technische universität
dortmund

Diplomarbeit

Lokale und globale Instruction Scheduling-Verfahren für den TriCore Prozessor

Thomas Pucyk

fakultät für
informatik
Technische Universität Dortmund
Lehrstuhl Informatik XII

28. Juni 2009

Gutachter:

Dipl.-Inform. Paul Lokuciejewski
Prof. Dr. Peter Marwedel

Vorwort

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich beim Schreiben dieser Diplomarbeit unterstützt haben.

Mein Dank gilt hierbei vor allem meinem Betreuer Paul Lokuciejewski, der mich durch seine ausdauernde Hilfe und Motivation stets bestmöglich unterstützt hat.

Zudem gilt mein Dank meiner Freundin Janith, die sehr viel Verständnis aufbrachte und die wenige freie Freizeit während der Erarbeitung dieser Diplomarbeit abwechslungsreich gestaltete. Ihr möchte ich daher diese Arbeit widmen.

Zuletzt möchte ich auch meiner Mutter Ottilie Pucyk danken, die mich stets bei meinem Studium unterstützt hat sowie immer ein offenes Ohr für mich hatte.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel soll den Leser in die vorliegende Diplomarbeit einführen. In Abschnitt 1.1 wird die Entwicklung eines Instruction Schedulers motiviert. Dazu werden im nachfolgenden Abschnitt 1.2 verschiedene Low-Level Optimierungen vorgestellt, die in Compilern angewendet werden. In Abschnitt 1.3 werden hiernach die Ziele der Diplomarbeit vorgestellt um im letzten Abschnitt 1.4 den Aufbau der Arbeit zu erläutern.

1.1 Motivation

Bis zum Ende der achziger Jahre war die Datenverarbeitung stark an große Mainframe-Computer gekoppelt. Mit fortschreitender Miniaturisierung der Chips und Platinen durch verbesserte Herstellungsverfahren gelang es in den neunziger Jahren immer mehr Verarbeitungskapazität und Speicherplatz auf immer kleinerem Platz zu ermöglichen. Dies hat viele Anwendungen ermöglicht, die vorher nicht realisierbar waren. So enthält der Großteil der heute verkauften Unterhaltungselektronik Datenverarbeitungsmöglichkeiten als Teil des gesamten Produkts. Diese sogenannten *Eingebetten Systeme* [1] haben charakteristische Anforderungen und Gemeinsamkeiten, auf die näher in 1.1.1 eingegangen werden soll.

Durch die Integration von Eingebetteten Systemen in Alltagsgegenstände (wie z.B. Handys, MP3-Player, Set-Top-Boxen, Fernseher etc.) haben sich dabei weitere Kategorien an Datenverarbeitung eröffnet. Dies ist zum einen das *ubiquitous computing*, zum anderen die *ambient intelligence*. Ubiquitous Computing soll dabei den Zugang zu Informationen an jedem Ort ermöglichen, indem z.B. tragbare Geräte einen solchen Zugriff ermöglichen. Dem gegenüber soll Ambient Intelligence eine Qualitätssteigerung des Lebens ohne das Zutun der betreffenden Person ermöglichen. So sollen *intelligente Häuser* mitunter das Klima jedes Raumes regeln und störende Sonneneinstrahlung durch Absenkung von Rolladen ohne Interaktion des Nutzers einleiten. All diese Funktionen sind erst durch den Siegeszug der Eingebetteten Systeme möglich geworden.

1.1.1 Eingebettete Systeme

Viele Eingebettete Systeme, vor allem die hohe Zahl der portablen Unterhaltungsanwendungen, ist durch den Einsatz von Batterien als Energiequelle starken Restriktionen unterworfen. Dies hat zur Folge, dass Eingebettete Systeme effizient mit der zur Verfügung stehenden Energiemenge umgehen müssen, um eine vernünftige Batterielaufzeit der Applikation zu ermöglichen. Es ist deshalb von nöten, das Gesamtsystem geeignet auf die gestellte Aufgabe zu dimensionieren. Dies bedeutet sowohl die Hardware als auch die Software effizient zu entwerfen.

Auf der Hardwareebene sind das z.B. die Taktrate und Komplexität des Prozessors sowie die Hierarchie des Speichersystems. Direkt hiermit verzahnt ist die Entwicklung der Software, welche an die Beschränkungen und Möglichkeiten der Hardware angepasst sein muss. So bieten viele Prozessoren durch Implementierung mehrerer Ausführungseinheiten auf Hardwareebene die Möglichkeit des Ausführens mehrerer gleichzeitiger Berechnungen, des sogenannten Instruction Level Paralellism (ILP). Durch effiziente Ausnutzung dieser Möglichkeit ist eine Verringerung der Taktrate eines Prozessors mit einergehender Verringerung des Energieverbrauchs möglich.

Ein weiteres wichtiges Kriterium von vielen Eingebetteten Systemen ist das Echtzeitverhalten. So stellen z.B. Audio- und Videoanwendungen, bedingt durch die menschliche Wahrnehmungspsychologie, hohe Anforderungen an die rechtzeitige Berechnung von Daten. Kann dies nicht gewährleistet werden, so führt das zu nicht ausgegebenen Bild- oder Toninformationen, was zu starker Verschlechterung der Qualität des Mediums führt. Um diese Berechnungen innerhalb der erlaubten Zeit fertigzustellen, werden Zeitschranken aufgestellt, die durch die durchschnittliche Programmlaufzeit, engl. Average Case Execution Time (ACET) dargestellt werden können. Bei harten Echtzeitsystemen die durch zeitkritische Applikationen, wie z.B. im Automobilbereich, repräsentiert werden, wird das Zeitverhalten durch die maximale Programmlaufzeit, engl. Worst-Case Execution Time (WCET), definiert. Durch die optimale Ausnutzung aller Möglichkeiten der Hardware ist auch hier die Einhaltung dieser Schranken bei gleichzeitiger Verringerung der Anforderungen an die Hardware möglich.

In den meisten Fällen ist die Einhaltung und Minimierung dieser Zeitschranken mit entsprechender Optimierung seitens des Entwicklers verbunden. So war es bei der Entwicklung Eingebetteter Systeme üblich, dass der Maschinencode von Softwareroutinen per Hand optimiert oder sogar gänzlich von Anfang an geschrieben wurde. Mit steigender Komplexität der Software und den Anforderungen von kürzeren Entwicklungszeiten ist das manuelle Optimieren des Codes nicht ohne weiteres möglich. Eine Automatisierung dieser Optimierungen innerhalb eines Hochsprachen-Compilers ist hier wünschenswert. Durch die höhere Abstraktion und die für einen Menschen natürlichere Syntax ist eine Entwicklung und Fehlersuche deutlich effizienter. Zudem ist eine Portierung des Programms auf eine neue Architektur einfacher.

1.1.2 Compiler

Compiler für Eingebettete Systeme unterscheiden sich hierbei in einigen Punkten deutlich von Compilern, die für Prozessoren in Personal Computern (PCs) geschrieben wurden:

- Prozessorarchitekturen in Eingebetteten Systemen verfügen meist über spezielle Fähigkeiten. Diese sollten durch den Compiler ausgenutzt werden.
- Ein hoher Grad an Optimierung ist wichtiger als geringere Kompilierzeiten. Daher sind komplexere und zeitaufwändigere Analysen und Optimierungen akzeptabel.
- Eine Minimierung des Energieverbrauchs durch ein genaues Modell des Prozessors und der benötigten Energiemenge pro Instruktion innerhalb des Compilers ist wünschenswert.
- Unterstützung mehrerer Prozessorarchitekturen durch Compiler für Eingebettete Systeme durch austauschbare Backends (sog. retargetable compiler). Dies ist ein wichtiger Faktor, da es innerhalb Eingebetteter Systeme viel mehr Prozessorarchitekturen gibt, die unterschiedliche Befehlssätze unterstützen.
- Compiler für Eingebettete Systeme sollten es ermöglichen, Laufzeitschranken zu validieren und zu erhalten. Dies sollte insbesondere auch durch die Codeformung und die Beachtung der Speicherhierarchie erreicht werden.

Diese Punkte werden genauer in [1] erläutert.

Optimierungen innerhalb eines Compilers können dabei an zwei verschiedenen Stellen realisiert werden. Diese wären:

- **High-Level Optimierungen:** Diese sind unabhängig vom Prozessormodell und werden auf einer Zwischendarstellung der Programmiersprache, der sogenannten High-Level Intermediate Representation (IR) durchgeführt. Diese können Optimierungen sein, wie Loop tiling, Loop blocking, Loop fusion/fission, Constant folding, Constant propagation usw.
- **Low-Level Optimierungen:** Dem gegenüber werden solche Optimierungen auf dem Assemblercode oder einer äquivalenten Low-Level Intermediate Representation (LLIR) durchgeführt. Solche Optimierungen sind Prozessorspezifisch, da sie explizit Möglichkeiten der Maschinensprache des Prozessors und Architekturgegebenheiten ausnutzen.

Im folgenden soll nun näher auf Low-Level Optimierungen eingegangen werden sowie eine Auswahl davon vorgestellt werden. Diese sind von Interesse, da Instruction Scheduling, als Thema dieser Diplomarbeit, eine Low-Level Optimierung ist.

1.2 Low-Level Optimierungen

Einige der genannten Optimierungen sind auch in Compilern für PCs nicht unüblich, andere wie Erzeugung von DSP Instruktionen oder Minimierung des Speicherverbrauchs nur bei Compilern für Eingebettete Systeme zu finden.

- **Ausnutzung von DSP-Instruktionen:** Code für Eingebettete Systeme soll, so weit wie möglich, effizient auf die Zielarchitektur optimiert sein. Da viele solche Architekturen auch Instruktionen von Digitalen Signalprozessoren (DSPs) unterstützten, sollten diese verwenden werden. Hierbei sind z.B. sogenannte Multiply-Accumulate (MAC) Instruktionen interessant, da diese eine Multiplikation mit nachfolgender Addition innerhalb einer einzigen Instruktion realisieren. Dadurch können wertvolle Ausführungszyklen gespart werden.

- **Alignment von Code:** Durch Ausrichtung von Code auf Grenzen, die das Prefetching oder die Cache- und/oder Speicherarchitektur begünstigen, ist es möglich die Programmausführung positiv zu beinflussen.

- **Minimierung des Speicherverbrauchs:** Verschiedene Prozessorarchitekturen bieten neben einem regulären Instruktionssatz, welcher bei RISC Architekturen eine bestimmte Bitbreite aufweist, einen reduzierten Instruktionssatz auf, welcher die meistbenutzten Instruktionen auf einen sekundären Instruktionssatz kleinerer Bitbreite abbildet. Durch Ausnutzung dieser Instruktionen kann der Speicherverbrauch und die Cacheauslastung optimiert werden.

- **Instruction Scheduling:** Prozessoren mit mehreren Ausführungseinheiten auf Hardwareebene können mehr als eine Instruktion in einem Takt ausführen. In vielen Fällen sind hierbei einige Regeln zu beachten, um eine solche parallele Ausführung zu ermöglichen, und so weit wie möglich, effizient ablaufen zu lassen. Eine solche Umstellung des originalen Assemblercodes wird *Instruction Scheduling* genannt. Durch solches Scheduling des Codes ist eine Reduktion der Programmlaufzeit möglich, wodurch die Effizienz des Gesamtsystems unter den schon vorgestellten Aspekten optimiert werden kann. Diese Optimierung soll als Thema dieser Diplomarbeit genauer vorgestellt werden.

1.2.1 Instruction Scheduling

Das Ziel des Instruction Schedulings ist es also das Instruction Level Parallelism der Zielarchitektur so weit wie möglich auszuschöpfen, indem Instruktionen innerhalb der durch den Code gegebenen Einschränkungen eine geeignete Ausführungsreihenfolge gefunden wird. Hierbei muss auf Eigenheiten der Architektur geachtet werden, da vielfach eine Kombination von direkt nacheinander ausgeführten spezifischen Instruktionen zu einem unerwünschten Wartezyklus der Prozessorpipeline(s), des sogenannten Stalls, führt. Dies muss durch den Instruktion Scheduler modelliert werden. Für manche Prozessorarchitekturen ist es zusätzlich notwendig mittels Einfügen von sogenannten

No-Operation (NOP) Instruktionen nach Instruktionen mit Ausführungslatenzen größer 1 die korrekte Initialisierung des Zielregisters zu gewährleisten. Dies wird dann nötig, wenn keine unabhängigen Instruktion zwischen die betreffende Instruktion und eine davon abhängige nachfolgende Instruktion geschedult werden können. Dies ist im Falle des Infineon TriCore Prozessors nicht notwendig, da hier ein automatisches Interlock zwischen abhängigen Instruktionen angewandt wird.

Ein erfolgreiches Scheduling sollte dabei eine Verringerung der ACET des Programms zur Folge haben. Da es auch innerhalb der Prozessoren Hardware-Fehler gibt, die unter bestimmten Umständen auftreten, sollte dabei auch ein Instruction Scheduler es vermeiden, Instruktionsabfolgen, die diese Fehler auslösen, zu erzeugen.

1.3 Ziele der Arbeit

Das Ziel dieser Diplomarbeit soll die Entwicklung und Evaluierung lokaler und globaler Scheduling-Verfahren für den Infineon TriCore Prozessor, genauer des Modells TC1796, sein. Dabei sollen zuerst Grundlagen erarbeitet werden, wie die Aufstellung aller nötigen Datenstrukturen, um eine effiziente Implementierung eines Schedulers zu ermöglichen.

Hiernach sollen Heuristiken erarbeitet werden, die ein Scheduling mittels des List Scheduling Verfahrens ermöglichen. Diese sollen sowohl generische Heuristiken umfassen, als auch um spezifisch auf die Zielarchitektur zugeschnittene, um die Möglichkeiten der betrachteten Architektur auszuschöpfen. Diese sollen mittels einer Pipeline-Simulation und ACET-Analyse des verwendeten Simulations-Tools evaluiert werden und, wenn nötig, optimiert werden. Des weiteren soll eine erweiterte Heuristik zum Schedulen von Lade- sowie Speicherinstruktionen auf Basis einer Werte-Analyse, die im verwendeten Compiler integriert ist, implementiert werden.

Die Güte des dabei erzeugten Schedules soll zusätzlich durch ein ILP-basiertes optimales Scheduling verifiziert werden.

Um die Möglichkeiten des Schedulers zu erweitern, soll in einem weiteren Schritt dieser um die Möglichkeit erweitert werden, innerhalb der möglichen Grenzen auch global Instruktionen zu schedulen, um die Effizienz des Codes noch weiter zu steigern. Die Grenzen werden dabei durch sogenannte Treegions dargestellt, die einen azyklischen Teilbaum des Kontrollflußgraphen eines Programms darstellen. Dieses Verfahren wurde gewählt, um mehrere Pfade betrachten zu können. Da jedoch keine Profiling-Informationen im betrachteten Compiler vorhanden sind, haben innerhalb der Treegions alle Pfade dieselbe Priorität.

Als Nebeneffekt der Diplomarbeit soll in den zugrundeliegenden Compiler das Erzeugen von MAC-Instruktionen implementiert werden, um das korrekte Scheduling dieser zu gewährleisten.

1.4 Aufbau der Arbeit

Die vorliegende Diplomarbeit gliedert sich folgendermaßen.

