschränkung auf die Datenabhängigkeiten kann List Scheduling nur zur lokalen Instruktionsanordnung verwendet werden. In Abschnitt 5.3.2 wird gezeigt, dass dem List Scheduling sehr ähnliche Algorithmen sowohl zur globalen als auch zyklischen Instruktionsanordnung verwendet werden können.

4.2.2 Trace Scheduling

Trace Scheduling ([Fis81]) ist ein globales Instruktionsanordnungsverfahren, welches nicht nur einzelne Operationen betrachtet, sondern versucht, ganze Pfade (Traces) des Programmes anzuordnen.
Die Idee beruht darauf, die Instruktionen aufeinanderfolgender Basisblöcke gemeinsam anzuordnen. Dazu benötigt man sog. Profiling-Informationen über die Ausführungshäufigkeit einzelner Programmteile, welche man über Messungen oder heuristische Schätzungen erlangen kann.
Die genauere Vorgehensweise des Trace Scheduling teilt sich in zwei Schritte:

- Zerlegung des Kontrollflussgraphen in disjunkte Teilpfade

- Anordnung der Teilpfade

Zur Zerlegung des Kontrollflussgraphen in disjunkte Teilpfade wird iterativ jeweils der Basisblock betrachtet, der am häufigsten ausgeführt wird. Für diesen muss nun entschieden werden, ob und wieviele seiner Nachfolger im Basisblockgraph zum Teilpfad hinzugenommen werden. Diese Entscheidung ist abhängig von der Zielarchitektur.
Hat man nun disjunkte Teilpfade berechnet, werden ihre Instruktionen angeordnet, indem ein Teilpfad ähnlich behandelt wird wie ein Basisblock. Dadurch tritt eventuell der Fall auf, dass eine Instruktion über die Grenzen ihres ursprünglichen Basisblockes hinaus verschoben wird. Bewirkt man dadurch eine Verschiebung über Kontrollabhängigkeiten hinweg, so sind semantikerhaltende Transformationen erforderlich. Verschiebt man z.B. eine Instruktion über einen Verschmelzungs- oder Verzweigungspunkt hinaus, benötigt man Kompensationskopien der Instruktion in Basisblöcken, die in diesen Teilpfad einmünden oder von ihm ausgehen.
Die hierbei kürzest mögliche Sequenz von Instruktionen ist minimal so lang wie der längste Pfad im Abhängigkeitsgraph. Diese kürzest mögliche Sequenz lässt sich aus der topologischen Sortierung desselben heraus bestimmen.
Dadurch kann man mittels Trace Scheduling globale Instruktionsanordnung implementieren. Die berechneten Teilpfade gehen hierbei allerdings nie über die Grenzen von Schleifen hinaus, weshalb das Verfahren zu den azyklischen Instruktionsanordnungsverfahren zählt. Man spricht bei diesem Problem auch von einer Barriere für die Codeverschiebung an der sog. Rückwärtskante[2] einer Schleife. Einen Nachteil stellt der unter Umständen enorme Aufwand für semantikerhaltende Transformationen dar. Um dies in Grenzen zu halten, wurden weitere, dem Trace Scheduling ähnliche Verfahren entwickelt, darunter *Superblock Scheduling*([HMC+93]) und *Hyperblock Scheduling*([MLC+92]).

4.3 Zyklische Verfahren

Zyklische Instruktionsanordnungsverfahren unterscheiden sich von den azyklischen Verfahren dadurch, dass sie auf zyklischen Kontrollfluss angewendet werden können. Zyklischer Kontrollfluss bringt eine neue Problematik für die Instruktionsanordnung ins Spiel. Wie bereits in Kapitel 2 erwähnt, erzeugen Schleifen neue Formen von Daten und Kontrollabhängigkeiten, sog. *schleifenabhängige* und *schleifenunabhängige Abhängigkeiten* (vgl. Definition 2.19). Mit den azyklischen Verfahren ist man in dem Fall auf die lokale Instruktionsanordnung beschränkt. Wie geht man auf diese Abhängigkeiten,die über die Grenzen von Iterationen einer Schleife hinweg gehen ein? Ein erster Ansatz der Lösung dieses Problems ist die Reduktion auf das Problem der azyklischen Instruktionsanordnung. Um den Körper einer Schleife von seiner zyklischen Eigenschaft zu befreien, muss man die Ausführung einer Schleife näher betrachten. Sie

[2]die Rückwärtskante, auch *Backedge* genannt, ist die rückwärts gerichtete Kante einer Schleife im Kontrollflussgraph. Sie läutet den Start einer neuen Iteration ein.

stellt die n-malige Hintereinanderausführung des Schleifenkörpers dar, wobei n die Anzahl der Iterationen ist. Kopiert man den Schleifenkörper n Mal hintereinander, erhält man eine azyklische Codesequenz, die genau der Abarbeitung der Schleife entspricht. Man spricht bei dieser Vorgehensweise von *Schleifenaufrollen(Loop-Unrolling)*.
Das Anordnen der Instruktionen dieser azyklische Codesequenz entspricht nun dem Problem der azyklischen Instruktionsanordnung und kann durch die in Abschnitt 4.2 vorgestellen Verfahren gelöst werden. Diese Vorgehensweise birgt allerdings zwei große Probleme:

- Codeexplosion durch das Aufrollen und eine
- eventuell schlechte Codequalität abhängig vom Eingabeprogramm.

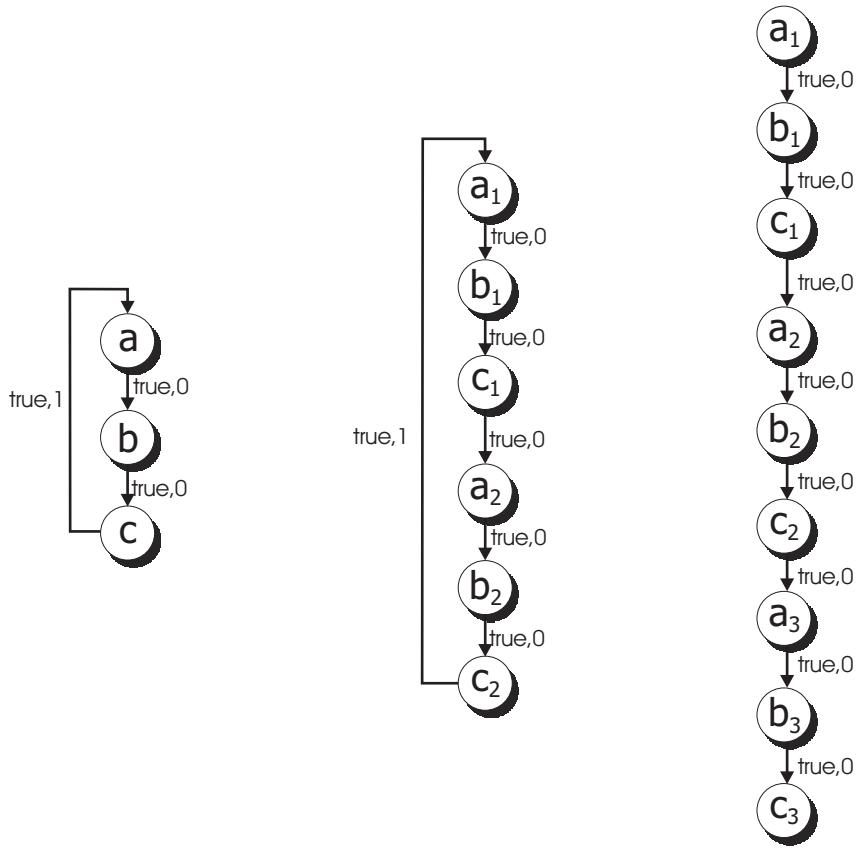
Die Codeexplosion ist vor allem im eingebetteten Bereich sehr problematisch aufgrund der harten Speicherplatzbeschränkungen. Beispiel 4.3 zeigt, dass man durch Anwendung dieser Methode in Abhängigkeit des Eingabeprogrammes eventuell keinen Gewinn an Performanz verzeichnen kann. In dem Fall ist der damit erzeugte Code qualitativ wie quantitativ schlechter als ohne Anwendung von *Loop-Unrolling*.

Bemerkung (Problematik von Loop-Unrolling*).* Abbildung 4.1 zeigt einen vereinfachten[3] Datenabhängigkeitsgraphen(DDG) einer Schleife (a). Rollt man nun den Schleifenkörper einmal aus, ergibt sich der DDG in (b). Neu hinzugekommen ist eine Datenabhängigkeit mit einer Distanz von Null, von Operation c_1 zu Operation a_2. Die schleifenabhängige Abhängigkeit mit der Distanz von eins geht nun von Operation c_2 zu Operation a_1. (c) zeigt den DDG der Schleife, wenn sie komplett ausgerollt worden ist unter der Annahme, dass die Gesamtanzahl der Iterationen drei ist. Die Änderungen der Datenabhängigkeiten ergeben sich analog zu den Änderungen von (b).
Die Datenabhängigkeiten zwischen den Operationen von (b) und (c) ergeben eine Kette, d.h. jede Operation kann erst ausgeführt werden, wenn ihr Vorgänger im DDG ausgeführt worden ist. Dies bedeutet, keine der Operationen kann durch List Scheduling parallelisiert werden.

4.4 Softwarepipelining

Einen zu *Loop-Unrolling* konträren Ansatz verfolgt man mit **Softwarepipelining**, einem zyklischen globalen Instruktionsanordnungsverfahren, das mittlerweile in vielen Übersetzern eingesetzt wird.
Softwarepipelining beruht auf der Idee, die Ausführung der verschiedenen Iterationen einer Schleife miteinander zu überlappen und dadurch die Codesequenz, die

[3] der Einfachheit wegen wurden nur Setzung-Benutzung-Abhängigkeiten verwendet



(a) Original-Schleife (b) 1x aufgeklappte Schleife (c) komplett aufgeklappte Schleife

Abbildung 4.1: Loop-Unrolling

tatsächlich iteriert werden muss, zu minimieren. Dadurch ergibt sich dann eine Leistungssteigerung bei verhältnismäßig geringer Erhöhung der Codegröße.
Softwarepipelining ist mittlerweile als eigenständiges Verfahren zur Instruktionsanordnung von Schleifen anerkannt und wird in vielen hochoptimierenden Übersetzern eingesetzt. So z.B. in den hochoptimierenden Übersetzern von *Intel®* für die *Itanium™* Architektur und einigen Übersetzern von *Texas Instruments*.
Diese Arbeit beschreibt, wie man Softwarepipelining auf Assemblerebene betreibt, ohne sich dabei auf eine Zielarchitektur zu beschränken. Dies ist in Kapitel 5 ausführlich erläutert.

Kapitel 5

Softwarepipelining

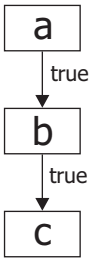
5.1 Einführung

Rechnerarchitekten benutzen u.a. sog. *Befehlsfliessbänder(Pipelines)* ([The04]), um Intraprozessorparalellität ausnutzen zu können. Die Grundlage bildet die Unterteilung der Ausführung einer Operation in eine Sequenz von Arbeitsschritten[1], sog. *Stufen*, sodass z.B. während sich eine Operation im zweiten Arbeitsschritt befindet, der erste Arbeitsschritt bereits für die nächste Operation begonnen werden kann. Man erhöht durch diese verzahnte Ausführung den durchschnittlichen Durchsatz an Operationen pro Zyklus (*Cycles per Instruction (CPI)*) des Prozessors. So wird dann z.B. im Schnitt in jedem Zyklus die Ausführung einer Operation beendet, obwohl die Ausführungszeiten der einzelnen Operationen durchaus länger als ein Zyklus sind. Softwarepipelining überträgt diese Idee auf die Operationen eines Schleifenkörpers, um so die verschiedenen Iterationen einer Schleife in ihrer Ausführung zu überlappen. Das heißt, das Verfahren geht direkt auf die zyklische Eigenschaften des Kontrollflusses ein und wird auch nur auf die Schleife selbst angewendet. Die Anwendung eines Softwarepipelining Verfahrens nennt man auch *Pipelining*.
Zuerst werden disjunkte Mengen von Operationen im Kontrollfluss eines Schleifenkörpers identifiziert, die im Rahmen ihrer Daten- und Kontrollabhängigkeiten parallelisiert werden können. Jeder dieser Mengen entspricht dabei einer *Pipeline-Stufe* der Schleife. Wie bei der Ausführung einer Maschinenoperation kann man die Ausführung einer Iteration in die Ausführung der einzelnen Stufen aufteilen. Ähnlich zu einer Pipeline moderner Prozessoren kann man hier die Ausführung von Iterationen miteinander verzahnen, indem man die Ausführung unterschiedlicher Stufen aufeinanderfolgender Iterationen parallelisiert. D.h. wird die zweite Stufe der ersten Iteration einer Schleife ausgeführt, kann man bereits die Ausführung der ersten Stufe der zweiten Iteration starten. Durch diese Parallelisierung von Stufen unterschiedlicher Iterationen wird ein neuer Schleifenkörper gebildet, der sog. **Kernel**. Wie bei den eingangs des Kapitels genannten Pipelines, wird hier bei jeder Ausführung einer Iteration des Kernels eine Iteration der Originalschleife beendet, da Stufen aus unterschiedlichen Iterationen verzahnt ausgeführt werden. Beispiel 5.1 veranschaulicht die Transformationen, denen eine Schleife durch Softwarepipelining unterliegt.

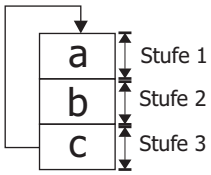
Beispiel 5.1. Gegeben sei ein VLIW-Prozessor, dessen Instruktionen bis zu drei Operationen umfassen können. Abbildung 5.1(a) zeigt vereinfacht den Datenabhängigkeitsgraphen einer Schleife, Abbildung 5.1(b) ihren Kontrollfluss. Die Ausführung einer Schleife kann hier in drei Phasen (Stufen) aufgeteilt werden, nämlich die Ausführung der Operation a, die von b und die von c. Abbildung 5.1 (c) zeigt die Schleife aus (b), wenn man ihren Schleifenkörper vier Mal aufklappt. Alle Operationen, die in einer Zeile dargestellt sind, bilden dabei zusammen eine Instruktion, werden also parallel ausgeführt. Hierbei zeigt sich, dass es eine Wiederholung der vierten Instruktion (c,b,a) gibt. Dadurch kann man den neuen Schleifenkörper, den Kernel, auf eben

[1] wie etwa das Dekodieren der Operation, das Lesen der Operanden, etc.

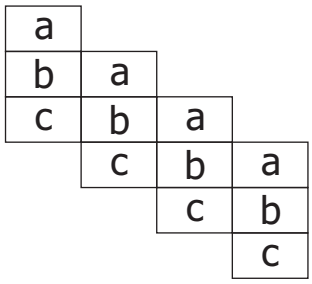
diese Instruktion reduzieren (vgl. Abbildung 5.1 (d)). Man sieht dabei, dass hier die unterschiedlichen Stufen der Schleife parallel zueinander ausgeführt werden. ■



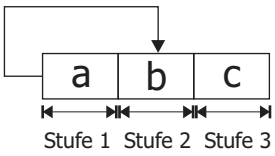
(a) DDG Originalschleife



(b) CFG Originalschleife



(c) 4x aufgeklappte Schleife



(d) Kernel

Abbildung 5.1: Konzeptansicht - Softwarepipelining

Wie man in Beispiel 5.1 auch sieht, kann der berechnete Kernel nicht allein als Ersatz für die ursprüngliche Schleife dienen, da bildlich gesprochen die Pipeline zu Beginn der Ausführung des Kernels gefüllt sein muss. Man muss hier dem Kernel noch eine Codesequenz vorschalten. Diese Sequenz bezeichnet man auch als **Prolog**. Analog dazu benötigt man auch eine Codesequenz, die dem Kernel nachgeschaltet werden muss. Diese nennt man **Epilog**. Die Notwendigkeit dieser Codesequenzen und ihre Berechnung werden genauer in Abschnitt 5.3.2 behandelt.

Nachteilig an Softwarepipelining ist, dass die Codegröße der Schleife durch das Pipelining anwächst, nämlich genau um die Anzahl der Operationen von Prolog und Epilog. In Abschnitt 5.3.2 wird aber gezeigt, dass es Verfahren von Softwarepipelining gibt, die das Anwachsen des Codes in einem verträglichen Rahmen halten.
Da der erzeugte Kernel mehr als eine Iteration der ursprünglichen Schleife während seiner Ausführung bearbeitet, können nur noch eine bestimmte Anzahl von Schleifeniterationen ausgeführt werden. Diese Beschränkung ist genauer in Abschnitt 5.3.2 erläutert.
Softwarepipelining steht nicht für einen einzigen Algorithmus sondern vielmehr für eine ganze Klasse von Algorithmen, denen die oben beschriebene Grundlage gemein ist. Im folgenden Abschnitt werden die unterschiedlichen Arten der Implementierung von Softwarepipelining genannt. Abschnitt 5.3 behandelt dann im Detail eine dieser Arten, das sog. **Modulo Scheduling**, welches in dieser Arbeit verwendet wurde, um Softwarepipelining auf Assemblerebene zu implementieren. In Abschnitt 5.5 wird beschrieben, wie man durch spezielle Hardware-Eigenschaften die Codeerzeugung für Softwarepipelining erleichtern und damit die mögliche Leistungssteigerung erhöhen kann. Abschnitt 5.4.1 beschreibt eine Erweiterung des Verfahrens für geschachtelte Schleifen.

5.2 Arten

Eine Möglichkeit, Softwarepipelining umzusetzen, ist das Verschieben einzelner Instruktionen über die Rückwärtskante aus einer Iteration in eine andere. Dieses Verfahren wird auch **move-then-schedule** genannt. Hierbei ist sowohl ein Verschieben in die nachfolgende als auch in die vorhergehende Iteration möglich. Findet eine solche Umordnung über die Grenzen einer Kontrollabhängigkeit hinweg statt, so muss man wie bei allen globalen Instruktionsanordnungsverfahren, semantikerhaltende Transformationen vornehmen. Dies entspricht hier dem Erzeugen von Kompensationkopien bei einer Verschiebung über eine Kontrollflussverzweigung bzw. dem Unifizieren zweier Operationen bei einer Verschiebung über eine Verschmelzung hinweg ([Ebc87, Jai91, GS92, ME92]).
Bei dieser Vorgehensweise ist aber nicht immer klar bestimmbar, welche Operation in welcher Richtung (vorwärts/rückwärts) über die Rückwärtskante verschoben werden soll/muss, um sich weiter dem Optimum zu nähern. Da die optimale Anordnung nicht bekannt ist, kann es passieren, dass man zu exzessive Codeverschiebung betreibt und sogar Performanz verliert.
Ein konträrer Ansatz zu *move-then-schedule* verfolgt das sog. **schedule-then-move** Verfahren, bei dem versucht wird, von Grund auf einen optimalen Schedule aufzubauen, indem einem partiellen Schedule solange Operationen hinzugefügt werden, bis alle Operationen der Schleife angeordnet sind. Dadurch sind die Codeverschiebungen

über die Rückwärtskante hinweg implizit im Algorithmus enthalten. Auch hier kann man zwischen zwei Vorgehensweisen unterscheiden: *Kernel Recognition* und *Modulo Scheduling*.
Kernel Recognition ([AJLA95]) Verfahren zeichnen sich durch folgende Vorgehensweise aus:

1. Abrollen der Schleife,
2. Anordnung der Operationen und anschließend
3. Suche nach sich wiederholenden Codesequenzen, dem Kernel.

Sollte sich keine wiederholende Codesequenz finden, werden die drei Schritte iteriert bis sich ein Kernel bildet. Algorithmen dieser Art sind *Perfect Pipelining* ([AN88]), *Petri Net Model* ([AJLA95]) und *Vegdahl's Technique* ([Veg92],[Veg82]).
Modulo Scheduling ist ein Rahmenwerk zur Implementierung von Softwarepipelining, das im Gegensatz zu Kernel Recognition die Schleife nicht aufrollt sondern aus den Daten- und Kontrollabhängigkeiten versucht, ein sog. *Initiierungsintervall* zu berechnen. Dieses gibt an, nach wievielen Zyklen eine neue Iteration gestartet wird.
Modulo Scheduling ist die am weitesten verbreitete Methode, um Softwarepipelining zu implementieren und kann zum einen durch eine Formulierung als ganzzahliges lineares Programm exakt gelöst werden. Aufgrund der $\mathcal{NP}$-Vollständigkeit des Problems gibt es zum anderen aber auch heuristische Lösungsverfahren, wie z.B. *Iterative Modulo Scheduling (IMS)* von Rau ([Rau94]), *Swing Modulo Scheduling (SMS)* ([LGAV96]), *Slack Modulo Scheduling (Slack)* ([Huf93]) und *Integrated Register-Sensitive Iterative Softwarepipelining (IRIS)* ([DRG98]). Iterative Modulo Scheduling ist detaillierter in Abschnitt 5.3.2 erläutert.
Ein weiterer recht komplizierter Ansatz für Softwarepipelining ist das sog. *Enhanced Pipeline Scheduling* von Allan ([AJLA95]). Dieses Verfahren basiert auf Perfect Pipelining, gehört aber nicht zu den Kernel Recognition Verfahren.

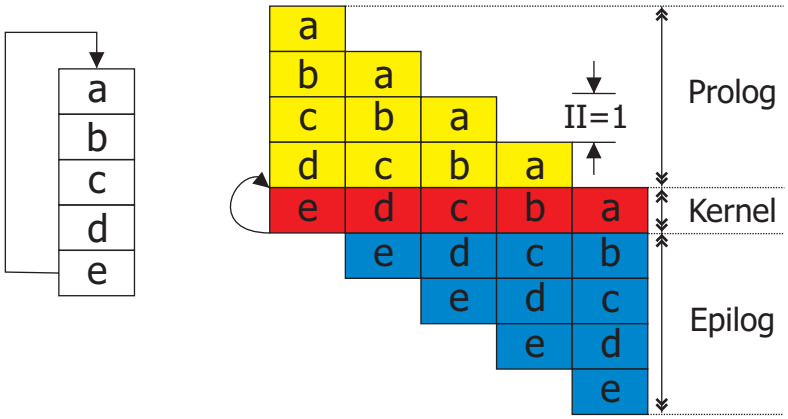
Im folgenden wird Modulo Scheduling näher erläutert, das als Grundlage dieser Arbeit dient.

5.3 Modulo Scheduling

5.3.1 Überblick

Wie bereits in Abschnitt 5.2 angeschnitten, versucht Modulo Scheduling eine überlappende Ausführung aufeinanderfolgender Iterationen einer Schleife zu erreichen, indem ein **Initiierungsintervall (II)** berechnet wird, das die Verzögerung zwischen

dem Start der Ausführung zweier aufeinanderfolgender Iterationen bestimmt. Aufbauend auf diesem Initiierungsintervall wird der eigentliche Schedulingprozess durchgeführt. Hier finden sich die größten Unterschiede in den verschiedenen Modulo Scheduling Techniken. Bei allen heuristischen Techniken muss der Fall betrachtet werden, dass eine Operation nicht mehr platziert werden kann aufgrund der bereits angeordneten Operationen und deren Abhängigkeiten. Dies kann man z.B. durch eine Form von Rücksprung, *Backtracking*, zu einem früheren Schedule-Zustand lösen oder indem man bestimmte bereits angeordnete Operationen wieder aus dem Schedule herausnimmt und später nochmal bearbeitet. Dadurch springt man quasi aus einer Sackgasse zu einem anderen Schedule-Zustand. Für letzteren Fall muss man Sorge tragen, dass das Verfahren terminiert, also nicht der Fall auftritt das zwei Operationen sich immer wieder gegenseitig aus dem partiellen Schedule entfernen.
Abbildung 5.2 zeigt beispielhaft die erzielte Überlappung durch Modulo Scheduling. Meist wird ein iterativer Ansatz verwendet, in dem für das Initiierungsintervall eine



(a) Schleife (b) Schleife nach Modulo Scheduling

Abbildung 5.2: Schleife nach Modulo Scheduling

untere Schranke berechnet und diese solange erhöht wird, bis ein legaler Schedule gefunden werden kann. Abbildung 5.3 zeigt in einem Flussdiagramm das allen Modulo Scheduling Techniken gemeinsame Rahmenwerk ihres Ablaufs. Hierbei liegt der Unterschied zwischen den verschiedenen Techniken in der Umsetzung der einzelnen Schritte.

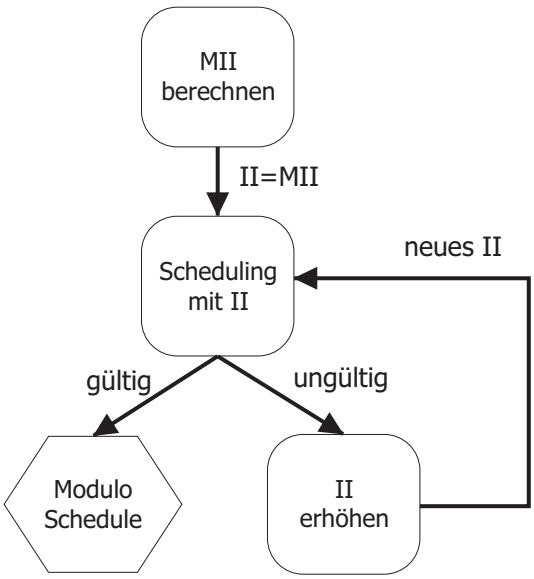


Abbildung 5.3: Rahmenwerk des Ablaufs von Modulo Scheduling

5.3.2 Iterative Modulo Scheduling

Iterative Modulo Scheduling wurde von Rau und Glaeser 1994 entwickelt ([Rau94]). Man könnte das Verfahren als *greedy* bezüglich der Ressourcen bezeichnen, da jeweils zum frühest möglichen Zeitpunkt möglichst viele Operation parallelisiert werden, ohne dabei bereits zu diesem Zeitpunkt zu beachten, ob dies zu einem späteren Zeitpunkt zu einem Konflikt führt. Die auftretenden Konflikte werden dann dadurch aufgelöst, dass Operationen wieder aus dem partiellen Schedule genommen werden. Außerdem wird ein iterativer Ansatz für den eigentlichen Scheduling-Prozess gewählt, indem eine untere Schranke für das Initiierungsintervall berechnet und diese bei Bedarf erhöht wird. Die Reihenfolge, in der die Operationen betrachtet werden, ist durch Benutzung einer Prioritätsfunktion definiert. Dadurch wird der Schedule in einem top-down Verfahren erzeugt.
Die Arbeitschritte sind die folgenden:

1. **IF-Conversion**
   Durch die *IF-Conversion* werden alle Kontrollabhängigkeiten in Datenabhängigkeiten verwandelt. Dazu wird jeder Operation ein Prädikat zugeordnet, das darüber entscheidet, ob sie überhaupt ausgeführt. Die vorkommenden technischen Umsetzungen (z.B. der Inhalt eines Registers interpretiert als Boole'scher Wert) solcher Prädikate finden sich in Abschnitt 5.5.1.
   Um die Kontrollabhängigkeiten durch Prädikate auszudrücken, nehmen die

Prädikate kontrolläquivalenter Operationen (vgl. Definition 2.16) die gleichen Werte an. Dadurch ist sichergestellt, dass die Prädikate genau die Kontrollabhängigkeiten ausdrücken.

2. **Berechnung des MII**
   Eine untere Schranke für das Initiierungsintervall wird in diesem Arbeitschritt berechnet. Sie basiert auf zwei Werten, dem sog. minimalen ressourcenbasierten Initiierungsintervall ($\mathbf{MII_{res}}$) und dem sog. minimalen datenabhängigkeitsbasierten Initiierungsintervall ($\mathbf{MII_{dep}}$). Ersteres ergibt sich daraus, dass die vorhandenen funktionalen Einheiten eine obere Schranke induzieren, die den möglichen Parallelität begrenzt. Das MII_{dep} wird aus den Datenabhängigkeiten[2] berechnet. Für die Berechnung selbst gibt es mehrere Möglichkeiten, die in Abschnitt 7.3.2 näher erläutert sind. Um diese beiden Werte konservativ zu vereinigen, berechnet sich das MII als das Maximum beider Werte:
   $$MII = \max\{MII_{res}, MII_{dep}\}$$

3. **Berechnung der Prioritäten aller Operationen**
   Die Reihenfolge, in der die Operationen angeordnet werden, wird durch die Vergabe von Prioritäten bestimmt. Als Gewichtungsfunktion wird hier eine leicht modifizierte Version der sog. **highest-level-first** Funktion aus dem List Scheduling verwendet. Die Modifikation ist in Abschnitt 7.4 genauer erklärt. Im Wesentlichen bekommt die Operation die höchste Priorität, die die größte Verzögerung[3] zum Ende der ursprünglichen Schleife hat.

4. **Berechnung des Flat Schedule**
   Auf Basis des II werden die Operationen in der durch die Gewichtungsfunktion festgelegten Reihenfolge angeordnet, wobei für jede Operation ein Zeitfenster in Zyklen, basierend auf den Datenabhängigkeiten zu den bereits angeordneten Operationen, berechnet wird. Innerhalb dieses Zeitfensters muss dann die Operation angeordnet werden. Dadurch wird die Eigenschaft erhalten, dass der partielle Schedule während seiner Erzeugung keine Datenabhängigkeiten zwischen den bereits angeordneten Operationen verletzt.
   Die Berechnung des Zeitfensters trägt dabei dem Initiierungsintervall Rechnung. Kann keine Instruktion innerhalb des Zeitfensters gefunden werden, in der die Operation ausgeführt werden kann, liegt ein Konflikt vor und es werden alle Operationen aus dem Schedule entfernt, die mit der aktuellen Operation in Konflikt stehen. Kann dadurch ein Konflikt nicht aufgelöst werden, so kann für das aktuelle II kein legaler Modulo Schedule berechnet werden und das II wird erhöht.

[2] samt den vorher durch die *IF-Conversion* umgewandelten Kontrollabhängigkeiten

[3] die Verzögerung errechnet sich aus den Datenabhängigkeiten auf dem längsten Pfad, dem so. kritischen Pfad

Um die Berechnungszeit zusätzlich in Schranken zu halten, existiert ein zusätzlicher Parameter, das sog. **Budget-Verhältnis**. Es ist definiert als das Verhältnis zwischen der Anzahl an maximal zulässigen Scheduleschritten und der Anzahl der Operationen der Schleife.
Der so berechnete sog. **Flat Schedule** entspricht also dem Ergebnis azyklischer Instruktionsanordnung mit der Ausnahme, dass das vorher berechnete Initiierungsintervall bereits berücksichtigt wird. Ebenso findet keine modulo Berechnung wie bei der Erzeugung des Kernels statt (vgl. folgender Abschnitt).

5. **Berechnung des Kernels**
   Ist der *Flat Schedule* (vgl. vorhergehender Abschnitt) berechnet, kann der Kernel daraus bestimmt werden, indem der zuvor berechnete Kontrollschritt modulo II gerechnet wird. Die Länge des Kernels ist hierbei gerade II Zyklen lang. Handelt es sich bei der Zielarchitektur um einen VLIW-Prozessor, so bedeutet das für den Kernel, dass er exakt II Instruktionen enthält.

6. **Modulo Variable Expansion**
   Der berechnete Kernel ist kein gültiger Code, wenn die maximale Lebensspanne einer Variablen im ursprünglichen Schleifenkörper die Länge[4] des Kernels überschreitet. Logischerweise überschreibt dann die verzahnte Ausführung einer neuen Iteration den noch benötigten Wert einer Variablen. Dies löst man durch Aufrollen des Kernels. Nun hat man mehrere Iterationen des Kernels hintereinandergeschaltet, sodass die Länge des neuen Kernels größer oder gleich der maximalen Lebensspanne dieser Variablen ist. Das alleine reicht noch nicht, denn die durch das Aufrollen erzeugten Kopien einer ursprünglichen Kerneliteration müssen nun ein andere Variable verwenden. Das Umbenennen der Variablennamen der erzeugten Kopien nennt man *Register-Renaming*. Modulo Variablen Expansion ist in [Lam88] kurz näher beschrieben.
   Nachteilig an diesem Schritt ist, dass die Anzahl der verwendeten Variablen steigt, damit auch die Anzahl der verwendeten Register im Programm. Man spricht in dem Fall auch davon, dass sich der Registerdruck erhöht. Modulo Variablen Expansion kann umgangen werden, wenn die Zielarchitektur rotierende Registerbänke unterstützt (vgl. Abschnitt 5.5.2).

7. **Erzeugung von Prolog und Epilog**
   Wie bereits in Abschnitt 5.1 und dort in Abbildung 5.1 gezeigt wurde, benötigt der neue Schleifenkörper zusätzlichen Code, um die Semantik des Eingabeprogrammes zu erhalten. Diese Codesequenzen können beide systematisch aus dem fertigen Kernel erzeugt werden, was in Abschnitt 7.7 und 7.8 detailliert erläutert wird.
   Nachteilig ist, dass durch Prolog und Epilog die Codegröße wächst. Um dies

[4] die Länge des Kernels ist die Anzahl seiner Instruktionen

zu verhindern, kann man sich prädikativer Ausführung (vgl. Abschnitt 5.5.1) bedienen, falls die Zielarchitektur dies unterstützt.

8. **Revertierung der IF-Conversion**
   Durch die *IF-Conversion* zu Anfang ist der Code frei von Kontrollabhängigkeiten, da sie alle in Datenabhängigkeiten transformiert wurden. Unterstützt die Zielarchitektur keine prädikative Ausführung, so muss dies nun wieder rückgängig gemacht werden. Ein Algorithmus hierfür ist in [WMHR93] gegeben.

9. **Vorschalten der Originalschleife**
   Da der erzeugte Kernel während einer Ausführung mehr als eine Iteration der ursprünglichen Schleife bearbeitet und pro Iteration des Kernels eine ursprüngliche Iteration beendet wird, ist es leicht einsichtig, dass man mit diesem Kernel nicht mehr ohne weiteres eine beliebige Anzahl von Iteration der ursprünglichen Schleife ausführen kann. Bearbeitet eine Iteration des Kernels n Iterationen der ursprünglichen Schleife, so können mit dem Kernel nur noch Vielfache von n Iterationen der ursprünglichen Schleife ausgeführt werden. Um dieses Problem zu lösen, muss man für die übrig bleibende Anzahl von Iterationen die ursprüngliche Schleife der optimierten Schleife vorschalten (vgl. auch Abschnitt 7.10).

Detailliertere Erläuterungen der einzelnen Arbeitschritte sowie die Beschreibung notwendiger Anpassungen für einen Postpass-Ansatz finden sich in Kapitel 7. Dort wird auch gezeigt, dass manche der einzelnen Arbeitsschritte bereits $\mathcal{NP}$-harte Probleme sind.

5.3.3 Weitere Modulo Scheduling Techniken

Neben Iterative Modulo Scheduling gibt es weitere Heuristiken, die Modulo Scheduling implementieren. Drei solcher werden im folgenden kurz mit ihren Unterschieden beschrieben.

Slack Modulo Scheduling Slack Modulo Scheduling (Slack) von Huff ([Huf93]) unterscheidet sich im wesentlichen dadurch von Iterative Modulo Scheduling, dass eine andere Form von Gewichtungsfunktion verwendet wird, welche hauptsächlich auf dem berechneten Zeitfenster einer Operation basiert. D.h. einer Operation, deren Zeitfenster größer ist als das Zeitfenster einer anderen Operation, wird eine höhere Priorität zugewiesen. Diese Gewichtungsfunktion ist dynamisch gegenüber der *highest-level-first* Priorität in Iterative Modulo Scheduling, da sie nach jeder Platzierung einer Operation neu berechnet werden müssen.
Außerdem wird kein top-down Ansatz verwendet, sondern es handelt sich vielmehr um

einen bidirektionale Strategie. Ebenso wie Iterative Modulo Scheduling ist der Algorithmus iterativ, d.h. kann eine Operation nicht angeordnet werden, werden andere bereits platzierte Operationen wieder aus dem partiellen Schedule entfernt.