In Kapitel 2 sollen die Grundlagen und Konzepte der Diplomarbeit vorgestellt werden. Es soll dabei zunächst näher auf die Zielarchitektur eingegangen werden und Eigenheiten dieser erläutert werden, auf die beim Instruction Scheduling geachtet werden muss. Nachfolgend soll auf den zugrundeliegenden Compiler eingegangen werden, in welchen das erarbeitete Instruction Scheduling implementiert wurde. Dabei werden die internen Datenstrukturen und die Integrationsstelle des Schedulers erläutert. Schliesslich soll das WCET-Analyse Tool, welches während der Entwicklung verwendet wurde, vorgestellt werden, da dieses eine Pipeline-Visualisierung bereitstellt, die bei der Entwicklung des Schedulers zum Verständnis der Arbeitsweise der TriCore Pipeline diente.

In Kapitel 3 wird dann in die Thematik des lokalen Schedulings eingeführt. Dazu sollen zunächst verwandte Arbeiten vorgestellt werden. Danach wird die Funktionsweise des vorliegenden Schedulers erläutert und dazu benötigte Datenstrukturen erläutert. Es wird in diesem Zusammenhang das Verfahren des List Scheduling und die im vorliegenden Scheduler implementierten Heuristiken vorgestellt. Danach wird auf zu beachtende Deteils beim Scheduling vor und nach der Registerallokation eingegangen um im nächsten Schritt auf das zusätzlich implementierte optimale Scheduling einzugehen. Zum Schluss wird der konzeptionelle Aufbau des implementierten Schedulers beleuchtet.

In Kapitel 4 widmet sich des Globalen Schedulings. Zuerst soll dabei auf verwandte Arbeiten eingegangen werden. Danach wird auf das im vorliegenden Scheduler implementierte globale Scheduling und nötige Erweiterungen der Datenstrukturen im Vergleich zum lokalen Scheduling eingegangen. Weiterhin werden zusätzliche Heuristiken vorgestellt und die beim vorliegenden globalen Scheduling benötigte erweiterte Prioritätsvergabe erläutert. Zum Schluss werden Mechanismen erklärt, welche die Korrektheit des implementierten globalen Schedulings gewährleisten.

Innerhalb des Kapitels 5 wird der entwickelte Scheduler mittels verschiedener Benchmarks evaluiert. Die dadurch erhaltenen Werte werden dann miteinander und mit Werten ohne Scheduling verglichen.

Kapitel 6 bietet dann eine Zusammenfassung der Diplomarbeit. Es soll dabei auch ein Ausblick auf mögliche Erweiterungen des Schedulers gegeben werden.

Kapitel 2

Grundlagen und Konzepte

2.1 Architektur

In diesem Abschnitt soll die Architektur des Infineon TriCore TC1796 Prozessors näher erläutert werden. Dazu wird im Abschnitt 2.1.1 zuerst auf die Befehlssatzarchitektur, die sogenannte Instruction Set Architecture (ISA), eingegangen. Dann wird in Abschnitt 2.1.2 genauer auf den Registersatz der Architektur eingegangen und mögliche Implikationen durch den Verwendungszweck von Registern beleuchtet. Unterabschnitt 2.1.3 stellt dann die Pipelines des TriCore vor und stellt grob die auf jeder Pipeline ausführbaren Instruktionen vor. Im letzten Abschnitt 2.1.4 sollen schließlich alle Gründe beleuchtet werden, die einen Stall in der Ausführung des Maschinencodes auf dem Prozessor verursachen können.

2.1.1 Befehlssatzarchitektur

Der Infineon TriCore TC1796 Prozessor ist eine kombinierte 32Bit RISC Microcontroller/DSP Architektur. Zu den wesentlichen Eigenschaften dieser Architektur gehören [2]:

- 32Bit Instruktionsbreite mit zusätzlichen 16Bit Instruktionen der meistverwendeten Instruktionen für verringerten Speicherbedarf.
- 3-fach superskalar, d.h. drei eigenständige Pipeline-Ausführungseinheiten.
- Latenzzeit der meisten Instruktionen von einem Taktzyklus.
- Sprungvorhersage bei allen bedingten Sprunganweisungen.
- 16 KByte Instruktionscache, aufgeteilt in Cache-Lines von 256 Bit.
- Niedrige Latenzzeiten bei Interruptbehandlung.

- Dedizierte Schleifen-Pipeline mit „Zero-Overhead“-Schleifeninstruktionen.
- DSP Instruktionen sowie optionale Fließkommaeinheit (FPU), welche beim Typ TC1796 standardmäßig enthalten ist.
- Auf dem Prozessordie eingebettete On-Chip Speicher für Programmcode [3]:
  - 2 Mbyte Program Flash (PFLASH)
  - 48 Kbyte Scratch-Pad RAM (SPM)

Die Latenzzeit ist dabei die Zeit zwischen dem Erzeugen eines Ergebnisse durch eine Instruktion und der Möglichkeit dieses zu verwenden.

2.1.2 Registersatz

Der Infineon TriCore besitzt 32 General Purpose Register (GPR), einen Program Counter (PC), sowie zwei 32Bit Register PSW und PCXI für Status-Flags, Programmkontext- sowie Speicherschutz-Informationen.

Hierbei ist die Klasse der General Purpose Register aufgeteilt in je 16 Daten- sowie Adressregister. Diese sind generell 32Bit breit. Es besteht zudem die Möglichkeit, je zwei aufeinanderfolgende Register zu einem erweiterten 64Bit Register zusammenzufassen, um 64Bit Daten aufzunehmen. Dabei muss das erste Register von gerader Indexzahl sein.

Bei Funktionsaufrufen gibt es bezüglich der Register eine klare Aufrufkonvention. So werden die Register A[10] bis A[15], D[8] bis D[15] sowie PSW und PCXI explizit bei einem Funktionsaufruf durch die CALL-Instruktionen in einem dedizierten Speicherbereich, der Context Save Area (CSA), abgespeichert und beim Rücksprung aus der Funktion durch die RET-Instruktion daraus wieder hergestellt. Diese Register werden „obere-Kontext-Register genannt. Die restlichen General Purpose Register bilden den unteren Kontext, welcher nach einer aufgerufenen Funktion keine definierten Inhalte enthält.

Diverse Register haben zudem spezielle Aufgaben innerhalb der Programmausführung. So sind die Register A[0], A[1], A[8] und A[9] als sogenannte „Globale Systemregister“ definiert. Register A[10] findet als Stackpointer (SP) Verwendung, A[11] dient der Speicherung der Rücksprungadresse (RA).

Um die Kodierung einiger 16Bit-Instruktionen zu erleichtern, benutzen diese implizit A[15] als Adress- bzw. D[15] als Datenregister.

Es gibt überdies keine dedizierten Fließkommaregister. Stattdessen werden Fließkommawerte in den Datenregistern abgespeichert, und bei Kontextwechseln automatisch abgespeichert und wiederhergestellt.

In Abbildung 2.1 ist der General Purpose Registersatz inklusive etwaiger Verwendungszwecke der jeweiligen Register nochmals bildlich dargestellt.

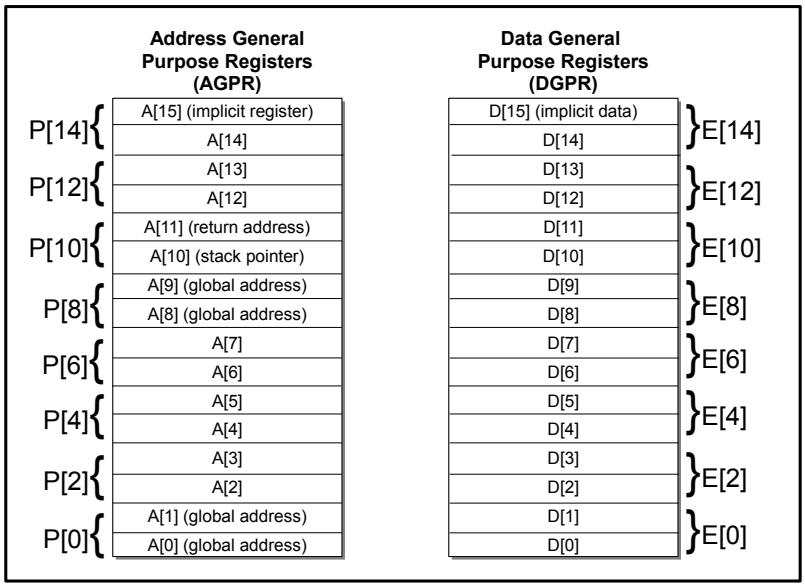

Abbildung 2.1: *General Purpose Registersatz*

2.1.3 Pipelines

Der Infineon TriCore besitzt drei voneinander unabhängige Pipelines. Jede dieser Pipelines hat 4 Ausführungsstufen Fetch (FE), Decode (DE), Execute (EX) sowie Writeback (WB) und führt Instruktionen in-Order aus, d.h. die Instruktionsreihenfolge wird nicht auf Hardwareebene verändert, um das Instruction Level Parallelism zu erhöhen.

Um Verzögerungen in der Ausführung bei Datenabhängigkeiten zu vermeiden, greifen die Pipelines des TriCore auf ein ausgeklügeltes Forwarding-System zurück. So ist es einer nachfolgenden Instruktion einer Pipeline möglich, auf das berechnete Ergebnis einer unmittelbaren vorhergegangenen Instruktion zuzugreifen, ohne dass diese ein Writeback in ein Register schon durchgeführt hat. Es findet also ein sogenanntes EX $\rightarrow$ EX Forwarding statt. Dies ist auch für alle anderen Ausführungsstufen möglich, da z.B. Adressberechnungen schon in der späten Decodephase durchgeführt werden, und hier ein EX $\rightarrow$ DEC Forwarding vonnöten ist, um Stalls zu vermeiden.

Die Details der einzelnen Pipelines stellen sich wie folgt dar:

IP-Pipeline

Die IP-Pipeline führt alle Instruktionen aus, die auf Datenregistern operieren. Daneben werden auch alle Instruktionen, welche zum Datenaustausch zwischen dem Datenregister- und Adressregistersatz dienen, auf der IP-Pipeline ausgeführt.

Die meisten Instruktionen haben eine Latenz von 1, mit Ausnahme der Multiply-Accumulate- und Multiply-Subtract-Instruktionen (auch MAC-Instruktionen genannt). Diese benötigen mindestens zwei bis maximal drei Taktzyklen in der Execute-Stufe. Die Division ist bei dieser Architektur in mehrere Instruktionen aufgeteilt (DVINIT, DVSTEP, DVADJ). Eine Division besteht aus einer DVINIT-, vier DVSTEP- und einer DVADJ-Instruktion. Eine DVSTEP Instruktion hat eine Latenz von 4 Taktzyklen [4].

Load/Store-Pipeline

Die Load/Store-Pipeline führt, mit Ausnahme von auf beiden Registersätzen operierenden Instruktionen, alle auf den Adressregistern operierenden Instruktionen aus. Eine weitere Ausnahme zu dieser Regel sind die auf den Speicher zugreifenden Funktionen, welche Datenregister beschreiben oder aus diesen lesen.

Alle auf dieser Pipeline ausführbaren Instruktionen haben eine Ausführungslatenz von einem Taktzyklus.

Loop-Pipeline

Die Loop-Pipeline führt spezialisierte bedingte (LOOP), sowie unbedingte Schleifeninstruktionen (LOOPU) aus. Dadurch ist es möglich, effizient Schleifenkonstrukte zu implementieren, da durch die Branch-Prediction die anderen Pipelines im optimalen Fall schon mit neuen Instruktionen gefüllt und ausführbereit sind.

Die Fetch-Stufe ist in der Lage, sowohl die IP- als auch die Load/Store-Pipeline (nachfolgend auch LS-Pipeline genannt) gleichzeitig mit Instruktionen zu versorgen. Es ist also möglich, einen Instruction per Clock (IPC) Wert von 2 zu erreichen. (Ein IPC-Wert von 3 ist trotz Loop-Pipeline nicht erreichbar, da Schleifeninstruktionen nur am Anfang und Ende eines Basisblocks vorkommen.) Hierbei ist allerdings einschränkend zu beachten, dass dafür die Instruktionen in einer bestimmten Weise im Maschinencode angeordnet werden müssen. Um eine nebenläufige Ausführung von IP- und LS-Instruktionen zu ermöglichen, muss die IP-Instruktion immer als erstes der Decode-Stufe vorliegen. Diese Voraussetzung der Bündelung muss ein Instruction Scheduler für den TriCore Prozessor beachten, um die bestmögliche Parallelität ausnutzen zu können.

Es gibt überdies Instruktionen, welche Teile sowohl der IP- als auch LS-Pipeline in Anspruch nehmen. Solche „Dual-Pipeline“ (nachfolgend auch DP-Instruktionen genannt) genannten Instruktionen sind nur allein ausführbar. Ein Versuch diese zusammen mit einer IP- oder LS-Instruktion zu schedulen würde zu einem Stall der IP-/LS-Instruktion führen, bis die dafür benötigte Pipeline wieder frei ist. Die wenigen, von einem Compiler erzeugten, Dual-Pipeline-Instruktionen wären die Funktionsaufruf- sowie Rücksprunginstruktionen und einige spezielle Adressierungsinstruktionen.

Mittels des Fließkomma-Koprozessors können auf Hardwareebene Fließkommaberechnungen durchgeführt werden. Die Fließkommaeinheit ist dann über einen Koprozessor-Port an den internen Bus des TriCore Prozessors gekoppelt und verfügt selbst über eine Fließkomma-Pipeline. Während der Durchführung von Fließkommaberechnungen ist keine nebenläufige Ausführung anderer Instruktionen möglich. Die Latenzzeiten von Fließkommainstruktionen variieren zwischen 2 bis 5 Taktzyklen für einfache Berechnungen sowie 16 Taktzyklen für die Division.

Nachfolgend sollen die Instruktionen der TriCore Architektur tabellarisch in die möglichen Instruktionstypen klassifiziert werden. Die FP-Instruktionen seien davon ausgeschlossen, da diese eindeutig der FPU zugewiesen werden können.

2.1.3.1 Klassifikation von Instruktionen

Tabelle 2.1: *Klassifikation von TriCore Instruktionen*

IP-Instruktionen	LS-Instruktionen	DP-Instruktionen
ABS*	ADD_A	ADDIH_A
ADD*, ausser ADD_A, ADDIH_A, ADDSC*	EQ_A	ADDSC*
AND*	GE_A	BISR*
CADD*	J	CACHEA*
CLO*	LD*	CALL*
CLS*	LEA	DEBUG*
CLZ*	LT_A	DISABLE
CMOV*	MOV_AA	DSYNC
CSUB*	MOVH_A	DVADJ
DEXTR	NE_A	DVINIT*
EQ*, ausser EQ_A	NEZ_A	DVSTEP*
EXTR*	NOP	ENABLE
GE* ausser GE_A	ST*	IMASK

Tabelle 2.1: *Klassifikation von TriCore Instruktionen*

IP-Instruktionen	LS-Instruktionen	DP-Instruktionen
INS*	SUB_A	ISYNC
LT*, ausser LT_A		J* ausser J
MADD*		MFCR
MAX*		MTCR
MIN*		RET*
MOV*, ausser MOV_AA, MOVH_A		RFE
MSUB*		RSLCX
MUL*		RSTV
NAND*		SVLCX
NE		SWAP
NOR*		SYSCALL
NOT*		TLB*
OR*		TRAP*
RSUB*		
SAT*		
SEL		
SH		
SHA*		
SUB*, ausser SUB_A		
XNOR		
XOR*		

Es gibt, wie in 2.1.3 beschrieben, einige wenige Instruktionen, die eine Latenzzeit größer als 1 aufweisen. Diese sind nochmals in Tabelle 2.2 aufgeführt. Alle Instruktionen ausser DVSTEP, MADD und MSUB sind hierbei FP-Instruktionen. Zu beachten ist ausserdem, dass Instruktionen mit einer Latenzzeit größer 1 die Ausführung von Instruktionen aller anderen Pipelines anhalten. Dies ist darin begründet, dass es eine einzige Fetch-Stufe für alle Pipelines gibt.

Tabelle 2.2: *Instruktionen mit höheren Latenzzeiten*

DVSTEP	MADD	MSUB	ADD_F	CMP_F	DIV_F	FTOI	FTOQ	FTOU
4	2	2	2	2	16	5	4	4
ITOF	MADD_F	MSUB_F	MUL_F	Q31TOF	QSEED	SUB_F	UPDFL	UTOF
4	4	4	4	4	4	4	4	4

Zusätzlich ist bei einer superskalaren Architektur von Interesse, wie viele Zyklen vergehen müssen,

bis ein Ergebnis einer Instruktionsklasse einer anderen Instruktionklasse zur Verfügung steht. Dieser Wert soll Abhängigkeitszyklen genannt werden. Tabelle 2.3 listet die Anzahl an Abhängigkeitszyklen für jede Instruktionsklasse des TriCore auf. Hierbei ist die Tabelle so zu lesen, dass Zeilen die Vorgängerinstruktion repräsentieren, Spalten die Nachfolgerinstruktion. D.h. das Ergebnis einer vorhergehenden IP-Instruktion ist für eine direkt nachfolgende LS-Instruktion in 0 Taktzyklen verfügbar. Dies bedeutet, dass eine IP-Instruktion mit einer davon abhängigen LS-Instruktion gebündelt und parallel ausgeführt werden kann. Dies unter der Annahme, daß kein Grund für einen zusätzlichen Stallzyklus, wie im folgenden Abschnitt 2.1.4 beschrieben, vorliegt. Dabei wird zusätzlich zwischen IP- und MAC-Instruktionen unterschieden, obwohl die MAC-Instruktionen eine Teilmenge der IP-Instruktionen bilden und keine eigene Instruktionsklasse bilden.

Tabelle 2.3: *Abhängigkeitszyklen zwischen verschiedenen Instruktionsklassen*

	IP	LS	DP	MAC	LOOP	FP
IP	1	0	1	1	1	1
LS	1	1	1	1	1	1
DP	1	1	1	1	1	1
MAC	2	0	2	2	2	2
LOOP	1	1	1	1	1	1
FP	1	1	1	1	1	1

2.1.4 Gründe für Stalls und Delays

Es gibt verschiedene Gründe für zusätzliche unerwünschte Taktzyklen bei der Programmausführung eines Prozessors. Diese werden Stalls genannt, wenn diese generell alle nachfolgend ausgeführten Instruktionen betreffen, oder Delays. Ein Delay ist ein Stall-Zyklus, welcher nur auftritt, wenn eine nachfolgende Instruktion abhängig von einer vorhergehenden ist und das von dieser Instruktion erzeugte Ergebnis nur nach einer Delay-Zeit verwendbar ist.

Im Folgenden werden Gründe von Stalls und Delays beim Infineon TriCore vorgestellt, zusammen mit eventuellen Möglichkeiten, diese zu vermeiden.

- Zugriff auf das selbe Register im gleichen Taktzyklus durch eine parallel ausgeführte IP- sowie LS-Instruktion [5]. Der Grund ist ein struktureller Hazard bedingt durch den gleichzeitigen Zugriff auf das selbe Register.

  Beispiel:

```
add d0, d1, d2
ld.w  d0, [a0]0      <-STALL
```

Abhilfe: Solche Bündel von IP- und LS-Instruktionen vermeiden.

- Load-Instruktionen, die Adressregister mit Werten beschreiben. Hierbei ist der Wert des Adressregisters erst nach einem Delay-Zyklus verfügbar [6].

  Beispiel:

  ```
  ld.a  a0, [a2]0
  add.a a0, 4      <-STALL
  ```

  Abhilfe: Eine nicht abhängige Instruktion zwischen Load und davon abhängiger Instruktion schedulen.

  Anmerkung: Load-Instruktionen, die Datenregister beschreiben, haben keinen Delay-Zyklus. Diese blockieren die weitere Programmausführung der Pipelines, bis das Datenregister beschrieben wurde.

- Zugriff auf die selbe Speicheradresse durch eine Store-Instruktion und eine sofort folgende Load-Instruktion [5]. Dies liegt darin begründet, dass Store-Instruktionen den Speicher in der Writeback-Phase beschreiben, Load-Instruktionen in der Execute-Phase lesen. Im Falle eines solchen Structural-Hazards würde die Load-Instruktion gestallt werden.

  Beispiel:

  ```
  add  d0, d2, d3
  st.w [a0]0, d1
  add  d4, d5, d3
  ld.w d4, [a0]0      <-STALL
  ```

  Abhilfe: Eine nicht abhängige Instruktion zwischen Store- und abhängiger Load-Instruktion schedulen.

- Eine direkt auf eine CALL- oder RET-Instruktion folgende LS-Instruktion verursacht einen Stall-Zyklus [6]. Dies liegt darin begründet, dass CALL/RET-Instruktionen Multizyklen-Instruktionen mit kombiniertem Laden bzw. Speichern des oberen Kontext sind. Diese Lade- respektive Speicheroperation kann noch im ersten Zyklus eines neuen Basisblocks die LS-Pipeline verwenden.

  Beispiel:

  ```
  call function
  add d0, d1, d2
  ld.w d4, [a0]0      <-STALL
  ```

  Abhilfe: Vermeidung des Schedules einer LS-Instruktion im ersten Zyklus nach einer CALL- oder RET-Instruktion. Dies kann dadurch erreicht werden, indem zuerst zwei IP-Instruktionen geschedult werden.

- Eine direkt vor einer RET-Instruktion auftretende Store-Instruktion verursacht einen Stall-Zyklus [6]. Dies liegt wie im vorhergehend genannten Stall-Grund darin begründet, dass die RET-Instruktion Widerherstellen des oberen Kontexts durchführt, und dadurch eine Ladeoperation in der LS-Pipeline stattfindet. Diese Ladeoperation wird gestallt, bis die vorhergehende Store-Instruktion beendet ist.

  Abhilfe: Vermeidung des Schedules einer Store-Instruktion unmittelbar vor einer RET-Instruktion.

  Beispiel:

  ```
  st.w [a0]0, d1
  ret                <-STALL
  ```

- Eine Store-Instruktion, die parallel zu einer MAC-Instruktion ausgeführt wird, verursacht einen Stall, wenn diese von der MAC-Instruktion abhängig ist [6]. Dies liegt darin begründet, dass MAC-Instruktionen mindestens zwei Taktzyklen benötigen, bis das Ergebnis berechnet wurde. Es ist somit kein EX→EX Forwarding möglich.

  Beispiel:

  ```
  madd d1, d1, d4, d6
  st.w [a0]0, d1       <-STALL
  ```

  Abhilfe: Vermeidung des Schedules einer abhängigen Store-Instruktion zusammen mit einer MAC-Instruktion.

- Stalls, verursacht durch das Lesen zweier Memory-Lines durch den Instruction Prefetch Mechanismus, wie in Abbildung 2.2 dargestellt. Dies liegt darin begründet, dass der Prefetch Mechanismus generell 64Bit einliest. Wenn die Anfangsadresse in einer Memory-Line liegt, die Endadresse dieser 64Bit in einer anderen Memory-Line, so kommt es zu einem Stallzyklus. Der Speicher des TriCore ist dabei in 256Bit große Memory-Lines partitioniert.

  Abhilfe: Bei linearem Code ist es möglich beim 128 Memory-Line-Bit ein 64Bit Bündel zu schedulen. Dadurch wird der Prefetch-Buffer geleert, somit muss der Prefetcher am 192. Memory-Line-Bit 64Bit einlesen. Dies verhindert effektiv Stalls. Bei Code mit vielen Verzweigungen ist dies leider nicht ohne weiteres umsetzbar, da hierfür alle möglichen Verzweigungen und z.B. die gesamte Codegröße einer Schleife in Betracht gezogen werden muss. Dies ist ohne Profiling-Informationen nicht ohne weiteres möglich.

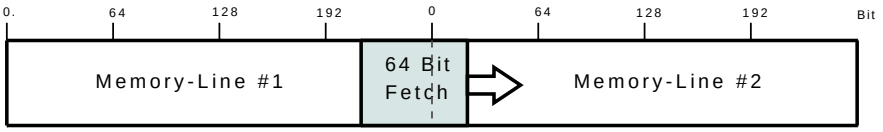

Abbildung 2.2: *Prefetch über zwei Memory-Lines, welches Stalls verursacht*

2.2 WCC

Um harte Realzeit-Systeme zu entwickeln, ist es in den meisten Fällen für den Entwickler notwendig, mehrmals die maximale Programmlaufzeit (WCET) zu bestimmen, und beim überschreiten der gewünschten Laufzeitschranke Optimierungen vorzunehmen, das Programm neu zu übersetzen und die WCET des Programms aufs neue zu überprüfen. Dieser Prozess kann sehr langwierig sein, weshalb es Anstrengungen gibt, den gesamten Prozess in den Compiler zu verlagern, um eine gewünschte Laufzeitschranke mittels automatisch vom Compiler gewählter Optimierungen zu gewährleisten.

Ein solcher Compiler wurde am Lehrstuhl 12 der Technischen Universität Dortmund unter dem Namen WCC [7], was für WCET-aware C Compiler steht, für den schon genannten Infineon TriCore Prozessor entwickelt. Dieser ist, wie in Abbildung 2.3 zu sehen, in ein Frontend gegliedert, welches auf der ICD-C [8] genannten High-Level IR operiert. Im ersten Schritt der Übersetzung wird der zu kompilierende Quelltext in diese Zwischendarstellung transformiert. Dabei bietet die ICD-C Datenstruktur verschiedene Optimierungs- und Analyseverfahren. Im zweiten Schritt wird die ICD-C IR dann in die ICD-LLIR [9] überführt. Diese Transformation wird durch den LLIR Code Selection durchgeführt. Dabei bietet auch die LLIR Optimierungs- und Analyseverfahren, es sind hierbei sowohl plattformunabhängige als auch architekturspezifische Low-Level-Optimierungen vorhanden. Auf Basis der LLIR bietet der WCC mittels eines WCET-Analyse-Tools zudem weitergehende, WCET-abhängige Optimierungen und Analysen.

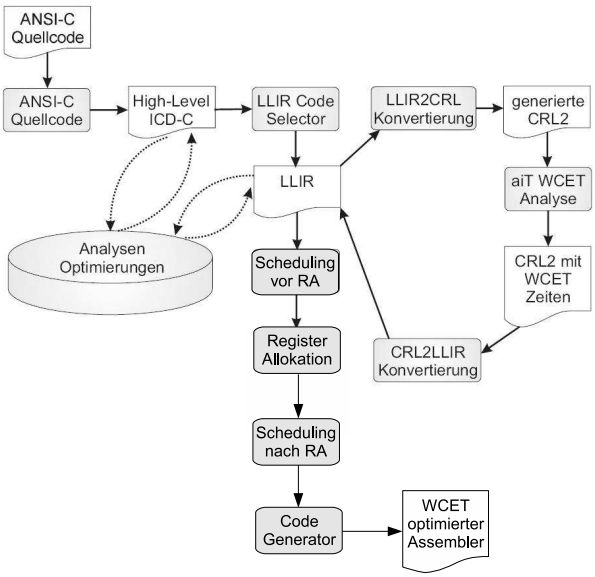

Abbildung 2.3: *Aufbau des WCC*

Als WCET-Analyse-Tool wurde aiT der Firma AbsInt [10] gewählt. Dieses wird noch näher in Ab-

schnitt 2.2.2.1 vorgestellt. Dieses liefert Laufzeitschranken eines kompilierten Programms. Diese können dann durch WCC benutzt werden, um automatisch Optimierungen auszuwählen und den optimierten Code einer neuen WCET-Analyse zuzuführen. Der Ablauf dieser automatischen WCET-Optimierung durch den WCC ist in Abbildung 2.3 ersichtlich. Die Koppelung von aiT an das WCC Framework wird hierbei in der LLIR2CRL Phase vorgenommen. Innerhalb dieser Phase wird die LLIR in ein AbsInt-proprietäres Format umgewandelt, welches den Kontrollflußgraphen des zu analysierenden Programms enthält. Nach der WCET-Analyse gibt aiT eine mit den WCET-Zeiten annotierte CRL2-Datei zurück, welche in der CRL2LLIR-Phase analysiert und die gewonnenen WCET-Zeiten in der LLIR annotiert werden.

Die Integration des in dieser Diplomarbeit vorgestellten Schedulers findet an zwei verschiedenen Stellen statt, nämlich vor sowie nach der Registerallokation (RA) innerhalb der im WCC verwendeten LLIR. Diese LLIR wurde am Informatik Centrum Dortmund (ICD) entwickelt und wird näher in Abschnitt 2.2.1 vorgestellt.

Durch die Integration des Schedulers sowohl vor als auch nach der Registerallokation ergeben sich unterschiedliche Möglichkeiten und Einschränkungen.

Scheduling vor der Registerallokation

Vor der Registerallokation sind alle Register innerhalb der LLIR virtuell, d.h. nicht in ihrer Anzahl durch die Zahl der realen Hardware-Register der Zielarchitektur limitiert. Somit werden nur gleiche Register für Instruktionen verwendet, die diese definiert haben oder als Parameter benötigen. Dadurch können keine falschen Abhängigkeiten entstehen, die ein Scheduling zusätzlich limitieren und damit Optimierungspotential verschenkt wird. Allerdings kann bei zu aggressivem Scheduling vor der Registerallokation die Anzahl der gleichzeitig lebendigen Register über den Wert der tatsächlich vorhandenen Hardwareregister ansteigen. Dies würde zur Generierung von sogenanntem Spill-Code innerhalb der Registerallokation führen. Dies sind zusätzliche Lade- und Speicherinstruktionen, die eingefügt werden, um Registerinhalte auf dem Stack zwischenzuspeichern. Durch diesen Spill-Code kann ein gutes Scheduling wieder zunichte gemacht werden und die Laufzeit des geschedulten Programms im Vergleich zum ungeschedulten Fall stark erhöhen.

Scheduling nach der Registerallokation

Ein Scheduling nach der Registerallokation hat dagegen den Vorteil, dass Spill-Instruktionen in den LLIR-Code eingefügt wurden und damit im Scheduling berücksichtigt werden können. Allerdings wirkt hier die schon erläuterte Limitierung durch falsche Abhängigkeiten einschränkend. Instruktionen, die nach der Registerallokation dieselben physikalischen Register beschreiben, können nicht mehr in ihrer Reihenfolge durch den Scheduler umgestellt werden.

Es sollen beide Verfahren innerhalb dieser Diplomarbeit implementiert und evaluiert werden. Da die

LLIR als Integrationspunkt des Schedulers innerhalb des WCC von großer Wichtigkeit ist, soll im nächsten Abschnitt näher auf die Struktur der LLIR eingegangen werden.