Swing Modulo Scheduling Swing Modulo Scheduling (SMS) von Llosa et al. ([LGAV96]) vergibt die Prioritäten an die Operationen basierend auf MII_{dep} und einem Faktor, der mit einbezieht, wie kritisch der Pfad der Operation ist. Dadurch wird erreicht, dass die Sortierung der Operationen folgende Eigenschaft für fast alle Operationen aufweist: betrachtet man eine Operation innerhalb der Sortierung[5], so sind alle 'kleineren'[6] Operationen entweder nur Vorgänger oder nur Nachfolger der aktuell betrachteten Operation im Datenabhängigkeitsgraph.
Ein weiterer gravierender Unterschied ist, dass SMS kein *Backtracking* beinhaltet und daher die Berechnungskomplexität nicht so hoch ist als bei den anderen Verfahren.

Integrated Register-Sensitive Iterative Softwarepipelining *Integrated Register-Sensitive Iterative Softwarepipelining (IRIS)* von Dani et al. ([DRG98]) ist eine Erweiterung von Iterative Modulo Scheduling. Die Erweiterung besteht darin, dass einige Heuristiken aus dem *Stage Scheduling*[7] ([ED95]) direkt in das Verfahren integriert wurden anstelle einer Postpass-Anwendung derselben. Dies beeinflusst die Entscheidung, ob eine Operation so früh wie möglich oder so spät wie möglich angeordnet wird, mit dem Ziel, den Registerdruck zu minimieren.

Modulo Scheduling durch ganzzahlige lineare Programmierung Um exakte Ergebnisse zu berechnen, kann man sich auch der ganzzahligen linearen Programmierung (ILP) bedienen. Einzelne Teilschritte des Modulo Scheduling Rahmenwerkes können dabei als ILP-Problem formuliert werden. Dies sind als erstes die Berechnung des Initiierungsintervalls und anschließend der Scheduling-Prozess selbst.

5.3.4 Bewertung

In Studien wurden Iterative Modulo Scheduling und die in Abschnitt 5.3.3 beschriebenen Techniken bewertet und verglichen ([CLG02]).
Bezüglich der extrahierten Parallelität zeigte sich, dass alle Techniken sehr ähnliche und gute Ergebnisse liefern. Bei sehr komplexen Architekturen[8] extrahiert Iterati-

[5] implementiert in Form einer verketteten Liste
[6] bezüglich der Ordnung, auf der die Sortierung beruht
[7] Menge von Heuristiken, die versuchen, den Registerdruck zu verringern, der durch die Anwendung von Modulo Variablen Expansion entsteht.
[8] auch einfache Operationen werden in der Instruktionspipeline verzahnt, viele irreguläre Eigenschaften

ve Modulo Scheduling etwas mehr Parallelität auf Instruktionsebene als die anderen Techniken.
Betrachtet man den sog. Registerdruck, d.h. wie viele Register das Verfahren benötigt, so zeigt sich, dass Swing Modulo Scheduling und Slack Modulo Scheduling am wenigsten Register benötigen, wohingegen Iterative Modulo Scheduling und IRIS mehr Register benutzen.
Alle Techniken erzielen eine Leistungssteigerung der optimierten Schleifen, d.h. sie können die Ausführungszeit verkürzen. Hierbei produzieren Swing Modulo Scheduling und Slack Modulo Scheduling die besten Schedules. Iterative Modulo Scheduling und IRIS erreichen hier signifikant schlechtere Schedules.
Bezüglich der Berechnungszeit zeichnet sich Swing Modulo Scheduling als günstigstes Verfahren aus.

5.4 Optimierungen

Neben den eigentliche Verfahren des Softwarepipelining kann man den erzielten Schedule für Schleifen weiter optimieren. Zwei Möglichkeiten der Optimierung sollen nun hier kurz vorgestellt werden.

5.4.1 Geschachtelte Schleifen

Die meisten Softwarepipelining Verfahren legen ihren Fokus auf die Optimierung von nichtgeschachtelten Schleifen. Einige wenige Verfahren beschäftigen sich mit der Anordnung von perfekt geschachtelten Schleifen ([Ram94]). Das in Kapitel 7 beschriebene Iterative Modulo Scheduling auf Assemblerebene kann ebenfalls erweitert werden, um perfekt geschachtelte Schleifen anordnen zu können. Bislang wird hier nur die innerste Schleife behandelt.
Um perfekt geschachtelte Schleifen ebenfalls behandeln zu können, bedient man sich eines ähnlichen Prinzipes wie es Lam in [Lam88] vorstellt, der *Hierarchische Reduktion*. Dazu ordnet man die geschachtelten Schleifen ausgehend von der innersten zu der äußersten hin an. Zur Anordnung der innersten kann Iterative Modulo Scheduling wie oben beschrieben angewandt werden. Sei L eine Schleife mit n enthaltenen inneren Schleifen $l_1, \ldots, l_n$, auf die bereits Iterative Modulo Scheduling angewandt wurde. Um nun L anzuordnen, muss man den Datenabhängigkeitsgraphen anpassen. Jede innere Schleife $l_1, \ldots, l_n$ wird nun als einziger Knoten $l'_1, \ldots, l'_n$ dargestellt. Diese Knoten stellen insofern *Metaknoten* dar, da sie stellvertretend für alle Operationen der Schleife stehen. Eine Kante ausgehend von einer Operation i, die außerhalb $l_1, \ldots, l_n$ liegt, zu einer Operation j innerhalb l_i wird nun als Kante von i nach l'_i dargestellt. Damit ist es nun möglich, für alle i, die minimale Anzahl der Zyklen relativ zum

Ausführungszeitpunkt von i zu berechnen und MII_{dep} für L kann, wie bereits in Abschnitt 5.3.2 beschrieben, berechnet werden. Bei der Berechnung von MII_{res} muss nun noch beachtet werden, dass es eine *Metaoperation* für jede innere Schleife l_i gibt, deren Ausführungszeit gerade die Ausführungszeit der gesamten inneren Schleife ist. Damit kann die untere Schranke MII wie oben beschrieben als das Maximum von MII_{dep} und MII_{res} berechnet werden.
Auf Basis dieses II kann man nun Iterative Modulo Scheduling auf L anwenden, wobei hier ebenfalls noch beachtet werden muss, dass es die bereits erwähnten Metaoperationen gibt.
Dieses Vorgehen kann nun iterativ weiter angewendet werden, je nachdem wie groß die Schachtelungstiefe der Schleifen im Eingabeprogramm ist, d.h. das Verfahren unterliegt keiner oberen Schranke bezüglich der Schachtelungstiefe.

5.4.2 Verzahnung von Prolog und Epilog mit umgebendem Code

Die für jede Schleife erzeugten Codesequenzen für Prolog und Epilog (vgl. Abschnitt 5.3.2) können mit dem sie umgebendem azyklischen Code vor bzw. nach der Schleife verzahnt werden. Dies kann man durch die nachträgliche Anwendung eines lokalen Instruktionsanordnungsverfahrens, wie z.B. List Scheduling, auf die entsprechenden Codestücke erreichen. Damit lassen sich die optimierten Schleifen besser in den restlichen Code integrieren.

5.5 Hardwareunterstützung für Softwarepipelining

Softwarepipelining ist ein Verfahren, dessen Leistungspotential beträchtlich gesteigert werden kann, wenn die Hardware bereits bestimmte Eigenschaften, wie *prädikative Ausführung*, *rotierende Registerbänke* und *spekulative Ausführung* anbietet, obgleich die vorgestellten Verfahren auch auf Prozessoren angewendet werden können, die diese Eigenschaften nicht unterstützen. Die genannten Eigenschaften und ihre positiven Auswirkungen auf durch Softwarepipelining optimierte Schleifen werden nun hier vorgestellt. Sie sind heutzutage in vielen Prozessoren vorhanden, da sie natürlich nicht nur Vorteile für optimierte Schleifen bieten sondern insgesamt Performanzgewinne ermöglichen.

5.5.1 Prädikative Ausführung

Wenn man für eine Maschinenanweisung dynamisch zur Laufzeit entscheiden kann, ob sie überhaupt ausgeführt wird, so nennt man diese Entscheidungsmöglichkeit **Prädikative Ausführung**, wobei die Bedingung für die Ausführung als **Prädikat**

bezeichnet. Ist das Prädikat einer Operation zur Laufzeit nicht wahr, so hat dies den Effekt, als würde man eine `nop` ausführen.
Es gibt mehrere Arten der prädikativen Ausführung. Zum einen muss man unterscheiden, auf welcher Ebene die Eigenschaft aufgesetzt wurde, d.h. ob man für jede Operation oder nur für eine ganze Instruktion (im Falle eines VLIW-Prozessors) ein Prädikat angeben kann. Orthogonal zu dieser Unterscheidung gibt es verschiedene Umsetzungen.
Bei vielen Architekturen, wie etwa ARM7 ([ARM01]), ADSP-2106x SHARC ([Ana95]) oder TMS320C33 ([Tex04]), gibt es spezielle Systemregister, die den Bedingungscode[9] enthalten. Dieser wird von speziellen Operationen gesetzt und von anderen Operationen ausgelesen. Diese Art der prädikativen Ausführung hat den Nachteil, dass die Bedingung, die durch den Bedingungscode repräsentiert ist, nur solange verfügbar ist, bis die Ausführung einer Operation den Bedingungscode erneut setzt. Listing 5.1 zeigt am Beispiel der ARM7-Architektur die Benutzung von prädikativer Ausführung. Hier setzt die Vergleichsoperation in Zeile zwei die Bedingungsbits, von denen dann die Ausführung der Operationen in Zeile 3 und 4 abhängig ist. Konkret werden die beiden Operationen nur ausgeführt, wenn Register r5 den Inhalt 0 hat.
Eine andere Art der Umsetzung sieht man beim Philips TriMedia TM1000 (vgl. Abschnitt 3.2). Hier ist das Prädikat durch ein beliebiges Register umgesetzt, dessen Inhalt zur Laufzeit entscheidet, ob die Operation ausgeführt wird. Dies hat den Vorteil, dass man die Entscheidung über die Ausführung einer Operation prinzipiell über den ganzen Programmcode hinweg erhalten kann. Das macht die Codeerzeugung flexibler und übersichtlicher.

```
  LDR   R5, [SP, #0]
2 CMP   r5, #0
  MOVEQ R0, #1
4 BEQ   0x800
```

Listing 5.1: Prädikative Ausführung bei der ARM7-Architektur

Durch prädikative Ausführung wird es möglich, die Anzahl der notwendigen Verzweigungen im Programm zu verringern, was auch die Performanz erhöht. Außerdem werden durch die Anwendung von prädikativer Ausführung Kontrollabhängigkeiten in Datenabhängigkeiten transformiert, wodurch die Instruktionsanordnung wesentlich flexibler und ein höherer Grad an Parallelität extrahierbar wird. Die Transformation von Kontroll- in Datenabhängigkeit mittels prädikativer Ausführung nennt man **IF-Conversion** ([SS02]). Aus diesem Grund ist auch ein erster Schritt von Iterative

[9] üblicherweise ein Register, bei dem jedes/viele Bits für eine bestimmte Eigenschaft, z.B. overflow, Vergleichsergebnis, etc., stehen.

Modulo Scheduling (vgl. Abschnitt 5.3.2) die Anwendung von *IF-Conversion* auf den Eingabecode.
Die Vorteile von prädikativer Ausführung für die Anwendung von Softwarepipelining liegen darin, dass man die Codesequenzen für den Prolog und Epilog nicht erzeugen muss sondern nur zusätzlichen Code braucht, der möglichst geschickt entsprechende Prädikate ausnutzt. Dies ist möglich, da der Pro- bzw. Epilog aus den Operationen des Kernels entsteht und die Operationen also bereits vorhanden sind. Dies verringert das Anwachsen der Codegröße, macht auf der anderen Seite aber die Codeerzeugung wesentlich komplexer, da der Bedingungscode zusätzliche Datenabhängigkeiten einführt. Um die Eigenschaft der Generizität zu erhalten, wurde in dem in Kapitel 7 beschriebenen Verfahren für Iterative Modulo Scheduling auf Assemblerebene kein Gebrauch davon gemacht.
Wie bereits erwähnt, wird prädikative Ausführung bereits von vielen modernen Prozessoren unterstützt und mittlerweile als *State-of-the-Art* für die Eigenschaften von Instruktionssätzen angesehen.

5.5.2 Rotierende Registerbänke

Eine **rotierende Registerbank** ist eine Registerbank, auf deren Register indirekt über einen **Basiszeiger** zugegriffen werden kann. Dieser Basiszeiger kann verändert werden. Die indirekten Registerverweise werden dann modulo der Größe der Registerbank berechnet, wodurch die zyklische Eigenschaft zustande kommt. Damit kann man die Registerbank segmentieren und auf jedes Segment einzeln zugreifen. Die IA-64 Architektur ([Int01]) beispielsweise nutzt solche rotierende Registerbänke, um beim Aufruf von Routinen die lokalen Werte in einem eigenen Bereich zu halten.
Für Schleifen, die durch Softwarepipelining optimiert wurden, ergibt sich das Problem, das die maximale Lebenspanne einer Variablen größer sein kann als die Länge des Kernels. Man benötigt also den alten Wert einer Variablen, der aufgrund des geringen II bereits durch eine Operation der darauffolgenden Iteration überschrieben wurde. Um solche alten Werte zu retten, kann man rotierende Registerbänke benutzen, indem für jede Iteration der Basiszeiger verändert wird. Für ein Beispiel wird auf [Huf93] verwiesen. Sind rotierende Registerbänke vorhanden, kann auf die Modulo Variablen Expansion (vgl. Abschnitt 5.3.2) verzichtet werden und so das Anwachsen der Codegröße verringert werden.

5.5.3 Spekulative Ausführung

Bei modernen Prozessoren mit ihren Befehlsfließbänder (Instruktionspipelines), wie anfangs in Abschnitt 5.1 beschrieben, und der Möglichkeit, mehrere Operationen parallel auszuführen (VLIW-Prozessoren), kann der Fall auftreten, dass ein Sprung ausgeführt wird, dessen Ziel noch nicht berechnet wurde oder von dem noch nicht bekannt

ist, ob der Sprung überhaupt genommen wird. Dadurch ist nicht klar, welche Operationen als nächstes ausgeführt werden, wodurch die Instruktionspipeline zum Erliegen kommt. Das bedeutet einen Verlust von Performanz. Die Operationen beider Programmpfade bezeichnet man in diesem Fall als **spekulativ**.
Spekulative Ausführung bedeutet die Ausführung spekulativer Operationen. Das Ergebnis dieser Operationen wird aber noch nicht zurückgeschrieben, da ja noch nicht klar ist, ob sie überhaupt ausgeführt werden sollten. Dies geschieht, wenn diese Entscheidung getroffen ist. Wird eine solche Operation nicht ausgeführt, so wird das Ergebnis der Ausführung verworfen. Der Begriff der Ausführung bekommt auf diese Art eine neue Bedeutung.
Wie in Abschnitt 5.3.2 erwähnt kann man mit dem erzeugten Kernel keine beliebige Anzahl von Iterationen der ursprünglichen Schleife ausführen. Mit spekulativer Ausführung ist dies möglich, da bis zu dem Zeitpunkt, an dem die Schleifengrenze bekannt ist, Operationen spekulativ ausgeführt werden. Hat man dann nach Bekanntwerden der Schleifengrenze Operationen spekulativ ausgeführt, die man sonst nicht ausgeführt hätte, kann man ihr Ergebnis einfach verwerfen.

Kapitel 6

Retargierbare Postpassoptimierungen

6.1 Einleitung

Dieses Kapitel befasst sich mit der Konzeption und Durchführung von **retargierbaren Postpassoptimierungen**. Der Ausgangspunkt ist die Assemblerebene, eine sog. *low-level* Eingabe, anstelle der sonst üblichen Zwischendarstellung innerhalb eines Übersetzers.
Retargierbar bedeutet, dass der Kernalgorithmus generisch, d.h. prozessorunabhängig, ist. Für einen konkreten Prozessor sind Spezialisierungen erforderlich, die häufig generativ durchgeführt werden können. Die dazu notwendigen prozessorspezifischen Informationen werden einer Spezifikationsdatei entnommen.
Außerdem müssen aus einer solchen Prozessorspezifikation heraus Werkzeuge generiert werden, die die Assemblereingabe analysieren und in ein generisches Zwischenformat überführen, wobei gewisse Programmanalysen erforderlich sind wie z.B. die Rekonstruktion des Kontrollflussgraphen. Auf diesem Zwischenformat werden dann die Optimierungen durchgeführt.
In [Käs00a] wurde das retargierbare PROPAN-*Framework* entwickelt, das genau dieser Konzeption entspricht und auch erfolgreich industriell eingesetzt werden konnte. PROPAN steht für *Postpass-oriented Retargetable Optimizer and Analyzer* und ist ein Framework, mit dessen Hilfe generische Optimierungen und Analysen definiert werden können. Die Besonderheit liegt hier zum einen in der Generizität und zum anderen im Postpass-Ansatz, d.h. die Eingabe ist ein Programm auf Assemblerebene.
Zur Spezifikation des Zielprozessors wird die Sprache TDL verwendet (vgl. Abschnitt 6.2). PROPAN stellt hierfür einen TDL-Parser zur Verfügung, der aus der Prozessorbeschreibung einen Assemblerparser für die Eingabeprogramme generiert. Außerdem werden aus den in der TDL-Beschreibung vorhandenen prozessorspezifischen Informationen bezüglich seiner Ressourcen, seines Instruktionssatzes und irregulärer Hardwareeigenschaften spezifische Datenstrukturen in ANSI-C generiert, die dann den generischen Optimierungen und Analysen zur Verfügung stehen.
Der generierte Assemblerparser transformiert das Eingabeprogramm in einen generischen annotierten Kontrollflussgraph unter Verwendung von CRL (vgl. Abschnitt 6.3). Aus diesem werden für die Optimierungen und Analysen notwendige Programmdarstellungen, Datenabhängigkeitsgraph, Kontrollabhängigkeitsgraph, Schleifenerkennung, etc., erzeugt. Auf diesem Kontrollflussgraph samt den dafür erzeugten Programmdarstellungen beginnt die Optimierungs- bzw. Analysephase. Diese verändert bzw. annotiert die CRL-Beschreibung des Programms.
Bei Optimierungen kann anschließend aus dem modifizierten Kontrollflussgraph mittels einer Assemblerrekonstruktion die optimierte Assemblerdatei erzeugt werden. Parallel dazu existiert eine Schnittstelle zum graphischen Visualisierungswerkzeug aiSee [Abs05], um die generierten Programmdarstellungen visualisieren zu können.
In [Käs00a] finden sich weitere detailliertere Informationen über PROPAN und wie man mithilfe des PROPAN-Systems globale Instruktionsanordnung auf der Basis von

ganzzahliger linearer Programmierung durchführt. Abbildung 6.1 zeigt schematisch die Struktur des PROPAN-Frameworks.

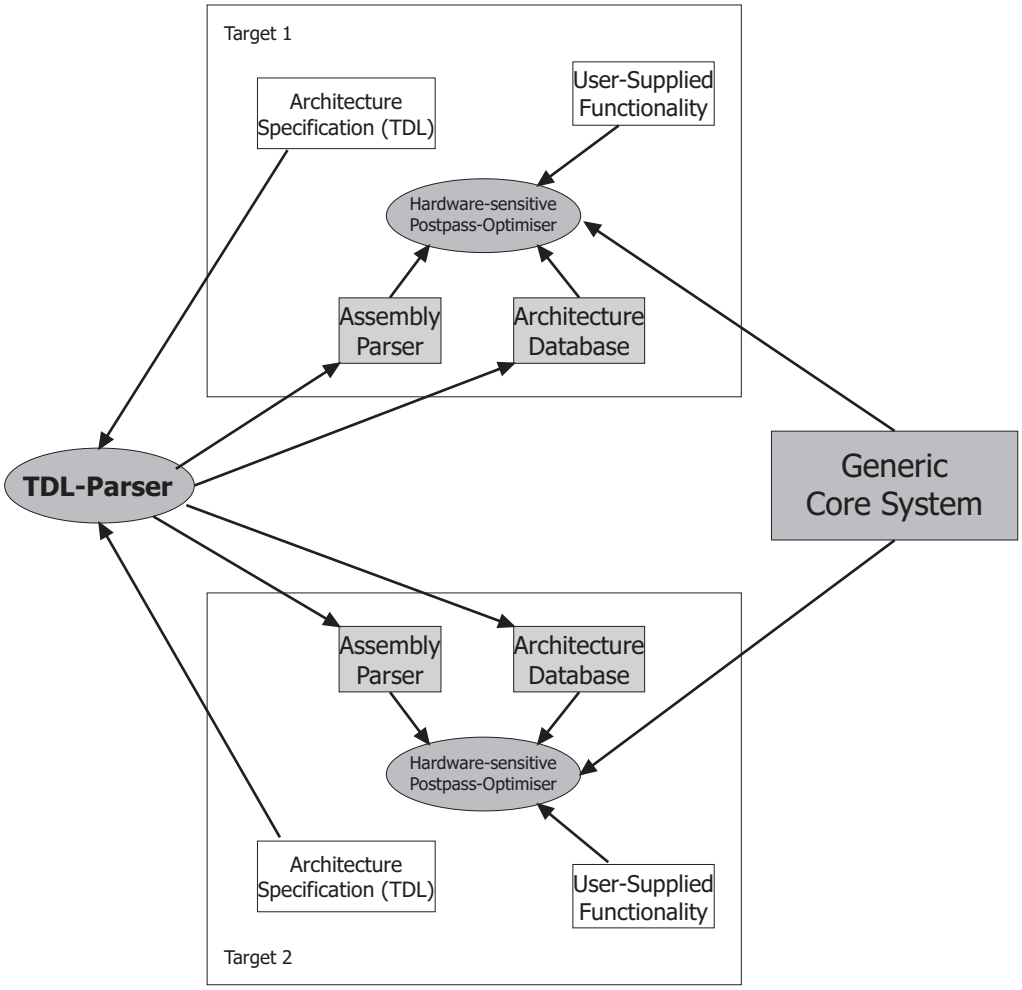


Abbildung 6.1: Struktur des PROPAN-Systems

In der vorliegenden Arbeit wurde das PROPAN-Framework als Ausgangsbasis benutzt, einen generischen Softwarepipelining-Algorithmus als Postpassoptimierung zu definieren und in das System zu integrieren. Das Algorithmus selbst wird in Kapitel 7 vorgestellt und erläutert, die Umsetzung der Integration ist in Kapitel 8 beschrieben. Der folgende Abschnitt beschreibt die Hardwarespezifikationssprache TDL, mit dessen Hilfe Prozessorspezifikationen für das PROPAN-Framework durchgeführt werden können. Darauf folgt in Abschnitt 6.3 eine Beschreibung der im PROPAN-System benutzen Zwischendarstellung CRL.

6.2 Target Description Language

Die *Target Description Language (*TDL*)* ist eine Beschreibungsprache, die es erlaubt, die Ressourcen und Assemblersprache eines Prozessors präzise zu spezifizieren. In einer TDL-Spezifikation sind außerdem alle Informationen enthalten, die zur Analyse und Optimierung von Programmen für diesen Prozessor notwendig sind. Dazu gehören die Eigenschaften von Ressourcen, Syntax und Semantik der Maschinenoperationen sowie zusätzliche Informationen über das zeitliche Verhalten von Operationen. Um einen großen Bereich von Prozessoren beschreiben zu können und viele verschiedene Optimierungen und Analysen zu unterstützen, ist TDL sehr einfach erweiterbar und flexibel gehalten.
TDL benutzt sowohl strukturelle als auch operationale Beschreibungsformalismen. So werden die Ressourcen in einem strukturellen Stil deklariert und beschrieben, wohingegen die Operationen mehr aus der Sicht ihres Verhaltens spezifiziert werden, d.h. man modelliert die Semantik einer Operation.
Das Beschreibungskonzept in TDL ist sehr modular; eine Spezifikation eines Prozessors ist in vier Abschnitte aufgeteilt. Eine **Ressourcenspezifikation**, in der die Ressourcen deklariert werden können, eine **Spezifikation des Instruktionssatzes**, einen Abschnitt zur **Spezifikation von irregulären Hardwareeigenschaften** und einen Abschnitt zur **Spezifikation von Assemblereigenschaften** des Prozessors. Diese Abschnitte sind im folgenden kurz näher erläutert. Eine detaillierte Erläuterung von TDL findet sich in [Käs03] und [Käs00b].

6.2.1 Ressourcenspezifikation

In der Ressourcenspezifikation einer TDL-Beschreibung werden die für hardwarespezifische Optimierungen und Analysen notwendigen Ressourcen deklariert. Dazu gehören z.B. die funktionalen Einheiten des Prozessors, eine Beschreibung seiner Registersätze, der Caches und des Speichers. Die speziellen Eigenschaften der einzelnen Ressourcen können durch einen erweiterbaren Attributmechanismus dargestellt werden. TDL bietet eine Menge von vordefinierten Ressourcentypen an, deren Eigenschaften durch eine Menge von vordefinierten Attributen beschrieben werden können. Diese Typen umfassen funktionale Einheiten (`FuncUnit`), Caches (`Cache`), Registerbänke (`Register`) und den Speicher (`Memory`). Hier sind Attribute verfügbar, um z.B. die Breite von Registern, die Größe des Speichers, Speicherzugriffsbreiten und Alignmentbedingungen anzugeben.
TDL unterliegt der Annahme, dass alle deklarierten Ressourcen unterschiedlichen Typs parallel zueinander arbeiten können. Das gilt auch für alle Instanzen eines Typs. Dadurch kann die in Abschnitt 3 vorgestellte VLIW-Architektur unterstützt werden. Architekturen ohne Parallelität auf Instruktionsebene können dabei als Sonderfall der VLIW-Architektur aufgefasst werden, in dem nur eine Ausführungseinheit vorhanden

```
Resources-Section
...
FuncUnit ALU 5;
Register gpr "r%d" [0:127] size=32, type=signed<32>;
SetProperties gpr[0] value=0x00000000;
SetProperties gpr[1] value=0x00000001;
Memory MEM "Mem" access=8, align=8, type=mixed;
Cache DataCache assoc=8, size=256, linesize=64, type=data;
Cache InstrCache assoc=8, size=512, linesize=64, type=instr;
...
```

Listing 6.1: Ressourcenspezifikation in TDL

ist.
Weiter kann der Entwickler einer TDL-Beschreibung eines Prozessors benutzerdefinierte Ressourcen und Attribute anlegen, um den Bereich der deklarierbaren Ressourcen beliebig zu erweitern.
Listing 6.1 zeigt beispielhaft einen Auszug aus der TDL-Beschreibung für den Philips TriMedia TM1000-Prozessor. Die Spezifikation der Ressourcen in TDL wird durch das Schlüsselwort `Resources-Section` eingeleitet. In Zeile 3 des Listings wird z.B. eine funktionale Einheit mit dem eindeutigen Namen ALU deklariert, von der genau fünf Instanzen mit gleichen Eigenschaften zur Verfügung stehen. Anschließend werden 128 Allzweckregister mit ihrer Assemblerrepräsentation definiert, die alle 32 Bit breit sind. Durch das Schlüsselwort `SetProperties` kann man bestimmte Eigenschaften einer Ressource setzen, wie z.B., dass der Inhalt von Register `r0` immer Null und der von Register `r1` immer eins ist. Für die genauere Erklärung der Bedeutung einzelner Attribute wird auf [Käs00b] verwiesen.

6.2.2 Spezifikation des Instruktionssatzes

Die Spezifikation des Instruktionssatzes eines Prozessors ist der zentrale Teil einer TDL-Spezifikation. Hier wird die Assemblerrepräsentation einer jeder Maschinenoperation definiert. Außerdem werden das Zeitverhalten und die Semantik für jede Operation beschrieben, d.h. welche Ressource zu welchem Zeitpunkt ihrer Ausführung von der Operation benutzt wird.
Die Definition des Instruktionssatzes erfolgt über eine attributierte Grammatik. Für eine detaillierte Erklärung der fundamentalen Konzepte attributierter Grammatiken wird auf [WM97] verwiesen. Die Terminale dieser Grammatik stellen die Maschinenoperationen des Instruktionssatzes dar.
Wie in der Spezifikation der Ressourcen bietet TDL auch hier eine Menge von vordefi-

```
InstructionSet-Section
...
DefineOp opIMUL "%!(optguard) imul %s %s \[->\] %s"
    {src1="$2" in {gpr}, src2="$3" in {gpr}, dst1="$4" in {gpr
       }},
    {ISlot2(exectime=3, latency=1, slots=0)
     |ISlot3(exectime=3, latency=1, slots=0); WbBus},
    {if (guard==true) {if (src3<0>==0b1) { signed<64> temp;
     temp:=_sext(src1,64) * _sext(src2,64); dst1:=_bext(temp
        ,31,0);} } };
...
```

Listing 6.2: Spezifikation des Instruktionssatzes in TDL

nierten Attributen an, mit denen das Zeitverhalten der Operationen definiert werden kann. Das Attribut `execution time` beschreibt z.B. die Anzahl an Kontrollschritten, die für die Ausführung einer Operation benötigt werden.
Die Semantik einer Operation wird durch Benutzung einer Registertransfersprache erledigt ([Käs00b]). Für die Operanden gibt es vordefinierte Attribute, wie z.B. `src1` oder `dst1` für Quell- und Zieloperanden.
Listing 6.2 zeigt am Beispiel der TDL-Beschreibung für den Philips TriMedia TM1000-Prozessor die Spezifikation einer Multiplikationsoperation. Die Spezifikation des Instruktionssatzes wird durch das Schlüsselwort `InstructionSet-Section` und die Definition einer Maschinenoperation durch das Schlüsselwort `DefineOp` eingeleitet Nichterminale, die keine eigenständigen Symbole sind, werden durch das Schlüsselwort `OpNT` deklariert. Jede Operation und jedes Nichterminal erhält einen eindeutigen Namen und eine Assemblerrepräsentation. Anschliessend werden die Eigenschaften der Operation in drei Attributgruppen definiert.
In der ersten Attributgruppe werden die Quell- und Zieloperanden deklariert. Die jeweilige Position jedes Operanden in der Assemblerrepräsentation wird dabei angegeben. So bedeutet `dst1=$4 in {gpr}`, das der vierte Platzhalter in der Assemblerrepräsentation ein Zieloperand und ein Register in der Menge `gpr` ist.
In der zweiten Attributgruppe wird die Ausführung der Instruktion samt ihrem Zeitverhalten durch einen Belegungsplanmechanismus beschrieben. Hierbei werden alle Alternativen an Ressourcen angegeben, auf der die Operation ausgeführt werden kann, wobei die Ressourcen durch `|` getrennt werden. Für jede Ressource wird das Zeitverhalten durch die drei Attribute `exectime`, `latency` und `slots` definiert. Das Attribut `exectime` bestimmt dabei die Ausführungszeit der Operation, `latency` das notwendige Zeitintervall zur nächsten Dateneingabe an diese Ressource und `slots` definiert die Anzahl an sog. Delay Slots für die Ausführung dieser Operation auf dieser Ressource.
Die letzte Attributgruppe beschreibt die Semantik der Operation in C-ähnlicher Syn-

```
Constraints-Section
...
op1 in {DMemspecClass} & op2 in {DMemClass}: !(op1 && op2);
...
```

Listing 6.3: Spezifikation irregulärer Hardwareeigenschaften in TDL

tax. Die Operation `opIMUL` in Listing 6.2 beispielsweise bewirkt nur dann eine Zustandsänderung des Prozessors, wenn ihr Prädikat (`guard`) wahr ist. In das Zielregister (`dst1`) wird dann das Ergebnis der Multiplikation der Quellregister (`src1`,`src2`) zusammen mit einigen Sign- und Bitextensions (`_sext`,`_bext`) geschrieben.

6.2.3 Spezifikation irregulärer Hardwareeigenschaften

Das Ausführungsmodell von TDL unterliegt der Annahme, dass alle funktionalen Einheiten parallel zueinander arbeiten können. Leider gibt es irreguläre Hardwarearchitekturen, die diese Parallelität einschränken. Diese können durch Ressourcen- oder Kodierungsbedingungen entstehen.
Solche irregulären Hardwareeigenschaften werden in TDL durch eine Menge von Regeln ausgedrückt, die jeweils aus Prämisse und Folge bestehen. Die Prämisse ist ein Boole'scher Ausdruck, der die Vorraussetzung für die Folge darstellt und die Eigenschaften von Operationen invariant gegenüber Scheduling und Registerallokation beschreibt. Sie muss statisch evaluierbar sein. Die Folge selbst ist auch ein Boole'scher Ausdruck, der während der Optimierung dynamisch ausgewertet wird.
Listing 6.3 zeigt eine solche irreguläre Eigenschaft am Beispiel des TriMedia TM1000-Prozessors. Die Spezifikation von irregulären Hardwareeigenschaften wird durch das Schlüsselwort `Constraints-Section` eingeleitet. In Listing 6.3 ist eine Regel gezeigt, die die parallele Ausführung von Operationen aus der Klasse `DMemClass` mit Operationen aus der Klasse `DMemClass` verbietet.

6.2.4 Spezifikation von Assemblereigenschaften

Die Spezifikation von Assemblereigenschaften umfasst syntaktische Details der Assemblersprache des Zielprozessors. Hier werden z.B. die Trennzeichen für Operationen und Instruktionen definiert. Außerdem wird hier die Syntax von Assemblerdirektiven festgelegt.
Listing 6.4 zeigt einen Ausschnitt der Assemblerspezifikation für den TriMedia TM1000-Prozessor. Hier wird das Zeichen , als Operationstrennzeichen (`OperationDelimiter`) und das Zeichen ; als Trennzeichen zwischen Instruktionen

(`InstructionDelimiter`) verwendet. Kommentare (`Comments`) werden durch die Zeichenfolge `(*` begonnen und durch die Zeichenfolge `*)` wieder beendet. Ausserdem werden einige Assemblerdirektiven spezifiziert. So definiert z.B. Zeile 6 einen Direktive mit der Assemblerrepräsentation `.data`, die den Start des Segments `data` darstellt.

```
Assembly-Section
...
OperationDelimiter ",";
InstructionDelimiter ";";
Comment ("(*", "*)");
DefineDirective DirData  "\[.data\]"  {type=SegStart, name="
   data"};
DefineDirective DirData1 "\[.data1\]" {type=SegStart, name="
   data1"};
DefineDirective DirText  "\[.text\]"  {type=SegStart, name="
   text"};
...
```

Listing 6.4: Spezifikation von Assemblereigenschaften in TDL

Die Spezifikation von Assemblereigenschaften wird durch das Schlüsselwort `Assembly-Section` eingeleitet.

6.3 Controlflow Representation Language (CRL)

Wie in Abschnitt 6.1 erwähnt, wird die Assemblereingabe in eine Zwischendarstellung überführt, auf der dann generische Optimierungen und Analysen durchgeführt werden können. Das PROPAN-Framework benutzt die *Controlflow Representation Language (CRL)* als Zwischendarstellung.
CRL beschreibt den Kontrollfluss eines Programmes hierarchisch gegliedert nach Routinen, Basisblöcken und Instruktionen. Basisblöcke sind in Routinen enthalten und Instruktionen in den Basisblöcken.
Kontrollfluss wird in CRL durch Kanten zwischen Basisblöcken dargestellt. Wird durch eine Verzweigung in einen anderen Basisblock gewechselt, so wird eine Kante ausgehend vom aktuellen Basisblock zum Zielblock definiert. Prozeduraufrufe werden durch Kanten zwischen Routinen dargestellt. Ruft also eine Instruktion eine Prozedur auf, so wird eine Kante ausgehend von der aktuellen Routine zur aufgerufenen Routine definiert.
CRL unterscheidet nicht zwischen Instruktion und Operation. Deshalb gibt es innerhalb von Basisblöcken nur noch Operationen. Um eine VLIW-Instruktion darzustellen, muss man mehrere Operationen definieren, die die gleiche Adresse haben. Eine Operation selbst besteht aus einer Zeile mit folgenden Angaben:

- Adresse der Instruktion,
- Assemblerrepräsentation,
- eine kommaseparierte Auflistung von Attribut-Wert-Paaren, um spezifische Eigenschaften der Operation anzugeben. In diesem Anwendungsfall entsprechen die Attribute denen der TDL-Beschreibung dieser Operation, also z.B. Angaben zu den verwendeten Operanden und ähnliches.

Listing 6.5 zeigt ein Beispiel für eine CRL-Beschreibung eines Programmes. Die Startroutine des Programmes ist Zeile 1 deklariert. Das Programm enthält nur eine Routine namens `puremat1x3`. Für eine Routine ist dann der Startbasisblock durch das vorangestellte Schlüsselwort `entry` definiert. Für einen Basisblock kann durch das Schlüsselwort `edges to` und eine anschließende Auflistung der Namen der Zielblöcke Kontrollfluss definiert werden, der von diesem Basisblock ausgeht. In dem Beispiel gibt es von Block `bmain_DT_0` nur eine ausgehende Kontrollflusskante, nämlich die zu Block `bmain_DT_1`. Durch das `contains`-Schlüsselwort beginnt dann eine Auflistung der Operationen eines Basisblockes. Für das Ende des Programmes gibt es einen speziellen Basisblock, nämlich den sog. *exit*-Block. Dieser wird durch das Schlüsselwort `exit` markiert.
Weitere Informationen über CRL finden sich in [Lan98].

```
start with puremat1x3;

routine puremat1x3: ismain="true"
{
  entry bmain_DT_0 {
   edges to bmain_DT_1;
   contains {
        0x0:0 "if r1 uimm ( __DATA_SECTION + 48 ) -> r34":
           label="__main_DT_0,_main", src3="r1", genname="
           opUIMM", dst1="r34", noreorder="0", mode="mDefault",
            type="otDefault", guard="gTrue", src1="
           __DATA_SECTION + 48";
        0x0:0 "if r1 ijmpi ( __main_DT_1 )": src3="r1", genname
           ="opIJMPI", target="__main_DT_1", dst1="dpc",
           noreorder="0", mode="mDefault", type="otBranch",
           guard="gTrue";
        ...
   }
  }
  block bmain_DT_1 {
   edges to bmain_DT_1, END;
   contains {
        0x10:0 "if r1 iaddi ( 1 ) r33 -> r39": src2="r33",
           label="__main_DT_1", src3="r1", genname="opIADDI",
           dst1="r39", noreorder="0", mode="mDefault", type="
           otDefault", guard="gTrue", src1="1";
        0x10:0 "if r1 uimm ( __DATA_SECTION ) -> r6": src3="r1
           ", genname="opUIMM", dst1="r6", noreorder="0", mode
           ="mDefault", type="otDefault", guard="gTrue", src1="
           __DATA_SECTION";
        ...
   }
  }
  exit END;
}
```

Listing 6.5: Beispiel CRL-Beschreibung

Kapitel 7

Iterative Modulo Scheduling auf Assemblerebene

Dieses Kapitel beschreibt detailliert die Anwendung von Iterative Modulo Scheduling (vgl. Abschnitt 5.3) auf die Schleifen von Assemblerprogrammen. Die *low-level* Eingabe unterscheidet sich hierbei von der üblicherweise verwendeten *high-level* Zwischendarstellung innerhalb eines Übersetzers. Dies macht einige Anpassungen und zusätzliche Berechnungen nötig, wie etwa die Rekonstruktion des Kontrollflussgraphen aus dem Eingabeprogramm.
Wie bereits in Abschnitt 5.3 erläutert, stellt Modulo Scheduling ein Rahmenwerk dar, um Softwarepipelining umzusetzen. Um eine der weiteren Heuristiken aus Abschnitt 5.3.3 zu implementieren, kann man viele der hier beschriebenen Arbeitsschritte wiederverwenden.
Die Abschnitte 7.1 bis 7.12 beschreiben die Arbeitsschritte des Verfahrens. Die Anwendung von Iterative Modulo Scheduling auf Assemblerebene resultiert in einigen spezifischen Eigenschaften, die in Abschnitt 7.13 beschrieben werden.
Die Umsetzung des in diesem Kapitel vorgestellten Modulo Scheduling Technik wurde als Postpass-Optimierung (vgl. Kapitel 6) im Rahmen dieser Arbeit implementiert und in das PROPAN-System ([Käs00a]) integriert. Die Implementierung ist in Kapitel 8 beschrieben.

7.1 Vorberechnungen

Aufgrund der *low-level* Eingabe und der Umsetzung von Modulo Scheduling als Postpass-Ansatz, stehen die normalerweise im Backend[1] vorhandenen Programmdaten (wie Kontrollflussgraph, Datenabhängigkeitsgraph, etc.) nicht direkt zur Verfügung. Um Modulo Scheduling anwenden zu können, müssen daher zuerst diese Daten aus der Assemblereingabe rekonstruiert werden. Diese Vorberechnungen können mittels des PROPAN-Systems generisch auf Basis der Beschreibungssprache TDL erledigt werden (vgl Kapitel 6).
Die notwendigen Vorberechnungen bauen zum Teil aufeinander auf und sind in den folgenden Abschnitten in ihrer Ausführungsreihenfolge aufgeführt.

7.1.1 Parsen der Assemblerdatei

Als ein erster Schritt muss die Assemblereingabe bezüglich der in ihr enthaltenen Instruktionen respektive Operationen analysiert werden. Dazu benötigt man einen Parser, der spezifisch für den Instruktionssatz der Zielarchitektur die Eingabe einliest und damit in geeignete interne Datenstrukturen überführt. Man erhält dadurch eine Liste der Operationen in der Reihenfolge, wie sie in der Eingabedatei enthalten sind. Um generisch bezüglich der Zielarchitektur zu bleiben, muss der notwendige Parser

[1] das Backend ist der für die Codeerzeugung verantwortliche Teil des Übersetzers

automatisch generiert werden. Die geschieht auf der in Abschnitt 6.2 vorgestellten Beschreibungssprache TDL.
Zur Berechnung der exakten Reihenfolge der Instruktionen/Operationen, muss in einem nächsten Schritt der Kontrollfluss des Eingabeprogramms rekonstruiert werden.

7.1.2 Rekonstruktion des Kontrollflussgraphen

Der Kontrollflussgraph stellt die Basis für fluss-sensitive Analysen und Optimierungen dar und wird zur Berechnung des Datenabhängigkeitsgraphen benötigt. Bei der Anwendung von Iterative Modulo Scheduling innerhalb eines Übersetzers für eine Hochsprache ist der Kontrollfluss meist explizit im Eingabeprogramm enthalten. Aufgrund der Anwendung des Verfahrens auf der Assemblerebene, muss der Kontrollflussgraph zuerst rekonstruiert werden, da Assembler-Code diesen nicht mehr explizit sondern durch die Übersetzung in Verzweigungsoperationen nur noch implizit enthält.
Um den Kontrollfluss eines Assemblerprogrammes zu rekonstruieren, muss man schrittweise den Aufrufgraphen[2] des Assemblerprogrammes und für dessen Routinen den Basisblockgraphen rekonstruieren. Ausgangspunkt dieser Analysen ist die Ausgabe des Assemblerparsers. Für die Ausgabe des berechneten Kontrollflussgraphen wird die in Abschnitt 6.3 vorgestellte Sprache CRL benutzt.
Die Rekonstruktion des Kontrollflussgraphen ist unter Umständen nicht exakt möglich. In diesem Fall wird eine konservative Approximation an den exakten Kontrollfluss berechnet. Für sehr detaillierte Informationen zur Kontrollflussrekonstruktion wird auf [The00] verwiesen. Generische Algorithmen dazu finden sich in [Wil01].

7.1.3 Rekonstruktion des Datenabhängigkeitsgraphen

Der Datenabhängigkeitsgraph kann auf der Grundlage des berechneten Kontrollflussgraphen (siehe vorhergehenden Abschnitt) rekonstruiert werden. Um globale Instruktionsanordnung zu ermöglichen, benötigt man den verallgemeinerten Datenabhängigkeitsgraphen (siehe Abschnitt 2.2). Um diesen zu berechnen, bedient man sich eines zweistufigen Verfahrens, das in entgegengesetzter Richtung des Kontrollflusses angewendet wird:

1. Erfassung der Ressourcenzugriffe für einen Basisblock b, die eine Datenabhängigkeit induzierten können und

2. Erfassung der tatsächlichen Datenabhängigkeiten für b.

[2] der Aufrufgraph eines Assemblerprogrammes enthält seine aufgerufenen Routinen ausgehend von der Startroutine

Der erste Schritt arbeitet auf Basisblöcken und basiert auf zwei rückwärts-gerichteten Datenflussanalysen: die *Analyse der exponierten Setzungen* und die *Analyse der aktiven Benutzungen*. Erstere ermittelt in einem Basisblock für eine Variable v die Menge aller Definitionen von v, die auf einem Programmpfad ausgehend von einem Programmpunkt p, der setzungsfrei bezüglich v ist, erreicht werden können. Die *Analyse der aktiven Benutzungen* berechnet für eine Variable v die Menge aller Benutzungen von v, die aufwärts exponiert sind, d.h. die auf einem Pfad ausgehend von einem Programmpunkt p, der benutzungsfrei bezüglich v ist, erreicht werden können. Die Idee solcher Datenflussanalysen ist es, Informationen von verschiedenen Kontrollflusspfaden an Verschmelzungspunkten zu vereinigen. Dadurch kann die exponentielle Berechnungszeit, die für die Iteration aller möglichen Pfade notwendig wäre, umgangen werden. Für detaillierte Informationen zu Datenflussanalysen wird auf [WM97, NNH99] verwiesen. Im ersten Schritt zur Rekonstruktion der Datenabhängigkeiten wird also für jeden Basisblock die Mengen der exponierten Setzungen und der aktiven Benutzungen am Ende desselben berechnet. Diese Information dient als Eingabe für Schritt zwei.
Der zweite Schritt stellt eine erweitere Version des Algorithmus in [WM97] zur Berechnung der Datenabhängigkeiten dar. Hier wird instruktionsweise auf den Basisblöcken gearbeitet, die mit den Informationen aus Schritt eins annotiert sind. Für jeden Basisblock wandert der Algorithmus über alle Instruktionen des Blockes in umgekehrter Reihenfolge ihrer Ausführung. Für jede Instruktion betrachtet man alle in ihr enthaltenen Operationen, indem man basierend auf den Informationen aus Schritt eins die Datenabhängigkeiten dieser Operationen erfasst und registriert.
Detaillierte Erläuterungen zur Rekonstruktion des Datenabhängigkeitsgraphen auf Basis des Kontrollflussgraphen finden sich in [Käs00a].

7.1.4 Rekonstruktion des Kontrollabhängigkeitsgraphen

Der Kontrollabhängigkeitsgraph (siehe Definition 2.15) ist notwendig, um sicherzustellen, dass globale Codeverschiebungen zwischen Basisblöcken semantikerhaltend sind (siehe auch Abschnitt 2.2).
Der Algorithmus berechnet zuerst für alle Operationen die Menge ihrer Postdominatoren auf Basis dessen dann die Kontrollabhängigkeiten berechnet werden können. Eine genaue Beschreibung des Algorithmus findet sich in [Käs00a].

7.1.5 Schleifenerkennung

Nachdem der Kontrollflussgraph rekonstruiert wurde, müssen nun alle Schleifen des Eingabeprogrammes identifiziert werden. Dies geschieht durch eine Tiefensuche nach rückwärtsgerichteten Kanten im Kontrollflussgraphen. Postdominiert der Zielknoten

einer rückwärts-gerichteten Kante den Quellknoten und der Zielknoten ist noch nicht als Schleifenknoten markiert, so hat man eine neue Schleife gefunden.
Dieses Vorgehen geht allerdings davon aus, dass alle Schleifen eindeutige Eintrittspunkte haben. Schleifen mit mehreren Eintrittspunkten werden dadurch nicht korrekt erkannt und müssen durch Benutzereingaben identifiziert werden. Assemblerbefehle, die durch einen Übersetzer erzeugt wurden, beinhalten zumeist nur Schleifen mit eindeutigen Eintrittspunkten, sodass diese Einschränkung akzeptabel ist.

7.1.6 Transformation der Kontrollabhängigkeiten

Das Erzeugen des neuen Schleifenkörpers durch Iterative Modulo Scheduling basiert auf Delays, die aus dem Datenabhängigkeitsgraphen berechnet werden. Wie bereits in Abschnitt 2.2 erwähnt, reicht die Betrachtung der Datenabhängigkeiten eines Programmes nicht für globale Instruktionsanordnung aus. Man muss zusätzlich noch die Kontrollabhängigkeiten berücksichtigen.
Um mittels Iterative Modulo Scheduling globale Instruktionsanordnung durchführen zu können, werden die Kontrollabhängigkeiten in Datenabhängigkeiten umgewandelt, sodass durch die Betrachtung letzterer globale Codeverschiebungen alleine auf der Basis des DDG möglich werden. Dazu wird der vorliegende Assemblercode in die sog. **prädikative Form** (siehe Abschnitt 5.5.1) gebracht. Das bedeutet, dass jeder Operation ein Prädikat zugeordnet wird, das darüber entscheidet, ob die Operation überhaupt ausgeführt wird. Hierbei müssen die Prädikate kontrolläquivalenter Operationen den gleichen Wahrheitswert annehmen. Weiter braucht man nun noch Operationen, die die jeweiligen Prädikate berechnen und dadurch entstehen Datenabhängigkeiten, die genau die Kontrollabhängigkeiten ausdrücken.
Diese Transformation von Kontrollabhängigkeiten in Datenabhängigkeiten nennt man auch **IF-Conversion**. Ein Algorithmus findet sich in [War94].
Man beachte, dass eine Schleife, die vor der *IF-Conversion* aus mehreren Basisblöcken bestanden hat, nun zu einem einzelnen großen Basisblock transformiert wurde.

7.2 Vorgeschaltete Optimierungen

Wenn man die notwendigen Programmdarstellungen der Eingabeschleife, wie Kontrollflussgraph, Datenabhängigkeitsgraph, etc., berechnet hat, könnte man mit dem eigentlichen Modulo Scheduling beginnen. Allerdings kann man noch von den Ergebnissen anderer Codeoptimierungen profitieren.
Der Scheduling-Prozess muss sicherstellen, dass keine Datenabhängigkeiten und Ressourcenbedingungen verletzt werden. Diese Bedingungen limitieren die aus der Schleife extrahierbare Parallelität. Dadurch wird klar, dass jedwede vorhergehende Optimierung, die die Anzahl dieser Bedingungen verringert, die Qualität desselben erhöht. Im

folgenden werden nun einige bekannte Optimierungen genannt, die hierzu verwendet werden können:

Common Subexpression Elimination sucht in einem Programm nach Berechnungen, die mehrfach durchgeführt werden und entfernt die redundanten Berechnungen aus dem Programm. Die Suche wird durch eine *Available Expressions Analyse* ([NNH99]) erledigt. Durch Common Subexpression Elimination werden Benutzung-Setzung-Abhängigkeiten aus dem Datenabhängigkeitsgraphen entfernt und auch Lebensspannen von Registern reduziert.

Constant Folding sucht in einem Programm nach Variablen/Registern, die auf jedem Ausführungspfad des Programmes einen konstanten Wert annehmen. Diese Auftreten können dann durch den konstanten Wert ersetzt werden, wodurch ebenfalls Lebensspannen von Variablen/Registern und Benutzung-Setzung-Abhängigkeiten entfernt werden. Welche Variablen bzw. Register konstante Werte annehmen, kann über die sog. *Copy Propragation Analyse* ([NNH99]) bestimmt werden.

Loop-Invariant Code Motion sucht nach Berechnungen innerhalb einer Schleife, deren Wert für die Ausführung des Schleifenkörpers invariant ist. In dem Fall, kann die komplette Berechnung vor die Schleife geschoben werden. Dies hat zwei Vorteile: zum einen wird die Schleife kleiner, sodass weniger Operationen iteriert werden müssen, und zum anderen entfallen dadurch Datenabhängigkeiten innerhalb der Schleife.

Strength Reduction ist eine Optimierung, die in Schleifen teure[3] Operationen durch äquivalente aber einfachere und damit schneller ausführbarere Operationen ersetzt. Beispielsweise ist für gewöhnlich eine Multiplikation teurer als eine Addition. Dies geschieht auf der Basis von Induktionsvariablen einer Schleife. Eine Induktionsvariable einer Schleife ist eine Variable, die in jeder Iteration systematisch, d.h. in der gleichen Weise, verändert wird. Beispielsweise stellt eine Variable i, die in jeder Iteration inkrementiert wird, eine Induktionsvariable für die Schleife dar. Listing 7.1 verdeutlicht *Strength Reduction*. Im linken Listing ist das Originalprogramm dargestellt. Im rechten Listing ist das Programm nach der *Strength Reduction* gezeigt. Dort ist die Multiplikation in Zeile drei des Originalprogrammes in eine Addition transformiert worden. Dafür wird eine temporäre Variable (`temp`) benutzt.

[3]eine Operation ist teurer als eine andere, wenn ihre Ausführung länger dauert

```
2 for (i=0 ; i<10 ; ++i) {
      n = i * (m+1);
4
  }
```

(Originalprogramm)

```
temp = 0;
for (i=0 ; i<10 ; ++i) {
   n = temp;
   temp = temp + (m+1);
}
```

(Programm nach Strength Reduction)

Listing 7.1: Beispiel für Strength Reduction

Dead Code Elimination entfernt Anweisungen bzw. Operationen, die während der Abarbeitung des Programmes nie ausgeführt werden. Solche „toten“ Anweisungen sind z.B. Zuweisungen von Werten zu einem Register, für die es keine korrespondierende Benutzung gibt. Solche Register/Variablen können durch eine *Live Variables Analyse* identifiziert werden. Die *Dead Code Elimination* entfernt zwar dadurch keine Datenabhängigkeiten aus dem Programm, die für die spätere Optimierung einer Schleife von Einfluss wären, kann aber die Anzahl der Operationen innerhalb einer Schleife verringern. Dadurch werden Ressourcen nicht unnötig belegt, was mehr Raum für die Optimierung der Schleife gibt.
Die hier genannten Codeoptimierungen werden üblicherweise bereits von den Übersetzern durchgeführt, sodass sie aufgrund des Postpass-Ansatzes in dieser Arbeit bereits in der Eingabeschleife umgesetzt sind. Bietet der vorhandene Übersetzer solche Optimierungen nicht an, so kann man sie an dieser Stelle auch im Postpass-Ansatz durchführen. Hier ist allerdings zu beachten, dass es einen wesentlichen Unterschied macht, ob eine Optimierung oder Analyse innerhalb des Übersetzers in der dort vorhandenen Zwischendarstellung oder als Postpass-Ansatz durchgeführt wird. Da eine Anweisung der Hochsprache meist in mehrere Operationen auf Assemblerebene übersetzt wird, ergibt sich eine andere Sicht des Programmes. So weist ein Assemblerprogramm meist wesentlich mehr Datenabhängigkeiten auf als sein entsprechendes Hochspracheprogramm. Dies führt dazu, dass es vorteilhaft sein kann, diegleiche Optimierung sowohl auf dem Hochspracheprogramm als auch im entsprechenden Assemblerprogramm durchzuführen.
Für detailliertere Beschreibungen und Erläuterungen zu den hier genannten Optimierungen wird auf [NNH99] verwiesen.

7.3 Initiierungsintervall

Wie in Abschnitt 5.3.2 beschrieben, bildet das Initiierungsintervall die Grundlage für Modulo Scheduling, da es die Verzögerung zwischen dem Start zweier aufeinanderfolgender Iterationen der ursprünglichen Schleife in Zyklen bestimmt. Durch das Initiie-

rungsintervall ist der Grad der Verzahnung der Iterationen festgelegt, d.h. je kleiner das Intervall ist desto enger werden die Iterationen miteinander verzahnt. Das beeinflusst auch die Anzahl an Iterationen der ursprünglichen Schleife, die später im erzeugten Kernel während einer einzigen Ausführung desselben bearbeitet werden und hat damit wesentlichen Einfluss auf die Dauer der Ausführungszeit der Schleife nach dem Pipelining.
Für den folgenden Scheduling-Prozess benötigt man eine untere Schranke für das Initiierungsintervall, welche man als **minimales Initiierungsintervall** (MII) bezeichnet. Dieses hängt von zwei Komponenten ab: dem minimalen ressourcenbasierten und dem minimalen datenabhängigkeitsbasierten Initiierungsintervall. Ersteres basiert auf der Anzahl an verfügbaren parallelen funktionalen Einheiten des Prozessors, wohingegen das minimale datenabhängigkeitsbasierte Initiierungsintervall von den im Eingabeprogramm vorhandenen Datenabhängigkeiten bestimmt wird.
Dadurch lässt sich die Berechnung des Initiierungsintervalls in die Lösung zweier Teilprobleme aufspalten; die Berechnung des sog. **minimalen ressourcenbasierten Initiierungsintervalls** (MII_{res}) und des sog. **minimalen datenabhängigkeitsbasierten Initiierungsintervalls** (MII_{dep}). Die Berechnung von MII_{res} und MII_{dep} ist in Abschnitt 7.3.1 bzw. 7.3.2 erläutert.
Hat man für beide Komponenten Werte bestimmt, so kann man das minimale Iterationintervall MII wie folgt definieren:

Definition 7.1 (minimales Initiierungsintervall). Seien MII_{res} und MII_{dep} Werte für das minimale ressourcenbasierte Initiierungsintervall bzw. das minimale datenabhängigkeitsbasierte Initiierungsintervall für eine Schleife L. Das **minimale Initiierungsintervall** (MII) für L ist dann konservativ definiert als das Maximum beider Werte:

$$MII = \max\{MII_{res}, MII_{dep}\}.$$

■

Es kann im späteren Scheduling-Prozess der Fall eintreten, dass für das berechnete MII kein gültiger Schedule gefunden werden kann. Wenn z.B. $MII = MII_{dep}$ gilt, kann es sein, dass ein darauf basierender Schedule dennoch eine Ressourcenbedingung[4] verletzt.

Bemerkung. Das berechnete MII muss nicht zwangsläufig zu einem gültigen Schedule führen sondern berechnet lediglich eine sichere untere Schranke. Kann kein gültiger Schedule gefunden werden, wird ein höheres Initiierungsintervall genommen (vgl. Abschnitt 7.5). Dies begründet sich in der Aufspaltung der Berechnung in die zwei Teilprobleme der Berechnung des *ressourcenbasierten Initiierungsintervalls* und das der Berechnung des *datenabhängigkeitsbasierten Initiierungsintervalls* und dem Bilden des Maximums.

[4]für eine Erklärung siehe Abschnitt 4.1

7.3.1 Ressourcenbasiertes Initiierungsintervall

Das **minimale ressourcenbasierte Initiierungsintervall** berechnet eine untere Schranke für die Verzögerung zwischen zwei aufeinanderfolgenden Iterationen der ursprünglichen Schleife unter Berücksichtigung der vorhandenen Ressourcen. Dabei nimmt man an, dass keine Datenabhängigkeiten diese maximale Auslastung einschränken. Das Ergebnis ist die minimale Anzahl an Instruktionen, die für die Operationen der Schleife benötigt werden, unter maximaler Auslastung der vorhandenen funktionalen Einheiten. Denn nutzt man die auf dem Prozessor mögliche Parallelität maximal aus, erhält man eine obere Schranke für die verzahnte Ausführung der Iterationen der Schleife. Einen höheren Grad an Parallelität kann man auf der Maschine für das Eingabeprogramm nicht extrahieren.
Jede Operation benutzt während ihrer Ausführung bestimmte funktionale Einheiten des Prozessors. Eine Additionsoperation z.B. belegt die arithmetische Einheit und anschließend den Bus, um ihr Ergebnis ins das angegebene Ziel zu schreiben. Da nur endlich viele funktionale Einheiten im Prozessor vorhanden sind, entstehen **Ressourcenbedingungen**, die bei der Berechnung des MII_{res} beachtet werden müssen. Dazu muss man für jede Operation erfassen, welche funktionalen Einheiten sie während ihrer Ausführung belegt.

Definition 7.2 (Belegungsplan). Sei p eine Maschinenoperation für einen Prozessor P, deren Ausführungszeit i Zyklen beträgt, und $\mathcal{R}$ die Menge aller auf P verfügbaren Ressourcen. $\mathcal{R}_p \subseteq \mathcal{R}$ bezeichnet dabei die Menge der durch die Operation p benutzten Ressourcen. Ein **Belegungsplan** ist eine Tabelle, die für jeden Zyklus ($\{0, \ldots, i-1\}$) der Ausführungszeit von p angibt, welche Ressource $r \in \mathcal{R}_p$ in diesem Zyklus von p benutzt wird.
Man unterscheidet zwischen den folgenden Arten von Belegungsplänen:

- einfacher Belegungsplan,
- Blockbelegungsplan und
- komplexer Belegungsplan.

Ein **einfacher Belegungsplan** ist ein Belegungsplan für eine Operation, die nur eine einzige Ressource einen Zyklus belegt, nämlich den Zyklus, in dem die Operation gestartet wurde.
Ein **Blockbelegungsplan** ist ein Belegungsplan für eine Operation, die nur eine einzige Ressource über mehrere, aufeinanderfolgende Zyklen belegt, beginnend mit dem Zyklus, in dem sie gestartet wurde.
Ein **komplexer Belegungsplan** ist ein Belegungsplan, der weder ein einfacher Belegungsplan noch ein Blockbelegungsplan ist. ■

Bemerkung. Alle Belegungspläne für Operationen moderner Prozessoren sind komplexe Belegungspläne.

Bemerkung. Bei vielen Prozessoren gibt es mehrere Belegungspläne für eine Operation.

Zyklus	Addierer	Multiplizierer	Bus
0			
1			
2			

Tabelle 7.1: Komplexer Belegungsplan

Beispiel 7.1 (Belegungsplan). Tabelle 7.1 zeigt einen komplexen Belegungsplan einer Operation. Die Ausführung der Operation benötigt insgesamt drei Zyklen, in denen im ersten Zyklus der Addierer, im zweiten der Multiplizierer und im dritten und letzten Zyklus der Systembus belegt wird. ■

Damit lässt sich die exakte Berechnung des MII_{res} auf die Lösung des sog. *Scheduling of Independent Tasks Problem (SIT)* zurückführen:

Definition 7.3 (Scheduling of Independent Tasks Problem). Sei $J = \{1, 2, \ldots, n\}$ eine Menge von n verschiedenen voneinander unabhängigen Jobs. Jeder Job i besitzt eine spezifische Ausführungszeit e_i. Ferner sei $M = \{m_1, m_2, \ldots, m_p\}$ eine Menge von p Maschinen. Eine Lösung für das **Scheduling of Independent Tasks Problem** weist jedem Job eine Maschine zu, sodass die Gesamtausführungszeit aller Jobs minimiert wird, d.h. es wird eine Funktion $S : \{1, \ldots, n\} \to \{1, \ldots, p\}$ berechnet, sodass

$$\max_{k \in M} \sum_{i:S(i)=k} e_i$$

minimiert wird.
Es wird also ein minimaler Schedule für die Jobs berechnet. ■

Satz 7.1. *Das SIT-Problem aus Definition 7.3 ist $\mathcal{NP}$-vollständig.* ■

Beweis. Für den Beweis von Satz 7.1 wird auf [GJ79, EF03] verwiesen. ■

Die Berechnung der minimalen Lösung des SIT-Problems stellt ein Optimierungsproblem (Minimierung der Länge des Schedules) dar. Aufgrund der Schwere des Problems, approximiert man die optimale Lösung.

Definition 7.4 (δ-Approximation). Sei I die Instanz eines Minimierungsproblems Π und $S(I)$ die Menge aller gültigen Lösungen für I. Die Funktion $\Phi : S(I) \to \mathbb{R}$

sei die Zielfunktion, deren Wert für eine Lösung $S(I)$ minimiert werden soll. Eine **δ-Approximation** für I ist eine Lösung A_I, für die gilt:

$$\forall S \in S(I), \delta \leq 1 : \Phi(A_I) \leq \delta * \Phi(S).$$

■

Das bedeutet, dass eine δ-Approximation eine Lösung eines Minimierungsproblems ist, die nicht schlechter als $\delta * L_{OPT}$ ist. L_{OPT} bezeichnet hierbei die bezüglich einer Zielfunktion Φ optimale Lösung.

Satz 7.2 ($\frac{4}{3}$-Approximation für Scheduling of Independent Tasks). *Folgender Algorithmus liefert eine $\frac{4}{3}$-Approximation an die optimale Lösung des* Scheduling of Independent Tasks Problem*:*

1. *Sortiere die Jobs in absteigender Reihenfolge ihrer Ausführungszeit*
2. *Weise nacheinander jedem Job in der berechneten Reihenfolge die Maschine zu, die zu dem Zeitpunkt am wenigsten benutzt wird.*

■

Beweis. Für den Beweis von Satz 7.2 wird auf [EF03] verwiesen. ■

Das Problem, das ressourcenbasierte Initiierungsintervall zu berechnen, lässt sich dann wie folgt auf das SIT-Problem zurückführen: die Maschinen entsprechen den freien Plätzen[5] für Operationen innerhalb einer Instruktion und die Jobs entsprechen den Operationen einer Schleife. Das vorliegende Problem ist sogar noch schwerer als das SIT-Problem, da bei der Zuweisung einer Operation zu einem freien Platz geprüft werden muss, dass entsprechend ihres Belegungsplanes genügend Ressourcen zu den entsprechenden Ausführungszeitpunkten vorhanden sind. Diese sind dann zu den entsprechenden Zeitpunkten als belegt zu markieren.

Definition 7.5 (Kontrollschritt). Sei σ ein Ablaufplan und seien alle Instruktionen von σ beginnend mit Null durchnummeriert in der Reihenfolge, in der sie auch abgearbeitet werden. Der **Kontrollschritt** ist der Index einer Instruktion gemäß dieser Nummerierung und bezeichnet damit einen Zeitpunkt. ■

Aufgrund der Schwere des Problem, versucht man, das MII_{res} so exakt wie möglich zu approximieren, indem man sich folgender Heuristik bedient:
Man beginnt mit einer leeren Instruktion und fügt der Reihe nach Operationen dieser

[5] bei VLIW-Prozessoren auch *Issue-Slots* genannt, siehe Kapitel 3

Instruktion hinzu. Hierbei führt man Buch über den jeweils verwendeten Belegungsplan, um gewährleisten zu können, dass durch das Anordnen einer Operation keine Ressourcenbedingung gegenüber bereits angeordneten Operationen verletzt werden. Dabei versucht man, die einzelnen Instruktionen möglichst komplett mit Operationen zu füllen. Kann aber eine Operation der aktuellen Instruktion nicht mehr hinzugefügt werden, weil diese bereits voll besetzt ist bzw. weil die Belegungspläne der bereits angeordneten Operationen keine Möglichkeit mehr für die Belegungspläne der aktuellen Operation zulassen, so wird eine neue Instruktion erzeugt und versucht, die Operation dort anzuordnen. War die vorhergehende Instruktion noch nicht voll mit Operationen besetzt, wird im weiteren Verlauf für alle noch zu bearbeitenden Operationen versucht, die noch freien Plätze in den Instruktionen zu füllen.
Mithilfe des Ergebnisses dieser Heuristik lässt sich das minimale ressourcenbasierte Initiierungsintervall wie folgt definieren:

Definition 7.6 (minimales ressourcenbasiertes Initiierungsintervall(MII_{res})). Seien die Operationen $op_1, \ldots, op_n$ einer Schleife L gegeben. Das **minimale ressourcenbasierte Initiierungsintervall** MII_{res}^L für die Schleife L ist die minimale Anzahl der Instruktionen, die benötigt werden, um die Operationen $op_1, \ldots, op_n$ auszuführen, unter alleiniger Berücksichtigung der Ressourcenbedingungen. ■

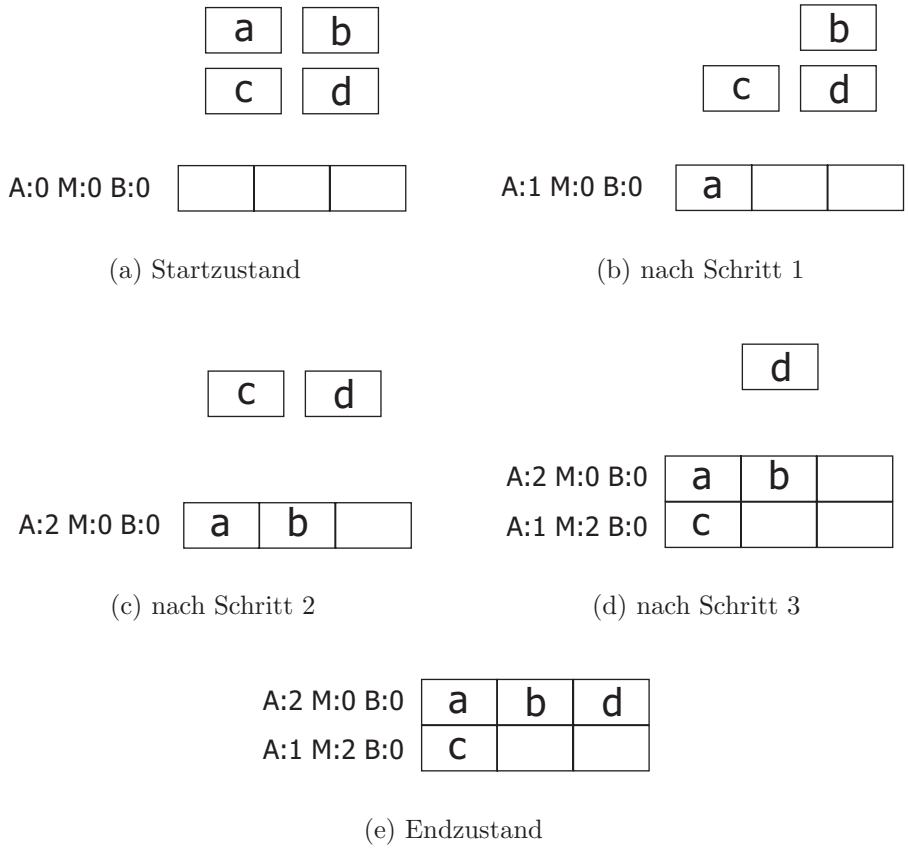
Beispiel 7.2. Gegeben sei ein VLIW-Prozessor (siehe Kapitel 3), der pro Instruktion maximal drei Operationen aufnehmen kann. Der Prozessor verfügt über folgende Ressourcen: zwei Addierer, zwei Multiplizierer und zwei Busse. Es soll nun das MII_{res} für eine Schleife mit vier Operationen (a,b,c,d) berechnet werden. Die Belegungspläne der Operationen a, b c sind in Tabelle 7.2 dargestellt, der Belegungsplan von Operation d in Tabelle 7.3.
Abbildung 7.1 zeigt schrittweise die Berechnung des MII_{res}. In (a) ist der Ausgangszustand dargestellt: man beginnt dabei mit einer leeren Instruktion, neben der die Benutzung der Ressourcen dargestellt ist. In Instruktion 1 wird momentan weder Addierer, Multiplizierer noch Bus benutzt. Abbildung 7.1 (b) zeigt den Zustand, nachdem Operation a angeordnet wurde. Man sieht, das in der ersten Instruktion nun ein Addierer (durch Operation a) benutzt wird. In (c) ist die Situation dargestellt, nachdem Operation b angeordnet wurde. Da noch ein Addierer frei ist, kann sie in die erste Instruktion gesetzt werden. In (d) ist dann die Situation nach dem Anordnen von Operation c dargestellt. Sie muss in eine neue Instruktion gesetzt werden, da keine Addierer im Kontrollschritt der ersten Instruktion verfügbar sind. Im zweiten Kontrollschritt sieht man auch die Belegung der Multiplizierer durch die Operationen a und b (siehe dessen Belegungspläne). Operation d kann nun aber in die erste Instruktion gesetzt werden, da sie zu Anfang ihrer Ausführung direkt einen Multiplizierer benötigt, der da noch frei ist. Der fertige Schedule ist in (e) gezeigt.
Man benötigt also zwei Instruktionen, um alle vier Operationen der Schleife anzuordnen, ohne Ressourcenbedingungen zu verletzen. Daher gilt hier $MII_{res} = 2$. ■

Zyklus	Addierer	Multiplizierer	Bus
0			
1			
2			

Tabelle 7.2: Belegungsplan für Operationen a,b,c

Zyklus	Addierer	Multiplizierer	Bus
0			
1			

Tabelle 7.3: Belegungsplan für Operation d



(a) Startzustand

(b) nach Schritt 1

(c) nach Schritt 2

(d) nach Schritt 3

(e) Endzustand

Abbildung 7.1: MII_{res}-Approximation

Mittels der beschriebene Heuristik erhält man ein positives **ganzzahliges Initiierungsintervall**. Da man aber im Allgemeinen nicht alle Instruktionen komplett mit Operationen besetzen kann, kann man das MII_{res} genauer bestimmen, wenn man auch **rationale Initiierungsintervalle** zulässt. Um solche für eine Schleife L zu berechnen, geht man wie folgt vor:

1. virtuelles Aufrollen der Schleife L zu L' um einen Faktor u,
2. Bestimmung des $MII_{res}^{L'}$
3. Berechnung von MII_{res}^{L} aus $MII_{res}^{L'}$.