2.2.1 LLIR

Die ICD-LLIR stellt die Assembler-nahe Zwischendarstellung des zu übersetzenden Programms dar. Diese Darstellung wird im WCC aus der C-nahen ICD-C Darstellung durch den Code Selector erzeugt. Danach liegt das Programm in Form von Basisblöcken in der LLIR-Darstellung vor. Der genaue Aufbau der LLIR ist in Abbildung 2.4 zu sehen.

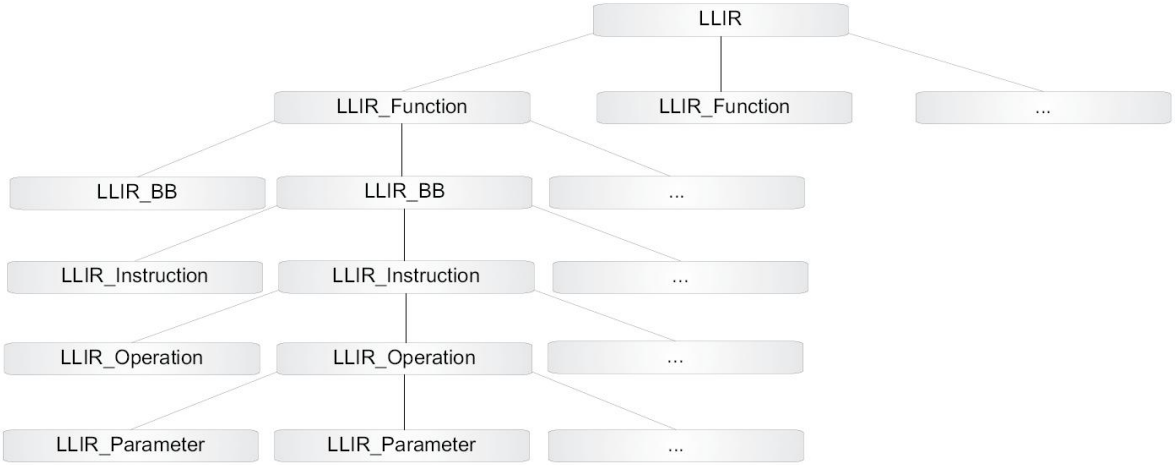

Abbildung 2.4: *Klassenhierarchie der ICD-LLIR*

Die Hierarchie der LLIR teilt sich dabei in folgende Klassen auf:

- LLIR - Die Oberklasse der LLIR. Entspricht einer einzelnen Quelldatei des zu übersetzenden Programms. Besteht aus Funktionen der Klasse LLIR_Function.
- LLIR_Function - Repräsentiert eine Funktion innerhalb des zu übersetzenden Programms. Besteht aus Basisblöcken der Klasse LLIR_BB.
- LLIR_BB - Entspricht einem Basisblock des zu übersetzenden Programms. Besteht aus hardwarenahen Anweisungen der Klasse LLIR_Instruction.
- LLIR_Instruction - Entspricht einer hardwarenahen Anweisung des Zielprozessors. Eine LLIR_Instruction kann hierbei aus mehreren Operationen bestehen, um Berechnungen eines Very Long Instruction Word (VLIW)-Prozessors darzustellen.
- LLIR_Operation - Ein einzelner Maschinenbefehl, ergänzt durch die benötigten sowie den durch die Operation zu definierenden Parameter, repräsentiert durch die Klasse LLIR_Parameter.

- LLIR_Parameter - Ein Parameter eines Maschinenbefehls. Es kann sich dabei um eine Integer-Konstante, ein Register, Label oder einen Operator handeln. Letztere dienen z.B. der Spezifikation von Adressierungsmodi.

Die Definition eines Basisblocks in der LLIR ist hierbei, dass alle Instruktionen zwischen zwei Verzweigungen des Kontrollflusses zu einem Basisblock gehören. Zu einer Verzweigung des Kontrollflusses gehören auch Funktionsaufrufe durch CALL-Instruktionen. Somit endet ein Basisblock auch durch einen Funktionsaufruf.

Eine weitere Fähigkeit der LLIR, die für den Scheduler von Wichtigkeit ist, besteht in der integrierten Def/Use-Analyse. Durch diese kann für jede Instruktion abgefragt werden, welche Register zur Zeit definiert sind, durch welche Instruktion diese zuletzt definiert wurden, durch welche Instruktion(en) diese verwendet werden und welche davon am Ende eines Basisblocks lebendig (alive) sind, d.h. eine Verwendung in einem nachfolgenden Basisblock erfahren.

2.2.2 Tools

Nachfolgend sollen nun Entwicklungs-Tools vorgestellt werden, die zur Entwicklung und Evaluierung des Schedulers verwendet wurden.

2.2.2.1 aiT

Das WCET-Analyse-Tool aiT der Firma AbsInt ermittelt statisch obere Laufzeitschranken eines Programms. Um dies zu erreichen, ist seine Arbeitsweise in mehrere Schritte aufgeteilt[11]. Abbildung 2.5 stellt dies anschaulich dar.

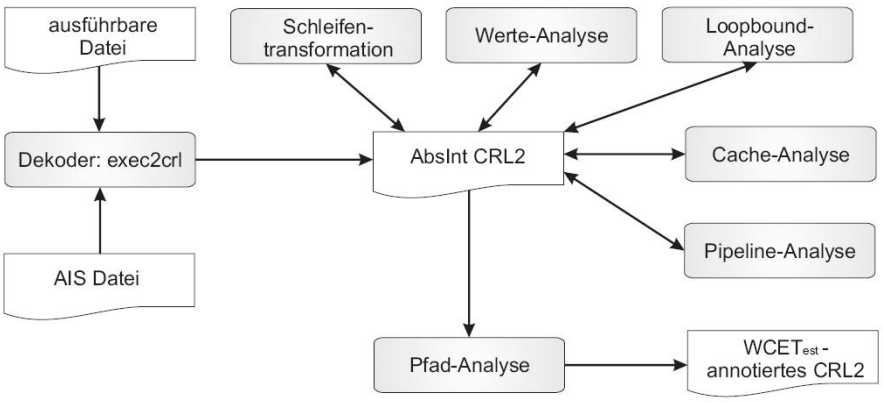

Abbildung 2.5: *Arbeitsweise des aiT WCET-Analyse-Tools*

Dem Tool wird durch den Entwickler eine ausführbare Datei des zu analysierenden Programms

übergeben, sowie eine AIS-Datei, in der benutzerdefinierte Annotationen zu finden sind. In dieser sind zusätzliche, für die WCET-Bestimmung benötigte, Informationen enthalten. In der exec2crl-Phase wird dann der Kontrollflußgraph des Programms mit benötigten Zusatzinformationen extrahiert. Es folgen mehrere Schritte, wie eine Transformation von Schleifen, um korrekt den Kontrollflußgraphen des zu analysierenden Programms zu bestimmen. Danach werden die Inhalte aller Register des zugrundeliegenden Prozessormodells für jeden Zustand des Programms bestimmt, um mittels dieser Informationen die Grenzen von Schleifen wie auch nicht ausführbarer Pfade (sog. infeasible paths) bestimmen zu können. Es folgen eine Cache-Analyse sowie als vorletzten Schritt die Simulation der Pipeline des Zielprozessors. All diese gewonnenen Informationen werden in jedem Schritt in die interne CLR2 Darstellung des Programms geschrieben und als letztes eine Pfadanalyse des Gesamtprogramms durchgeführt. Nach Bestimmung des Worst-Case Ausführungspfades, engl. Worst Case Execution Path (WCEP), kann die CRL2-Darstellung mit all diesen Zusatzinformationen für weitere Optimierungen verwendet werden. Der WCET-Pfad ist hierbei der Pfad eines Programms, durch dessen Ausführung die größtmögliche Ausführungszeit zustande kommt.

Werte-Analyse

Es ist so möglich, durch die gewonnenen Informationen der Werte-Analyse zu bestimmen, auf welche physikalische Speicherstelle Lade- sowie Speicher-Instruktionen Zugriff nehmen. Dies wurde im vorzustellenden Scheduler zum aggressiveren Scheduling von eben diesen Instruktionen verwendet. Dadurch lassen sich Datenabhängigkeiten durch gemeinsam genutzte Speicheradressen erkennen.

Pipeline-Visualisierung

Weitere Möglichkeiten ergibt sich durch die Pipeline-Analyse, welche durch ein weiteres Tool der Firma AbsInt, den Graphvisualisierer AiSee, unterstützt wird. Diese Analysie simuliert die Ausführungseinheiten des TriCore Prozessors in Software. Es wird dadurch möglich, die Ausführung eines Programms für den TriCore Prozessor Zyklus für Zyklus auf Pipeline-Ebene nachzuverfolgen. Hierdurch erhält man besseren Einblick in die internen Abläufe des Prozessors und kann Zusammenhänge sowie Einschränkungen bei der Ausführung spezifischer Instruktionssequenzen besser erfassen. Dies wurde im Rahmen der Entwicklung des Schedulers zur Bestimmung von Instruktionslatenzen sowie der Entwicklung und Verifikation des korrekten und effizienten Schedulings genutzt.

2.2.2.2 CoMET

CoMET ist ein Entwicklungstool der Firma VaST Systems Technology [12]. Es ermöglicht dem Entwickler das Erzeugen von virtuellen System-Prototypen für Eingebette Systeme. Dadurch ist eine gleichzeitige Entwicklung und Optimierung der Hardware und Software eines solchen Systems zu

betreiben. Dazu bietet CoMET eine Simulationsumgebung verschiedener Hardwarearchitekturen darunter der des Infineon TriCore Prozessors.

Die Simulationsumgebung wird innerhalb des WCC verwendet, um akkurate Werte der durchschnittlichen Laufzeit eines Programms zu bestimmen. Diese Werte sollen innerhalb dieser Diplomarbeit dazu dienen die Effektivität des entwickelten Schedulers zu bestimmen.

Kapitel 3

Lokales Scheduling

In folgendem Abschnitt soll das innerhalb der Diplomarbeit erarbeitete lokale Scheduling vorgestellt werden. Von einem lokalen Scheduling spricht man, wenn das Scheduling nur innerhalb der Grenzen eines Basisblocks stattfindet und keine Instruktionen zwischen unterschiedlichen Basisblöcken verschoben werden.

Zuerst werden in Abschnitt 3.1 verwandte Arbeiten vorgestellt. Danach wird in Abschnitt 3.2 die Funktionsweise des vorliegenden Schedulers erläutert. Als nächstes wird in Abschnitt 3.3 auf die Bestimmung des Datenabhängigkeitsgraphen eingegangen, um danach in Abschnitt 3.4 in das List Scheduling-Verfahren einzuführen. Hierzu soll dann in Abschnitt 3.5 auf die Prioritätsbestimmung von Instruktionen und die dabei verwendeten Heuristiken eingegangen werden, sowie auf weitere implementierte Heuristiken. In Abschnitt 3.6 soll dann auf Details eingegangen werden, die beim Scheduling vor und nach der Registerallokation beachtet werden muss. Abschnitt 3.7 stellt dann den als Teil des gesamten Schedulers integrierten optimalen Scheduler vor. Abschnitt 3.8 soll dann einen Überblick in den Aufbau der Implementation des Schedulers geben, indem wichtige Klassen und deren Methoden vorgestellt werden.

3.1 Verwandte Arbeiten

In diesem Abschnitt soll ein Überblick über die bisher veröffentlichten Forschungsarbeiten gegeben werden, die sich mit dem Thema des lokalen Schedulings befasst haben.

Das Problem des Instruction Schedulings für heutige superskalare Prozessoren ist als solches NP-vollständig. Dies wurde durch Hennessy und Gross in [13] gezeigt. Es wurde deshalb innerhalb weiterer Forschung versucht möglichst gute approximative Algorithmen zu finden. So haben Hennesy at al. einen der ersten Algorithmen für das Instruction Scheduling Problem entwickelt [13]. Gibbons und Muchnick haben auf Basis dieses Algorithmus einen effizienteren Algorithmus entwickelt, den *List Scheduling* Algorithmus [14]. Durch Wilken et. al wurde dann in [15] gezeigt, dass für eine

Prozessorarchitektur mit einer maximalen Latenzzeit von 3 Ausführungszyklen der List Scheduler Algorithmus mit einer Auswahl an Heuristiken in 99,7% der Basisblöcke der SPEC95 Benchsuite optimale Ergebnisse lieferte im Vergleich zu einem optimalen Scheduling.

Innerhalb dieses Kapitels soll ein List Scheduler für den TriCore Prozessor und mehrere Heuristiken vorgestellt werden. Zusätzlich wird ein optimaler Scheduler für den Infineon TriCore entwickelt, welcher als Referenz zum vorgestellten List Scheduler dienen soll.

3.2 Funktionsweise

Der beispielhafte Ablauf des Schedulers soll durch die folgende Abbildung 3.1 visualisiert werden. Zuerst wird für den zu scheduleden Basisblock der Datenabhängigkeitsgraph bestimmt. Die nachfolgenden Schritte sind Teil des List Scheduling-Verfahrens. Der erste Schritt des List Schedulings besteht darin, eine Liste der ausführbereiten Instruktionen, die sog. *Ready-List*, zu einem Zeitpunkt zu bestimmen. Falls mehrere Instruktionen zu einem bestimmten Zeitpunkt ausführbereit sind, wird die geeigneteste mittels Prioritäten, die durch unterschiedliche Heuristiken berechnet werden, bestimmt. Die Wahl der Prioritäten ist ein wichtiges Maß für die Güte des Schedulers. Das List Scheduling wird so lange ausgeführt, bis alle Instruktionen geschedulet wurden. In den nächsten Abschnitten soll näher auf diese Schritte eingegangen werden.

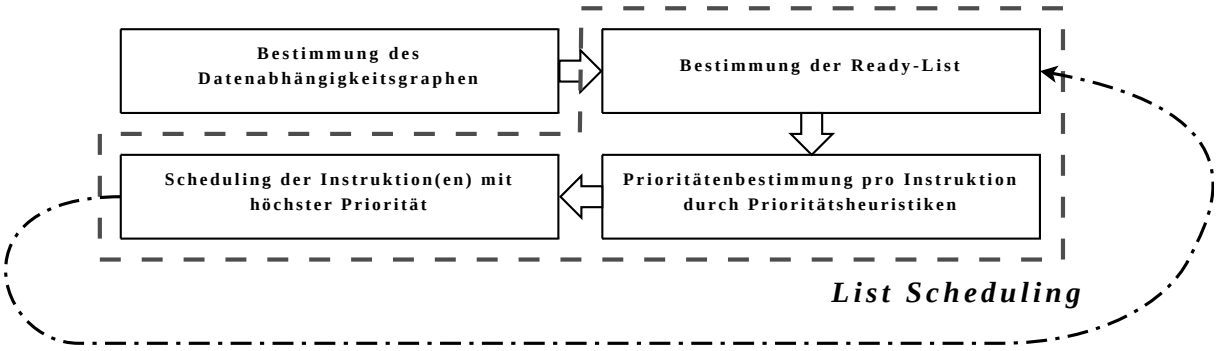

Abbildung 3.1: *Beispielhafter Ablauf eines lokalen Schedules*

3.3 Datenabhängigkeitsgraph

Der Datenabhängigkeitsgraph (DAG) ist ein azyklischer Graph, welcher Datenabhängigkeiten zwischen Instruktionen eines Basisblocks darstellt. Hierbei werden zwei Instruktionen i und j als Knoten dargestellt, eine mögliche Abhängigkeit zwischen diesen Instruktionen als Kante zwischen Knoten i und j. Abbildung 3.2 zeigt als Beispiel ein Codestück und den korrespondierenden DAG. Die Pfeilenden an Instruktion i8 sollen angedeutete Kontrollflußkanten zwischen jeder anderen Instruktion im DAG und i8 darstellen.