Das virtuelle Aufrollen von L erreicht man, indem der Schleifenkörper einfach u Mal kopiert wird. Das $MII_{res}^{L'}$ für die Schleife L' berechnet sich dann wie in der Heuristik oben beschrieben. MII_{res}^{L} berechnet sich dann durch

$$MII_{res}^{L} = \left\lceil \frac{MII_{res}^{L'}}{u} \right\rceil .$$

7.3.2 Datenabhängigkeitsbasiertes Initiierungsintervall

Analog zum minimalen ressourcenbasierten Initiierungsintervall berechnet das **minimale datenabhängigkeitsbasierte Initiierungsintervall (MII_{dep})** die minimale Verzögerung zwischen dem Start zweier aufeinanderfolgender Iterationen der ursprünglichen Schleife unter Berücksichtigung der in der Schleife vorhandenen Datenabhängigkeiten. Die Grundlage der Berechnung des MII_{dep} sind die minimalen Abstände zwischen den Operationen, die durch die Datenabhängigkeiten induziert werden. Hier muss man beachten, dass Schleifen neue Datenabhängigkeiten, die sog. schleifenabhängigen Datenabhängigkeiten, zu den bereits im azyklischen Schleifenkörper vorhandenen einführen (vgl. Definition 2.19). Mit diesen minimalen Zeitabständen kann man berechnen, wie stark zwei Iterationen miteinander verzahnt werden können, ohne eine Datenabhängigkeit zu verletzen.
Um die minimale Verzögerung zwischen dem Start zweier Operationen zu bestimmen, muss man sich die Art der Datenabhängigkeit zwischen beiden ansehen.

Definition 7.7 (Verzögerung einer Datenabhängigkeit). Die **Verzögerung** $Delay(e)$ einer Datenabhängigkeit $e = (p, q)$ zwischen zwei Operationen p und q definiert die zeitlichen Verzögerung in Zyklen zwischen den Startzeitpunkten von p und q. $Exec(p)$ bezeichne dabei die Ausführungszeit einer Operation p in Zyklen. $Delay(e)$ hängt dabei vom Typ der Datenabhängigkeit ab:

- Setzung-Benutzung-Abhängigkeit: $Delay(e) = Exec(p)$

- Benutzung-Setzung-Abhängigkeit: $Delay(e) = 1 - Exec(q)$
- Setzung-Setzung-Abhängigkeit: $Delay(e) = 1 + Exec(p) - Exec(q)$

■

Die Berechnung der Abstände zwischen dem Start zweier Operationen sind in Abbildung 7.2 dargestellt. Hierbei wird von einer einfachen vierstufigen Instruktionspipeline ausgegangen, sodass die Ausführung einer Operation in die Stufen *Fetch(F)*, *Decode(D)*, *Execute(E)* und *Write-Back(W)* unterteilt ist. Die Operanden einer Operation werden in der ersten Phase (*Fetch*) gelesen und in der letzten Stufe (*Write-Back*) zurückgeschrieben. Die Tiefe und Untergliederung einer Instruktionspipeline unterscheidet sich stark von Prozessor zu Prozessor, sodass z.B. neue Registerwerte bereits nach der *Execute*-Phase verfügbar sein können. Die Berechnung der Verzögerung einer Datenabhängigkeit in Definition 7.7 geht von dem Fall aus, dass die Operanden in der ersten Phase gelesen und neue Werte in der letzten Phase zurückgeschrieben werden. Diese Situation ist typisch für VLIW-Prozessoren.
Für superskalare Prozessoren ist die folgende konservative Definition der Verzögerung

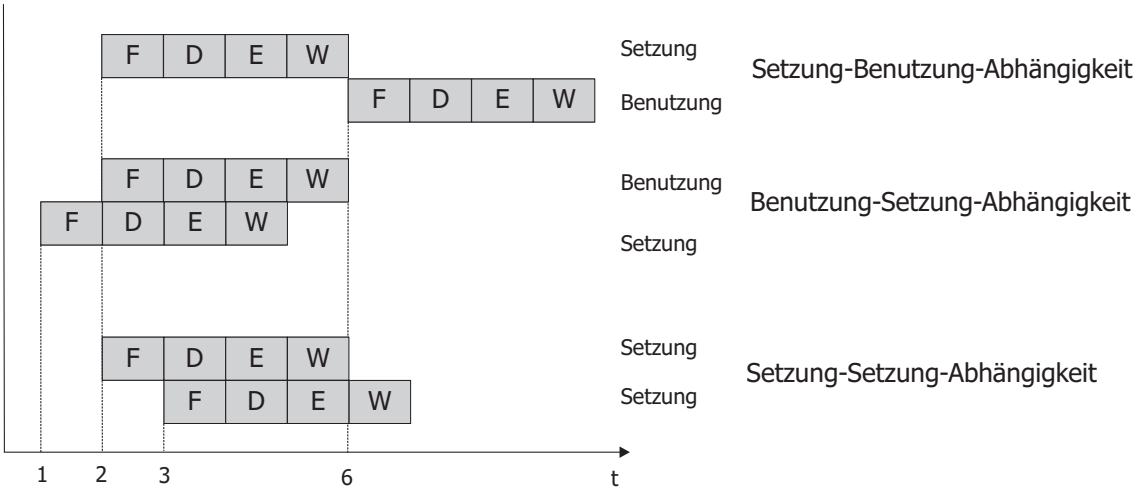


Abbildung 7.2: Datenabhängigkeitsinduzierte Verzögerungen

geeigneter. Sie macht nur die Annahme, dass die Ausführungszeit einer Operation nicht kleiner als ein Zyklus ist.

Definition 7.8 (konservative Verzögerung einer Datenabhängigkeit). Die **konservative Verzögerung** $Delay_{cons}(e)$ einer Datenabhängigkeit $e = (p, q)$ zwischen zwei Operationen p und q definiert die zeitliche Verzögerung in Zyklen zwischen

den Startzeitpunkten von p und q. $Exec(p)$ bezeichne dabei die Ausführungszeit einer Operation p in Zyklen. $Delay_{cons}(e)$ hängt dabei vom Typ der Datenabhängigkeit ab:

- Setzung-Benutzung-Abhängigkeit: $Delay_{cons}(e) = Exec(p)$
- Benutzung-Setzung-Abhängigkeit: $Delay_{cons}(e) = 0$
- Setzung-Setzung-Abhängigkeit: $Delay_{cons}(e) = Exec(p)$

■

Durch die schleifenabhängigen Datenabhängigkeiten kommt es zu Zyklen im Datenabhängigkeitsgraph. Will man Operationen der nachfolgenden Iteration einer Schleife bereits früher ausführen, sind eben diese Zyklen wichtig, da man aus ihnen Bedingungen für das Initiierungsintervall ableiten kann (siehe unten). Daher braucht man eine Aussage über die minimalen Verzögerung auf einem solchen Zyklus.

Definition 7.9 (elementarer Zyklus). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife. Ein **elementarer Zyklus** c in G_{dd} ist eine Sequenz von Knoten $n_1, n_2, \ldots, n_k \in N_{dd}, k \geq 0$, für die folgende Bedingungen gelten:

- die Knoten $n_1, \ldots, n_{k-1}$ sind paarweise verschieden:

$$\forall n_i, n_j, i \neq j, i, j \in \{1, \ldots, k-1\} : n_i \neq n_j$$

- es gibt einen Pfad von n_1 zu n_k in G_{dd}:

$$\forall i, 1 \leq i \leq k-1 : (n_i, n_{i+1}) \in E_{dd}$$

- Anfangsknoten und Endknoten der Knotensequenz sind gleich:

$$n_1 = n_k$$

■

Definition 7.10 (Verzögerung eines elementaren Zyklus). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $\mathcal{C}$ bezeichne die Menge aller elementaren Zyklen in G_{dd}. Die **Verzögerung** $Delay(c)$ **eines elementaren Zyklus** $c \in \mathcal{C}$ mit $c = n_1, n_2, \ldots, n_k$ ist definiert als die Summe der Verzögerungen aller Datenabhängigkeiten, die den Zyklus definieren.

$$Delay(c) = \sum_{i=1}^{k-1} Delay(n_i, n_{i+1})$$

■

Analog zur Verzögerung eines elementaren Zyklus kann man dann auch die Distanz eines Zyklus bestimmen.

Definition 7.11 (Distanz eines elementaren Zyklus). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $\mathcal{C}$ bezeichne die Menge aller elementaren Zyklen in G_{dd}. Die **Distanz** $Dist(c)$ **eines elementaren Zyklus** $c \in \mathcal{C}$ mit $c = n_1, n_2, \ldots, n_k$ ist definiert als die Summe der Distanzen aller Datenabhängigkeiten, die den Zyklus definieren.

$$Dist(c) = \sum_{i=1}^{k-1} Dist(n_i, n_{i+1})$$

■

Man dann die Bedingung ableiten, dass die tatsächliche Verzögerung im Schedule zwischen zwei aufeinanderfolgenden Ausführungen derselben Operation auf einem solchen Zyklus c einer Schleife größer oder gleich der minimal benötigten Verzögerung, $Delay(c)$, sein muss, wobei hierbei das Initiierungsintervall zu berücksichtigten ist.

Satz 7.3. *Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $\mathcal{C}$ bezeichnet die Menge aller elementaren Zyklen in G_{dd}. Jeder Zyklus $c \in \mathcal{C}$ induziert folgende Bedingung für das zu berechnende Initiierungsintervall MII_{dep}:*

$$MII_{dep} * Dist(c) \geq Delay(c)$$

■

Um das MII_{dep} zu berechnen, muss man nun alle Zyklen im Datenabhängigkeitsgraphen betrachten. Der Zyklus, der die stärkste Bedingung für das Initiierungsintervall darstellt, d.h. das MII_{dep} maximiert, stellt eine untere Grenze für MII_{dep} dar.

Definition 7.12 (MII_{dep}). Sei der Datenabhängigkeitsgraph $G_{dd} = (N_{dd}, E_{dd})$ einer Schleife gegeben und $\mathcal{C}$ die Menge aller elementaren Zyklen in G_{dd}. Das **minimale datenabhängigkeitsbasierte Initiierungsintervall (MII_{dep})** berechnet sich dann als

$$MII_{dep} = \max_{c \in \mathcal{C}} \left(\left\lceil \frac{Delay(c)}{Dist(c)} \right\rceil \right).$$

■

Um das MII_{dep} gemäß Definition 7.12 zu berechnen, muss man zuvor alle elementare Zyklen im Datenabhängigkeitsgraphen identifizieren.

Beispiel 7.3. Abbildung 7.3 zeigt einen Ausschnitt eines Datenabhängigkeitsgraphen einer Schleife. Jede Datenabhängigkeit e ist dabei mit einem Tripel (t, de, di) beschriftet, wobei t für den Typ, de für die Verzögerung $Delay(e)$ und di für die Distanz

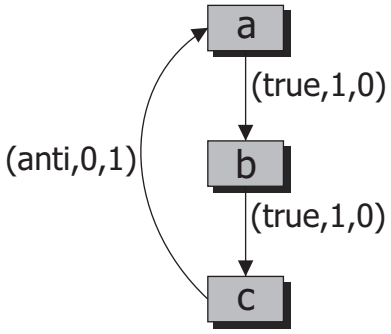


Abbildung 7.3: Ausschnitt Datenabhängigkeitsgraph

$Dist(e)$ steht. Im DDG ist der Zyklus $c = (a, b, c, a)$ enthalten, dessen Verzögerung $Delay(c) = 2$ und Distanz $Dist(c) = 1$ ist. Gemäß Satz 7.3 induziert dieser Zyklus also folgende Bedingung:

$$MII_{dep} * 1 \geq 2.$$

Das bedeutet, dieser Zyklus macht ein minimales MII_{dep} von zwei erforderlich. ■

Neben dieser Definition kann man das MII_{dep} auch durch die Formulierung als ganzzahliges lineares Programm bestimmen. Hier ist das MII_{dep} selbst die zu minimierende Zielfunktion und folgende Bedingungen müssen für alle Kanten $e = (i, j) \in E_{dd}$ gelten:

$$T(j) - T(i) \geq Delay(e) - MII_{dep} * Dist(e),$$

wobei $T(i)$ den Zeitpunkt angibt, an dem die Operation i gestartet wird. Die Differenz $T(j) - T(i)$ gibt die tatsächliche Verzögerung zwischen dem Start beider Operationen an. Diese Verzögerung muss mindestens so groß sein wie die durch die Datenabhängigkeit $(i, j) \in E_{dd}$ induzierte, wobei man das verwendete Initiierungsintervall berücksichtigen muss.
Eine weitere Möglichkeit, das MII_{dep} zu berechnen, ist die Formulierung als ein sog. **minimal cost-to-profit ratio cycle Problem** ([Huf93]).
Die Idee dieser Lösungsvariante ist es, eine Distanzmatrix zu berechnen, die aufbauend auf einem gewählten Initiierungsintervall für zwei Operationen i und j das minimal notwendige Intervall in Zyklen angibt, das zwischen dem Start der Operationen liegen muss.

Definition 7.13 (MinDist-Matrix). Sei P die Menge der Operationen einer Schleife mit $|P| = n$. Diese seien durchnummeriert, sodass jeder Operation p ein Index i mit $1 \leq i \leq n$ zugeordnet wird. Die **MinDist-Matrix** ist definiert als eine $n \times n$-Matrix, sodass Eintrag $[i, j]$ das minimale Zeitintervall in Zyklen bezeichnet, das zwischen den Operationen p_i und p_j (der gleichen Iteration) liegen muss. Das Zeitintervall zwischen

beiden Operationen ist die positive Differenz zwischen dem Startzeitpunkt beider Operationen. ■

Die Berechnung der $MinDist - Matrix$ erfolgt in zwei Phasen:

1. Initialisierung der Matrix und
2. Berechnung der endgültigen Werte der Matrix durch Betrachtung transitiver Datenabhängigkeiten.

Die Initialisierung der Matrix erfolgt nach folgender Berechnungsvorschrift:

$$MinDist\,[i,j] = \begin{cases} -\infty & (p_i, p_j) \notin E_{dd} \\ \max\limits_{e=(i,j)\in E_{dd}} \{Delay(e) - MII_{dep} * Dist(e)\} & sonst \end{cases}$$

Dies berechnet die minimal notwendige Verzögerung zwischen zwei Operationen p_i und p_j bei Betrachtung aller direkten Kanten zwischen den beiden Operationen im Datenabhängigkeitsgraph.
Die Berechnung der Matrix-Einträge unter zusätzlicher Betrachtung der transitiven Abhängigkeiten zwischen zwei Operationen erfolgt durch den in Abschnitt 8.2.2 beschriebenen *minimal cost-to-profit ratio cycle Algorithmus* ([EF03]), in dem auch die Initialisierung der Matrix enthalten ist. Dieser Algorithmus stellt fest, ob das verwendete Initiierungsintervall zu groß oder zu klein ist. Ersteres ist der Fall, wenn alle Einträge auf der Diagonalen der $MinDist - Matrix$ kleiner als Null sind. Dann muss man die Berechnung mit einem kleineren Wert für das Initiierungsintervall wiederholen. Ist mindestens ein Diagonaleintrag echt größer als Null, würde das bedeuten, dass eine Operation um genau den Wert des Eintrages nach sich selbst angeordnet werden müsste, was unmöglich ist. In dem Fall ist das verwendete Initiierungsintervall zu klein und man muss die Berechnung mit einem größeren Wert wiederholen. Das kleinste gültige Initiierungsintervall ist also gefunden, wenn mindestens ein Diagonaleintrag der Matrix gleich Null ist und alle anderen kleiner gleich Null sind.
Durch wiederholtes Berechnen mit unterschiedlichen Initiierungsintervallen lässt sich dann das gesuchte MII_{dep} bestimmen. Um die Anzahl der Matrix-Neuberechnungen zu minimieren, kann man das für eine Neuberechnung zu verwendende größere bzw. kleinere Initiierungsintervall mithilfe einer binären Suche bestimmen. Man beginnt als Startwert mit dem MII_{res} (siehe Abschnitt 7.3.1) und verdoppelt das Initiierungsintervall, wenn das verwendete zu klein war, solange, bis es zu groß ist. An diesem Punkt hat man ein Intervall, nämlich das zwischen dem letzten verwendeten (zu kleinen) Wert und dem aktuellen, gefunden, auf dem eine binäre Suche angewandt werden kann.

Eine weitere Methode der Berechnung des MII_{dep}, die sog. **Path Algebra**, ist in [SS02] aufgezeigt. Sie zeigt eine alternative mathematische Formulierung des Problems

und berechnet ähnlich zur Formulierung als *minimal cost-to-profit ratio cycle Problem* eine Distanzmatrix.

7.4 Priorisierung der Operationen

Die Priorisierung weist jeder Operation der Schleife eine Priorität zu, mit deren Hilfe später die Reihenfolge, in der die Operationen angeordnet werden, festgelegt wird. Prinzipiell gibt es für das Verfahren keine Einschränkung in der Art der Priorisierung. Dennoch erzielt man gute Ergebnisse, wenn man den Operationen auf dem kritischen Pfad[6] eine höhere Priorität zuweist. Dies erreicht man zum Beispiel durch Verwendung der in Abschnitt 5.3.2 genannten **highest-level-first** Priorisierung, die auch im List Scheduling verwendet wird. Um den zyklischen Abhängigkeiten im DDG einer Schleife Rechnung zu tragen, muss man kleine Anpassungen vornehmen, die im folgenden beschrieben sind.
Die im List Scheduling verwendete Priorisierungsfunktion, die jeder Operation eine Priorität zuweist, ist wie folgt definiert:

Definition 7.14 (azyklische Priorität). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife L gegeben und $p \in N_{dd}$ eine Operation, deren Nachfolger im Datenabhängigkeitsgraph durch die Menge $Succ(p)$ bezeichnet sind. Die Priorisierungsfunktion $Height : N_{dd} \rightarrow \mathbb{N}$ weist p eine Priorität nach folgender Berechnungsvorschrift zu:

$$Height(p) = \begin{cases} 0 & Succ(p) = \emptyset \\ \max\limits_{q \in Succ(p)} (Height(q) + Delay(p,q)) & sonst \end{cases}$$

■

Diese Definition berechnet für eine Operation *op* den längsten Pfad[7] von *op* im Datenabhängigkeitsgraphen G_{dd} beginnend mit *op*. Je größer der Wert von $Height(op)$ ist, desto länger, also kritischer, ist ihr Pfad und desto weniger Freiraum hat man, um die Operation anzuordnen. Will man den erzielten Schedule nicht unnötig erhöhen, so muss man die Operationen mit der höchsten Priorität zuerst anordnen.
Man könnte sich die Frage stellen, warum man dann nicht einfach alle Operationen auf dem kritischsten Pfad zuerst anordnet. In dem Fall würde man Operationen anordnen, bevor alle ihre Vorgänger im DDG angeordnet worden wären. Um aber eben die Abhängigkeiten zwischen diesen Vorgängern und den Operationen auf dem kritischsten Pfad zu erfüllen, kann der in Abschnitt 5.3.2 beschriebene Konfliktfall auftreten,

[6] der kritische Pfad ist der längste azyklische Pfad im DDG beginnend bei der aktuellen Operation
[7] im Sinne von Ausführungszeit

was den Berechnungsaufwand noch erhöhen würde. Und diesen Fall sollte man vermeiden.
Um nun die in einer Schleife vorhandenen schleifenabhängigen Datenabhängigkeiten zu berücksichtigen, muss man die Definition der Verzögerung zwischen zwei Operationen etwas anpassen. Eine schleifenabhängige Datenabhängigkeit beschreibt eine Abhängigkeit über eine Iterationsgrenze hinweg. Dies bedeutet, dass die Verzögerung zusätzlich durch das Initiierungsintervall beeinflusst wird. Deshalb wird der Begriff der „effektiven Verzögerung“ eingeführt:

Definition 7.15 (effektive Verzögerung). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph und $e = (p, q) \in E_{dd}$ eine schleifenabhängige Datenabhängigkeit von p nach q mit einer Distanz $Dist(e)$ von d. Ferner bezeichne II das verwendete Initiierungsintervall. Der **effektive Verzögerung** *EffDelay* von e ist dann definiert durch

$$EffDelay(e) = Delay(e) - II * Dist(e).$$

■

Aufbauend darauf lässt sich nun eine zyklische Priorität definieren, die auch schleifenabhängige Datenabhängigkeiten berücksichtigt.

Definition 7.16 (zyklische Priorität). Sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife L gegeben und $p \in N_{dd}$ eine Operation, deren Nachfolger im Datenabhängigkeitsgraph durch die Menge $Succ(p)$ bezeichnet sind. Die Priorisierungsfunktion $HeightR : N_{dd} \to \mathbb{N}$ weist p eine Priorität nach folgender Berechnungsvorschrift zu:

$$HeightR(p) = \begin{cases} 0 & Succ(p) = \emptyset \\ \max\limits_{q \in Succ(p)} (HeightR(q) + EffDelay(p, q)) & sonst \end{cases}$$

■

Da es sich um eine rekursive Definition handelt, muss zur Umsetzung eine Fixpunktiteration durchgeführt werden. Die direkte Umsetzung als solche würde allerdings nicht terminieren, da Datenabhängigkeitsgraphen eines zyklischen Kontrollflussgraphen in der Regel zyklisch sind. Diesem Problem entgegnet man, indem der zyklische Datenabhängigkeitsgraph in einen azyklischen Graphen transformiert wird.
Zyklen im Datenabhängigkeitsgraphen einer Schleife definieren starke Zusammenhangskomponenten (SCC). Betrachtet man alle starken Zusammenhangskomponenten als *Super-Knoten*, so ist der resultierende Graph azyklisch. In diesem kann dann für alle Knoten die Priorität mit einer DFS-Suche über den Graphen berechnet werden. Abbildung 7.4 zeigt die Transformation eines solchen zyklischen DDG in einen azyklischen. Die Berechnung der Priorität eines Knotens, der nicht zu einer starken Zu-

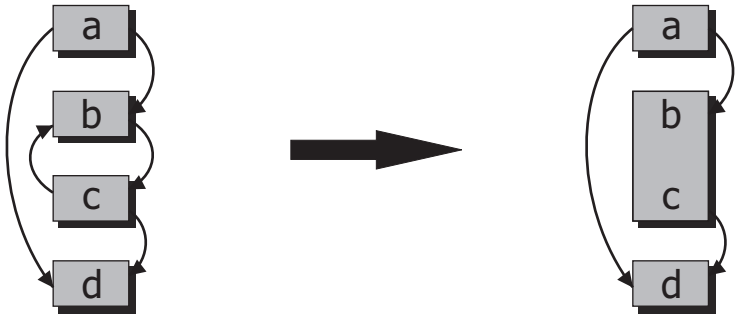


Abbildung 7.4: Transformation des Datenabhängigkeitsgraphen

sammenhangskomponenten gehört, d.h. der kein *Super-Knoten* ist, funktioniert durch die Anwendung von Definition 7.16. Die Berechnung der Priorität aller Knoten innerhalb einer starken Zusammenhangskomponenten muss speziell behandelt werden. Wie schon erwähnt kann die Priorität für solche Knoten nicht durch die Anwendung der Definition von zyklischer Priorität erfolgen, da in einer starken Zusammenhangskomponenten jeder Knoten ein Nachfolger eines jeden anderen Knotens ist.
Wird während eines DFS-Laufs über den zyklischen Datenabhängigkeitsgraphen der erste Knoten r einer noch nicht besuchten starken Zusammenhangskomponente bearbeitet, so bildet dieser Knoten r die Wurzel des kleinsten DFS-Teilbaumes, der die ganze SCC enthält. Dieser DFS-Teilbaum enthält auch alle Nachfolger der SCC. Gemäß Definition 7.16 (zyklischen Priorität) kann man die Priorität eines jeden Knotens einer SCC erst dann berechnen, wenn die Priorität für alle Nachfolger dieser SCC berechnet wurde. So kann man während eines DFS-Laufs durch den Datenabhängigkeitsgraphen alle Knoten einer starken Zusammenhangskomponenten sammeln und gemäß der Tiefensuche zuerst die Priorität der Nachfolger bestimmen und dann die Priorität für die Knoten der starken Zusammenhangskomponenten berechnen. Dies geschieht über eine Fixpunktiteration gemäß Definition 7.16, die terminiert, da die Priorität aller Nachfolger der starken Zusammenhangskomponente bereits berechnet wurde.
Der Algorithmus zur Berechnung der zyklischen Priorität einer Operation in einem zyklischen Datenabhängigkeitsgraphen ist in den Listings 7.2 und 7.3 dargestellt. Für jede Operation *op* wird die Priorität *HeightR*(*op*) mit $-\infty$ initialisiert und `New[`*`op`*`]` wird mit `true` initialisiert. `New[`*`op`*`]` bezüglich einer Operation zeigt an, ob die Operation bereits während des DFS-Laufs betrachtet wurde, d.h. `New[`*`op`*`]` ist `true` genau dann, wenn *op* noch nicht betrachtet wurde. Die Funktion *ComputeHeightR* setzt `New[`*`op`*`]` auf `false` für die Operation, mit der die Funktion aufgerufen wurde, um zu verhindern, dass diese Operation mehrfach bearbeitet wird. Die Knoten aller identifizierten starken Zusammenhangskomponenten werden in einem globalen Stack gespeichert. Für die Wurzel eines DFS-Teilbaumes einer starken Zusammenhangskomponenten wird die Funktion *FinalizeSCCHeights* aufgerufen, die per Fixpunktiteration

```
void FinalizeSCCHeights ( )
{
  operation first = tiefste operation im Stack, welche zur
                    gleichen starken Zusammenhangskomponenten
                    geh{\"o}rt wie das oberste Element des
                       Stacks
  bool changed;

  do {
    changed = false;
    for op = first to Top(Stack) do {
      for each child of op do {
        int new_prio = Max(HeightR[op], HeightR[child] +
                           Delay(op,child)-Dist(op,child)*II);
        if ( HeightR(op) != new_prio ) {
          HeightR(op) = new_prio;
          changed = true;
        }
      }
    }
  }
  while changed;
}
```

Listing 7.2: Funktion FinalizeSCCHeights

die Prioritäten der Knoten der starken Zusammenhangskomponenten berechnet. Anschließend werden alle Knoten der bearbeiteten starken Zusammenhangskomponenten vom Stack entfernt.

Alternativ kann man hier aber die Ergebnisse aus der Berechnung des datenabhängigkeitsbasierten Initiierungsintervalls benutzen, falls man dort die Umsetzung als *minimal cost-to-profit ratio cycle Problem* verwendet hat (siehe Abschnitt 7.3.2). In Kapitel 4 ist beschrieben, dass der durch die Instruktionsanordnung erzielte Schedule eine topologische Sortierung der Operationen des Eingabeprogrammes darstellt.

Satz 7.4. *Wenn zur Berechnung von HeightR die konservative Verzögerung (vgl. Definition 7.8) benutzt wird, dann definiert die Priorisierungsfunktion HeightR eine topologische Sortierung der Operationen.* ■

Dies ist darin begründet, dass per Definition der konservativen Verzögerung einem Nachfolgeknoten q nie eine geringere Priorität zugeordnet wird als seinem Vorgänger p im Datenabhängigkeitsgraph.

```
void ComputeHeightR ( operation op )
{
  New[op] = false;
  if ( op has no children )
    HeightR(op) = 0;
  else {
    for each child of op do {
      if ( New[child] )
        ComputeHeightR(child);
      HeightR(op) = Max(HeightR(op),HeightR(child)+
                        Delay(op,child)-Dist(op,child)*II);
    }
  }
  if ( op geh{\"o}rt zu einer \index{starke
     Zusammenhangskomponente} starken Zusammenhangskomponenten
     ) {
    FinalizeSCCHeights();
    while ( Top(Stack) geh{\"o}rt zur gleichen starken Zus-komp
       . wie op )
      Pop();
  }
}
```

Listing 7.3: Funktion ComputeHeightR

Da man wie oben beschrieben mit *HeightR* eine Prioritätsfunktion verwendet, die versucht, Konfliktfälle im späteren Scheduling-Prozess zu vermeiden, erhält man dadurch einen effizienten Scheduler, der auch pfadsensitiv bezüglich des kritischen Pfades der Schleife ist.

7.5 Flat Schedule

Wie in Abschnitt 5.3.2 beschrieben stellt der **Flat Schedule** einen Ablaufplan dar, der auch die schleifenabhängigen Datenabhängigkeiten berücksichtigt, ohne allerdings irgendeine Berechnung modulo eines Initiierungsintervalls zu machen. Aus diesem kann in einem späteren Schritt der eigentliche Kernel berechnet werden. Dort findet auch die Modulo-Berechnung statt. Dies kann nicht in einem Schritt geschehen, da sonst falsche Schedule-Positionen berechnet werden würden. Genauer gesagt, könnte man die Kontrollschritte zweier Operationen nicht mehr vergleichen, wenn einer bereits modulo des Initiierungsintervalls gerechnet worden wäre und der andere noch nicht.

Die Berechnung des Flat Schedule ist iterativ in der Hinsicht, dass vor Beginn nicht sicher ist, ob das verwendete Initiierungsintervall einen gültigen Schedule zulässt. Denn wie in Abschnitt 7.3 beschrieben, stellt das Initiierungsintervall nur eine untere Schranke dar, die noch nicht das reale Minimum darstellen muss. Schlägt die Berechnung für ein II fehl, wird von neuem mit einem größeren II begonnen. Da meistens die Approximation an das real mögliche Initiierungsintervall nur sehr gering unterschätzt (siehe Kapitel 9), wenn überhaupt, reicht es hier aus, das II zu inkrementieren.
Um sicherzustellen, dass das Verfahren terminiert und um außerdem die Laufzeit einzuschränken, wird ein zusätzlicher Parameter benutzt, das sog. **Budget-Verhältnis**. Es definiert eine obere Schranke für die Anzahl der Anordnungsversuche (vgl. Abschnitt 5.3.2). Dieser iterative Ablauf ist in Abbildung 5.3 in Abschnitt 5.3 dargestellt.
In jedem Schritt wird versucht, für das aktuelle Initiierungsintervall einen gültigen Schedule zu erzeugen. Dazu beginnt man mit dem leeren Schedule und versucht iterativ für alle Operationen eine gültige Instruktion zu finden, in der die Operation angeordnet werden kann, ohne Ressourcenbedingungen oder Datenabhängigkeiten[8] zu verletzen. Dadurch entsteht ein sog. **partieller Schedule**, der für die in ihm enthaltenen Instruktionen bereits gültig ist.
Die notwendigen Arbeitsschritte zum Suchen einer gültigen Instruktion für eine Operation sind in Abbildung 7.5 dargestellt. Zuerst wird für eine Operation ein sog. **Zeit-**

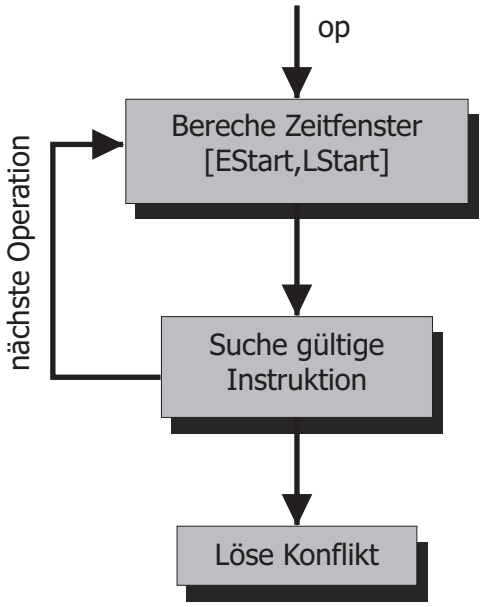


Abbildung 7.5: Berechnung Flat Schedule

[8] man erinnere sich, dass die Kontrollabhängigkeiten in Datenabhängigkeiten mittels *IF-Conversion* umgewandelt wurden

fenster berechnet, das ein Intervall von Kontrollschritten darstellt, innerhalb dessen die Operation angeordnet werden muss, um keine Datenabhängigkeiten zu verletzen. Anschließend wird innerhalb dieses Zeitfensters versucht, eine gültige Instruktion zu finden. Dabei wird die Strategie verfolgt, die Operation so früh wie möglich anzuordnen. Gelingt es, kann der Prozess mit dem Anordnen der nächsten Operationen fortfahren. Andernfalls ist der sog. Konfliktfall (siehe Abschnitt 7.5.3) aufgetreten. Dieser wird dann durch das erneute Entfernen einer oder mehrerer Operationen aus dem partiellen Schedule und das Anordnen der aktuellen Operation gelöst. Diese einzelnen Arbeitsschritte sind in den folgenden Unterabschnitten ausführlich erläutert.

7.5.1 Berechnung Zeitfenster

Um eine Operation *op* in den partiellen Schedule einfügen zu können, ohne Datenabhängigkeiten zu verletzen, muss man die Abhängigkeiten von *op* von den im partiellen Schedule enthaltenen Operationen betrachten. Aus jeder bestehenden Datenabhängigkeit ergeben sich Bedingungen, wie früh oder spät *op* angeordnet werden kann bzw. muss. Das sich daraus berechnende Zeitfenster ist relativ zum aktuellen partiellen Schedule und teilt sich auf in die Berechnung eines frühest möglichen Zeitpunktes (**Early Start**) und eines spätest möglichen Zeitpunktes (**Late Start**).
Die Berechnung dieser beiden Werte ist in den beiden folgenden Unterabschnitten erläutert.

7.5.1.1 Early Start

Die Berechnung von *Early Start* kann auf zwei verschiedene Weisen erfolgen, wobei die Berechnungsvorschrift sich darin unterscheidet, dass einmal nur die Vorgänger von *op* im DDG betrachtet werden, die im partiellen Schedule bereits enthalten sind, oder alternativ alle Vorgänger von *op* betrachtet werden. Ersterer Wert wird als *immediate Early Start* bezeichnet, die Alternative als *transitive Early Start*.
Die Berechnungsvorschrift für *immediate Early Start* sieht dabei wie folgt aus:

Definition 7.17 (immediate Early Start). Sei ein partieller Schedule σ_{part} gegeben. Ferner sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $p \in N_{dd}$ eine Operation der Schleife. $SchedTime(i)$ bezeichnet den Kontrollschritt[9] einer Operation i. $EffDelay(i)$ ist definiert wie in Definition 7.15. Der Wert *immediate Early Start* ist dann definiert durch

$$EStart(p) = \max_{(q,p)\in E_{dd}} \left(\begin{cases} 0 & q \notin \sigma_{part} \\ \max\left(0, SchedTime(q) + EffDelay(q,p)\right) & sonst \end{cases} \right).$$

[9] vergleiche Abschnitt 7.3

Wenn eine Operation *op* keine Vorgänger im DDG besitzt, ist $EStart(op) = 0$. ■

Der Berechnungsaufwand für *immediate Early Start* liegt in $\mathcal{O}(n)$, wobei n die Anzahl der Operationen in der Schleife ist.
Die Definition der alternativen Berechnungsweise für den frühest möglichen Kontrollschritt einer Operation ist wie folgt:

Definition 7.18 (transitive Early Start). Sei ein partieller Schedule σ_{part} gegeben. Ferner sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $p \in N_{dd}$ eine Operation der Schleife. $SchedTime(i)$ bezeichne den Kontrollschritt[10] einer Operation i. $EffDelay(i)$ ist definiert wie in Definition 7.15. Der Wert *transitive Early Start* ist dann definiert durch

$$EStart_{trans}(p) = \max_{(q,p)\in E_{dd}} \left(\begin{cases} \max(0, EStart_{trans}(q) + EffDelay(q,p)) & q \notin \sigma_{part} \\ \max(0, SchedTime(q) + EffDelay(q,p)) & sonst \end{cases} \right).$$

Wenn eine Operation *op* keine Vorgänger im DDG besitzt, ist $EStart_{trans}(op) = 0$. ■

Die Berechnung von *transitive Early Start* ist wesentlich aufwendiger als die von *immediate Early Start*, da bei *transitive Early Start* eine Fixpunktiteration gelöst werden muss.

Bemerkung. Um die Fixpunktiteration für *transitive Early Start* zu lösen, muss man anfangs für alle Operationen *op* das $EStart_{trans}(op)$ mit 0 initialisieren.

7.5.1.2 Late Start

Die Berechnung von *Late Start* erfolgt analog zur Berechnung von *Early Start*, d.h. auch hier unterscheidet man ein *immediate Late Start* und ein *transitive Late Start*. Die Berechnung des *immediate Late Start* für eine Operation p berücksichtigt dabei nur die Nachfolger von p im DDG, die auch aktuell im partiellen Schedule enthalten sind, wohingegen die Berechnung von *transitive Late Start* auch die restlichen Nachfolger von p im DDG mit einbezieht.
Die Berechnungsvorschrift für *immediate Late Start* sieht dabei wie folgt aus:

Definition 7.19 (immediate Late Start). Sei ein partieller Schedule σ_{part} gegeben. Ferner sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $p \in N_{dd}$ eine Operation der Schleife. $SchedTime(i)$ bezeichne den Kontrollschritt[11] einer Operation i. $EffDelay(i)$ ist definiert wie in Definition 7.15. Das *immediate Late Start* ist

[10] vergleiche Abschnitt 7.3
[11] vergleiche Abschnitt 7.3

dann definiert durch

$$LStart(p) = \min_{(p,q)\in E_{dd}} \left(\begin{cases} \infty & q \notin \sigma_{part} \\ \min(\infty, SchedTime(q) - EffDelay(p,q)) & sonst \end{cases} \right).$$

Wenn eine Operation *op* keine Nachfolger im DDG besitzt, ist $LStart(op) = \infty$. ■

Der Berechnungsaufwand für *immediate Late Start* liegt in $\mathcal{O}(n)$, wobei n die Anzahl der Operationen in der Schleife ist.
Die Definition der alternativen Berechnungsweise für den spätest möglichen Kontrollschritt einer Operation ist wie folgt:

Definition 7.20 (transitive Late Start). Sei ein partieller Schedule σ_{part} gegeben. Ferner sei $G_{dd} = (N_{dd}, E_{dd})$ ein Datenabhängigkeitsgraph einer Schleife und $p \in N_{dd}$ eine Operation der Schleife. $SchedTime(i)$ bezeichne den Kontrollschritt[12] einer Operation i. $EffDelay(i)$ ist definiert wie in Definition 7.15. Das *transitive Late Start* ist dann definiert durch

$$LStart_{trans}(p) = \min_{(p,q)\in E_{dd}} \left(\begin{cases} \min(\infty, LStart_{trans}(q) - EffDelay(p,q)) & q \notin \sigma_{part} \\ \min(\infty, SchedTime(q) + EffDelay(p,q)) & sonst \end{cases} \right).$$

Wenn eine Operation *op* keine Nachfolger im DDG besitzt, ist $LStart_{trans}(op) = \infty$. ■

Die Berechnung von *transitive Late Start* ist wesentlich aufwendiger als die von *immediate Late Start*, da bei ersterem eine Fixpunktiteration gelöst werden muss.

Bemerkung. Um die Fixpunktiteration für *transitive Late Start* zu lösen, muss man anfangs für alle Operationen *op* das $LStart_{trans}(op)$ mit ∞ initialisieren.

Nach [Rau94] benötigt die Fixpunktiteration zur Berechnung von *transitive Early Start* und *transitive Late Start* statistisch gemessen ca. 2.5 Iterationen. Die Laufzeit der jeweiligen Berechnung wäre dann $2.5 * N$, wobei N die Anzahl der Anordnungsversuche darstellt.
Eine alternative Methode zur Berechnung von *transitive Early Start* bzw. *transitive Late Start* ist die Benutzung der Ergebnisse aus der Berechnung des datenabhängigkeitsbasierten Initiierungsintervalls, falls man dort das Problem als **minimal cost-to-profit ratio cycle Problem** formuliert hat. Dann berechnet sich $EStart_{trans}(p)$ durch

$$EStart_{trans}(p) = \max_{q\in\sigma_{part}} (SchedTime(q) + MinDist(q,p))$$

[12] vergleiche Abschnitt 7.3

und $LStart_{trans}(p)$ durch

$$LStart_{trans}(p) = \min_{q \in \sigma_{part}} \left(SchedTime(q) - MinDist(p, q) \right).$$

7.5.2 Suche im Zeitfenster

Die berechneten Werte für *Early Start* und *Late Start* legen ein Intervall von Kontrollschritten im partiellen Schedule fest, das man als **Zeitfenster** bezeichnet. Innerhalb dieses Intervalls wird nun nach einer Instruktion gesucht, die die Operation aufnehmen soll und kann. Bei *Multi-Issue Prozessoren* beinhaltet dies die Suche nach einem gültigen *Issue-Slot* (vgl. Kapitel 3).

Definition 7.21 (Gültiger Issue-Slot). Sei *op* eine Maschinenoperation und σ_{part} ein partieller Schedule. Ein *Issue-Slot* i ist genau dann **gültig** für *op*, wenn i in σ_{part} noch nicht durch eine andere Operation besetzt ist und wenn das Anordnen von *op* in i keine Ressourcenbedingungen verletzt gegenüber den anderen Operationen im partiellen Schedule σ_{part}. ■

Bemerkung. Die Gültigkeit eines *Issue-Slots* hängt nach obiger Definition deshalb nicht von der Verletzung einer Datenabhängigkeit ab, da diese Bedingung bereits durch die Berechnung des Zeitfensters sichergestellt wurde.

Mit Definition 7.21 lässt sich nun definieren, wann eine Instruktion für eine anzuordnende Operation gültig ist.

Definition 7.22 (Gültige Instruktion). Sei *op* eine Operation, σ_{part} ein partieller Schedule und i eine Instruktion. Man nennt die Instruktion i eine **gültige Instruktion** für *op* genau dann, wenn ein *Issue-Slot* in i existiert, der für *op* gültig ist. ■

Das Prüfen der Ressourcenbedingungen ist hierbei abhängig von der Zielarchitektur. Auf VLIW-Prozessoren kann es z.B. vorkommen, dass in einer Instruktion noch ein *Issue-Slot* frei ist, dieser aber für die anzuordnende Operation nicht benutzt werden kann. Das ist eine Folge der heterogenen Bindung von funktionalen Einheiten zu *Issue-Slots* (vgl. Abschnitt 3.2.7). Da die meisten Operationen in mehreren verschiedenen *Issue-Slots* ausgeführt werden können, ist es unter Umständen möglich, durch eine Umverteilung (Permutation) der Operationen dennoch einen freien *Issue-Slot* für die aktuelle Operation zu finden, indem man die Zuweisung von Operationen zu Issue Slots in der Instruktion ändert. Beispiel 7.4 verdeutlicht eine solche Permutation, die aus einer ungültigen Instruktion eine gültige macht.

Beispiel 7.4 (Permutation der Issue-Slot-Bindung). Gegeben sei ein VLIW-Prozessor, der maximal fünf Operationen pro Instruktion aufnehmen kann. Ferner seien fünf Operationen $(1, 2, 3, 4, 5)$ und eine Instruktion gegeben, in der bereits vier

Operationen angeordnet wurden. Tabelle 7.4 zeigt für jede Operation die Issue-Slots, in der sie ausgeführt werden kann.
Will man nun in die in Abbildung 7.6 (a) Operation 5 anordnen, so stellt man fest, das der einzige freie *Issue-Slot* für Operation 5 nicht benutzt werden kann. Aber es existiert eine gültige Permutation gemäß Definition 7.22, die in Abbildung 7.6 (b) dargestellt ist. ■

Operation	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5
1					
2					
3					
4					
5					

Tabelle 7.4: Verfügbare Issue-Slots

(a) Instruktion

(b) Instruktion nach Issue-Slot Issue-Slot Permutation

Abbildung 7.6: Permutation der Issue-Slot-Bindung

Um innerhalb des berechneten Zeitintervalls nach einer gültigen Instruktion für eine Operation *op* zu suchen, beginnt man bei der Instruktion am Kontrollschritt von *Early Start* und prüft von dort an alle Instruktionen auf Gültigkeit für *op*.

7.5.3 Konfliktfall

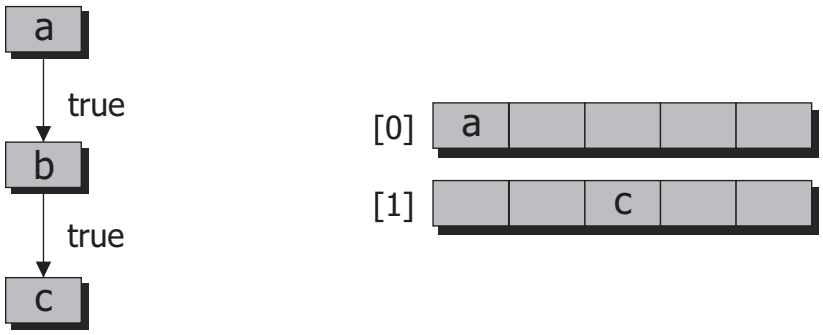
Während der Suche nach einer gültigen Instruktion für eine Operation kann es zu einem sog. **Konflikt** kommen, wenn keine gültige Instruktion gefunden werden konnte.

Definition 7.23 (Konflikt). Sei *op* eine Maschinenoperation und σ_{part} ein partieller Schedule. Ein **Konflikt** liegt genau dann vor, wenn in σ_{part} keine Instruktion existiert, die für *op* gültig ist. ■

Das Auftreten eines Konflikts bedeutet also, dass der partielle Schedule nicht um die aktuell anzuordnende Operation erweitert werden kann, ohne Ressourcenbedingungen oder Datenabhängigkeiten zu verletzen. Als Ursache kommen zwei Möglichkeiten in

Frage.
Zum einen kann das berechnete Zeitfenster ungültig sein, d.h. der Wert von *Late Start* ist kleiner als der von *Early Start*. Dies wird dadurch verursacht, dass die Verzögerung zwischen den Vorgängern und den Nachfolgern der aktuell anzuordnenden Operation im Datenabhängigkeitsgraph zu gering ist. Beispiel 7.5 verdeutlicht diese Situation. Zum anderen kann der Fall auftreten, dass das berechnete Zeitfenster gültig ist, innerhalb diesem aber keine gültige Instruktion gefunden werden kann. Da die Berechnung des Zeitfensters auf den Datenabhängigkeiten basiert, verursacht jeder noch freie *Issue-Slot* einen Ressourcenkonflikt.

Beispiel 7.5. Sei der Datenabhängigkeitsgraph eines Programmes wie in Abbildung 7.7 (a) und der aktuelle partielle Schedule wie in Abbildung 7.7 (b). Ferner beträgt die Ausführungszeit aller Operationen jeweils einen Zyklus. Für das Zeitfenster von



(a) Datenabhängigkeitsgraph (b) partieller Schedule

Abbildung 7.7: Konfliktfall

Operation b ergeben sich folgende Werte:

$$\begin{aligned} EStart(b) &= SchedTime(a) + Delay(a,b) \\ &= 0 + 1 = 1 \\ LStart(b) &= SchedTime(c) + Delay(b,c) \\ &= 1 - 1 = 0 \end{aligned}$$

Das berechnete Zeitfenster $[1, 0]$ ist ungültig, da keine gültige Instruktion im partiellen Schedule auf Basis der Datenabhängigkeiten gefunden werden kann. ■

Bemerkung. Man beachte, dass Beispiel 7.5 konstruiert ist. In diesem einfachen Fall würde, durch die Priorisierung der Operationen gesteuert, die Operationen in der Reihenfolge a, b, c angeordnet werden. Dann würde der beschriebene Konfliktfall nicht auftreten. Um das Beispiel möglichst einfach zu halten, wurde hier ein partieller Schedule gegeben, in dem Operation c bereits angeordnet ist.

Ein solcher Konflikt einer Operation *op* mit einem partiellen Schedule σ_{part} wird durch folgende Schritte aufgelöst:

1. Berechnung eines zwingenden Kontrollschrittes c für *op*
   Diese Berechnung hängt davon ab, ob *op* vorher bereits einmal angeordnet worden ist oder nicht.

   $$c = \begin{cases} EStart(op) & \text{op wird zum ersten Mal angeordnet} \\ \max\{c_{old} + 1, EStart(op)\} & \text{sonst} \end{cases}$$

   c_{old} bezeichnet dabei den Kontrollschritt, der *op* beim letzten Anordnen zugewiesen worden war.

2. Identifikation der mit *op* in Kontrollschritt c konfliktierenden Operationen aus σ_{part}
   Für jeden Vorgänger und jeden Nachfolger p von *op* im Datenabhängigkeitsgraph, der im partiellen Schedule enthalten ist, muss die zeitliche Verzögerung zwischen dem Start von p und *op* geprüft werden. Ist dieser nicht mindestens so groß, wie es die Datenabhängigkeit zwischen beiden Operationen erfordert (vgl. Definition 7.7), ist p konfliktierend mit *op*.
   Wenn in einer Instruktion für eine Operation kein Platz mehr frei ist, so kann dies entweder bedeuten, dass die Instruktion bereits komplett gefüllt ist, oder, dass von den freien Plätzen keiner für *op* genutzt werden kann aufgrund von Ressourcenbedingungen des zugrundeliegenden Prozessors. Zur präzisen Berechnung der minimalen Menge von Operationen, die mit *op* konfliktierend sind, müsste man alle Kombinationsmöglichkeiten der Belegungspläne in Betracht ziehen. Um dies zu vermeiden, werden alle Operationen der Instruktion als konfliktierend markiert. Dies vereinfacht den Algorithmus.

3. Entfernen dieser konfliktierenden Operationen aus σ_{part}

4. dem Anordnen von *op* im berechneten Kontrollschritt c

Dadurch wird der *partielle Schedule* aus einer Sackgasse in einen anderen Zustand versetzt, in dem der Scheduling-Prozess weitergeführt werden kann. Dieses Entfernen von Operationen aus dem partiellen Schedule zugunsten einer anderen Operation bezeichnet man auch als *Backtracking* (vgl. Abschnitt 5.3). Die entfernten Operationen

besitzen eine höhere Priorität als alle aktuell noch nicht angeordneten Operationen, da sie bereits angeordnet worden waren. Deshalb werden die entfernten Operationen direkt nach dem Entfernen durch den normalen Scheduling-Prozess wieder bearbeitet. An dieser Stelle muss man aber gewährleisten, dass sich nicht zwei Operationen gegenseitig wiederholend aus dem partiellen Schedule entfernen. Dies wird in Schritt 1 sichergestellt, indem eine Operation nie mehrmals im gleichen Kontrollschritt angeordnet wird.

7.5.4 Pipeline-Stufen

Die **Pipeline-Stufen** sind disjunkte Mengen von Operationen der ursprünglichen Schleife. Im resultierenden Modulo Schedule entsprechen die Operationen verschiedener Stufen Operationen verschiedener Iterationen der ursprünglichen Schleife, was die verzahnte Ausführung der Iterationen der ursprünglichen Schleife widerspiegelt.
Die Stufen einer Schleife können direkt aus dem *Flat Schedule* abgelesen werden.

Definition 7.24 (Länge eines Schedule). Sei L eine Schleife und $\mathcal{F}$ der entsprechende Flat Schedule von L. Die **Länge** $|\mathcal{F}|$ von $\mathcal{F}$ ist definiert als die Anzahl der Instruktionen, die $\mathcal{F}$ beinhaltet. ■

Eine Pipeline-Stufe besteht aus höchstens II vielen Instruktionen. Die Stufen erhält man, indem jeweils II Instruktionen des *Flat Schedule* zusammengefasst werden. Je nach Verhältnis zwischen der Länge des *Flat Schedule* und dem Wert des II kann die letzte Stufe auch weniger als II Instruktionen enthalten. Dadurch findet die Berechnung der Stufen implizit durch die Berechnung des Initiierungsintervalls und die Berechnung des *Flat Schedule* statt.
Im berechneten Kernel finden sich also später Operationen aus verschiedenen Iterationen der ursprünglichen Schleife wieder. Zur Berechnung des Prologes und Epiloges ist es wichtig, zu wissen, wieviele verschiedene Iterationen der ursprüngliche Schleife im Kernel vertreten sind.

Definition 7.25 (Pipeline-Tiefe). Sei $\mathcal{F}$ ein *Flat Schedule* einer Schleife und II das für seine Berechnung verwendete Initiierungsintervall.Die **Pipeline-Tiefe (SC)** von $\mathcal{F}$ ist dann definiert durch

$$StageCount = \left\lceil \frac{|\mathcal{F}|}{II} \right\rceil .$$

■

Bemerkung. Die Pipeline-Tiefe wird oft auch als **Spanne** bezeichnet.

Je höher die Pipeline-Tiefe ist, desto mehr Iterationen werden durch die Ausführung des Kernels parallel ausgeführt. Dadurch ist die Pipeline-Tiefe ein Gradmesser für die aus der ursprünglichen Schleife extrahierte Parallelität, was seine Wichtigkeit zur Bewertung eines Modulo Schedule hervorhebt.

7.6 Kernel

Der in Abschnitt 7.5 berechnete *Flat Schedule* berücksichtigt bereits das zu verwendende Initiierungsintervall. Nun muss aus diesem *Flat Schedule* noch der eigentliche Kernel der Schleife als Ergebnis des Pipeline-Prozesses berechnet werden. Dies kann nicht zusammen mit der Berechnung des *Flat Schedule* erledigt werden, da man ihn sonst nicht korrekt berechnen würde. Das Zeitfenster einer Operation berechnet sich relativ zu den bereits berechneten Kontrollschritten der Operationen im partiellen Schedule (vgl. Abschnitt 7.5.1). Eben diese Informationen gehen durch die im folgenden beschriebene Erzeugung des Kernels verloren.
Der Kernel unterscheidet sich vom *Flat Schedule* dadurch, dass die berechneten Kontrollschritte modulo des Initiierungsintervalls berechnet werden, wodurch die Verzahnung der Iterationen realisiert wird.

Definition 7.26 (Modulo-Kontrollschritt). Sei $\mathcal{F}$ ein *Flat Schedule* und $p \in \mathcal{F}$ eine Operation aus dem *Flat Schedule*, deren Kontrollschritt im Flat Schedule gemäß Definition 7.5 mit $Cstep_{\mathcal{F}}(p)$ bezeichnet ist. Der **Modulo-Kontrollschritt** $ModCstep_{\mathcal{F}}(p)$ von p für den *Flat Schedule* $\mathcal{F}$ ist definiert als der positive ganzzahlige Rest, der durch die Division von $Cstep(p)$ mit dem Initiierungsintervall II entsteht.

$$ModCstep_{\mathcal{F}}(p) = Cstep_{\mathcal{F}}(p) \bmod II$$

■

Definition 7.27 (Kernel). Sei $\mathcal{F}$ ein *Flat Schedule*. Der **Kernel** $\mathcal{K}$ des *Flat Schedule* $\mathcal{F}$ ist die Sequenz von Instruktionen, die entsteht, wenn jeder Operation $p \in \mathcal{F}$ mit Kontrollschritt $Cstep_{\mathcal{F}}(p)$ der Modulo-Kontrollschritt $ModCstep_{\mathcal{F}}(p)$ zugewiesen wird.
■

Satz 7.5 (Länge des Kernels). *Die Länge des Kernels* $|\mathcal{K}|$ *ist genau die Größe des verwendeten Initiierungsintervall* II*.*

$$|\mathcal{K}| = II$$

■

Gemäß Definition 7.27 wird der **Kernel** in einem iterativen Prozess ausgehend vom leeren Schedule erzeugt, indem nacheinander für jede Operationen p des *Flat Schedule* $\mathcal{F}$ folgende Arbeitsschritte durchgeführt werden:

1. Berechnung von $ModCstep_{\mathcal{F}}(p)$ für die aktuelle Operation p,
2. Suche nach einem gültigen *Issue-Slot* in der dem Modulo Kontrollschritt entsprechenden Instruktion und

3. Anordnen der Operation in diesem *Issue-Slot*.

Abbildung 7.8 verdeutlicht noch einmal die Berechnung des Kernels.
Wie bei der Berechnung des *Flat Schedule* kann auch hier bei VLIW-Prozessoren der Fall auftreten, dass eine Operation in einer Instruktion angeordnet werden soll, in der noch ein *Issue-Slot* frei ist, dieser aber von der anzuordnenden Operation nicht benutzt werden kann. Durch eine Permutation der Bindung der *Issue-Slots* zu den Operationen kann möglicherweise eine gültige Anordnung der Operation gefunden werden. Beispiel 7.4 zeigt eine solche Permutation der Bindungen.
Für den Fall, dass keine gültige Permutation der Issue-Slot Bindung gefunden werden kann, muss eine neue Instruktion hinter der aktuellen eingefügt werden. In dieser wird dann die aktuelle Operation eingefügt. Ein solcher Konflikt, der nicht durch eine Permutation der Issue-Slot Bindung aufgelöst werden kann, verschlechtert allerdings die Qualität des fertigen Schedule wesentlich. Die Ergebnisse in Kapitel 9 zeigen jedoch, dass dieser Fall nicht häufig auftritt.
Man beachte, dass alle Datenabhängigkeiten[13] bereits aufgrund der Berechnung des *Flat Schedule* erfüllt werden.
Aufgrund der Konstruktion des *Flat Schedule* (siehe Abschnitt 7.5) wird versucht, eventuell vorhandene **Delay-Slots** der Verzweigungsoperationen mit anderen Operationen der Schleife zu füllen. Inwieweit dies gelingt, hängt von den Datenabhängigkeiten zwischen den Operationen der Schleife, die die Verzweigungsbedingung berechnen, und den Verzweigungsoperationen selbst ab. Nachdem der Kernel erzeugt wurde, muss nun überprüft werden, ob noch leere Instruktionen am Ende des Kernels eingefügt werden müssen. Die nächste Iteration des Kernels kann nicht ausgeführt werden, bevor bekannt ist, ob die Verzweigung für die Ausführung der nächsten Iteration überhaupt genommen wird oder ob überhaupt aus der Schleife herausgesprungen wird. Gibt es nicht genügend Instruktionen nach der Instruktion, die die letzte Verzweigungs-Operation enthält, so müssen zusätzlich genügend leere Instruktionen an das Ende des Kernels angehängt werden.

7.7 Prolog

Der in Abschnitt 7.6 erzeugte Kernel bearbeitet genau *SC* viele Iterationen der ursprünglichen Schleife parallel während seiner Ausführung. Deshalb kann der Kernel selbst nicht alleine die ursprüngliche Schleife ersetzen. Zunächst muss die Softwarepipeline „gefüllt“werden. Die Anzahl der Iterationen der ursprünglichen Schleife, die im Prolog enthalten sein müssen, ist dabei vom Pipeline-Tiefe des Kernels abhängig:

Satz 7.6 (Iterationen im Prolog). *Sei $\mathcal{K}$ ein Kernel gemäß Definition 7.27 und SC seine Pipeline-Tiefe. Die Anzahl der Iterationen, die zur Ausführung des Kernels*

[13] bis auf die in Abschnitt 7.9 beschriebene Ausnahme

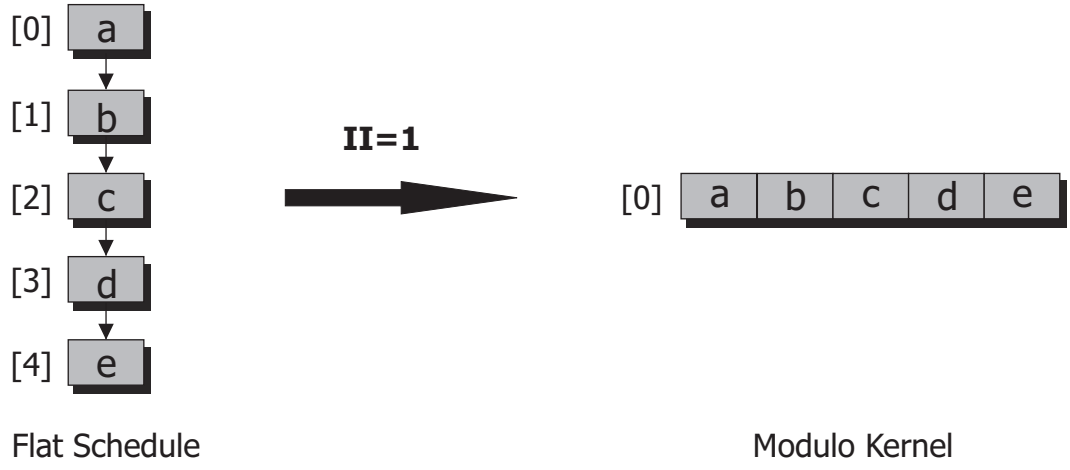


Abbildung 7.8: Berechnung Modulo Kernel

begonnen werden müssen, berechnet sich dann als

$$SC - 1.$$

■

Beweis. Da bei jeder Ausführung des Kernels auch eine Iteration der ursprünglichen Schleife begonnen wird und im Kernel SC viele Iterationen der ursprünglichen Schleife enthalten sind, müssen vor der ersten Ausführung des Kernels $SC-1$ viele Iterationen gestartet werden. ■

Bemerkung. Es gibt zu jeden Kernel exakt einen eindeutig bestimmten Prolog unter der Annahme, dass die ursprüngliche Schleife nur einen Eintrittspunkt besitzt. Dies ist bei übersetzer-generiertem Code der Fall.

Während der Ausführung des Kernels befindet sich in jedem seiner SC vielen Stufen eine andere Iteration der ursprünglichen Schleife. Daraus folgt, dass die Iterationen im Prolog auch zu unterschiedlichen Zeitpunkten begonnen werden müssen, um bei Start der Ausführung des Kernels in der richtigen Pipeline-Stufe zu sein.
Das bedeutet, dass man $SC - 1$ Mal die Operationen aus der ersten Stufe des *Flat Schedule* benötigt, $SC - 2$ Mal die Operationen aus der zweiten Stufe, u.s.w.

Satz 7.7. *Sei $\mathcal{K}$ der Kernel für den* Flat Schedule *$\mathcal{F}$ und SC bezeichne die Pipeline-Tiefe für $\mathcal{K}$. Ferner seien die Stufen in $\mathcal{F}$ in der Reihenfolge ihres Auftretens in $\mathcal{F}$ beginnend mit 1 nummeriert, wobei die i-te Stufe mit S_i bezeichnet ist. Der zu erzeugende Prolog für $\mathcal{K}$ muss $SC - i$ Kopien der Operation aus S_i enthalten.* ■

Beweis. Dies folgt direkt aus der Tatsache, dass der Prolog $SC - 1$ Iterationen starten muss und dass die im Kernel zuerst ausgeführten Operationen dementsprechend häufiger benötigt werden. ■

Damit kann man den Prolog aus dem Kernel wie folgt erzeugen:

Definition 7.28 (Prolog). Sei $\mathcal{K}$ ein Kernel nach Definition 7.27 und SC die Pipeline-Tiefe für $\mathcal{K}$. Der eindeutige **Prolog** $\mathcal{P}$ für $\mathcal{K}$ ist die Hintereinanderausführung von $SC - 1$ Kopien des Kernels, die durch folgende Arbeitschritte entstehen:

1. Für jede Operation von $\mathcal{K}$ wird ein Zähler eingeführt, wobei der Zähler der Operationen aus S_i in $\mathcal{F}$ mit $SC - i$ initialisiert wird.

2. Der Kernel wird $SC - 1$ Mal kopiert und die Operationen, deren Zähler kleiner gleich Null geworden ist, werden aus der aktuellen Kopie entfernt.

3. Die Kopien werden in umgekehrter Reihenfolge ihrer Erzeugung ausgeführt.

■

Abbildung 7.9 zeigt das Ergebnis des Algorithmus auf einen Kernel bestehend aus einer Instruktion mit insgesamt fünf Operationen.

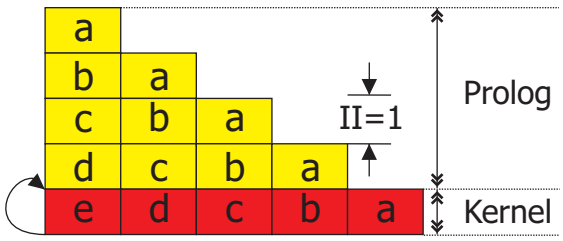


Abbildung 7.9: Berechnung des Prologes aus dem Kernel

Im Gegensatz zu dem hier vorgestellen Verfahren, muss der Prolog bei *Kernel Recognition*-Verfahren nicht explizit erzeugt werden, sondern wird automatisch durch die Berechnung des Kernels erzeugt. Dies liegt darin begründet, dass *Kernel Recognition*-Verfahren umgekehrt vorgehen wie in der hier beschriebenen Methode. Sie klappen die Schleife aus und suchen nach einem sich wiederholenden Muster von Operationen. Hat man dieses Muster gefunden, stellt der aufgeklappte Code vor dem Kernel-Muster den Prolog dar.

7.8 Epiloge

Ebenso wie der Kernel einen Prolog benötigt, wird auch ein sog. **Epilog** gebraucht, der die Softwarepipeline nach der letzten Ausführung des Kernels wieder leert. Allerdings kann eine vom Übersetzer generierte Schleife mehrere Austrittspunkte haben, weshalb mehrere Epiloge für einen Kernel erzeugt werden müssen.
Wird die Schleife am Ende des Kernels bzw. am Ende des Prologes beendet, so befindet man sich im dem Zustand einer gefüllten Pipeline, d.h. zu diesem Zeitpunkt befinden sich SC viele Iterationen der ursprünglichen Schleife in der Ausführung. Der zugehörige Epilog bezeichnet man als **kompletten Epilog**.
Ähnlich wie bei der Erzeugung des Prologes (vgl. Abschnitt 7.7) werden auch hier abhängig von der Pipeline-Tiefe unterschiedlich viele Operationen des Kernels benötigt, um die Pipeline zu leeren:

Satz 7.8 (Iterationen im kompletten Epilog). *Sei $\mathcal{K}$ ein Kernel gemäß Definition 7.27 und SC seine Pipeline-Tiefe. Die Anzahl der Iterationen, die im kompletten Epilog beendet werden müssen, berechnet sich als*

$$SC - 1.$$

■

Beweis. Da bei jeder Ausführung des Kernels auch eine Iteration der ursprünglichen Schleife beendet wird und im Kernel SC viele Iterationen der ursprünglichen Schleife enthalten sind, müssen $SC - 1$ viele Iterationen beendet werden. ■

Satz 7.9. *Sei $\mathcal{K}$ der Kernel für den* Flat Schedule *$\mathcal{F}$ und SC bezeichnet die Pipeline-Tiefe für $\mathcal{K}$. Ferner seien die Stufen in $\mathcal{F}$ in der Reihenfolge ihres Auftretens in $\mathcal{F}$ beginnend mit 0 nummeriert, wobei die i-te Stufe mit S_i bezeichnet ist. Der zu erzeugende Prolog für $\mathcal{K}$ muss i Kopien der Operation aus S_i enthalten.* ■

Beweis. Dies folgt direkt aus der Tatsache, dass der komplette Epilog $SC-1$ Iterationen beenden muss und dass die im Kernel zuerst ausgeführten Operationen dementsprechend weniger häufig benötigt werden. ■

Damit kann man den kompletten Epilog aus dem Kernel wie folgt erzeugen:

Definition 7.29 (kompletter Epilog). Sei $\mathcal{K}$ ein Kernel nach Definition 7.27 und SC die Pipeline-Tiefe für $\mathcal{K}$. Der eindeutige **komplette Epilog** $\mathcal{E}$ für $\mathcal{K}$ ist die Hintereinanderausführung von $SC-1$ Kopien des Kernels, die durch folgende Arbeitschritte entstehen:

1. Für jede Operation von $\mathcal{K}$ wird ein Zähler eingeführt, wobei der Zähler der Operationen aus S_i in $\mathcal{F}$ mit i initialisiert wird. Man beachte, dass die Nummerierung im Gegensatz zum Prolog mit Null beginnt.

2. Der Kernel wird $SC - 1$ Mal kopiert und die Operationen, deren Zähler kleiner gleich Null geworden ist, werden aus der aktuellen Kopie entfernt.

3. Die Kopien werden in der Reihenfolge ihrer Erzeugung ausgeführt.

■

Abbildung 7.10 zeigt das Ergebnis der Erzeugung eines kompletten Epiloges aus einem Kernel. Besitzt die ursprüngliche Schleife Austrittpunkte, die nicht am Anfang oder

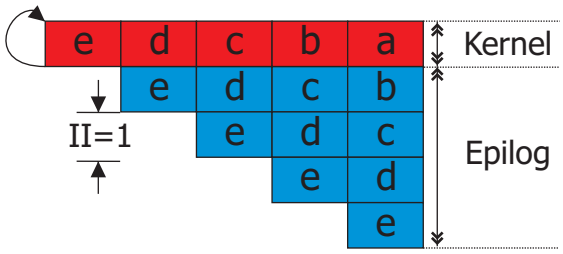


Abbildung 7.10: Berechnung des kompletten Epiloges

Ende einer Iteration liegen, so muss für jeden möglichen Austrittspunkt ein eigener sog. **partieller Epilog** erzeugt werden. Hier muss man unterscheiden, wo die optimierte Schleife verlassen wird und für jeden Austrittspunkt die zu dem Zeitpunkt noch nicht beendeten Iterationen vervollständigen. Die Berechnung der partiellen Epiloge ist wesentlich komplexer als die Erzeugung des kompletten Epiloges, da zuerst berechnet werden muss, welche Iterationen noch in Ausführung sind und welche Codesequenzen zu deren Beendigung benötigt werden. Muss zusätzlich eine Modulo-Registerexpansion, MRE, (vgl. Abschnitt 7.9) durchgeführt werden, so können die partiellen Epiloge erst danach erzeugt werden, da durch die MRE der Kernel modifiziert (aufgeklappt) wird und dadurch bei Austrittspunkten im Kernel selbst andere partielle Epiloge erzeugt werden müssen als das ohne die MRE der Fall gewesen wäre. Die notwendigen Codeschemata zur Erzeugung der Epiloge für alle möglichen Austrittspunkte der optimierten Schleife sind in [R+92] dargestellt. Hierbei wird deutlich, dass jeder Epilog einen Suffix besitzt, der gleich dem Suffix des kompletten Epiloges ist. Die gleichen Enden sind also redundant und können durch eine Verzweigung zum entsprechenden Suffix des kompletten Epiloges ersetzt werden. Für weitere Erläuterungen zum Erzeugen der partiellen Epiloge wird auf [R+92] verwiesen.

7.9 Modulo-Registerexpansion

Betrachtet man die Register in der ursprünglichen Schleife, so stellt man fest, dass sie oft Werte beinhalten, deren Lebensspannen größer sind als das zur Berechnung des Kernels verwendete Initiierungsintervall. Dies führt zu dem in Abschnitt 5.3.2 beschriebenen Problem, dass Registerwerte bei einem sehr kleinen Initiierungsintervall durch die nächste Iteration überschrieben werden, obwohl der alte Wert in der aktuellen Iteration noch benötigt wird.
Um gültigen Code zu erzeugen, muss der Kernel bei konsistenter Umbenennung solcher schleifenvarianten Register aufgeklappt werden.