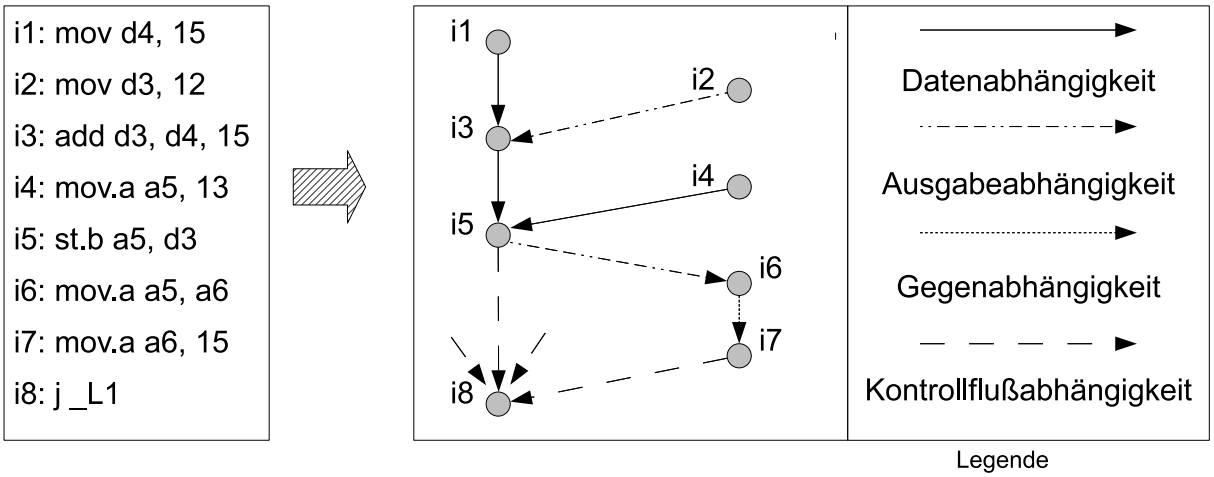

Abbildung 3.2: *Beispiel eines Datenabhängigkeitsgraphen*

Dabei stellt der DAG eine Halbordnung der im Codestück enthaltenen Instruktionen dar. Eine Instruktion *j*, welche abhängig von einer vorhergehenden Instruktion *i* ist, kann nicht vor Instruktion *j* ausgeführt werden.

Der DAG eines Basisblocks wird aufgestellt, indem für jede darin enthaltene Instruktion ein Knoten im DAG erzeugt wird, um nachfolgend für jede Nachfolge-Instruktion die Abhängigkeit zwischen beiden zu bestimmen. Dabei gilt es, wie aus Abbildung 3.2 ersichtlich, mehrere mögliche Abhängigkeiten zu ermitteln. Die möglichen Abhängigkeiten sind:

- **Datenabhängigkeit:** Eine Instruktion j ist datenabhängig von einer Instruktion i, wenn Instruktion j einen Parameter verwendet, der von Instruktion i berechnet wird. Zusätzlich werden zwischen Speicher-Instruktionen und nachfolgenden Lade-Instruktionen Datenabhängigkeiten modelliert, um den korrekten Datenfluß des Programms zu gewährleisten. Eine Umstellung solcher Instruktionen innerhalb eines Schedules kann eine Änderung des Datenflusses zur Folge haben, sollten beide die selbe Speicheradresse lesen/beschreiben. Eine in 3.5.1 vorgestellte Heuristik wird mittels Informationen, die durch eine Werte-Analyse gewonnen wurden, nur zwischen tatsächlich abhängigen Speicher- und Lade-Instruktionen Abhängigkeiten modellieren.
- **Ausgabeabhängigkeit:** Eine Instruktion j ist ausgabeabhängig von einer Instruktion i, wenn beide Instruktionen das selbe Zielregister beschreiben. Zusätzlich werden zwischen Diese Abhängigkeit ist eine falsche Abhängigkeit, die nur durch eine limitierte Anzahl an Registern entsteht. Somit tritt diese Abhängigkeit nur nach der Registerallokation auf. Diese Abhängigkeit wird auch zwischen Lade- und nachfolgenden Speicher-Instruktionen modelliert. Diese werden, wie auch im Falle der durch Speicher- und Lade-Instruktionen induzierten Datenabhängigkeiten, durch eine auf der Werte-Analyse basierenden Heuristik, nur auf die notwendigen Fälle reduziert.
- **Gegenabhängigkeit:** Eine Instruktion j ist gegenabhängig von einer Instruktion i, wenn In-

struktion i ein Parameter verwendet, das von Instruktion j berechnet wird. Diese Abhängigkeit ist, wie die Ausgabeabhängigkeit, eine falsche Abhängigkeit.

- **Kontrollflußabhängigkeit:** Eine Instruktion j ist kontrollflußabhängig von einer Instruktion i, wenn Instruktion j nach Instruktion i ausgeführt werden muss. Diese Art der Abhängigkeit kommt zum tragen, um zu gewährleisten, dass Instruktionen nicht hinter eine Sprung-Instruktion am Ende eines Basisblocks verschoben werden. Ein solches Verschieben würde die Semantik des Programms verändern, da die berechneten Ergebnisse der verschobenen Instruktionen nur noch innerhalb eines der nachfolgenden Kontrollfluss-Pfade definiert wären.

Als weitere Information innerhalb des DAG wird an jede Kante $\langle i, j \rangle$ zwischen zwei Instruktionen i und j die Latenz, zwischen beiden Instruktionen annotiert. Somit kann bei der nachfolgenden Bestimmung der Ready-List schnell auf diese zugegriffen werden.

Wie schon in Abschnitt 2.1.4 erläutert wurde, induzieren spezielle Abfolgen von Instruktionen zusätzlich Stall-Zyklen oder es finden sich Delay-Zyklen zwischen der Ausführung einer Instruktion i und der möglichen Verwendung des Registers, welches durch i beschrieben wird. Diese zusätzlichen Zyklen werden zur minimalen Latenz zwischen i und j addiert, um sowohl unerwünschte Stall-Zyklen durch genannte Instruktionsabfolgen zu vermeiden, als auch Delay-Zyklen durch Einfügen zu diesem Zeitpunkt ausführbereiter Instruktion auszunutzen.

$$latency(i, j) = minlatency(i, j) + stall(i, j) \tag{3.1}$$

Hierbei ist $minlatency(i, j)$ ein Wert aus einer der beiden Tabellen 2.3 und 2.2. $stall(i, j)$ ist ein möglicher zusätzlicher Stallzyklus, wie in Abschnitt 2.1.4 erläutert.

3.4 List Scheduling

List Scheduling basiert auf der Idee, dass für jeden Ausführungszyklus möglichst optimale Instruktionen für die Ausführung gefunden werden, die in den Ausführungseinheiten des Prozessors ausgeführt werden sollen. Dazu wird, solange sich in der Ready-List für den aktuellen Ausführungszyklus ausführbereite Instruktionen befinden, eine oder mehrere Instruktionen mittels Prioritäts-Heuristiken bestimmt. Im Falle des TriCore Prozessors wird versucht, wenn möglich, ein Bündel aus IP- und LS-Instruktionen zu schedulen. Nachdem die Ready-List für alle im Basisblock enthaltenen Instruktionen aufgestellt wurde, startet das List Scheduling im Ausführungszyklus 1. Der nachfolgende Pseudecode stellt den generellen Ablauf eines List Schedulings dar.

```
1 Eingabe: DAG
2 Ausgabe: Instruction Schedule
3
4 Bestimme Ready-List;
```

```
5  Ausführungszyklus = 1;
6  while( noch Instruktionen in Ready-List )
7  {
8     if ( keine Instruktionen im Ausführungszyklus verfügbar )
9       Ausführungszyklus++;
10    else {
11     Bestimme Prioritäten für ausführbereite Instruktionen im
          gegenwärtigen Ausführungszyklus;
12     Wähle Instruktion oder Bündel von IP-/LS-Instruktionen mit größter
          Priorität aus;
13     Schedule diese Instruktion(en), indem diese innerhalb der LLIR
          verschoben werden;
14     Erhöhe Ausführungszyklus um die Latenz der geschedulten Instruktion.
           Bei Bündeln nimm die größte Latenz, die eine der beiden
          Instruktionen aufweist;
15     Aktualisiere Ready-List;
16   }
17 }
```

Nachfolgend soll nun genauer auf die Aufstellung und Aktualisierung der Ready-List eingegangen werden.

3.4.1 Aufstellung der Ready-List

Die Ready-List der Region enthält für jede darin enthaltene Instruktion, ab welchem Zyklus diese zum ersten Mal ausführbar ist. Der Cycle-Wert einer Instruktion j muss hierbei folgender Gleichung genügen:

$$\forall i \in Pred(j) : Cycle(j) \geq Cycle(i) + latency(i, j) \tag{3.2}$$

Hierbei ist $Pred(j)$ die Menge der Vorgänger-Instruktionen einer Instruktion j im DAG, von denen j abhängig ist. $latency(i, j)$ ist die Latenz zwischen Instruktion i und nachfolgender Instruktion j. Dadurch wird gewährleistet, dass keine Instruktion vor der Berechnung ihrer benötigten Parameter geschedulet wird.

Es gibt hierbei zwei unterschiedliche Algorithmen, die zur Bestimmung der Cycle-Werte einer Instruktion verwendet werden können. Zum einen ist dies der „As Soon As Possible“ (ASAP) - Algorithmus und zum anderen der „As Late As Possible“ (ALAP) - Algorithmus. Beide bestimmen dabei maßgeblich das weitere Scheduling, da durch die bestimmten Cycle-Werte die frühestmögliche Ausführung einer Instruktion und somit das Intervall, in dem diese verschoben werden kann, festge-

legt wird.

Beide Algorithmen werden vom entwickelten Scheduler unterstützt und sollen nachfolgend vorgestellt werden.

ASAP

Der ASAP-Algorithmus weist einer Instruktion j einen frühestmöglichen Cycle-Wert zu. Es gilt dabei folgende Gleichung:

$$Cycle(j) = \begin{cases} 1 & : \quad Pred(j) = \emptyset \\ Max_{i \in Pred(j)}(Cycle(i) + latency(i,j)) & : \quad sonst \end{cases} \tag{3.3}$$

Den Cycle-Werten des Codestücks aus Abbildung 3.2 werden durch den ASAP-Algorithmus folgende, in Abbildung 3.3 ersichtliche, Werte zugewiesen.

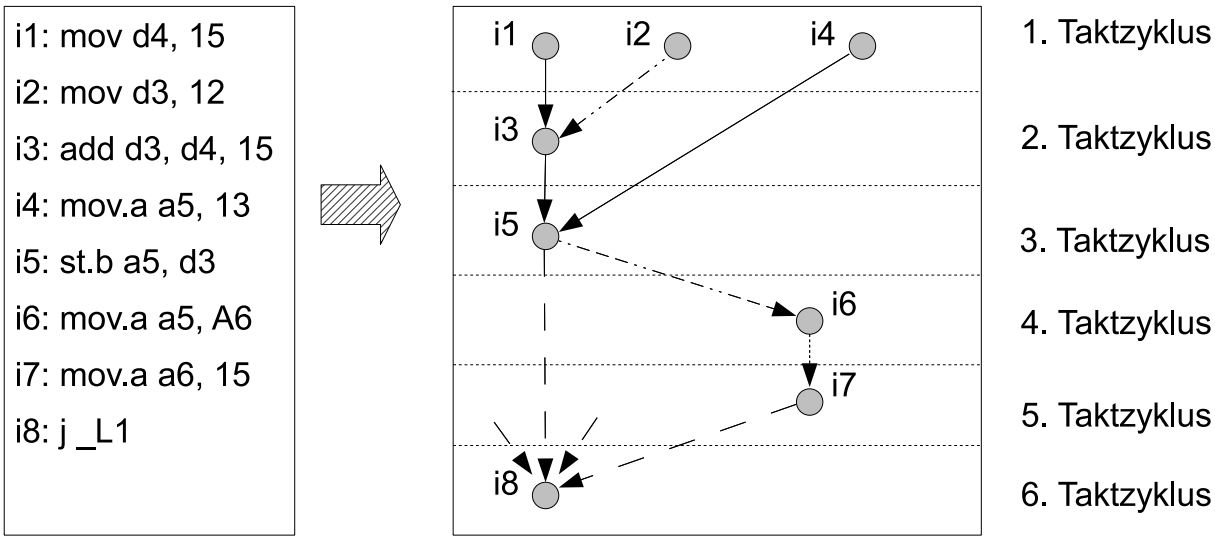

Abbildung 3.3: *Beispiel eines ASAP-Schedules*

ALAP

Der ALAP-Algorithmus weist dem gegenüber einer Instruktion i einen spätestmöglichen Cycle-Wert zu. Es gilt folgende Gleichung:

$$Cycle(i) = \begin{cases} maxASAPCycle & : \quad Succ(i) = \emptyset \\ Min_{j \in Succ(i)}(Cycle(j) - latency(i,j)) & : \quad sonst \end{cases} \tag{3.4}$$

Hierbei ist $Succ(i)$ die Menge aller Nachfolger-Instruktionen einer Instruktion i im DAG, von denen i abhängig ist. $maxASAPCycle$ ist der größte Cycle-Wert, der durch einen vorhergehenden Durchlauf des ASAP-Algorithmus bestimmt wurde.

Den Cycle-Werten des Codestücks aus Abbildung 3.2 werden durch den ALAP-Algorithmus entsprechend folgende, in Abbildung 3.4 ersichtliche, Werte zugewiesen.

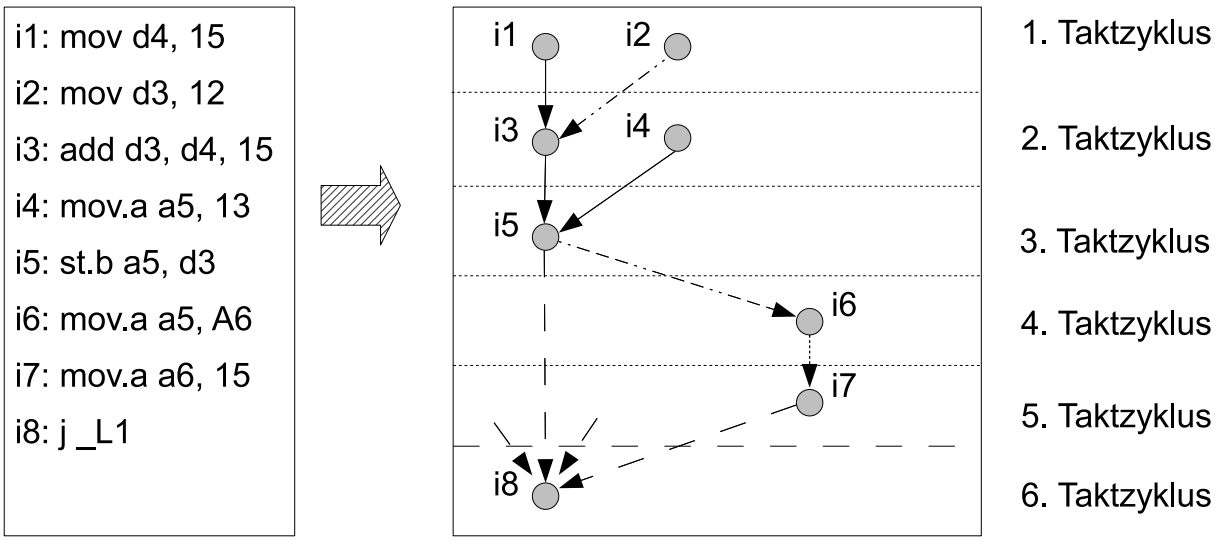

Abbildung 3.4: *Beispiel eines ALAP-Schedules*

Es ist im Vergleich zwischen den ASAP und ALAP Cycle-Werten ersichtlich, dass Instruktion i4 unterschiedlich gescheduled werden kann. Dies kann im Falle des Infineon TriCore, wegen der in Kapitel 2 genannten Einschränkung bzgl. der Parallelisierung von IP- sowie LS-Instruktionen, je nach Klassifikation von i4 und der gleichzeitig ausführbaren restlichen Instruktionen zu einer guten Parallelisierung führen, oder nicht. Dies soll in Kapitel 5 evaluiert werden.

In Tabelle 2.3 von Latenzen zwischen Instruktionsklassen ist hierbei von Interesse, dass IP-Instruktionen 0 Taktzyklen an Latenz zu einer nachfolgenden, von dieser abhängigen, LS-Instruktion aufweist. Dadurch erhalten miteinander parallel ausführbare Instruktionen gleiche Cycle-Werte zugewiesen, wenn alle Abhängigkeiten zu diesem Taktzyklus erfüllbar sind. Somit können Paare an Instruktionen gebildet werden, welche gleichzeitig ausführbar sind.

Der Scheduler bestimmt für jede im Basisblock enthaltene Instruktion dabei sowohl den ASAP- als auch den ALAP-Wert. Dies ist in zweierlei Hinsicht notwendig. Zum einen ist bei der Bestimmung der ALAP-Werte eine obere Grenze der Zyklen notwendig, die dann Instruktionen zugewiesen wird, die keine Abhängigkeiten innerhalb der Region aufweisen. Diese Grenze kann am einfachsten durch die Verwendung des Maximalwerts der Zyklen innerhalb des ASAP-Algorithmus für diese Region bestimmt werden. Zum anderen sind sowohl ASAP- und ALAP-Wert pro Instruktion für die Bestimmung der sogenannten Mobilität (engl. Mobility) einer Instruktion notwendig. Auf die Bestimmung dieser und des sogenannten „Maximum Delay“ einer Instruktion soll nachfolgend eingegangen werden. Bei der Bestimmung der ASAP- bzw. ALAP-Werte wird hierbei von jeder Instruktion innerhalb des Basisblocks rekursiv eine Bestimmung dieser gestartet. Der Anfangswert für jede Instruktion ist dabei im Falle von ASAP 1, im Fall von ALAP die obere Grenze der ASAP-Bestimmung.

Im Falle des ersten Zyklus eines Basisblocks wird ausserdem überprüft, ob dieser Basisblock ein

Nachfolger eines Basisblocks ist, welcher mit einer CALL- oder RET-Instruktion endet. Wie schon in 2.1.4 aufgeführt wurde, ist die Ausführung einer LS-Instruktion direkt nach einer der beiden Instruktionen nicht möglich. Diese Einschränkung ist nicht innerhalb des Abhängigkeitsgraphen kodierbar, da dieser sich nur auf einen Basisblock bezieht. Stattdessen wird der ersten LS-Instruktion mindestens ein Cycle-Wert von 2 zugewiesen.

Nach dem Schedule einer Instruktion oder eines Bündels ist es wichtig, dass die Ready-List aktualisiert wird. Dies ist notwendig, da durch die Ready-List die meisten Abhängigkeiten zwischen Instruktionen gewährleistet werden. Die Aktualisierung der Ready-List findet durch folgende Schritte statt. Hierbei ist zu beachten, dass die Aktualisierung erst nach erfolgter Inkrementierung des Ausführungszyklus stattfindet.

- Aktualisierung der Cycle-Werte aller Instruktionen in der Ready-List, welche einen kleineren Cycle-Wert als den neuen Ausführungszyklus-Wert aufweisen auf den neuen Wert. Aktualisiere davon ausgehend rekursiv auch alle Nachfolgeinstruktionen über die Abhängigkeitskanten im DAG. Davon ausgenommen sind die geschedulten Instruktionen.
- Aktualisierung der Cycle-Werte der Nachfolger der geschedulten Instruktionen auf den neuen Ausführungszyklus-Wert, wenn diese einen kleineren Cycle-Wert aufweisen.
- Entfernung der geschedulten Instruktion(en) aus der Ready-List.

3.5 Heuristiken

Um aus mehreren ausführbereiten Instruktionen eine möglichst optimale auszuwählen, wurden im vorliegenden Scheduler eine Reihe von Heuristiken implementiert. Diese werden zur Bestimmung der Priorität einer Instruktion verwendet. Einige, wie die „Maximum Delay“-Heuristik sind generische Heuristiken, die architekturunabhängig sind. Daneben gibt es TriCore-spezifische Heuristiken, die das Schedulen eines effizienten Codes für den TriCore Prozessor gewährleisten. Es ist dabei möglich, mehrere Heuristiken gleichzeitig zu verwenden. Die Summe dieser Prioritätswerte werden dann als Gesamtpriorität der betreffenden Instruktion zugewiesen. Hierbei wird zur Zeit jeder Prioritäts-Heuristik die selbe Gewichtung innerhalb der Gesamtpriorität zugeteilt. Ausnahmen hierzu sind die TriCore-spezifische Heuristik „Instruction Priority“ sowie die Heuristik der Minimierung der Registerzahl, welche als Faktor zu den bereits bestimmten Prioritäten multipliziert werden. Dies gilt auch für die Mobility-Heuristik. Eine Verwendung dieser Heuristiken ist ohne Wahl einer zusätzlichen anderen Prioritäts-Heuristik nicht sinnvoll.

3.5.1 Prioritäts-Heuristiken

Innerhalb der Prioritäts-Heuristiken gibt es eine Unterscheidung zwischen statischen und dynamischen Heuristiken. Statische Heuristiken bestimmen die Prioritäten aller Instruktionen einmalig am

Anfang eines Schedules, dynamische aktualisieren dagegen die Priorität jeder Instruktion in jedem neuen Ausführungszyklus. Es sollen nachfolgend alle beim lokalen Scheduling verwendbaren Heuristiken vorgestellt werden.

3.5.1.1 Generische Heuristiken

Maximum Delay

Die Maximum Delay-Heuristik weist einer Instruktion die Länge des maximalen Pfades zwischen der Senke eines Basisblocks und der betreffenden Instruktion im DAG, wobei die Senke eines Basisblocks z.B. die am Ende dessen zu findende Sprung-Instruktion ist. Dabei gilt, dass der Maximum Delay-Wert der Senke 1 ist und davon ausgehend rekursiv über die Abhängigkeitskanten des DAG durch die an den Kanten annotierten Instruktionslatenzen inkrementiert wird. Es gilt dabei folgende Gleichung:

$$maxDelay(i) = \begin{cases} 1 & : \quad Succ(i) = \emptyset \\ Min_{j \in Succ(i)}(maxDelay(j) + latency(i,j)) & : \quad sonst \end{cases} \tag{3.5}$$

Instruktionen, die auf dem sogenannten *kritischen Pfad* innerhalb des Basisblocks liegen, wird dadurch eine höhere Prioriät zugeteilt. Als kritischer Pfad wird der längste Pfad innerhalb eines Codes genannt. Dies hat den Vorteil, dass Instruktionen, die viele Nachfolger haben, früh geschedulet werden und somit auch die Bedingungen für die Ausführung vieler weiterer Nachfolger erfüllt sind. Dadurch wird der kritische Pfad nicht unnötig in der Ausführung verzögert und die Instruktionen abseits des kritischen Pfades können zur Formung von Instruktionsbündeln mit den Instruktionen auf dem kritischen Pfad hinzugezogen werden. Innerhalb der maxDelay-Werte ist der kritische Pfad der größte maxDelay-Wert einer Instruktion. Der längste Pfad zwischen dieser und der Senke des Basisblocks ist der kritische Pfad.

Diese Heuristik wird statisch bestimmt.

Mobility

Die Mobility-Heuristik bestimmt die mögliche Beweglichkeit einer Instruktion. Die Idee hinter dieser Heuristik ist es, Instruktionen, die eine niedrige oder keine Mobilität innerhalb des Codes aufweisen, eine hohe Priorität zuzuweisen. Generell weisen Instruktionen innerhalb des kritischen Pfades eines Codes eine niedrige Mobilität auf, somit werden auf dem kritischen Pfad liegende Instruktionen durch die Mobility-Heuristik bevorzugt. Der Mobility-Wert einer Instruktion j wird mittels folgender Gleichung berechnet:

$$mobility(j) = ALAP(j) - ASAP(j) \tag{3.6}$$

$ALAP(j)$ ist dabei der vom ALAP-Algorithmus, $ASAP(j)$ der vom ASAP-Algorithmus für j bestimmte Cycle-Wert.

Innerhalb des vorgestellten Schedulers wird der Mobility-Wert in Kombination mit der Maximum Delay- oder Number of Child-Instructions-Heuristik dazu verwendet, mögliche Delay-Zyklen zu erkennen und so gut wie möglich zu füllen. Dazu wird Instruktionen, die die selbe Maximum Delay-Priorität aufweisen mittels der Mobility-Heuristik weiter differenziert. Eine solche mögliche Situation ist in Abbildung 3.5 ersichtlich.

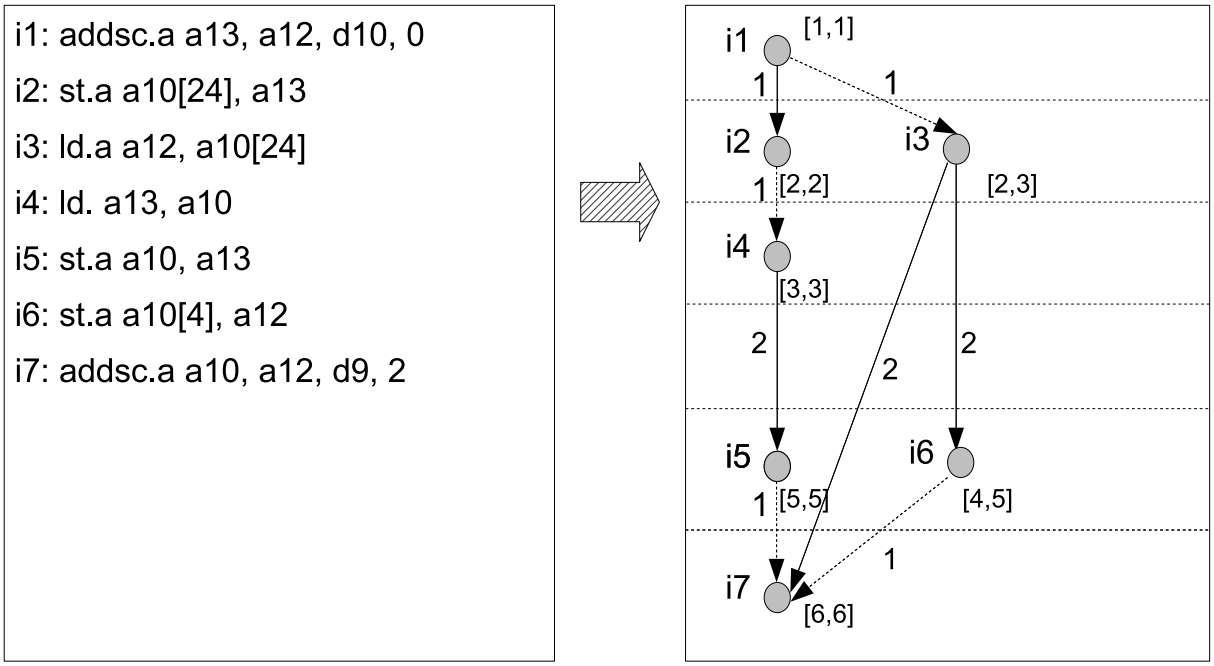

Abbildung 3.5: *Beispielhafte Situation der Verwendung der Mobility-Heuristik*

In der Abbildung ist rechts vom Beispielcode der DAG des Codes zu sehen. An den Abhängigkeitskanten sind zusätzlich die Latenzen zwischen den einzelnen Instruktionen annotiert. Wie hier zu sehen ist, weisen sowohl Instruktion i3 als auch i4 eine Latenz von 2 auf, da hier ein Delay-Zyklus zwischen der Ausführung der Ladeinstruktionen und der nachfolgenden Benutzung der Werte innerhalb der beschriebenen Register auftritt (siehe 2.1.4). Innerhalb der eckigen Klammern an jeder Instruktion sind deren ASAP- sowie ALAP-Wert ablesbar. Wie zu sehen ist, weist Instruktion i4 dabei einen Mobility-Wert von 0 auf, während Instruktion i3 einen Mobility-Wert von 2 aufweist. Beide Instruktionen weisen den selben Maximum Delay-Wert auf, wodurch zwei verschiedene Schedules von i3 und i4 möglich wären. Mit der Mobility-Heuristik wird i3 nach i4 geschedulet und füllt den Delay-Zyklus zwischen i4 und i5 aus. Instruktion i6 wird zwischen i5 und i7 geschedulet. Ein Vergleich zwischen originalem Code und geschedultem Code ist in Abbildung 3.6 zu sehen. Durch die Umstellung von i3 und i4 wurde ein Taktzyklus gespart.

Diese Heuristik wird dynamisch in jedem neuen Ausführungszyklus bestimmt. Verwendung nur zusätzlich zur Mobility- oder Number of Child-Instructions-Heuristik möglich.

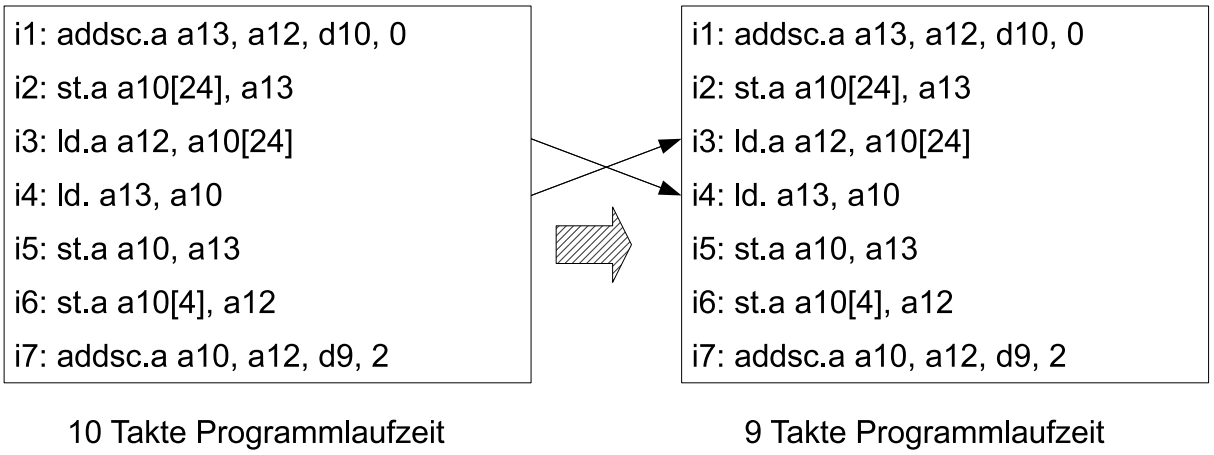

Abbildung 3.6: *Schedule des Beispielcodes mittels Maximum Delay- und Mobility-Heuristik*

Number of Child-Instructions

Die „Number of Child-Instructions“-Heuristik weist jeder Instruktion die Anzahl der von ihr abhängigen Instruktionen innerhalb des DAG als Priorität zu. Hierdurch werden Instruktionen, von denen eine größere Anzahl an Nachfolger-Instruktionen abhängen, bevorzugt. Durch Scheduling einer Instruktion mit einer höheren Anzahl an Nachfolgern steigt die Wahrscheinlichkeit, dass im nächsten Ausführungszyklus mehr Instruktionen zur Auswahl stehen. Dadurch steigen auch die Möglichkeiten der Bündelung von Instruktionen im Falle des TriCore Prozessors.