Definition 7.30 (schleifenvariantes Register). Sei $G_{dd} = (N_{dd}, E_{dd})$ der Datenabhängigkeitsgraph einer Schleife L, $\mathcal{F}$ der aus L berechnete *Flat Schedule* und r ein Register. Das Register r ist genau dann ein **schleifenvariantes Register**, wenn es zwei Operationen i, j in L gibt, für die folgendes gibt:

- $\exists e = (i, j, r, t) \in E_{dd} : Dist(e) = 0$, Setzung-Benutzung-Abhängigkeit und
- $\exists e' = (j, i, r, a) \in E_{dd} : Dist(e') = 1$, Benutzung-Setzung-Abhängigkeit.

Die Menge aller schleifenvarianten Register sei mit $\mathcal{R}$ bezeichnet. ■

Definition 7.31 (Lebensspanne eines schleifenvarianten Registers). Sei $G_{dd} = (N_{dd}, E_{dd})$ der Datenabhängigkeitsgraph einer Schleife L, $\mathcal{F}$ der *Flat Schedule* für L und $\mathcal{R}$ die Menge aller schleifenvarianten Register in $\mathcal{F}$. Die **Lebensspanne** $LifeSpan_{\mathcal{F}}(r)$ eines schleifenvarianten Registers $r \in \mathcal{R}$ bezüglich des Flat Schedules $\mathcal{F}$ ist definiert als die maximale Differenz der Kontrollschritte zwischen zwei Operationen i, j, die eine Benutzung-Setzung-Abhängigkeit bezüglich r definieren.

$$LifeSpan_{\mathcal{F}}(r) = \max_{e=(i,j,r,t)\in E_{dd}} (|Cstep_{\mathcal{F}}(i) - Cstep_{\mathcal{F}}(j)|)$$

Hierbei bezeichnet $Cstep_{\mathcal{F}}(i)$ den Kontrollschritt einer Operation i gemäß Definition 7.5. ■

Das Maximum der Lebensspannen aller schleifenvarianten Register stellt nun eine untere Schranke für die Größe des Kernels dar. Daraus kann man nun den Faktor berechnen, wie oft der Kernel aufgeklappt werden muss, um diese Größe zu erreichen.

Definition 7.32 (minimaler Unroll-Faktor). Sei $\mathcal{R}$ die Menge aller schleifenvarianten Register einer Schleife L, deren *Flat Schedule* $\mathcal{F}$ und Kernel $\mathcal{K}$ sei. II bezeichne dabei das für die Berechnung von $\mathcal{F}$ verwendete Initiierungsintervall. Der **minimale**

Unroll-Faktor $u_{min}(\mathcal{K})$ für den Kernel $\mathcal{K}$ ist definiert als das Maximum der aufgerundeten Quotienten, die aus der Division der Lebensspannen aller schleifenvarianten Register r und dem Initiierungsintervall II entstehen.

$$u_{min}(\mathcal{K}) = \max_{r \in \mathcal{R}} \left(\left\lceil \frac{LifeSpan_{\mathcal{F}}(r)}{II} \right\rceil \right)$$

■

Hat man nun berechnet, wie oft der Kernel aufgeklappt werden muss, um mindestens die Größe der längsten Lebensspanne eines schleifenvarianten Registers zu haben, reicht das aber noch nicht aus, da der aufgeklappte Kernel immer noch die gleichen Register benutzt. Hier müssen in allen durch das Aufrollen entstandenen Kopien des ursprünglichen Kernels andere noch freie Register verwendet werden. Dies erreicht man durch konsistente Umbenennung, auch **Register-Renaming** ([Lam88]) genannt.
Die Umbenennung der schleifenvarianten Register erfolgt derart, dass aufeinanderfolgende Definitionen eines schleifeninvarianten Registers nach der Umbenennung verschiedene Register benutzen. Hat man den Kernel n Mal aufgeklappt, so muss man in den drei n erzeugten Kopien des Kernels jeweils alle schleifenvarianten Register umbenennen. Dazu muss man für jedes schleifenvariante Register r folgendes tun:

1. Umbenennung aller Definitionen von r in $r^\star$, wobei $r^\star$ ein neues freies Register ist.

2. Identifikation der Position der ersten Redefinition von r innerhalb der erzeugten Kopie.

3. Umbenennung aller Benutzungen von r vor dieser ersten Redefinition werden in eine Benutzung von r' umbenannt, wobei r' das umbenannte Register in der vorherigen Kopie ist. Ist die aktuelle Kopie die erste erzeugte, so ist $r' = r$.

4. Umbenennung aller Benutzungen von r nach der ersten Redefinition werden in eine Benutzung von $r^\star$ umbenannt.

Bemerkung. Bei einer VLIW-Instruktion, in der sowohl eine Benutzung als auch eine Definition eines schleifenvarianten Registers r vorkommen, werden alle Benutzungen als vor der Definition von r vorkommend betrachtet.

Das beschriebene Umbenennungsverfahren für Register erweitert ein schleifenvariantes Register im Kernel zu einem Vektor von Registern, wobei in jeder erzeugten Kopie ein anderes Register verwendet wird. Dies kommt einer Emulation von sog. **rotierenden Registerbänken** nahe, sodass man diese softwareseitigen vektorisierten Register auch als **rotierende Software-Registerbänke** bezeichnen kann.

Durch das Aufrollen des Kernels wächst seine Größe um den Faktor $u_{min}(\mathcal{K})$. Dieses Anwachsen des Kernelcodes kann durch die Verwendung von rotierenden Registerbänken vermieden werden (vgl. Abschnitt 5.5.2). Gleichzeitig mit dem Anwachsen des Kernels wächst aber auch die Pipeline-Tiefe des neuen Kernels.

Satz 7.10. *Die Länge des Kernels $|\mathcal{K}|$ nach der Anwendung von MRE berechnet sich durch*

$$|\mathcal{K}_{MRE}| = |\mathcal{K}| * u_{min}(\mathcal{K}).$$

Gleichzeitig steigt die Pipeline-Tiefe SC um den gleichen Faktor.

$$SC_{MRE} = SC * u_{min}(\mathcal{K})$$

■

7.10 Vorschalten der Originalschleife

Wie bereits in Abschnitt 5.3.2 erwähnt wurde, kann man mit der optimierten Schleife nicht mehr eine beliebige Anzahl von Iterationen der ursprünglichen Schleife ausführen.

Satz 7.11 (Einschränkung der Iterationszahl). *Sei $\mathcal{K}$ der Kernel einer Schleife, SC seine Pipeline-Tiefe und $u_{min}(\mathcal{K})$ der für die Modulo Register Expansion notwendige Unroll-Faktor. Dann gilt für die optimierte Schleife (Prolog,Kernel,Epilog) möglichen Iterationszahlen $N_{\mathcal{K}}$ der ursprünglichen Schleife:*

$$N_{\mathcal{K}} = u_{min}(\mathcal{K}) * i + (SC - 1)\,,$$

für ein $i \geq 0$. ■

$SC - 1$ viele Iterationen werden durch den Prolog gestartet, da der Kernel SC viele Iterationen der ursprünglichen Schleife gleichzeitig bearbeitet. Nach jeder Ausführung des Kernels wurde eine weitere Iteration gestartet und eine Iteration wurde beendet. Will man eine von dieser Form abweichende Anzahl an Iterationen der ursprüngliche Schleife ausführen, so muss man genügend viele Iterationen der ursprünglichen Schleife vorher ausführen bis die verbleibende Iterationszahl von der angegebenen Form ist, damit dann die optimierte Schleife ausgeführt werden kann. Man schaltet also die Originalschleife der optimierten vor.
Die Anzahl der Iterationen, die man mittels der Originalschleife vorschalten muss, lässt sich wie folgt berechnen:

Satz 7.12 (Iterationszahlen). *Sei N_{all} die Anzahl der gewünschten Schleifeniterationen in der Eingabeschleife L und sei $\mathcal{K}$ der Kernel, der aus L berechnet wurde. SC sei die Pipeline-Tiefe von $\mathcal{K}$ und $u_{min}(\mathcal{K})$ der Unroll-Faktor für $\mathcal{K}$. Die Anzahl der*

Iterationen N_{prec}, die der Ausführung der optimierten Schleife vorgeschaltet werden müssen berechnet sich durch

$$N_{prec} = \begin{cases} N_{all} & N_{all} < SC - 1 \\ [N_{all} - (SC-1)] \bmod u_{min}(\mathcal{K}) & sonst \end{cases}.$$

Die Anzahl der Iterationen $N_{\mathcal{K}}$, die dann in der optimierten Schleife ausgeführt werden, ist die Differenz aus der gewünschten Anzahl an Gesamtiterationen und der Anzahl an vorgeschalteten Iterationen.

$$N_{\mathcal{K}} = N_{all} - N_{prec}$$

■

Die vorgeschalteten N_{prec} Iterationen werden dabei verhältnismäßig langsam, da nicht optimiert, ausgeführt. Die restlichen $N_{\mathcal{K}}$ Iterationen hingegen profitieren dann voll von der extrahierten Parallelität.

7.11 Basisblockstruktur

Nachdem nun die einzelnen Komponenten (Prolog, Kernel und die Epiloge) für eine Schleife berechnet und erzeugt wurden, muss man diese in die ursprüngliche Basisblockstruktur der Schleife einsetzen.
Durch die Anwendung der *IF-Conversion* (vgl. Abschnitt 7.1.6) wurde aus der Eingabeschleife mit den in ihr vorhandenen Basisblöcken ein einziger großer Basisblock. Dieser diente als Eingabe für die Berechnungen von Initiierungsintervall, Flat Schedule, etc. Die Schleife nach dem Pipelining hat eine etwas andere Struktur. Sie besteht aus drei Blöcken, jeweils einem für Prolog, Kernel und kompletten Epilog. Besitzt die Schleife mehrere Austrittspunkte, die nicht am Ende des Kernels bzw. am Ende des Prologes liegen, kommen noch weitere Blöcke für die partiellen Epiloge (vgl. Abschnitt 7.8) hinzu. Abbildung 7.11 zeigt die Transformation der Basisblockstruktur. Um diese Blockstruktur herzustellen, werden die ursprünglich vorhandenen Basisblöcke der Schleife aus dem Kontrollfluss entfernt und durch die neuen Blöcke ersetzt. Wird die Originalschleife weiterhin benutzt, um die gewünschte Anzahl an Iterationen der ursprünglichen Schleife zu erreichen (vgl. Abschnitt 7.10), so kommt zusätzlich noch ein Basisblock hierfür hinzu.
Die einzelnen Blöcke bestehen immer noch aus prädikativen Operationen aufgrund der zu Anfang des Verfahrens durchgeführten *IF-Conversion* (vgl. Abschnitt 7.1.6). In Abschnitt 7.12 wird erläutert, dass eventuell als Abschluss des Verfahrens diese prädikative Form wieder entfernt werden muss. In dem Fall werden die zuvor in Datenabhängigkeiten transformierten Kontrollabhängigkeiten wiederhergestellt, weshalb

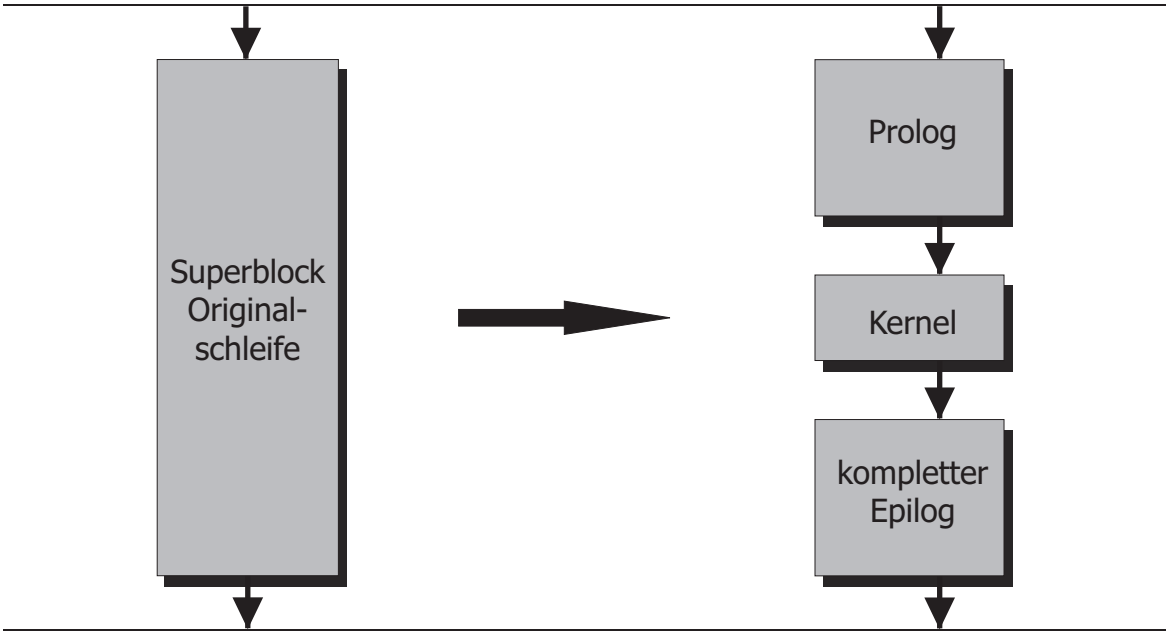


Abbildung 7.11: Transformation der Basisblockstruktur

innerhalb der neu erzeugten Basisblöcke wiederum neue Blöcke entstehen können.

7.12 Revertierung der IF-Conversion

Wenn die Zielarchitektur keine prädikative Ausführung unterstützt (vgl. Abschnitt 5.5.1), muss zum Abschluss des Verfahrens diese prädikative Form wieder entfernt werden. Das bedeutet die Rücktransformation von Kontrollabhängigkeiten aus Datenabhängigkeiten. Ein Algorithmus ist in [WMHR93] aufgeführt.

7.13 Charakteristische Eigenschaften des Postpass-Ansatzes

Da das Verfahren in einem Postpass-Ansatz angewandt wird und dadurch die Eingabe nicht in Form einer *high-level* Darstellung (wie im Übersetzer) erfolgt sondern als Assemblerprogramm, gibt es einige Besonderheiten für das Verfahren, die es zu dem von Rau in [Rau94] vorgestellten abgrenzen.

7.13.1 Integration in den umgebenden Kontrollfluss

Wendet man Iterative Modulo Scheduling in der Optimierungsphase eines Übersetzers an, so brauchen die notwendigen Verzweigungsoperationen erst nach Abschluss des Verfahrens erzeugt zu werden. Auf der Assemblerebene existieren bereits Verzweigungen, die den Kontrollfluss der Originalschleife implementieren. Nach dem Pipelining der Schleife sind die Verzweigungsziele dieser Operationen allerdings nicht mehr gültig, da sich die Struktur der Schleife durch die Optimierung geändert hat. In Abschnitt 7.11 wurde beschrieben, welche Basisblockstruktur die Verzweigungsoperationen in der optimierten Schleife definieren müssen. Beispielsweise sind durch die Änderungen in der Basisblockstruktur die Sprungziele zurück zum Schleifenanfang nicht mehr gültig und müssen angepasst werden. Gleiches gilt prinzipiell für alle Sprünge innerhalb der Schleife. Zusätzlich müssen neue Verzweigungsoperationen in den Code eingefügt werden, so z.B. um nach Ausführung des Prologes in den Kernel zu verzweigen. Man kann das mit dem Entfernen und Einfügen von Elementen in eine doppelt verkettete Liste vergleichen. So sind hier alle Basisblöcke durch Verzweigungen miteinander verknüpft, was in der Liste der Verzeigerung entspricht. Diese muss nun gültig transformiert werden.
Konkret macht das folgende Änderungen an den Sprungzielen bestehender Verzweigungsoperationen notwendig:

- rückwärts-gerichtete Verzweigungsoperationen:
  alle Verzweigungsoperationen, die in der Originalschleife eine neue Iteration starten, müssen nun auf den Start des Kernels zeigen.

- Schleifenaustrittsverzweigungen:
  alle Verzweigungsoperationen, die die Ausführung der Originalschleife beenden, müssen nun auf den Start des kompletten Epiloges zeigen bzw. auf den entsprechenden partiellen Epilog, falls es sich um einen sog. vorzeitigen Austritt aus der Schleife handelt (siehe Abschnitt 7.8).

- Schleifeneintrittsverzweigungen:
  Schleifeneintrittsverzweigungen sind Verzweigungsoperationen, die die Schleife starten. Beim Modifizieren dieser Verzweigungen muss auch berücksichtigt werden, dass die Schleife theoretisch von allen Stellen des Programmes angesprungen werden kann, d.h. alle Verzweigungen, die im Originalprogramm an den Start der Originalschleife verzweigen, müssen nun auf den Prolog bzw. den Start der vorgeschalteten Kopie der Originalschleife (vgl. Abschnitt 7.10) zeigen.

7.13.2 Kontrollflussrekonstruktion

Wie in Abschnitt 7.1.2 beschrieben ist der Kontrollfluss eines Programmes auf Assemblerebene nicht mehr explizit durch die Operationen ausgedrückt sondern implizit

durch die verschiedenen Verzweigungsoperationen umgesetzt.
Basis aller Kontrollfluss-sensitiven Analysen und Optimierungen ist der Kontrollflussgraph, weshalb dieser aus dem vorliegenden Assemblercode rekonstruiert werden muss. Dieser Prozess ist zum Teil sehr komplex und schwierig, da die Sprungziele der Verzweigungsoperationen aufgelöst werden müssen. Dies ist einfach bei absoluten Sprüngen und Prozeduraufrufen, deren Argumente Sprungmarken sind, da hier die Ziele statisch berechenbar sind. Übersetzer generieren allerdings auch sog. *Kontrollflussindirektionen*. Das sind Sprünge, deren Argumente Register sind. Um das Sprungziel zu ermitteln, muss der Inhalt des Registers bekannt sein, wofür man Informationen über die dynamischen Eigenschaften der Eingabeprogramme benötigt. Diese Informationen sind für eine statische Analyse nicht immer vorhanden ([Wil01]).
Die Generierung solcher Kontrollflussindirektionen ist notwendig, wenn die Verzweigung mehr als ein Sprungziel haben kann. Dies ist in prozeduralen Sprachen typischerweise bei der Übersetzung von `switch-case`-Konstrukten und Prozedurzeigern, d.h. Verweisen auf Prozeduren, der Fall. Bei diesen Sprachelementen wird das Ziel der Verzweigung dynamisch zur Laufzeit berechnet, wofür Übersetzer entsprechenden Assemblercode generieren. Deshalb werden solche Verzweigungen auch *berechneter* Sprung bzw. *berechneter* Prozeduraufruf genannt.
In [Wil01, The00] sind diese Problematiken detailliert und mit Beispielen beschrieben.

7.13.3 Registerallokation und Registerzuweisung

Die Registerallokation ist während des Postpass-Schedulings bereits abgeschlossen. Das bedeutet, die Entscheidung, welche Werte in Registern und welche im Speicher gehalten werden, ist vom Übersetzer bereits getroffen worden. Die Registerzuweisung kann allerdings im Postpass-Ansatz noch verändert werden, was in der Modulo-Registerexpansion (vgl. Abschnitt 7.9) auch notwendig ist.
In der Modulo-Registerexpansion werden Register umbenannt und auch zusätzliche Register verwendet. Diese dürfen keine Werte beinhalten, die während der Ausführung des Programmes noch benötigt werden.
Die einfachste Lösung ist es, Register zu verwenden, die im ganzen Programm noch nicht benutzt wurden. Für diese ist klar, dass ihre Inhalte nicht mehr von anderen Operationen gelesen werden. Allerdings kann man durch eine *Lebendigkeitsanalyse* eventuell noch weitere Register identifizieren, die zwar im Programm bereits benutzt werden, aber keine Informationen enthalten, die überschrieben werden könnten. Dazu berechnet man für jeden Programmpunkt, welche Register an diesem Punkt lebendig sind.

Definition 7.33 (Lebendigkeit eines Registers). Sei $G_{cf} = (N_{cf}, E_{cf})$ der Kontrollflussgraph eines Programmes gegeben. Ferner bezeichne Π die Menge aller Pfade in G_{cf} und r ein Register des Zielprozessors. Die Mengen $Uses(r)$ und $Defs(r)$ beinhalten alle Programmpunkte, an denen das Register r benutzt bzw. definiert wird.

Das Register r ist an einem Programmpunkt $p \in N_{cf}$ **lebendig**, wenn es einen Pfad $\pi = p_1, p_2, \ldots, p_k \in \Pi$ in G_{cf} gibt, der die folgenden Eigenschaften aufweist:

- π beginnt mit einer Definition von r:

$$p_1 \in \mathit{Defs}(r)$$

- π endet mit einer Benutzung von r:

$$p_k \in \mathit{Uses}(r)$$

- π ist setzungs- und benutzungsfrei bezüglich r:

$$\forall p_i, 1 < i < k : p_i \notin \mathit{Uses}(r) \wedge p_i \notin \mathit{Defs}(r)$$

■

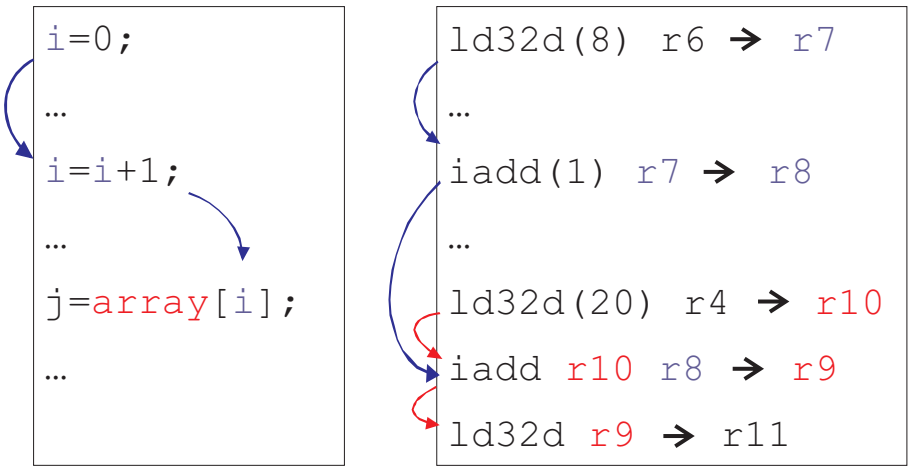
Die Datenflussanalyse zur Bestimmung der lebendigen Register für alle Programmpunkte eines Programmes nennt man **Live Variables Analyse** und ist in [NNH99] genau beschrieben.
Für die *Modulo-Registerexpansion* kann man nun Register verwenden, die in der Schleife nicht lebendig sind.

7.13.4 Datenabhängigkeiten auf Assemblerebene

Vergleicht man die Datenabhängigkeitsgraphen eines Programmes für seine Repräsentationen in der Hochsprache und auf Assemblerebene, so stellt man fest, dass der Datenabhängigkeitsgraph für den Assemblercode zusätzliche Abhängigkeiten enthält. Dies ist eine Folge aus der Übersetzung von Hochsprachekonstrukten in äquivalente Assembleroperationen, da hier keine höherstufigen Datenstrukturen mehr vorhanden sind sondern nur noch auf Register- bzw. Speicherinhalten gearbeitet wird. Außerdem wird häufig eine einzige Anweisung der Hochsprache in mehrere Assembleroperationen übersetzt.
Beispiel 7.6 verdeutlicht die Zunahme von Datenabhängigkeiten bei der Übersetzung von Hochspracheprogrammen selbst für einfache Beispiele.

Beispiel 7.6. Abbildung 7.12 (a) zeigt Ausschnitte eines C-Programmes, in dem ein Zugriff auf eine Feld-Datenstruktur durchgeführt wird. Die Pfeile markieren die vorhandenen Datenabhängigkeiten zwischen den Anweisungen, die durch die Setzungen und Benutzungen der Variable i entstehen (jeweils Setzung-Benutzung-Abhängigkeit). In Abbildung 7.12 (b) sind äquivalente Assembleroperationen zu den C-Anweisungen aus Abbildung 7.12 (a) dargestellt. `ld32d` liest Daten aus dem Speicher in ein Register

ein und `iadd` addiert eine Konstante auf ein Register und speichert das Ergebnis in ein Register. Die Datenabhängigkeiten aus Abbildung 7.12 (a) sind auch im Assemblercode enthalten. Zusätzlich entstehen aber noch zwei weitere Abhängigkeiten (für Register `r9` und `r10`). Diese beiden Register sind notwendig, um die Basisadresse (`r10`) des Feldes und die spezifische Adresse des Feldelementes (`r9`) zu berechnen. ■



(a) DDG für Hochsprache-programm

(b) DDG für Assemblerprogramm

Abbildung 7.12: Zunahme von Datenabhängigkeiten auf Assemblerebene

Abbildung 8.2 (b) in Abschnitt 8.3 zeigt den Datenabhängigkeitsgraphen einer Matrixmultiplikation. Der Graph zeigt, dass bereits für 29 Operationen 231 Datenabhängigkeiten entstehen können.

Kapitel 8

Implementierung

Dieses Kapitel beschreibt die prototypische Implementierung des in Kapitel 7 erläuterten Verfahrens Iterative Modulo Scheduling.
Die Implementierung wurde als Postpassoptimierung am Beispiel des in Abschnitt 3.2 beschriebenen TriMedia TM1000 VLIW-Prozessors durchgeführt. Grundlage ist das in Kapitel 6 beschriebene PROPAN-Framework, mit welchem die notwendigen Programmdarstellungen erzeugt werden können. Durch dieses Setup war es möglich, einen generischen Softwarepipeliner zu entwickeln, der aufbauend auf einer TDL-Beschreibung (siehe Abschnitt 6.2) und einem Eingabeprogramm auf Assemblerebene Iterative Modulo Scheduling auf die Schleifen des Eingabeprogrammes anwenden kann.
In Abschnitt 8.1 wird die Integration des Verfahrens in das PROPAN-Framework beschrieben, Abschnitt 8.2 beschreibt einige wichtige Komponenten des Verfahrens selbst und Abschnitt 8.3 die möglichen Visualisierungen der generierten Programmdarstellungen und Modulo Schedules. Das Kapitel schließt dann mit einem Abschnitt über die Berechnungskomplexität des implementierten Verfahrens.

8.1 Integration in PROPAN

In Kapitel 6 ist das PROPAN-Framework als generisches Rahmenwerkzeug zur Erstellung von Postpass-Optimierern und -Analysatoren beschrieben und erläutert. Darauf aufbauend kann aus einer TDL-Spezifikation ein prozessorspezifischer Modulo Scheduler erzeugt werden, was am Beispiel des TriMedia TM1000 durchgeführt wurde.
Die Optimierung eines Eingabeprogrammes (Assembler) läuft dabei in folgenden Schritten ab:

1. Erzeugung von prozessorspezifischem Assemblerparser und Datenstrukturen
2. Rekonstruktion des Kontrollflussgraphen
3. Generierung notwendiger Programmdarstellungen
4. Anwendung von Modulo Scheduling auf die Schleifen im Kontrollflussgraphen
5. Visualisierung (optional)
6. Rekonstruktion der optimierten Assemblerdatei

Abbildung 8.1 zeigt den schematischen Aufbau des PROPAN-Frameworks samt der Integration des generischen Modulo Schedulers. Der oben beschriebene Ablauf der Postpassoptimierung ist ebenfalls in dem Schema erkennbar. Die Implementierung des PROPAN-Frameworks selbst ist samt der Integration des Modulo Schedulers als Bibliothek gehalten, sodass die Anwendung des Verfahrens sehr flexibel ist und z.B. auf bestimmte Programmteile beschränkt werden kann.

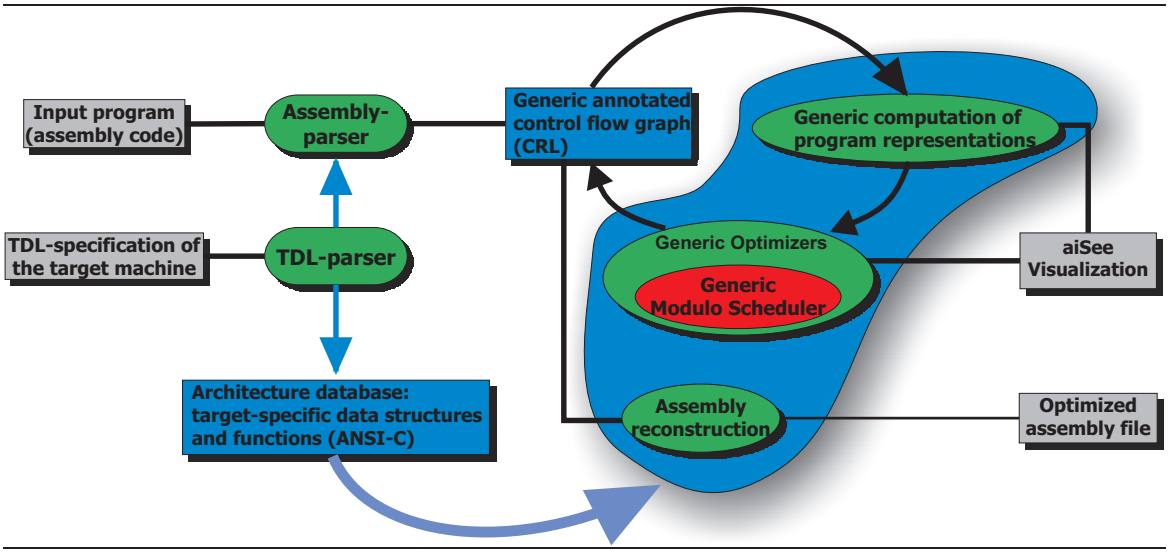


Abbildung 8.1: Struktur PROPAN-Framework mit integriertem Modulo Scheduler

8.2 Iterative Modulo Scheduling

8.2.1 Ressourcenbasiertes Initiierungsintervall

Wie in Abschnitt 7.3.1 beschrieben wird das ressourcenbasierte Initiierungsintervall durch eine Heuristik approximiert, da die Berechnung des exakten Wertes eine Berechnung der exakten Lösung des *Scheduling of Independent Tasks* Problemes erforderlich machen würde.
Eine vereinfachte Darstellung der implementierten Heuristik ist in Listing 8.1 dargestellt. Zuerst werden die Operationen in absteigender Anzahl der Issue-Slots, auf denen sie ausgeführt werden können, sortiert. Dies geschieht vor dem Hintergrund, dass eine Operation, die nur auf wenigen Issue-Slots ausgeführt werden kann, weniger Spielraum zum Anordnen zulässt. Im späteren Verlauf des Algorithmus hat man aufgrund der bereits angeordneten Operationen weniger freie Plätze und man kann dann Operationen, die auf möglichst vielen Issue-Slots ausgeführt werden können, besser anordnen.
Anschließend wird für jede Operation der Schleife in der berechneten Reihenfolge versucht, eine gültige Instruktion zu finden, wobei immer mit der ersten Instruktion begonnen wird. Dadurch wird versucht, jede Operation zum frühest möglichen Zeitpunkt anzuordnen. Zur Suche nach einer gültigen Instruktion iteriert man über alle bisher erzeugten Instruktionen und sucht in jeder nach einem gültigen Issue-Slot. Diese Suche wiederum betrachtet die Ressourcenbenutzungen der bisher angeordneten Operationen (nicht nur in der aktuellen Instruktion), um festzustellen, ob ein Ressourcenkonflikt entstehen würde. Besteht keiner, wird die Operation in den Issue-Slot

```
int ComputeRessourceMII ( list operations )
{
  array instructions;

  // sortiere die Operationen in absteigender Anzahl an Issue-
     Slots
  // auf denen sie ausgeführt werden können
  Sort(operations);

  // Anfangszustand
  instructions[0] = CreateEmptyInstruction();

  for each op in operations do {
    int i = 0;
    instruction ins = instructions[0];
    issue_slot s = NULL;

    // Suche nach gültiger Instruktion
    do {

      // erzeuge neue Instrution falls notwendig
      if ( NULL == ins ) {
        instructions[i] = CreateEmptyInstruction();
      }

      // Suche nach gültigem Issue-Slot in aktueller
         Instruktion
      s = SearchFeasibleIssueSlot(i,op);

      // betrachte nächste Instruktion
      ins = instructions[++i];

    } until ( s != NULL );

  }

  return MaxIndex(instructions);
}
```

Listing 8.1: Berechnung MII_{res}

aufgenommen. Dabei wird Buch über die Ressourcenbenutzungen zu den verschiedenen Kontrollschritten geführt, um die Suche nach Ressourcenkonflikten zu beschleunigen. Kann in den bisher erstellten Instruktionen keine gültige für die aktuell anzuordnende Operation gefunden werden, so wird eine neue leere Instruktion erzeugt, in der dann die aktuelle Operation platziert wird.

8.2.2 Datenabhängigkeitsbasiertes Initiierungsintervall

Die Implementierung zur Berechnung des datenabhängigkeitsbasierten Initiierungsintervalls nutzt die in Abschnitt 7.3.2 beschriebene Methode der Formulierung des Problems als *minimal cost-to-profit ratio cycle* Problem. Das bedeutet, dass ausgehend vom ressourcenbasierten Initiierungsintervall mittels des *minimal cost-to-profit ratio cycle* Algorithmus (siehe [EF03]) bestimmt werden kann, ob das zur Berechnung verwendete Initiierungsintervall zu klein, zu groß, oder in der gewünschten Gleitkommapräzision korrekt ist. Wie in Abschnitt 7.3.2 beschrieben wird dazu eine Distanzmatrix berechnet, die am Ende in Eintrag $[i, j]$ die minimal notwendige Verzögerung zwischen dem Start von Operation i und Operation j beinhaltet.
Die Berechnung der Distanzmatrix erfolgt in zwei Schritten: die Matrix wird derart initialisiert, dass in Eintrag $[i, j]$ die minimal notwendige Verzögerung zwischen dem Start der Operationen i und j enthalten ist, wenn man nur direkte Datenabhängigkeiten zwischen i und j in Betracht zieht. Dies kann durch eine Iteration über alle Paare von Operationen erreicht werden, wobei man für jedes Paar über alle von ihm definierten Kanten im DDG iterieren muss. Im zweiten Schritt betrachtet man zusätzlich transitive Abhängigkeiten zwischen den Operationen, indem man für alle Operationenpaare (i, j) nach einem Pfad von i nach j im DDG sucht, der über eine andere Operation k verläuft. Dieses k wird ebenfall über alle Operationen iteriert, wodurch alle transitiven Abhängigkeiten erfasst werden. Ist die notwendige Distanz zwischen i und j über die Operation k größer als die bisher gespeicherte, so wird der betreffende Matrix-Eintrag aktualisiert.
Eine vereinfachte Darstellung des *minimal cost-to-profit ratio cycle* Algorithmus ist in Listing 8.2 aufgelistet. Die Funktion `ComputeMinDist` liefert `true` zurück, wenn das benutzte Initiierungsintervall zu klein war. Liefert die Funktion `false` zurück und sind alle Diagonaleinträge der Distanzmatrix echt kleiner als Null, so war das gewählte Initiierungsintervall zu groß. Wie in Abschnitt 7.3.2 beschrieben, kann man durch binäre Suche und wiederholtes Aufrufen der Funktion `ComputeMinDist` mit verschiedenen Initiierungsintervallen das datenabhängigkeitsbasierte Initiierungsintervall berechnen.

8.2.3 Flat Schedule

Die Berechnung des *Flat Schedule* hält sich unmittelbar an die in Abschnitt 7.5 beschriebene Struktur. Es wird versucht, alle Operationen anzuordnen, bis ein gültiger

```
bool ComputeMinDist ( int II, int n )
{
  // II : Initierungsintervall
  // n  : Anzahl der Operationen der Schleife

  // Initialisierung der MinDist-Matrix
  for (int i=1 ; i<=n ; ++i) {
    for (int j=1 ; j<=n ; ++j) {
      MinDist[i,j] := -∞;
      for each edge e ∈ E_dd mit e = (p_i, p_j) do
        MinDist[i,j] = Max(MinDist[i,j], Delay(e) - II * Dist(e
           ));
    }
  }

  // Berechnung der endgültigen Matrix-Einträge
  for (int k=1 ; k<=n ; ++k) {
    // betrachte Pfad von p_i nach p_j über p_k
    for (int i=1 ; i<=n ; ++i) {
      for (int j=1 ; j<=n ; ++j) {
        int dist = MinDist[i,k] + MinDist[k,j];
        if ( dist > MinDist[i,j] ) {
          MinDist[i,j] = dist;
          if ( i==j && dist>0) {
            // positiven Zyklus gefunden
            return true;
          }
        }
      }
    }
  }

  return false;
}
```

Listing 8.2: Berechnung MII_{dep}

Flat Schedule erzeugt wurde oder die maximale Anzahl an Schedule-Operationen überschritten wurde. Diese ist durch das Budget-Verhältnis in Abhängigkeit zu der Anzahl der Operationen der Schleife bestimmt, da sich die maximale Anzahl von Schleifenoperationen durch das Produkt aus Budget-Verhältnis und der Anzahl der Operationen der Schleife ergibt.
In jedem Schedule-Schritt wird die noch nicht angeordnete Operation gewählt, die aktuell die höchste Priorität hat. Für diese Operation wird das Zeitfenster berechnet und anschließend innerhalb dieses Zeitfensters nach einem gültigen Issue-Slot gesucht. Dies geschieht über eine Iteration über alle Instruktionen des Zeitfensters. Konnte eine gültige Instruktion gefunden werden, so wird die gewählte Operation darin platziert. Anderenfalls wird ein Kontrollschritt berechnet, in dem die Operation zwangsweise angeordnet wird. Bevor dies aber geschieht, werden alle Operationen, mit denen ein Konflikt entstehen würde, aus dem partiellen Schedule entfernt. Ein Konflikt entsteht dadurch, dass entweder eine Ressourcenbedingung oder eine Datenabhängigkeit verletzt wird, wenn die aktuelle Operation in diesem Kontrollschritt ausgeführt wird. Die Funktion gibt dann den erzeugten *Flat Schedule* an den Aufrufer zurück.
Listing 8.3 zeigt eine vereinfachte Darstellung der Implementierung in Pseudo-Code.

8.2.4 Modulo Kernel

Hat man den *Flat Schedule* berechnet, ist die Erzeugung des zugehörigen Modulo Kernel recht einfach. Für jede Operation wird ihr Modulokontrollschritt (vgl. Abschnitt 7.6) berechnet, der sich aus dem Kontrollschritt im *Flat Schedule* modulo des Initiierungsintervalls ergibt. Die Operation wird dann im berechneten Modulokontrollschritt angeordnet. Listing 8.4 zeigt vereinfacht den Algorithmus zur Erzeugung des Kernels. Im Listing ist allerdings aus Gründen der besseren Darstellbarkeit der Konfliktfall ausgelassen, dass nicht alle Operationen desselben Modulokontrollschrittes auch in der gleichen Instruktion des Modulo Kernels einen gültigen Issue-Slot bekommen können (vgl. Abschnitt 7.6). In diesem Fall fügt man Instruktionen in den Kernel ein, wodurch viele Indexanpassungen im Algorithmus vorgenommen werden müssen.

8.2.5 Prolog und kompletter Epilog

Die Erzeugung von Prolog und komplettem Epilog sind annähernd gleich, sodass sie hier zusammengefasst am Beispiel des Prologes behandelt werden.
Zuerst wird in der Reihenfolge ihrer Ausführung über die Operationen im *Flat Schedule* iteriert. Der Prolog muss entsprechend der Pipeline-Tiefe $(SC-1)$ viele Iterationen in einer durch das Initiierungsintervall festgelegten Verzögerung zueinander starten. Jede im Kernel befindliche Iteration befindet sich somit in einem anderen Zustand der Ausführung. Die erste Iteration, die der Prolog startet, wird in der ersten Iteration

```
array ComputeFlatSchedule ( list operations, int II, int
   max_attemps )
{
  array instructions;
  list unscheduled_ops = operations;

  while ( unscheduled_ops is not empty ) {
    int estart,lstart;
    operation op = Pick operations with highest priority;
    issue_slot s;

    estart = ComputeEStart(op,instructions);
    lstart = ComputeLStart(op,instructions);

    for ( int i=estart ; i <= lstart ; ++i ) {
      s = SearchFeasibleIssueSlot(instructions[i],op);
      if ( NULL != s )
        break;
    }
    if ( NULL == s ) {
      // Konfliktfall
      int forced = ComputeForcedControlStep(op);
      list conflicts;

      conflicts = GetConflictingOperations(forced);
      for each op in conflicts {
        UnScheduleOp(op);
      }
      s = SearchFeasibleIssueSlot(instructions[forced],op);
      ScheduleOp(op,s);
    }
    else {
      ScheduleOp(op,s);
    }
    --max_attemps;
    if ( 0 > max_attemps )
      return NULL;
  }
  return instructions;
}
```

Listing 8.3: Berechnung Flat Schedule

```
array ComputeModuloKernel ( array flat_schedule, int II )
{
  array kernel;

  for ( int i=0 ; i<= MaxIndex(flat_schedule) ; ++i) {

    int modulo_cstep = i mod II;
    instruction ins = flat_schedule[i];

    for each operation op in ins do {
      ScheduleOp(kernel[modulo_cstep],op);
    }
  }

  return kernel;
}
```

Listing 8.4: Berechnung Modulo Kernel

des Kernels schon beendet, die zweite im Prolog gestartete Iteration wird erst in der zweiten Iteration des Kernels beendet, u.s.w. Dadurch wird klar, dass die Operationen, die am Anfang des *Flat Schedule* stehen, häufiger im Prolog auftreten werden als diejenigen, die weiter am Ende des *Flat Schedule* stehen. Da der Flat Schedule durch das Initiierungsintervall in mehrere Stufen (*Stages*) aufgeteilt wird, werden die Operationen der gleichen Stufe gleich häufig im Prolog auftauchen.
Der Prolog wird unmittelbar und direkt aus dem Kernel heraus durch Kopieren der Operationen erzeugt. Dafür wird jede Operation im Kernel mit einem Zähler initialisiert, der anfangs die Nummer der Stufe beinhaltet, in dem sich die Operation im *Flat Schedule* befindet. Die Zähler aller Operationen in den ersten II Instruktionen im *Flat Schedule* erhalten also den Wert $SC - 1$, die Operationen der nächsten II Instruktionen im *Flat Schedule* $SC - 2$, und so weiter.
Die eigentliche Erzeugung des Prologes ist ein iterativer Algorithmus. Da man $SC - 1$ viele Iterationen starten muss, ist die Iterationszahl für diesen Prozess ebenfalls $SC - 1$. In jedem Schritt wird der Zähler aller Operationen im Kernel dekrementiert; anschliessend wird der Kernel komplett kopiert bis auf die Operationen, deren Zähler unter Null gefallen ist. Zum Schluss ergibt die in umgekehrter Reihenfolge ihrer Erzeugung entstandene Sequenz von Kernelkopien den fertigen Prolog. Das Kopieren des Kernels geschieht in der Implementierung operationsweise. Hier muss mit vielen Indizes gearbeitet werden, weshalb Listing 8.5 nur eine vereinfachte Darstellung der wirklichen Implementierung auflistet und das Kopieren einer Operation hinter einer Funktion versteckt.

Zur Erzeugung des kompletten Epiloges kann genauso vorgegangen werden, außer dass die Zähler hier genau umgekehrt initialisiert werden. Die Operationen der ersten II Instruktionen des *Flat Schedule* werden mit 1, die nächsten II Instruktionen mit 2 initialisiert, u.s.w.

```
array ComputePrologue ( array flat_schedule,
                        int stage_count,
                        int II )
{
  array prologue;
  instruction ins;

  // Initialisierung der Counter
  for each ins in flat_schedule do {
    operation op;
    for each op in ins do {
      int counter = stage_count - ControlStep(op)/II;
    }
  }

  // Erzeuge Prolog
  for ( int i=stage_count ; i>0 ; --i) {
    for each ins in kernel do {
      operation op;
      for each op in ins do {
        DecrementCounter(op);
        if ( Counter(op) > 0 ) {
          CopyOperationIntoPrologue(op,prologue);
        }
      }
    }
  }

  return prologue;
}
```

Listing 8.5: Berechnung Prolog

8.2.6 Modulo-Registerexpansion

Wie in Abschnitt 7.9 beschrieben, kann die maximale Lebensspanne eines Registers größer sein als die Länge des berechneten Kernels. In dem Fall ist *Modulo-Registerexpansion* notwendig, um den Kernel soweit aufzuklappen, dass es zu keiner

Verletzung von Datenabhängigkeiten kommt.
Listing 8.6 zeigt vereinfacht den Algorithmus zur Implementierung der *Modulo-Registerexpansion*. Zu Beginn wird die maximale Lebenspanne eines Registers im *Flat Schedule* bestimmt, was durch Iteration über alle Datenabhängigkeiten geschieht. Hierbei ist zu beachten, dass man an den Lebensspannen von schleifenvarianten Registern interessiert ist, d.h. zu einer Benutzung-Setzung-Abhängigkeit muss hier auch eine Setzung-Benutzung-Abhängigkeit über die Iterationsgrenze hinweg vorhanden sein. Beim TriMedia TM1000-Prozessor z.B. gibt es Register, denen ein fester Inhalt zugeordnet ist. Deren Lebensspanne erstreckt sich immer über das komplette Programm hinweg, aber ihr Inhalt wird nie überschrieben. Daher muss man solche Lebensspannen ignorieren. Während der Berechnung der maximalen Lebensspanne eines Registers werden auch gleichzeitig schon die Register erfasst, die später beim Aufrollen umbenannt werden müssen.
Anschliessend wird der Unroll-Faktor bestimmt, der angibt, wie oft der Kernel aufgeklappt werden muss. Ein Faktor von zwei bedeutet beispielsweise, dass der Kernel einmal aufgeklappt werden muss, um seine zweifache Größe zu erhalten.
Für jedes notwendige Aufrollen, wird der gesamte Kernel kopiert und die vorher erfassten Register je nach Art ihres Auftretens als Benutzung oder Setzung umbenannt. Die Art der Umbenennung ist in Abschnitt 8.2.6 ausführlich beschrieben.

```
void ModuloRegisterExpansion (array kernel, array flat_schedule
   )
{
  int max_lifespan;
  int unroll;

  max_lifespan = ComputeMaximumLifeSpan(flat_schedule);
  unroll = ceil(max_lifespan/II);

  for (int i=unroll ; i>0 ; --i) {
    UnrollAndRenameKernel(kernel);
  }
}
```

Listing 8.6: Berechnung Modulo-Registerexpansion

8.2.7 Basisblockstruktur

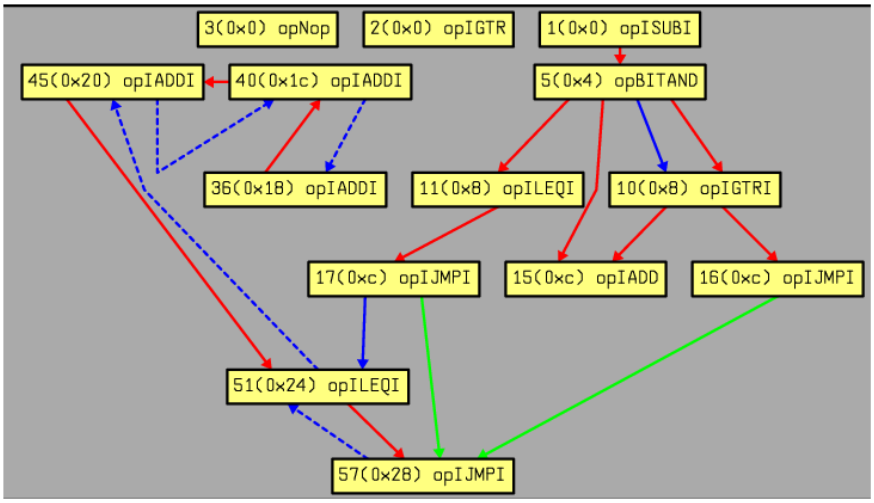
Die Basisblockstruktur der optimierten Schleife nach dem Pipelining besteht wie in Abschnitt 7.11 beschrieben aus mindestens drei Blöcken: einem für den Prolog, einem für den Kernel selbst und einem für den Epilog. Nun muss dafür gesorgt werden, dass diese neue Blockstruktur derart in den restlichen Kontrollfluss integriert wird, dass

alle Verzweigungen im Programm weiterhin korrekt sind. An dieser Stelle setzt die prototypische Implementierung voraus, dass alle Verzweigungen mit absoluten Zielen (Labels im Assembler) versehen sind. Dies macht die Rekontruktion des Kontrollflusses automatisierbar, kann aber umgangen werden, wenn man Benutzerannotationen zu nicht automatisch herleitbaren Sprungzielen zuläßt. Deshalb ist diese Vorraussetzung für den Prototyp akzeptabel. Als Startlabel für den Prolog wird das ursprüngliche Label verwendet, das den Start der Originalschleife markierte. Dadurch ist sichergestellt, dass alle Programmpunkte, die die Originalschleife aufrufen, auch jetzt noch korrekt verzweigen. Der Block für den Kernel wird direkt an den Block des Prologes angehangen, sodass hier kein spezieller Sprung von Prolog zu Kernel notwendig ist. Gleiches gilt für den Übergang vom Kernel zu komplettem Epilog. Die eventuell notwendigen partiellen Epiloge wurden in der prototypischen Implementierung nicht berücksichtigt. Nun muss man nur noch die Sprungziele anpassen, die in der Originalschleife vorhanden sind. Alle Verzweigungen, die in der ursprünglichen Schleife eine neue Iteration starten, verzweigen nun an den Anfang des Kernelblockes, der mit einem eindeutigen Label versehen wird. Ebenso zeigen nun alle Schleifenausgänge im Kernel auf den Start des Epilogblockes. Am Ende des Epiloges werden dann die Sprünge zum Nachfolgeblock der Schleife angehangen.

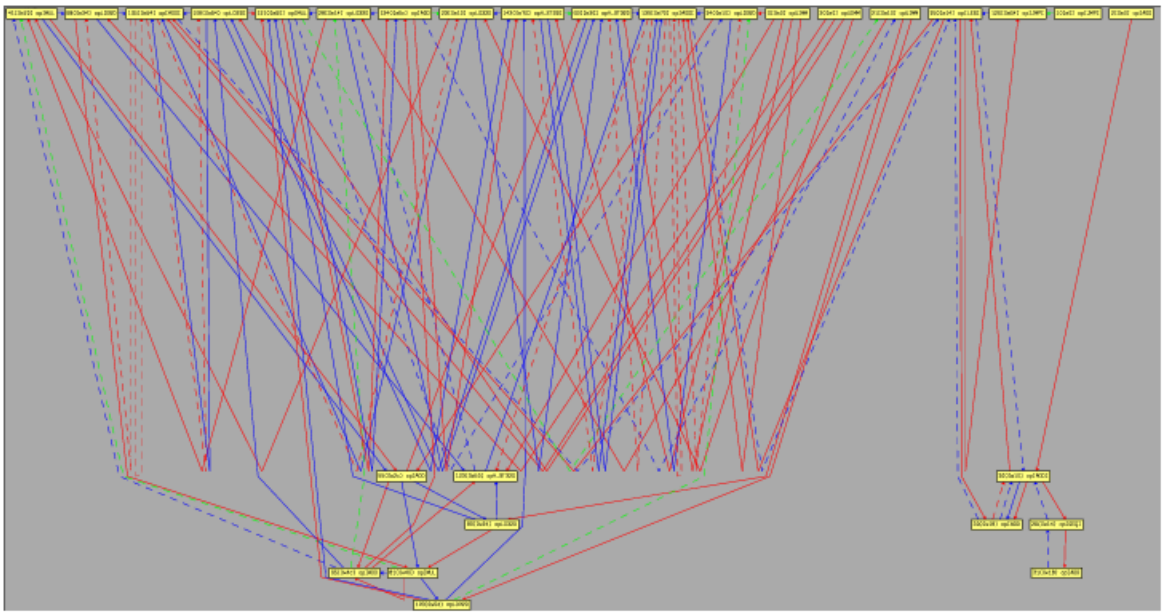
Dadurch werden die Basisblöcke der Originalschleife durch drei große Basisblöcke für Prolog, Kernel und kompletten Epilog ersetzt. Macht man die prädikative Form des Codes wieder rückgängig (vgl. Abschnitt 7.12), so wird aus jedem Basisblock jeweils eine Menge von Basisblöcken.

8.3 Visualisierungen

Die Implementierung bietet die Möglichkeit, sowohl die Ergebnisse des Modulo Schedulers als auch die vorher erzeugten Programmdarstellungen interaktiv zu visualisieren. Die visualisierbaren Programmdarstellungen umfassen den Kontrollflussgraphen, den Datenabhängigkeitsgraphen, den Kontrollabhängigkeitsgraphen sowie eine Darstellung des Instruktionssatzes für den Zielprozessor. Die Daten werden jeweils unter Benutzung der Graph-Beschreibungssprache GDL (siehe `http://www.aisee.com/gdl/nutshell`) in eine Datei exportiert. Diese kann mittels der interaktiven Graph-Visualisierungssoftware **aiSee** ([Abs05]) der Firma *AbsInt Angewandte Informatik GmbH* (`http://www.absint.com`) angezeigt. Dadurch ist es möglich, bestimmte Teile der Programmdarstellungen zu durchwandern. Abbildung 8.2 zeigt beispielhaft die aiSee-Darstellung eines Datenabhängigkeitsgraphen. Die Knoten stellen die Operationen dar und die Kanten zwischen den Operationen Datenabhängigkeiten, die zwischen den Operationen bestehen, die sie miteinander verbinden. Die Farben lassen dabei eine Unterscheidung zwischen den verschiedenen Typen von Datenabhängigkeiten zu. Die Visualisierung der Ergebnisse des Modulo



(a) einfacher Datenabhängigkeitsgraph



(b) komplexer Datenabhängigkeitsgraph

Abbildung 8.2: Visualisierungen eines Datenabhängigkeitsgraphen mit aiSee

Schedulers spalten sich in drei Teile auf: eine Visualisierung des Prologes, des Kernels und des kompletten Epiloges. Ebenso wie die Visualisierung der Programmdarstellungen werden die Daten in eine GDL-Datei exportiert, um dann mit aiSee visualisiert zu werden.
Die Abbildung 8.3 zeigt Beispiele der Darstellung eines Prologes (a), Kernels (b) und Epiloges (c). Hier stellt jeder Knoten eine VLIW-Instruktion dar, deren Subknoten die Operationen repräsentieren. Aufgrund der besseren Übersichtlichkeit werden nicht besetzte *Issue-Slots* nicht dargestellt. In jeder Operation findet man ihre Assemblerrepräsentation sowie die Bezeichnung des Issue-Slot, in dem sie ausgeführt wird. Die Kanten zwischen Instruktionen sind als Kontrollfluss zu interpretieren, d.h. eine Kante von Instruktion i zu einer Instruktion j bedeutet, dass i vor j ausgeführt wird.

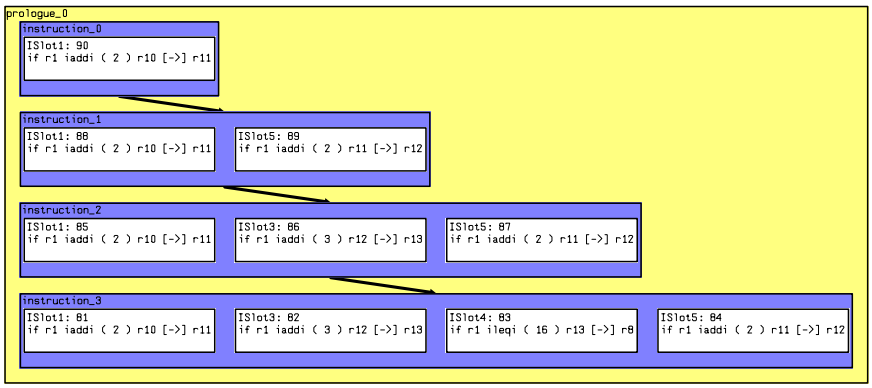
8.4 Komplexität

Der Berechnungsaufwand von Softwarepipelining im Allgemeinen und Modulo Scheduling im Speziellen ist relativ hoch im Vergleich zu azyklischen Verfahren. Dem entgegen steht allerdings, dass es sich um ein zyklisches Instruktionsanordnungsverfahren handelt, was alleine schon eine erhöhte Komplexität erklärt.
Das Instruktionsanordnungsverfahren ist $\mathcal{NP}$-vollständig. Daher werden in dem hier vorgestellten Verfahren optimale Lösungen approximiert (wie z.B. bei der Berechnung der unteren Schranke des Initiierungsintervalls), sodass dadurch die worst-case Laufzeit beschränkt werden kann. Die Komplexität des hier vorgestellten Verfahrens liegt

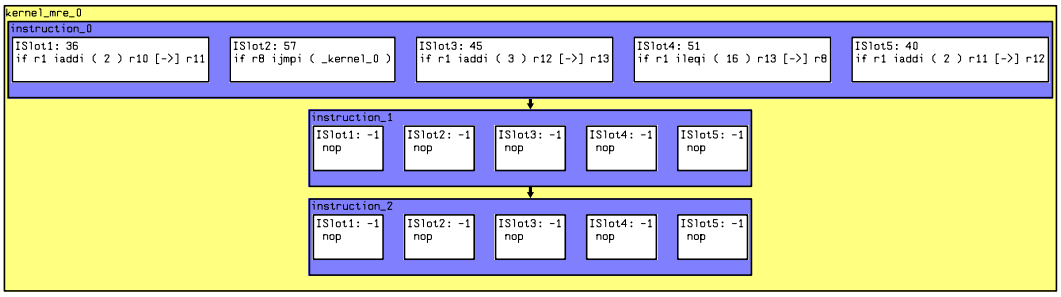
Teilproblem	**Komplexität**
MII_{res}	$\mathcal{O}(n^2)$
MII_{dep}	$\mathcal{O}(n^3)$
$HeightR$	$\mathcal{O}(n^2)$
Flat Schedule	$\mathcal{O}(n^2)$
Modulo Kernel	$\mathcal{O}(n)$
Prolog	$\mathcal{O}(\|\mathcal{K}\|) = \mathcal{O}(n)$
kompletter Epilog	$\mathcal{O}(\|\mathcal{K}\| = \mathcal{O}(n))$
Modulo-Registerexpansion	$u_{min} * \mathcal{O}(\|\mathcal{K}\| = \mathcal{O}(n))$

Tabelle 8.1: Worst-case Komplexität von Iterative Modulo Scheduling

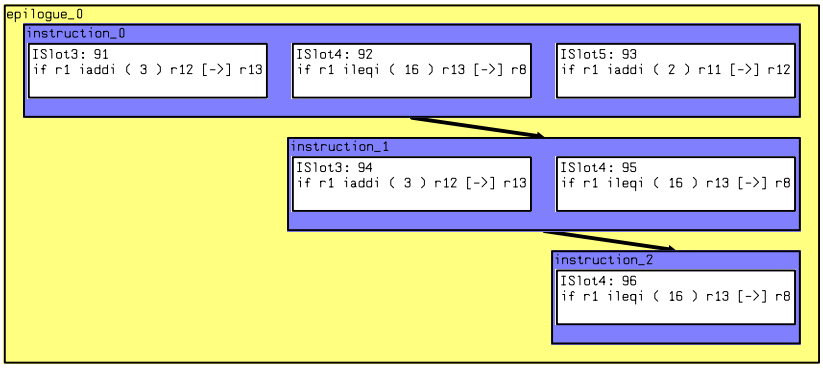
in $\mathcal{O}(n^3)$, wobei n die Anzahl der Operationen in der Eingabeschleife bezeichnet. Tabelle 8.1 zeigt, wie sich die Komplexität auf die einzelnen Teilprobleme aufspaltet. Hierbei ist zu beachten, dass die kubische Gesamtkomplexität nur dadurch zustande kommt, dass in der vorliegenden Implementierung das datenabhängigkeitsbasierte minimale Initiierungsintervall durch die Formulierung als *minimal cost-to-profit ratio cycle* Problem (vgl. Abschnitt 7.3.2) formuliert wurde. Alle anderen Berechnungen



(a) Visualisierung eines Prologes



(b) Visualisierung eines Modulo Kernels



(c) Visualisierung eines Epiloges

Abbildung 8.3: Visualisierungen des Modulo Schedules mit aiSee

sind höchstens quadratisch zur Anzahl der Operationen der Eingabeschleife, manche sogar linear.
Weitere Details zur Laufzeitanalyse des Verfahrens finden sich in [Rau94].

Kapitel 9

Experimentelle Ergebnisse

Dieses Kapitel enthält die Ergebnisse und Interpretation einiger Experimente, die mit der in Kapitel 8 beschriebenen Implementierung für den TriMedia TM1000-Prozessor (siehe Abschnitt 3.2) durchgeführt wurden.
In Abschnitt 9.1 wird ein Überblick über die verwendeten Testprogramme und die Erzeugung des Assembler gegeben. Der vom Übersetzer erzeugte Assembler muss noch vorverarbeitet werden, um ihn direkt in den Optimierer eingeben zu können, was in Abschnitt 9.2 beschrieben ist. Im Anschluss werden die erzielten Ergebnisse vorgestellt, aufgeteilt nach verschiedenen Kriterien wie z.B. Leistungssteigerung oder Zuwachs des Programmcodes. Im darauf folgenden Abschnitt schließt sich die Interpretation der Ergebnisse an.

9.1 Testprogramme

Die verwendeten Testprogramme stammen aus dem DSPSTONE-Benchmark ([ŽMSM94]), dem MiBench-Benchmark ([GRE+01]) und zwei handgeschriebenen Assemblerprogrammen. Die Programme des *DSPSTONE-Benchmark* implementieren typische Algorithmen für die digitale Signalverarbeitung, wie z.B. die Berechnung digitaler Filter, Konvolution, Fast-Fourier-Transformation, u.s.w. ([ŽMSM94]). Der MiBench-Benchmark stellt eine repräsentative Suite von Programmen für den Bereich eingebetteter Systeme dar, wie z.B. Verschlüsselungsverfahren (*pgp*, *blowfish*, etc.) und Funksignalkodierung (*gsm*). Die verwendeten Programme lagen im C-Quellcode vor. Die zwei selbst geschriebenen Assemblerprogramme enthalten recht einfache Schleifen, die aber durch azyklische Verfahren nicht parallelisiert werden können. Außerdem sollen diese Programme die Verarbeitbarkeit von handgeschriebenem Assembler zeigen.
Die im C-Code vorliegenden Testprogramme wurden unter Verwendung des Philips `tmcc`-Compilers in der Version V5.5.4 in TM1000-Assembler (siehe Abschnitt 3.2) übersetzt. Diese Version des Compilers unterstützt kein Softwarepipelining. Der Compiler führt mit den verwendeten Optionen `-O2 -Xunroll=0` bereits folgende Programmoptimierungen während der Übersetzung durch:

- *IF-Conversion* (siehe Abschnitt 5.5.1), wodurch bereits durch den Scheduler des Philips Compilers Parallelität auf Instruktionsebene extrahiert wird,
- lokale Optimierungen auf Basisblockebene,
- lokale Optimierungen auf Entscheidungsbaumebene[1] und
- die Haltung von Variablen in Registern soweit dies möglich ist.

[1] ein Entscheidungsbaum ist hier eine interne Datenstruktur des Philips Compilers

Leider gibt Philips keine genaueren Auskünfte über die durch den Übersetzer durchgeführten Optimierungen in den Handbüchern.
Für die hier verwendeten Testprogramme wurden vorzeitige Schleifenaustrittspunkte ignoriert und demnach nur der komplette Epilog erzeugt.

9.2 Vorverarbeitung der Assemblereingabe

Der durch den verwendeten Philips `tmcc`-Compiler erzeugte TM1000-Assembler der Testprogramme muss vor Beginn des Modulo Schedulings in die erwartete Assemblerform gebracht werden, um alle Prozedurbezeichner identifizieren zu können und alle Kontrollflussoperationen klassifizieren[2] zu können (vgl. [Wil01]). Das impliziert die manuelle Durchführung folgender Arbeitsschritte:

- Alle Sprünge, die zu einem Label verzweigen, das nicht im Assembler definiert ist, werden als Sprünge zu externen Prozeduren klassifiziert.
- Alle Sprünge, die nicht in `_main` liegen und die zum Inhalt von Register `r2` springen, werden als Rücksprunganweisungen klassifiziert. Dies ist eine Konvention des verwendeten Compilers.
- Alle berechneten Sprünge, die in Schleifen liegen und den Kontrollfluss der Schleife beeinflussen (neue Iteration oder Verlassen der Schleife), werden zu absoluten Sprüngen modifiziert. Dies vereinfacht die Erzeugung der neuen Basisblockstruktur (vgl. Abschnitt 7.11) und deren Integration in den umgebenden Kontrollfluss.

Die ersten beiden Vorverarbeitungen sind für die automatische Rekonstruktion des Kontrollflusses notwendig, wohingegen die dritte manuelle Änderung für das Modulo Scheduling selbst erforderlich ist.

9.3 Ergebnisse

Die prototypische Implementierung wurde unter folgenden Gesichtspunkten analysiert:

- Welche Leistungssteigerung kann mit Softwarepipelining im Postpass-Ansatz erreicht werden?
- Wie verändert sich die Größe des Programmcodes?

[2] in Prozeduraufrufe, Sprünge und Rücksprunganweisungen

- Wie gut ist die Approximation an das MII (vgl. Abschnitt 7.3), d.h. kann mit der berechneten unteren Schranke des Initiierungsintervalls bereits ein gültiger Schedule erzeugt werden? Und wenn nicht, wie oft musste das zur Berechnung verwendete Initiierungsintervall erhöht werden?

Die erzielten Ergebnisse werden nach diesen Gesichtspunkten geordnet in den folgenden Abschnitten vorgestellt.

9.3.1 Leistungssteigerung

Um die Leistungssteigerung einer optimierten Schleife quantifizieren zu können, wird eine Metrik benötigt, die die Ausführungsdauer von Originalschleife und ihrer optimierten Fassung vergleicht. Da in beiden Schleifen die gleichen Operationen enthalten sind, kommt es also zu keiner Veränderung in den Ausführungszeiten der Operationen selbst, wenn man von Pipeline-Effekten absieht. Die hier verwendete Metrik vergleicht die Anzahl der auszuführenden Instruktionen für jeweils 100 Iterationen der Originalschleife und der entsprechenden Anzahl der optimierten Schleife.
Für die Originalschleife ist diese Anzahl einfach zu berechnen, sie ergibt sich aus dem Produkt von 100 und der Anzahl der Instruktionen in der Originalschleife. Instruktionen der optimierten Schleife, die immer ausgeführt werden, sind die Instruktionen des Prologes und Epiloges. Die Pipeline-Tiefe (SC) (vgl. Definition 7.25) ist ein Maß für die Pipelinetiefe der optimierten Schleife und gibt an, wieviele verschiedene Iterationen der Originalschleife im Kernel enthalten sind. Demnach beendet der komplette Epilog ($SC-1$) viele Iterationen der ursprünglichen Schleife. Das bedeutet, die Iterationzahl des Kernels muss gerade so groß sein, dass $100-(SC-1)$ viele Iterationen der ursprünglichen Schleife ausgeführt wurden, um am Ende auf die Gesamtausführung von 100 Iterationen der Originalschleife zu kommen. Eine Iteration des Kernels selbst beendet ohne angewandte Modulo-Registerexpansion (MRE) genau eine Iteration der Originalschleife und mit angewandter MRE u_{min} (vgl. Definition 7.32) viele Iterationen der Originalschleife. Demnach muss der Kernel noch exakt $\frac{100-SC-1}{u_{min}}$ viele Male ausgeführt werden. Die Gesamtanzahl der notwendigen Instruktionen der optimierten Schleife,um 100 Iterationen der Originalschleife auszuführen, berechnet sich also durch

$$|\mathcal{P}| + |\mathcal{E}| + \left(\frac{100 - SC - 1}{u_{min}} * |\mathcal{K}| \right).$$

Tabelle 9.1 zeigt die erzielten Pipeline-Tiefen der Testprogramme. Der TriMedia TM1000-Prozessor kann maximal fünf Operationen pro Instruktion ausführen, sodass eine erzielte Pipeline-Tiefe von fünf den maximalen Wert für diesen Prozessor darstellt. Dieser wurde für `chain` erreicht, da der Datenabhängigkeitsgraph dieses Programmes eine kettenartige Struktur aufweist. Für die Matrixmultiplikation `dmat1x3` konnte ebenfalls eine sehr hohe Pipeline-Tiefe (von vier) erreicht werden. Die erzielten

Programm	**Pipeline-Tiefe**
bitstrng	1
chain	5
chain2	3
dfir	1
dlms	1
dmat1x3	4
dmatrix1	2
FFT	2
isqrt1	2
isqrt2	1
mamu2	3
Ø	**2**

Tabelle 9.1: Pipeline-Tiefen

Werte für `chain2` und `mamu2` sind mit drei noch über dem Durchschnitt von zwei. Für `dlms`, `bitstrng`, `dfir` und `isqrt2` hingegen konnte keine sehr hohe Pipeline-Tiefe erreicht werden.
Tabelle 9.2 zeigt die erzielte Leistungssteigerung der Testprogramme. Die zweite Spalte gibt jeweils die Anzahl der benötigten Instruktionen für die Ausführung von 100 Iterationen der Originalschleife an und die dritte Spalte zeigt die entsprechende Anzahl an Instruktionen der optimierten Schleife an. In der vierten Spalte ist die sich daraus ergebende Beschleunigung, um die sich der Schedule verkürzt hat, dargestellt. Wie man sieht, wurden die Schleifen um bis zu einem Faktor von 3,13 beschleunigt (Maximum für `dmat1x3`). Ebenfalls hohe Beschleunigungen von über zwei wurden für `chain`, `chain2`, `dmatrix1` und `FFT` erzielt. Die durchschnittliche Beschleunigung liegt bei einem Faktor von 1,82. Keine Leistungssteigerung konnte für `dfir` und `dlms` erreicht werden. Die Ergebnisse sind in Abbildung 9.1 visualisiert.
Ein weiteres Maß für die Leistungssteigerung ist der erzielte Grad an Parallelität auf Instruktionsebene. Eben dieser soll durch die Berechnung des Kernels maximiert werden. Ein Maß dafür ist die dort vorhandene durchschnittliche Anzahl an **Operationen pro Instruktion (OPI)**, da diese Operationen parallel ausgeführt werden. Tabelle 9.3 zeigt die dafür erzielten Werte für die Testprogramme. Für jedes Programm ist die OPI einmal mit Berücksichtigung der Delay Slots (zweite Spalte) und einmal ohne Berücksichtigung der Delay Slot (dritte Spalte) angegeben. So befindet sich bei `chain` und `chain2` eine Verzweigungs-Operation in der letzten Instruktion des Schedules. Sie kann nicht vorher gestartet werden, da erst hier einer ihrer Operanden verfügbar ist. Deshalb müssen am Ende des Kernels noch zwei leere Instruktionen eingefügt werden, um die Delay Slots zu berücksichtigen. Dies senkt den Wert für OPI. Betrachtet man jeweils die OPI-Werte ohne die eingefügten leeren Instruktionen (dritte Spal-

Programm	**#Instruktionen Originalschleife** [Instruktionen]	**#Instruktionen optimierte Schleife** [Instruktionen]	**Beschleunigung**
bitstrng	1000	798	1,25
chain	800	292	2,74
chain2	900	295	3,05
dfir	1200	1200	1,00
dlms	1000	1000	1,00
dmat1x3	2500	799	3,13
dmatrix1	4000	1985	2,02
FFT	5400	2386	2,26
isqrt1	1100	796	1,38
isqrt2	800	792	1,01
mamu2	1100	895	1,23
Ø			1,82

Tabelle 9.2: Erzielte Beschleunigung

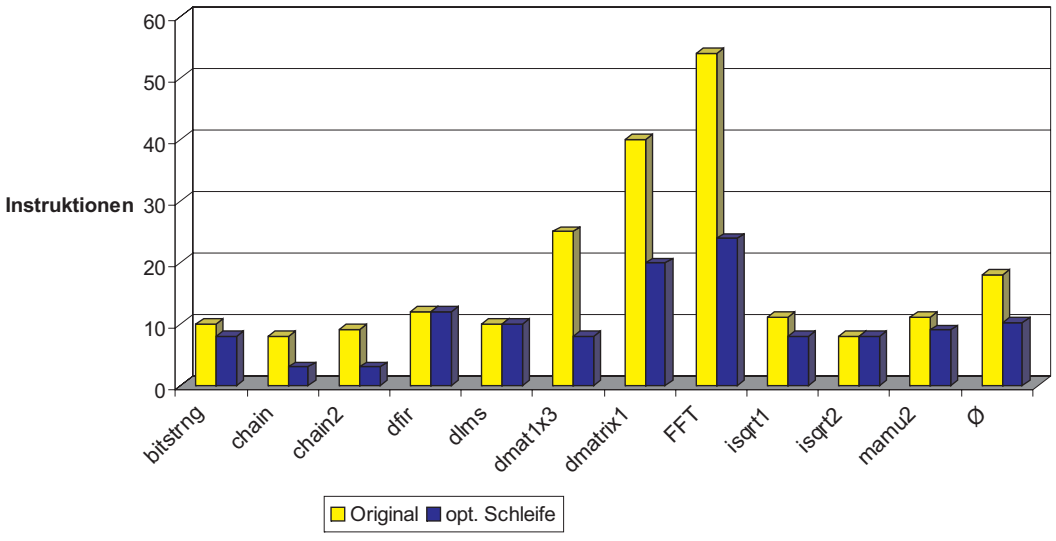


Abbildung 9.1: Leistungssteigerung

te), so stellt man fest, dass alle Instruktionen mit fünf Operationen gefüllt werden konnten, was für den TriMedia TM1000-Prozessor der maximal mögliche Wert ist (vgl. Abschnitt 3.2). Für alle anderen Programme befindet sich keine Verzweigungs-Operation in den letzten beiden Instruktionen[3], sodass sich hier die Werte für OPI mit Berücksichtigung der Delay Slots nicht von denen ohne Berücksichtigung derselben unterscheiden. Hohe Werte von über 3,0 konnten für `chain`, `chain2`, `dmatrix1`, `dmat1x3`, `isqrt1` und `mamu2` erzielt werden. Der durchschnittliche OPI-Wert liegt bei 3,5 Operationen pro Instruktion. Da die Schedules von `dfir` und `dlms` nicht verkürzt werden konnten (siehe Abschnitt 9.3.1), hat sich hier der OPI-Wert nicht verändert. Gleiches gilt für `isqrt2` und `mamu2`, wo zum einen nur eine sehr geringe Leistungssteigerung erzielt wurde und zum anderen der OPI-Wert der Originalschleife bereits sehr hoch ist. Für `dmat1x3` beispielsweise konnte der OPI-Wert aber von 1,0 auf 3,1 erhöht werden, was einen drastischen Anstieg an Parallelisierung darstellt.
Abbildung 9.2 visualisiert die Werte aus Tabelle 9.3.