Diese Heuristik wird dynamisch in jedem neuen Ausführungszyklus bestimmt.

Registerzahl-Minimierung

Beim Scheduling vor der Registerallokation sind die Möglichkeiten der Bündelung an Instruktion durch das Nichtvorhandensein falscher Abhängigkeiten im Vergleich zum Scheduling nach der Registerallokation erheblich höher. Allerdings kann ein Greedy-Ansatz der Bündelung von Instruktionen zwecks Maximierung der Anzahl an Bündeln dazu führen, dass sehr viele Register gleichzeitig lebendig innerhalb des Codes sind. Dies hat zur Folge, dass während der Registerallokation Spill-Code eingeführt wird, wodurch zusätzliche Ausführungszyklen innerhalb des Programmcodes benötigt werden. Dies kann den Vorteil eines Schedulings negieren. Es sollte also durch den Scheduler versucht werden die Anzahl an gleichzeitig lebendigen Registern so niedrig wie möglich zu halten. Dies soll durch die Registerzahl-Minimierungs-Heuristik realisiert werden. Diese bevorzugt Instruktionen, die über einen Pfad innerhalb des DAG von den zuvor geschedulten Instruktionen aus erreichbar sind. Somit werden Instruktionen bevorzugt, die potenziell Teil einer Kette von Instruktionen sind, welche zu einem Gesamtergebnis führen. Durch ein solches Scheduling werden Lebenszeiten von Registern so niedrig wie möglich gehalten. Abbildung 3.7 zeigt einen DAG, in dem zwei Ketten von Instruktionen gestrichelt umrandet sind. Die Kette [i1, i2, i4, i6] berechnet da-

bei einen Parameter von i8, Kette [i3, i5, i7] einen weiteren. Um die Anzahl gleichzeitig lebendiger Register zu minimieren, sollte eine Kette zuerst geschedult werden, danach die andere. Die hier vorgestellte Heuristik würde im Falle von schon geschedulten Instruktionen i1 und i2 erkennen, dass eine Abhängigkeit besteht zwischen i4 und i2 und dieser Instruktion eine höhere Priorität geben. Die Heuristik betrachtet dabei das zuletzt geschedulte Bündel an Instruktionen und prüft auf Pfade im DAG zwischen den Instruktionen im zuletzt geschedulten Bündel und der zu prioritisierenden Instruktion. Durch Betrachten des zuletzt geschedulten Bündels von IP- und LS-Instruktion ist es dabei möglich Abhängigkeiten sowohl innerhalb der Daten- als auch Adressregister festzustellen.

Diese Heuristik wird dynamisch in jedem neuen Ausführungszyklus bestimmt. Verwendung nur zusätzlich zur Mobility- oder Number of Child-Instructions-Heuristik möglich.

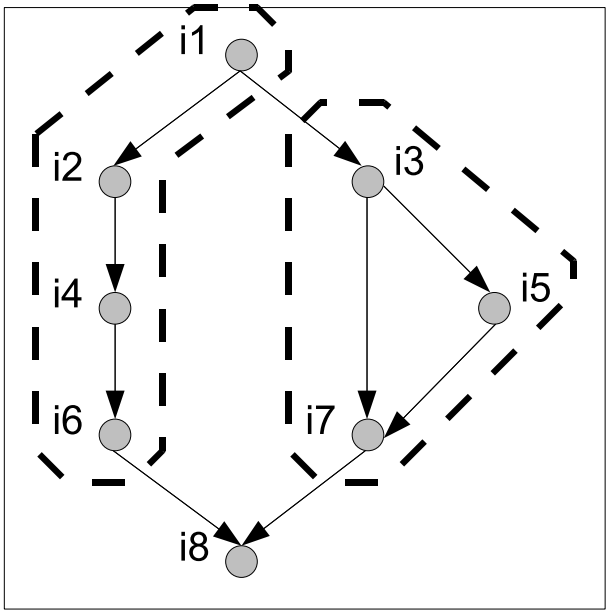

Abbildung 3.7: *Beispiel eines DAGs mit Ketten von Instruktionen*

3.5.1.2 TriCore-spezifische Heuristiken

Instruction Priority

Um einen möglichst hohen IPC-Wert zu erreichen und dadurch die mittlere Ausführungszeit des zu schedulenden Programms zu minimieren, ist im Falle des TriCore Prozessors eine Bündelung von IP- sowie LS-Instruktionen unerlässlich. Um anzuzeigen, dass ein Bündel aus zwei Instruktionen beim Scheduling zu bevorzugen ist, wird dem betreffenden Bündel die Summe der Prioritäten beider Einzel-Instruktionen zugewiesen. Zusätzlich wird, abhängig von der Instruktionsklasse der zuletzt geschedulten Instruktion, die Priorität jeder Instruktion innerhalb des Bündels mit einem bestimmten Faktor multipliziert. Diese Heuristik wird Instruction Priority-Heuristik genannt. Tabelle 3.1 soll die verwendeten Faktoren und die Fälle, in denen sie verwendet werden, vorstellen.

Tabelle 3.1: *Faktoren der Instruction Priority-Heuristik*

Vorgänger-Instruktionsklasse	Instruktionsklasse der derzeitigen Instruktion		
	Integer-Instruktion	Load/Store-Instruktion	Dual-Instruktion
Anfang eines Basisblocks	2	1	3
Integer-Instruktion	1	3	2
Load/Store-Instruktion	2	1	3
Dual-Instruktion	2	1	3

Wie aus der ersten Zeile der Tabelle zu erkennen ist, werden Dual-Instruktionen am Anfang eines Basisblocks bevorzugt, indem die Priorität dieser mit Faktor 3 multipliziert wird. Dies hat den Hintergrund, dass Dual-Instruktionen mit keiner anderen Instruktion bündelbar sind, es aber IP- und LS-Instruktionen geben kann, die vom Ergebnis dieser abhängen. Somit ist ein Scheduling dieser Instruktionen so früh wie möglich vorteilhaft. Integer-Instruktionen erhalten einen Faktor von 2. Da LS-Instruktionen immer mit IP-Instruktionen zu bündeln sind, ist ein Scheduling einer LS-Instruktion direkt am Basisblock-Anfang mit Faktor 1 zu benachteiligen.

Tritt nach einer Integer-Instruktion eine LS-Instruktion auf, so wird der LS-Instruktion ein Faktor von 3 zugewiesen. Dadurch wird eine Bündelung von IP- und LS-Instruktionen, wenn möglich, erzwungen. Die Situation nach dem Scheduling einer LS- sowie Dual-Instruktion ist äquivalent zur Situation am Anfang des Basisblocks. Es gelten die selben Faktoren.

Nachfolgend soll nochmals ein Beispiel der Prioritätsbestimmung eines Bündels von IP- und LS-Instruktionen mit sowohl der Maximum Delay- als auch der Instruction Priority-Heuristik vorgestellt werden. Es soll angenommen werden, dass die Bündelung am Anfang eines Basisblocks stattfindet.

$$\left.\begin{array}{lll} \text{mov d4, 24} & \longrightarrow & maxDelay : 10 \\ \text{st.w [a0]0 , d1} & \longrightarrow & maxDelay : 8 \end{array}\right\} \text{Gesamtpriorität = 10 * 2 + 8 * 3 = } \mathbf{46}$$

Diese Heuristik wird dynamisch in jedem neuen Ausführungszyklus bestimmt. Verwendung nur zusätzlich zur Mobility- oder Number of Child-Instructions-Heuristik möglich.

3.5.1.3 Gewährleistung der Korrektheit innerhalb der Prioritätsbestimmung

Die gewählte Implementierung der Ready-List, alle Instruktionsklassen zentral innerhalb einer gemeinsamen Ready-List zu speichern, hat sowohl Vorteile als auch Nachteile. Bedingt durch die Anzahl an Abhängigkeitszyklen von 0 Taktzyklen zwischen IP- und nachfolgenden LS-Instruktionen werden innerhalb der Ready-List einer von einer IP-Instruktion abhängigen LS-Instruktion die gleichen Ausführungszyklen zugewiesen, wie es die vorhergehende IP-Instruktion aufweist. Dieser Fall tritt auch auf, wenn mehr als eine IP-Instruktion im gleichen Taktzyklus ausführbar sind und beide

Register beschreiben, die von einer sofort nachfolgenden LS-Instruktion verwendet werden. Eine Ausführung der LS-Instruktion in diesem Fall mit einer der IP-Instruktionen ohne vorhergehende Ausführung der zweiten IP-Instruktion würde eine Datenabhängigkeit verletzen. Ein Auflösen dieses Problems kann hier an zwei Stellen erfolgen. Zum einen wäre eine Erhöhung der Latenz der betreffenden LS-Instruktion innerhalb der Aufstellung der Ready-List möglich, wenn ein solcher Fall erkannt wird. Zum anderen ist es möglich diesem Fall bei der Bündelung von LS-Instruktionen mit IP-Instruktionen zu erkennen und solche Bündel zu verwerfen, bis alle benötigten Registerwerte zur Ausführung der LS-Instruktion verfügbar sind. Es wurde innerhalb der Konzeptphase für den vorliegenden Scheduler zugunsten der letzteren Variante entschieden.

Weiterhin ist zu beachten, dass LS-Instruktionen, die von einer im gleichen Ausführungszyklus ausführbaren IP-Instruktion abhängig sind, nicht mit einer anderen IP-Instruktion gebündelt werden. Dadurch würde eine Datenabhängigkeit verletzt werden. Dies ist ein Spezialfall des vorhergehenden Falls. Solche Bündel an Instruktionen werden allen anderen Bündeln vorgezogen, um die Semantik des Progamms zu gewährleisten.

3.5.2 Weitere Heuristiken

Verbesserte Bestimmung von Lade-/Speicher-Abhängigkeiten

Wie schon im Abschnitt 3.3 über den Aufbau des DAG beschrieben, werden Abhängigkeiten zwischen Lade- und Speicher-Instruktionen sowie Speicher- und Lade-Instruktionen generell modelliert, da ohne weitere Informationen bezüglich der von den Instruktionen verwendeten Registerwerte kein Rückschluss auf die zugegriffenen Speicheradressen gezogen werden kann. Dadurch werden konservative Abhängigkeiten eingeführt, die zu einer unnötigen Einschränkung innerhalb des Schedulins der auf den Speicher zugreifenden Instruktionen führen. Diese redundanten Abhängigkeiten sollen durch die Verwendung von zusätzlichen, aus der Werte-Analyse des WCET-Analyse-Tools aiT (siehe 2.2.2.1) bezogenen Informationen vermieden werden. Nach erfolgter Werte-Analyse durch aiT werden die physikalischen Adressen, auf welche die Speicherzugriffe stattfinden, innerhalb der LLIR mittels sogenannten Pragmas annotiert. Diese Informationen werden dann verwendet, um nur für tatsächlich auf die gleiche Speicheradresse zugreifenden Lade- sowie Speicher-Instruktionen Abhängigkeiten zu modellieren.

3.6 Lokales Scheduling vor und nach der Registerallokation

Um zu gewährleisten, dass der ungeschedulte, durch den LLIR Code Selector erzeugte, Programm-Code innerhalb der LLIR weiterhin wie vom Register-Allokator erwartet aufgebaut ist, müssen beim Scheduling vor der Registerallokation weitere Abhängigkeiten innerhalb des DAG eingeführt werden. Diese betreffen den ersten Basisblock jeder Funktion des zu übersetzenden Programms. Durch

den LLIR Code Selector werden in diesen Basisblöcken Instruktionen eingefügt, die das Register A[10], welches als Stackpointer fungiert, intialisiert. Da es innerhalb der LLIR keine weiteren Abhängigkeiten zum Register A[10] vor der Registerallokation gibt, würde der Scheduler diese Instruktionen mit hoher Wahrscheinlichkeit verschieben. Durch die Verwendung des Stacks innerhalb des durch die Registerallokation erzeugten Spill-Codes führt ein Verschieben dieser Instruktionen allerdings möglicherweise zu einem nicht initialisierten Stackpointer innerhalb der Lade- und Speicherinstruktionen, welche Register auf den Stack auslagern. Um dies zu verhindern, wird zwischen den Instruktionen zur Initialisierung des Stackpointers und weiteren darauf folgenden Instruktionen Kontrollflußabhängigkeiten eingeführt. Dies deswegen, da es sich hier um keine Datenabhängigkeiten im eigentlichen Sinne handelt. Durch diese Abhängigkeiten wird ein Verschieben der Stackpointer initialisierenden Instruktionen verhindert. Solche zusätzlichen Abhängigkeiten sind im Falle des Schedulings nach der Registerallokation nicht notwendig. Hier ist möglicher Spill-Code schon vorhanden und es werden korrekt Abhängigkeiten dazwischen modelliert. Wurde ein Scheduling vor der Registerallokation durchgeführt, so bietet es sich an zusätzlich ein Scheduling nach der Registerallokation durchzuführen. Etwaige Spill-Code-Instruktionen werden so besser innerhalb des Codes verteilt, um die Parallelität des Codes zu verbessern.

3.7 Optimales Scheduling

Innerhalb des TriCore-Schedulers wurde zusätzlich zum List Scheduling ein auf Integer Linear Programming (ILP) basierender optimaler Scheduler implementiert. Dieser stellt auf Basis des Datenabhängigkeitsgraphen, welcher in Abschnitt 3.3 vorgestellt wurde, Systeme von Gleichungen und Ungleichungen auf, die von einem sogenannten Solver für ILP-Gleichungssysteme gelöst werden. Als Solver werden zum einen das kommerzielle CPLEX [16] der Firma ILOG sowie das frei erhältliche lp_solve [17] unterstützt. Durch eine Implementation eines optimalen Schedulers ist es möglich Aussagen über die mögliche Optimalität des lokalen Schedulings aufzustellen. Es wird hierzu eine erweiterte Statistik innerhalb des TriCore-Schedulers geführt, die Aussagen über die Anzahl der erzeugten IP/LS-Bündel sowie dadurch gesparter Ausführungszyklen, durch den im Vergleich zum List Scheduling erfolgten optimalen Scheduling, macht. Die Aufstellung des linearen Gleichungssystems ist angelehnt an [18], wurde aber dahingehend erweitert, dass die 3-fach superskalare Architektur des TriCore Prozessors berücksichtigt wird.