Programm	**OPI** (Originalschleife)	**OPI** (mit Delay Slots)	**OPI** (ohne Delay Slots)
bitstrng	2,2	2,8	2,8
chain	1,0	1,7	5,0
chain2	0,8	1,7	5,0
dfir	3,3	3,3	3,3
dlms	3,6	3,6	3,6
dmat1x3	1,0	3,1	3,1
dmatrix1	1,0	3,1	3,1
FFT	1,2	2,8	2,8
isqrt1	2,6	3,6	3,6
isqrt2	3,3	3,3	3,3
mamu2	3,0	3,0	3,0
Ø	2,1	2,9	3,5

Tabelle 9.3: Parallelität auf Instruktionsebene

9.3.2 Zuwachs des Programmcodes

Das zweite Merkmal des erzeugten Modulo Schedules ist die Vergrößerung des Programmcodes. Tabelle 9.4 stellt dazu die Länge des Schedules der Originalschleife der Länge des erzeugten Modulo Schedules gegenüber. Die Länge des Modulo Schedules ist dabei aufgespalten in Prolog, Kernel und kompletten Epilog. Alle Zahlen geben jeweils die Länge in Instruktionen wieder.

[3]die Kontrollflussoperationen des TriMedia TM1000-Prozessors besitzen drei Delay Slots

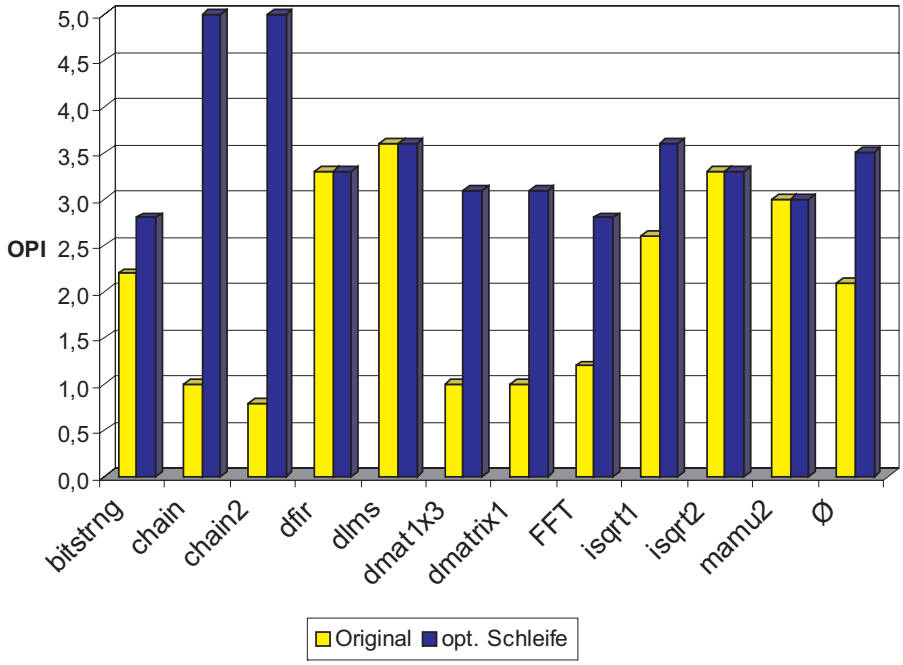


Abbildung 9.2: Parallelität auf Instruktionsebene

Der Zuwachs des Programmcodes liegt für `dmatrix1` und `FFT` bei sehr niedrigen 13% bzw. 7%. Für `chain2` konnte der Programmcode sogar um 22% reduziert werden. Die maximale Zunahme liegt bei 120% für `bitstrng`. Im Durchschnitt stieg der Programmcode um 33% an. Die Werte aus Tabelle 9.4 sind in Abbildung 9.3 dargestellt. Betrachtet man jeweils nur die Schleifenkörper, d.h. nur die zu iterierenden Codesequenzen, so bemerkt man eine deutliche Reduktion ihrer Längen. So wurden die Größen der noch zu iterierenden Instruktionen bei `chain2`, `chain`, `dmat1x3`, `dmatrix1` und `FFT` um über 50% reduziert.
Da für `dfir`, `dlms` und `isqrt2` gar keine bis geringe Leistungssteigerung erzielt wurde, hat sich hier die Anzahl der Instruktionen auch nicht verändert. Im Durchschnitt wurde aber die Größe des Schleifenkörpers um 18% reduziert.
Dieses Verhältnis der zu iterierenden Codesequenzen zueinander ist in Tabelle 9.5 dargestellt.

9.3.3 Gültigkeit des MII

Der Berechnungsaufwand für das minimale Initiierungsintervall ist recht hoch (vgl. Abschnitt 7.3). Deshalb stellt sich die Frage, wie nahe die berechnete untere Schranke

Programm	Länge Originalschleife	Länge Prolog	Kernel	Epilog	∑	Zuwachs
bitstrng	10	6	8	8	22	120%
chain	8	4	3	3	10	25%
chain2	9	2	3	2	7	- 22%
dfir	12	0	12	0	12	0%
dlms	10	0	10	0	10	0%
dmat1x3	25	17	8	14	39	56%
dmatrix1	40	12	20	13	45	13%
FFT	54	19	24	15	58	7%
isqrt1	11	6	8	6	20	82%
isqrt2	8	0	8	0	8	0%
mamu2	11	14	9	8	31	82%
Ø						33%

Tabelle 9.4: Zuwachs des Programmcodes

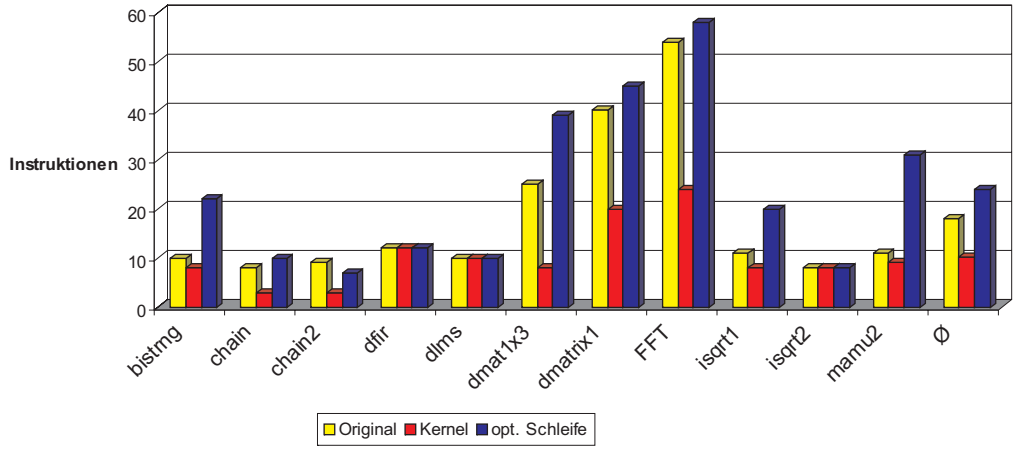


Abbildung 9.3: Zuwachs des Programmcodes

Programm	**Länge Originalschleifenkörper** [Instruktionen]	**Länge Kernel** [Instruktionen]	**Reduktionsrate** [%]
Bitstrng	10	8	20%
chain	8	3	63%
chain2	9	3	67%
dfir	12	12	0%
dlms	10	10	0%
dmat1x3	25	8	68%
dmatrix1	40	20	50%
FFT	54	24	56%
isqrt1	11	8	27%
isqrt2	8	8	0%
mamu2	11	9	18%
Ø			34%

Tabelle 9.5: Größenverhältnisse der Schleifenkörper

an dem tatsächlichen Initiierungsintervall liegt, mit dem ein gültiger Modulo Schedule erzeugt werden kann.
Erfreulicherweise kann in den meisten Fällen mit dem berechneten MII bereits gültiger ein Modulo Schedule erzeugt werden. Tabelle 9.6 verdeutlicht, dass bei 64% der Testprogramme das heuristisch ermittelte MII zu einem gültigen Modulo Schedule geführt hat. In den anderen Fällen liegt die durchschnittliche Abweichung bei 10%. Die größte Abweichung tritt bei `dfir` auf, wo das heuristisch ermittelte MII mit einem Wert von acht um vier Punkte kleiner ist als das tatsächliche MII mit zwölf.
Abbildung 9.4 visualisiert die Werte aus Tabelle 9.6.

9.3.4 Laufzeit

Die Laufzeitmessungen wurden auf einem Linux-System mit einem Intel® Pentium™ 4 mit 3,06 GHz und 512 MB DDR Arbeitsspeicher durchgeführt. Tabelle 9.7 zeigt die erzielten Laufzeiten für die Testprogramme. Die Laufzeiten sind aufgeteilt in die Berechnung des MII, die Berechnung des Modulo Kernels, des Prologes und des Epiloges sowie in die Integration des Schedules in den umgebenden Kontrollfluss des Programmes dargestellt. Zusätzlich ist die Gesamtlaufzeit, die zur Optimierung des Programmes benötigt wurde, angegeben.
Die maximale Gesamtlaufzeit eines Testlaufs liegt bei 8,5s für `FFT`, die minimale bei 0,4s für `chain` und `chain2`. Durchschnittlich benötigt die Optimierung 1,9s.

Programm	MII	II	Abweichung
bitstrng	5	6	20%
chain	1	1	0%
chain2	1	1	0%
dfir	8	12	50%
dlms	8	10	25%
dmat1x3	5	5	0%
dmatrix1	12	12	0%
FFT	19	19	0%
isqrt1	6	7	16%
isqrt2	8	8	0%
mamu2	5	5	0%
Ø			10%

Tabelle 9.6: Gegenüberstellung heuristisches MII zu tatsächlichem MII

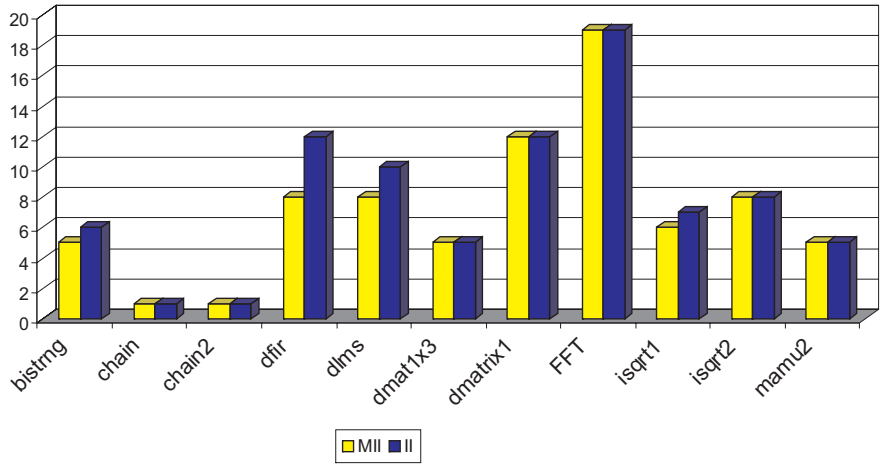


Abbildung 9.4: Gegenüberstellung heuristisches MII zu tatsächlichem MII

Den größten Anteil an der Laufzeit hat die Berechnung des MII mit einer durchschnittlichen Laufzeit von 1,8s und einem Maximum von 8,2s für `FFT`.
Außerdem ist erkennbar, dass die Berechnung von Prolog und komplettem Epilog wesentlich schneller ist als die des Kernels. Die benötigte Zeit zur Integration des Schedules in den umgebenden Kontrollfluss ist relativ unabhängig von der Größe der Eingabeschleife.
Man beachte, dass die angegebene Gesamtlaufzeit nicht einfach die Summe der einzelnen Teillaufzeiten ist, da nicht alle Berechnungen einzeln gemessen wurden. Z.B. wurde nicht die benötigte Zeit zur Rekonstruktion des Datenabhängigkeitsgraphen oder zur Rekonstruktion der optimierten Assemblerdatei, etc. angegeben. Die angegebenen Gesamtlaufzeiten beziehen sich auf den gesamten Prozess vom Parsen der Eingabedatei bis hin zur Ausgabe der optimierten Fassung als Assemblerdatei mit Ausnahme der Kontrollflussrekontruktion, die in einem separatem Schritt abläuft. Für Laufzeitmessungen der Kontrollflussrekontruktion wird auf [Wil01] verwiesen.

Programm	**MII** [ms]	**Prolog** [ms]	**Kernel** [ms]	**Epilog** [ms]	**Integration** [ms]	**Gesamt** [ms]
bitstrng	266,99	0,25	6,53	0,25	0,14	275,14
chain	3,02	0,12	0,33	0,07	0,11	4,27
chain2	3,15	0,06	0,31	0,03	0,11	4,22
dfir	1574,55	0,05	33,03	0,11	0,04	1608,51
dlms	1244,43	0,41	25,76	0,14	0,14	1271,78
dmat1x3	442,20	0,60	10,70	0,29	0,18	455,39
dmatrix1	6371,95	0,62	169,06	0,50	0,22	6544,03
FFT	8231,24	0,86	263,55	0,54	0,26	8498,72
isqrt1	671,14	0,29	22,73	0,25	0,13	695,41
isqrt2	472,86	0,04	15,29	0,03	0,03	488,84
mamu2	1023,41	0,67	20,73	0,27	0,16	1046,36
Ø	1845,90	0,36	51,64	0,23	0,14	1899,33

Tabelle 9.7: Laufzeiten der Optimierung

9.4 Interpretation

Die in Abschnitt 9.3 beschriebenen Experimente und deren Ergebnisse zeigen, dass signifikante Leistungsverbesserungen in der Ausführungszeit von Schleifen möglich sind gegenüber den Ergebnissen azyklischer Instruktionsanordnungsverfahren, die der Philips `tmcc`-Compiler erzielt. Hier wurden um bis 3,13 Mal schnellere Schedules erzeugt (vgl. Abschnitt 9.3.1), wobei die maximale Parallelität (von fünf Operationen

pro Instruktion) des TriMedia TM1000-Prozessors weitestgehend ausgenutzt werden konnte. Diese guten Ergebnisse konnten für Eingabeprogramme erzielt werden, deren Datenabhängigkeitsgraph Abhängigkeitsketten enthält, die es azyklischen Verfahren unmöglich machen, weitere Parallelität auf Intruktionsebene zu erzeugen. Weniger gute Ergebnisse wurden für Programme erzielt, für die bereits azyklische Verfahren einen hohen Grad an Parallelität erzeugen konnten. Dies ist darin begründet, dass in dem Fall durch azyklische Verfahren bereits gute Schedules mit der auf dem Prozessor verfügbaren Parallelität erzeugt wurden. Dadurch bleibt nur wenig Freiraum für weitere Optimierung.
Ein weiteres Qualitätskriterium für Schedulingverfahren ist die erzeugte Größe des Programmcodes. In den durchgeführten Experimenten wurde die Anzahl der zu iterierenden Instruktionen um bis zu 68% reduziert. Dieser Reduktion steht die notwendige Generierung von Prolog und Epilog entgegen, sodass es insgesamt zu einem Zuwachs der Codegröße um bis zu 120% kommt. Durchschnittlich wächst der Programmcode aber nur um 33%. Im Vergleich würden durch die Anwendung von *Loop-Unrolling* Zuwachsraten von mehr als 100% auftreten, sodass man von einem moderaten Zuwachs der Codegröße sprechen kann. Einer durchschnittlichen Leistungssteigerung von 34% steht ein Zuwachs der Codegröße von durchschnittlich 33% gegenüber.
Der hohe Aufwand für die Berechnung des MII macht durchschnittlich 97% der Gesamtlaufzeit aus. Doch der hohe Berechnungsaufwand zahlt sich dadurch aus, dass die berechnete Approximation des MII in 64% der Fälle bereits die Erzeugung eines gültigen Modulo Schedules zulässt. Ist dies nicht der Fall, so kann mit einem durchschnittlich 10% höheren Initiierungsintervall ein gültiger Schedule berechnet werden. Das bedeutet, der hohe Berechnungsaufwand für die Approximation verringert die Berechnungszeit für den gültigen Schedule, da im Falle eines zu kleinen II der Vorgang wiederholt werden muss.
Die gemessenen Laufzeiten liegen für realistische Eingabeprogramme aus dem eingebetteten Bereich zwischen 0,5s und 8,5s Gesamtlaufzeit. Diese Zeiten zeigen, dass das Verfahren trotz seiner hohen Komplexität durchaus in die Toolchains von Produktionsverfahren integriert werden kann.

Kapitel 10

Zusammenfassung und Ausblick

Die vorliegende Diplomarbeit befasst sich mit dem Softwarepipelining Algorithmus *Iterative Modulo Scheduling*, insbesondere dem Einsatz von Softwarepipelining als Postpass-Optimierung. Das Verfahren wird generisch definiert, was die Voraussetzung für den Einsatz in retargierbaren Systemen darstellt. Die vorgestellte Implementierung wendet das beschriebene Verfahren auf die Schleifen von Programmen in Assemblerdarstellung an und erzeugt einen um bis zu 3,13 Mal schnelleren Schedule für die Eingabeschleifen, der als optimiertes Assemblerfile ausgegeben wird. Dies wird bei nur moderat ansteigender Codegröße erzielt. Die Eigenschaft der Retargierbarkeit wird dabei durch die Integration des Verfahrens in das PROPAN-Framework erreicht, das speziell für generische Postpass-Optimierungen entwickelt wurde. Dadurch beschränkt sich die Anpassung an eine neue Zielarchitektur auf das Erstellen einer TDL-Spezifikation des Prozessors.

Softwarepipelining ist ein statisches globales und zyklisches Instruktionsanordnungsverfahren, das den Schedule einer Schleife durch die Konstruktion eines neuen Schleifenkörpers (*Kernel*) verbessert. Dessen Anzahl an Instruktionen, die zu iterierende Codesequenz, wird dabei minimiert. Die Konstruktion des neuen Schleifenkörpers basiert auf der überlappenden Ausführung (Verzahnung) aufeinanderfolgender Schleifeniterationen bei gleichzeitiger Extraktion von Parallelität auf Instruktionsebene. Der neue Schleifenkörper erfordert dabei jeweils eine vorgeschaltete und nachgeschaltete azyklische Codesequenz (*Prolog* und *Epilog*).
Als Ausgangspunkt der Umsetzung wurde das Rahmenwerk *Modulo Scheduling*, speziell der von Rau entwickelte Algorithmus *Iterative Modulo Scheduling* verwendet. Hier wird die überlappende Ausführung aufeinanderfolgender Schleifeniteration durch die Berechnung eines Initiierungsintervalls erzielt, das den zeitlichen Abstand zwischen dem Start zweier aufeinanderfolgender Iterationen definiert.

Die Anwendung von Softwarepipelining im Allgemeinen und Modulo Scheduling im Speziellen auf Assemblerebene macht einige Anpassungen des existierenden Verfahrens notwendig. Beim Postpass-Ansatz ist der Kontrollfluss der Originalschleife durch im Eingabeprogramm vorgegebene Verzweigungsoperationen festgelegt. Durch die Opti-

mierung der Schleife werden die Ziele dieser Verzweigungen ungültig und müssen angepasst werden. Außerdem gibt es auf irregulären Architekturen für Verzweigungsoperationen Delay Slots, die bei der Erzeugung des neuen Schleifenkörpers berücksichtigt werden müssen.
Die Eingabe des Verfahrens ist das vom Übersetzer erzeugte Assemblerprogramm. Zu Beginn müssen daher zur Durchführung notwendige Informationen (Kontrollflussgraph, Datenabhängigkeitsgraph, Kontrollabhängigkeitsgraph) aus der Eingabedatei mittels Programmanalysen rekonstruiert werden. Gerade die Rekonstruktion des Kontrollflussgraphen ist ein schwieriger und komplexer Vorgang ([The00, Wil01]). Außerdem ist der Datenabhängigkeitsgraph auf Assemblerebene wesentlich komplexer und flacher, da zum einen keine höherstufigen Datenstrukturen mehr sichtbar sind sondern nur noch Zugriffe auf die Inhalte von Registern und des Speichers und zum anderen eine Anweisung in der Hochsprache teilweise in mehrere äquivalente Assembleroperationen übersetzt wird. Auch die prädikative Ausführung von Operationen erzeugt Datenabhängigkeiten, die das Ergebnis des Verfahrens beeinflussen.
Eine weitere Eigenschaft der Assemblereingabe ist die bereits durchgeführte Registerallokation, d.h. die Entscheidung, welche Werte in Registern und welche im Speicher gehalten werden müssen, wurde vom Übersetzer bereits getroffen. Dies ist eine Beschränkung, die nicht überwunden werden kann. Zumindest die Registerzuweisung kann aber durch Registerumbenennungen im Postpass-Ansatz noch modifiziert werden.

In den mit der Implementierung durchgeführten Experimenten wurden die Schleifen von Testprogrammen verschiedener Benchmarks (*DSPSTONE-Benchmark* [ŽMSM94, SWS95] und *Mibench-Benchmark* [GRE+01]) optimiert. Die Ergebnisse zeigen eine Leistungssteigerung der optimierten Schleifen bis zum 3,13-fachen in der Ausführungszeit. Betrachtet man die Codegröße, so zeigt sich, dass der zu iterierende Code um bis zu 68% reduziert wird bei einer Gesamtzunahme des Programmcodes (durch Prolog und Epilog) von durchschnittlich 88%. Die gemessenen Laufzeiten liegen zwischen 0,5s und 8,5s.

Das vorgestelle Verfahren optimiert die innersten Schleifen eines Programmes. Eine vorstellbare Optimierung wäre die Ausweitung der Methode auf geschachtelte Schleifen. Hierzu müsste man eine innere Schleife als „Meta-Instruktion“behandeln (vgl. Abschnitt 5.4.1). Vorher müsste durch entsprechende Experimente untersucht werden, ob durch diese Erweiterung noch weitere Parallelität erzeugt werden kann. Dies hängt sicherlich auch von der Entwicklung der Halbleiterindustrie ab. Wenn der Trend zu Maschinen mit noch größerer Parallelität geht, stellt die Behandlung von geschachtelten Schleifen sicherlich eine sinnvolle Erweiterung dar.
Eine weitere von der Marktentwicklung unabhängige Optimierung ist die Verzahnung von Prolog und Epilog mit den die Schleife umgebenden Codesequenzen. Dies kann durch den Einsatz existierender azyklischer Instruktionsanordnungsverfahren (wie etwa List Scheduling) erreicht werden.

Diese Arbeit zeigt, dass Softwarepipelining auf Assemblerebene möglich ist und auch nutzbare Ergebnisse liefert, da *signifikante Leistungsteigerungen* bei nur *moderat steigender Größe des Programmcodes* erzielt werden können.
Das Verfahren stellt eine Programmoptimierung dar, mit der die Reaktionszeit von eingebetteten Systemen durch eine beschleunigte Ausführungszeit verbessert werden kann. Dadurch sind in einem vorgegebenen Zeitraum auch komplexere Berechnungen durchführbar. Gleichzeitig wächst die Programmgröße nur moderat im Vergleich mit Verfahren wie *Loop-Unrolling* an. Dies spart Speicherplatz, die teuerste Komponente in eingebetteten Systemen. Durch die Integration in das PROPAN-Framework und die damit vorhandene Anbindung zu TDL ist das Verfahren *retargierbar* und kann die Vorteile von PROPAN voll ausnutzen. Diese sind gleichermaßen das Optimieren über Bibliotheksgrenzen hinweg als auch das Optimieren von handgeschriebenem Assembler, wie er im Bereich der eingebetteten Systeme immer noch vorkommt. Außerdem kann dadurch Softwarepipelining mit anderen Optimierungen, z.B. der Instruktionsanordnung durch ganzzahlige lineare Programmierung ([Käs00a]), kombiniert werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[Abs05] AbsInt Angewandte Informatik GmbH. *aiSee. Graph Visualization User's Documentation*, 2005.

[AJLA95] Allan, V.H., Jones, R.B., Lee, R.M. und Allan, S.J. *Software Pipelining. ACM Computing Surveys*, 27(3):367–432, September 1995.

[AN88] Aiken, A. und Nicolau, A. *Perfect Pipelining: A New Loop Parallelization Technique*. In Harald Ganzinger (Herausgeber), *ESOP'88, 2nd European Symposium on Programming*, Band 300 von *Lecture Notes in Computer Science*, Seiten 221–235. Springer, Nancy, France, March 1988.

[Ana95] Analog Devices. ADSP-2106x Sharc *User's Manual*, 1995.

[ARM01] ARM Limited. *ARM7TDMI(Rev.4) Technical Reference Manual*, 2001.

[ASU86] Aho, A.V., Sethi, R. und Ullman, J.D. *Compilers: Principles, Techniques, Tools*. Addison-Wesley, 1986.

[Bas95] Bashford, S. *Code Generation Techniques for Irregular Architectures*, 1995. Universität Dortmund.

[Bra91] Bradlee, D.G. *Retargetable Instruction Scheduling for Pipelined Processors*. Technical Report TR-91-08-07, University of Washington, Department of Computer Science and Engineering, August 1991.

[CLG02] Codina, J.M., Llosa, J. und González, A. *A Comparative Study of Modulo Scheduling Techniques*. In *ICS*, Seiten 97–106. 2002.

[DRG98] Dani, A., Ramanan, V. und Govindarajan, R. *Register-Sensitive Software Pipelining*. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, Seiten 194–198. IEEE Computer Society, Los Alamitos, March 1998.

[Ebc87] Ebcioglu, K. *A Compilation Technique for Software Pipelining of Loops with Conditional Jumps*. In *Proc. ACM Workshop on Microprogramming (Micro-20)*. ACM Press, Ft. Collins, CO, December 1987.

[ED95] Eichenberger, A.E. und Davidson, E.S. *Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule*. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Seiten 338–349. IEEE Computer Society TC-MICRO and ACM SIGMICRO, Ann Arbor, Michigan, November 1995.

[EF03] Eisenbrand F., Funke S. *Optimization*. Lecture Notes, 2003. Saarland University.
URL `http://www.mpi-sb.mpg.de/~eisen/opt2003/index.html`

[Fis81] Fisher, J.A. *Trace Scheduling: A Technique for Global Microcode Compaction*. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[GJ79] Garey, M.R. und Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York;London;Toronto, 1979. ISBN 0-7167-1045-5.

[GRE+01] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. und Brown, R.B. *MiBench: A free, commercially representative embedded benchmark suite*. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. December 2001.
URL `http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf`

[GS92] Gasperoni, F. und Schwiegelshohn, U. *Scheduling Loops on Parallel Processors: A Simple Algorithm with Close to Optimum Performance*. *Lecture Notes in Computer Science*, 634:625, 1992.

[HMC+93] Hwu, W., Mahlke, S.A., Chen, W.Y., Chang, P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G. und Lavery, D.M. *The Superblock: An Effective Technique for VLIW and Superscalar Compilation*. *The Journal of Supercomputing*, 7:229–248, 1993.

[Huf93] Huff, R.A. *Lifetime-sensitive modulo scheduling*. *ACM SIGPLAN Notices*, 28(6):258–267, June 1993.

[Int01] Intel. *Intel®Itanium™Architecture Assembly Language Reference Guide*, 2001.

[Jai91] Jain, S. *Circular Scheduling: A New Technique to Perform Software Pipelining*. *ACM SIGPLAN Notices*, 26(6):219–228, June 1991.

[Käs00] Kästner, D. *A Retargetable System for Postpass Optimisations and Analyses*. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems*. 2000.

[Käs03] Kästner, D. *TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses*. In *GPCE*, Seiten 18–36. 2003.

[KL99] Kästner, D. und Langenbach, M. *Code Optimization by Integer Linear Programming*. In Stefan Jähnichen (Herausgeber), *Proceedings of the 8th International Conference on Compiler Construction CC99*, Seiten 122 – 136. Springer LNCS 1575, March 1999.

[Käs97] Kästner, D. *Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x*. Diplomarbeit an der Universität des Saarlandes FB 14 (Wilhelm), Universität des Saarlandes, Saarbrücken, 1997.

[Käs00a] Kästner, D. *Retargetable Postpass Optimisation by Integer Linear Programming*. Doktorarbeit (Wilhelm), Universität des Saarlandes, Saarbrücken, 2000.

[Käs00b] Kästner, D. *TDL - A Hardware and Assembly Description Language*. Technischer Bericht, Universität des Saarlandes, 2000. Transferbereich 14.

[Lam88] Lam, M. *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*. *ACM SIGPLAN Notices*, 23(7):318–328, July 1988.

[Lan97] Langenbach, M. *Instruktionsanordnungen unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x*. Diplomarbeit an der Universität des Saarlandes FB 14 (Wilhelm), Universität des Saarlandes, Saarbrücken, 1997.

[Lan98] Langenbach, M. *CRL – A Uniform Representation for Control Flow*. Technischer Bericht, Universität des Saarlandes, 1998.

[Leu98] Leupers, R. *Retargierbare Codeerzeugung für digitale Signalprozessoren*, 1998.

[LGAV96] Llosa, J., González, A., Ayguade und Valero, M. *Swing Modulo Scheduling: A Lifetime-Sensitive Approach*. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, Seiten 80–86. IEEE Computer Society Press, Boston, Massachusetts, October 1996.

[ME92] Moon, S. und Ebcioğlu, K. *An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors*. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, Seiten 55–71. 1992.

[MLC+92] Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E. und Bringmann, R.A. *Effective Compiler Support for Predicated Execution Using the Hyperblock*. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, Seiten 45–54. 1992.

[NNH99] Nielson, F., Nielson, H.R. und Hankin, C. *Principles of Program Analysis*. Springer, Berlin;Heidelberg;New York, 1999. ISBN 3-540-65410-0.

[Phi97] Philips Electronics North America Corporation. *TriMedia* TM1000 *Preliminary Data Book*, 1997.

[R+92] Rau, B.R. u. a.. *Code Generation Schema for Modulo Scheduled DO-Loops and WHILE-Loops*. HP Labs Technical Report HPL-92-47, Hewlett-Packard Laboratories, Palo Alto, CA, 1992.

[Ram94] Ramanujam, J. *Optimal Software Pipelining of Nested Loops*. In *Proceedings of the 8th International Symposium on Parallel Processing*, Seiten 335–343. IEEE Computer Society Press, Los Alamitos, CA, USA, April 1994.

[Rau94] Rau, B.R. *Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops*. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, Seiten 63–74. ACM SIGMICRO and IEEE Computer Society TC-MICRO, San Jose, California, November 1994.

[RF93] Rau, B.R. und Fisher, J.A. *Instruction-Level Parallel Processing: History, Overview, and Perspective*. *Journal of Supercomputing*, 7:9–50, May 1993.

[SS02] Srikant, Y. N. und Shankar, Priti (Herausgeber). *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 2002.

[SWS95] Schraut, H., Willems, M. und Schoenen, R. *DSPs, GPPs, and Multimedia Applications - An Evaluation Using DSPstone*, December 1995.

[Tex04] Texas Instruments. *TMS320C3x User's Guide*, March 2004.

[The00] Theiling, H. *Extracting Safe and Precise Control Flow from Binaries*. In *RTCSA*, Seiten 22–30. 2000.

[The04] Thesing, S. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. Doktorarbeit (Wilhelm), Universität des Saarlandes, Saarbrücken, 2004.

[VDL+98] Valderrama, C., Donawa, C., Liem, C., Paulin, P.G. und Sutarwala, S. *Trends in Embedded Systems Technology: An Industrial Perspective*, January 1998.

[Veg82] Vegdahl, S.R. *Local code generation and compaction in optimizing microcode compilers*. Doktorarbeit, CMU, 1982.

[Veg92] Vegdahl, S. R. *A Dynamic-Programming Technique for Compacting Loops*. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, Seiten 180–188. 1992.

[War94] Warter, N.J. *Modulo Scheduling With Isomorphic Control Transformations*. Doktorarbeit, University of Illinois, January 1994.

[Wil01] Wilhelm, S. *Generische Rekonstruktion von Kontrollflußgraphen aus Assemblerprogrammen*. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes, Saarbrücken, 2001.

[Win04] Winkel, S. *Optimal Global Instruction Scheduling for the Itanium Processor Architecture*. Doktorarbeit (Wilhelm), Universität des Saarlandes, Saarbrücken, 2004.

[WM92] Wilhelm, R. und Maurer, D. *Übersetzerbau*. Springer, Berlin;Heidelberg;New York, 1992. ISBN 3-540-55704-0.

[WM97] Wilhelm, R. und Maurer, D. *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite, überarbeitete und erweiterte Auflage*. Springer, Berlin;Heidelberg;New York, 1997. ISBN 3-540-61692-6.

[WMHR93] Warter, N.J., Mahlke, S.A., Hwu, W.W. und Rau, B.R. *Reverse If-Conversion*. In Robert Cartwright (Herausgeber), *Proceedings of the Conference on Programming Language Design and Implementation*, Seiten 290–299. ACM Press, 1993.

[ŽMSM94] Živojnović, V., Martínez, J., Schläger, C. und Meyr, H. *DSPstone: A DSP-Oriented Benchmarking Methodology*. In *Proc. of ICSPAT'94 - Dallas*. October 1994.

Index

F

G

H

I

K

L

M

O

P

R

S

T

U

V

W

Z