Vor Aufstellung der linearen Gleichungen und Ungleichungen werden zuerst eine obere sowie eine untere Ausführungsschranke aufgestellt. Die obere Ausführungsschranke O ist hierbei die normalisierte Anzahl an Taktzyklen, die ein Schedule des betreffenden Basisblocks durch den List Scheduler benötigt. Normalisiert bedeutet hierbei, dass für alle Instruktionslatenzen größer 1 der Wert 1 angenommen wird. Dies ist im Falle des TriCore Prozessors möglich, da Instruktionen mit Latenz ≥ 1 generell die Ausführung weiterer Instruktionen anhalten, bis die letzte EX-Phase dieser beendet wurde. Durch diese Maßnahme ist eine Vereinfachung des linearen Gleichungssystems möglich, was zu verringerten Laufzeiten des Solvers führt. Der Grund dafür soll im weiteren Verlauf erläutert

werden. Davon ausgenommen sind mögliche Stall- oder Delay-Zyklen, da diese unabhängig davon modelliert werden müssen, um ein gutes Schedule zu erhalten.

Die untere Ausführungsschranke U ist die unter optimalen Bedingungen erreichbare Anzahl an Ausführungszyklen, die das zu schedulende Programm benötigt. Diese wird durch folgende Gleichungen bestimmt.

$$\begin{aligned} InstructionBound &= Max(TotalInsCount - LSInsCount, TotalInsCount - IPInsCount) \\ U &= Max(CriticalPathLength, InstructionBound) \end{aligned} \tag{3.7}$$

TotalInsCount is hierbei die Anzahl aller Instruktionen innerhalb des Basisblocks, $IPInsCount$ die Anzahl an IP-Instruktionen und $LSInsCount$ die Anzahl an LS-Instruktionen im Basisblock.
$CriticalPathLength$ ist die normalisierte Länge des kritischen Pfades innerhalb des Basisblocks. D.h. auch hier werden Latenzen größer 1 als 1 aufgefasst. Die $CriticalPathLength$ wird durch die Maximum Delay-Heuristik bestimmt. Gilt dabei $O = U$, so ist der durch den List Scheduler geschedulte Basisblock optimal im Sinne der möglichen Ausnutzung des Instruction Level Paralellism des Prozessors. Gilt dagegen $O > U$, so wird ein ILP-Gleichungssystem für $O - 1$ Ausführungszyklen aufgestellt, wie unten beschrieben. Ist dieses vom Solver lösbar, so wird, um die Berechnungszeit niedrig zu halten, bestimmt, wie viele Ausführungszyklen N dieses optimale Schedule für das Programm benötigt. Es wird dann ein ILP-Gleichungssystem für $N - 1$ Ausführungszyklen aufgestellt und zu lösen versucht. Dies wird solange wiederholt, wie $N > U$ ist und der Solver eine optimale Lösung bestimmen kann. Ist eines der beiden Kriterien nicht mehr erfüllt, so wird die Iteration abgebrochen.

Um ein Schedule von m Taktzyklen für einen n Instruktionen enthaltenden Basisblock zu erstellen, werden für jede Instruktion und jeden Taktzyklus binäre Entscheidungsvariablen x_i^j eingeführt. Diese Entscheidungsvariablen repräsentieren die Entscheidung, ob eine Instruktion i im Ausführungszyklus j geschedulet wird (1) oder nicht (0). Damit die Korrektheit des Basisblocks und alle Abhängigkeiten zwischen den darin enthaltenen Instruktionen erfüllt bleiben, werden dazu weitere Bedingungen in Form linearer Gleichungen aufgestellt. Die erste Bedingung erzwingt den Schedule jeder im Basisblock enthaltener Instruktion.

$$\sum_{j=1}^{m} x_i^j = 1 \tag{3.8}$$

Durch diese Gleichung wird festgelegt, dass genau eine binäre Entscheidungsvariable der Instruktion i innerhalb aller Ausführungszyklen wahr (1) sein muss, d.h. Instruktion i muss aus jeden Fall ausgeführt werden. Zusätzlich müssen Bedingungen eingeführt werden, die die Ausführungskapazitäten des Zielprozessors korrekt modellieren. Im Falle des TriCore Prozessors sind das die IP-, LS- sowie Loop-Pipeline. Für FP-Instruktionen wird, um die Komplexität niedrig zu halten, angenommen, dass diese auf der IP-Pipeline und LS-Pipeline ausgeführt werden. Dies ist möglich, da bei der Ausführung von FP-Instruktionen auf der FPU jede andere Pipeline gestallt ist, bis die FP-Instruktion

Tabelle 3.2: *Instruktionsklassen beim optimalen Scheduling*

ILP-Instruktions-klassen $ExType(p)$	TriCore Instruktionsklassen $type(i)$				
	IP	LS	DP	LOOP	FP
IP	1	0	1	0	1
LS	0	1	1	0	1
LOOP	0	0	0	1	0

beendet wurde. Es wird für jede Pipeline eine Gleichung pro Ausführungszyklus j eingeführt:

$$\forall p : \sum_{type(i) \in ExType(p)} x_i^j \leq 1 \tag{3.9}$$

Die Variable p nimmt dabei für jede der modellierten Pipelines unterschiedliche Werte an. $ExType(p)$ ist die Menge der von p ausführbaren Instruktionsklassen. $type(i)$ ist die Instruktionsklasse von i. Die Tabelle 3.2 soll die Aufteilung der TriCore Instruktionsklassen auf die drei Instruktionsklassen des ILP-Gleichungssystems verdeutlichen:

Die Tabelle ist hierbei so zu lesen, dass ein Tabelleneintrag von 1 zu einer Abbildung der Instruktionsklasse $type(i)$ einer Instruktion i auf die Instruktionsklasse des ILP-Gleichungssystems $ExType(p)$ abgebildet wird. Um hierbei durch diese Gleichung die in 2.1.4 beschriebene Einschränkung des TriCore Prozessors beim Ausführen von LS-Instruktionen nach einer CALL- oder RET-Instruktion zu modellieren, wird diese für die ILP-Instruktionsklasse, welche die LS-Pipeline modelliert zum Ausführungszyklus 0 auf 0 gesetzt. Dadurch wird vermieden, dass ein Scheduling von LS-Instruktionen im ersten Ausführungszyklus des ILP-Schedules stattfindet.

Eine weitere Bedingung soll schliesslich gewährleisten, dass alle Abhängigkeiten innerhalb des Basisblocks berücksichtigt werden. Dies wird durch folgende Gleichung erreicht, welche für jede Instruktion i aufgestellt wird, von der eine weitere Instruktion k abhängig ist:

$$\sum_{j=1}^{m} j * x_k^j + L_{ki} \leq \sum_{j=1}^{m} j * x_k^j \tag{3.10}$$

Da durch die erste Bedingung 3.8 gewährleistet ist, das nur eine binäre Entscheidungsvariable einer Instruktion den Wert 1 einnehmen kann, wird die Summe über alle Ausführungszyklen mit Multiplikation des Werts des derzeitigen Ausführungszyklus und der betreffenden Entscheidungsvariable für Instruktion i den vom ILP-Solver für diese Instruktion bestimmten Ausführungszyklus zurückgeben. Eine Instruktion k kann durch diese Gleichung nur einen Ausführungszyklus größer als dem ermittelten Ausführungszyklus von i plus der Latenz L_{ki} zwischen beiden Instruktionen aufweisen.

3.7.1 Minimierung der Komplexität des Gleichungssystems

Um die Laufzeit des Solvers beim Lösen des Gleichungssystems so gering wie möglich zu halten, muss die Anzahl der Gleichungen so niedrig wie möglich gehalten werden. Latenzen größer als 2 erhöhen die Komplexität des Gleichungssystems zusätzlich. So ist laut [19] kein Algorithmus bekannt, der Instruktionen mit Latenzen > 2 optimal in polynomieller Zeit schedulen kann. Das optimale Schedulen von Instruktionen mit Latenzen ≤ 2 ist dagegen in polynomieller Zeit möglich [19]. Im Falle des optimalen Schedulers für den TriCore Prozessor ist dies durch eine Beschränkung auf Instruktionslatenzen ≤ 2 erreicht, die trotzdem alle möglichen Stalls und Delay-Zyklen innerhalb des TriCore Prozessors korrekt modelliert. Dies ist möglich, da Instruktionen mit einer Latenzzeit größer als 1 alle weiteren Pipelines anhalten, siehe 2.1.3. Eine weitere Möglichkeit der Minimierung der Komplexität des Gleichungssystems, die in [18] vorgestellt wird, ist die Entfernung redundanter Kanten aus dem DAG. Eine Kante ist dabei redundant, wenn ein weiterer Pfad zwischen den Start- und End-Knoten der Kante besteht, dessen Pfadlänge größer oder gleich der Pfadlänge der Kante ist. Die Pfadlänge ist hierbei die Summe aller Latenzen auf dem Pfad. Abbildung 3.8 zeigt ein Beispiel für eine redundante Kante. Die Kante zwischen Knoten B und E ist hierbei redundant.

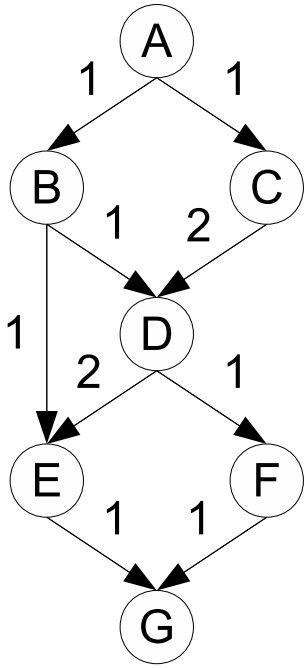

Abbildung 3.8: *Beispiel einer redunanten Kante zwischen Knoten B und E*

Die Möglichkeit der Bestimmung einer redundanten Kante innerhalb des DAG ohne direkte Bestimmung von Pfadlängen ist durch Heranziehen der bestimmten ASAP- und ALAP-Werte der als Start- und End-Knoten fungierenden Instruktionen möglich. Eine redundante Kante liegt dann vor, wenn folgende Gleichung gilt:

$$ALAP(i) + latency(i, j) \geq ASAP(j) \tag{3.11}$$

Hierbei ist *i* die Instruktion des Start-Knotens und *j* die Instruktion des End-Knotens der betreffenden Kante innerhalb des DAG.

Durch diese Möglichkeit der Entfernung redundanter Kanten ist eine substanzielle Verringerung der Komplexität des ILP-Gleichungssystems möglich. Im Kapitel 5 werden diesbezüglich auch separat Laufzeiten vorgestellt. Trotzdem kann bei großen Basisblöcken mit einer hohen Anzahl von Instruktionen und Abhängigkeiten zwischen diesen das Lösen des ILP-Modells sehr lange dauern. Für diesen Fall ist die Lösung des ILP-Modells auf 30 CPU-Minuten beschränkt. Überschreitet der Solver diese Zeitschranke, so liefert er entweder ein suboptimales Ergebnis zurück oder es wird keine Lösung zurückgegeben.

3.8 Konzeptioneller Aufbau

Der Scheduler gliedert sich, wie aus Abbildung 3.9 ersichtlich, in verschiedene spezialisierte Klassen, die nachfolgend bezüglich ihres Zwecks und ihrer Funktion einzeln vorgestellt werden sollen.

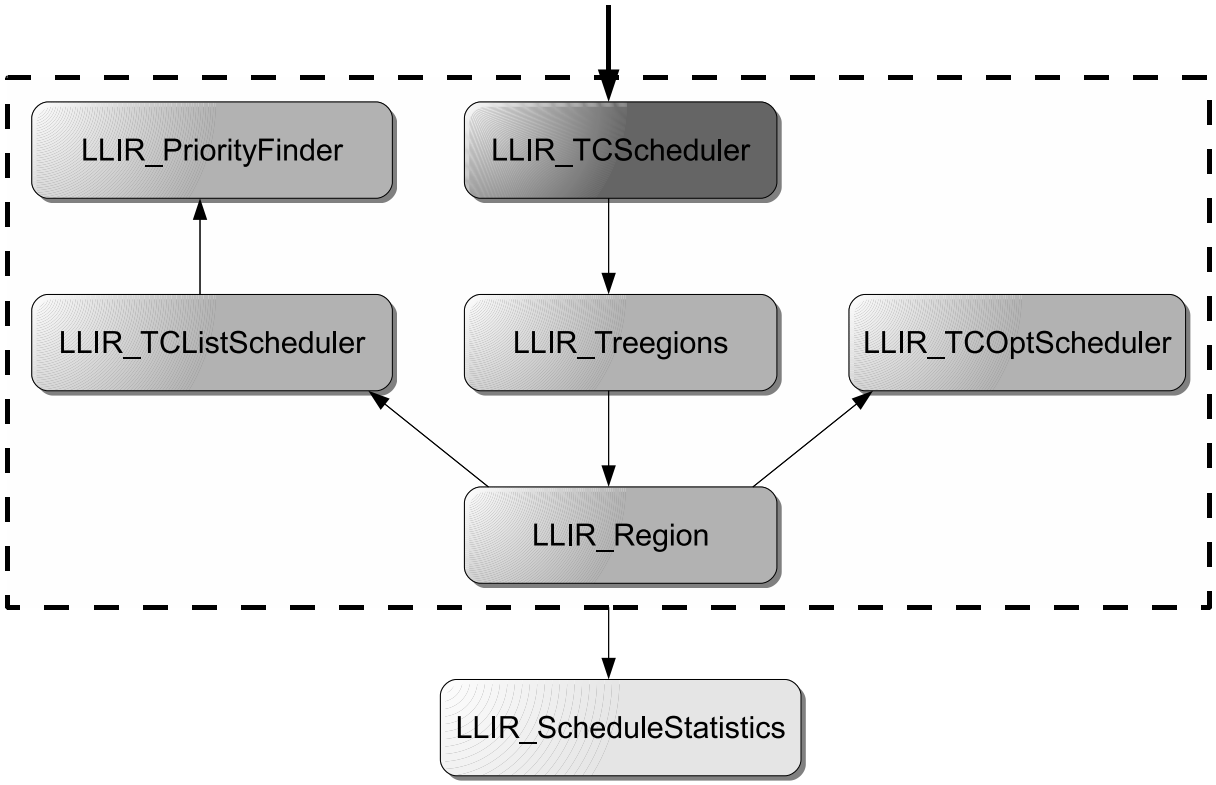

Abbildung 3.9: *Klassendiagramm des TriCore Schedulers*

Die Schnittstelle zwischen dem Framework des WCC und des Schedulers bildet hier die Klasse *LLIR_TCScheduler*. Diese bekommt innerhalb des WCC eine Liste von LLIRs übergeben, welche die zu übersetzenden Quelltexte repräsentieren. Der Scheduler arbeitet hierbei auf sogenannten *Regions*. Diese können aus einem einzelnen, oder aus mehreren, innerhalb eines Programms durch

dessen Kontrollfluß zusammenhängenden, Basisblöcken bestehen. Dadurch können sowohl lokale als auch globale Schedulingverfahren auf diesen Regionen implementiert werden. Im Falle des lokalen Schedulings wird hier ein einzelner Basisblock eingetragen. Innerhalb der Klasse wird über die Liste von LLIRs iteriert und mittels der *LLIR_Treegions*-Klasse Regionen aufgestellt. Diese Klasse unterstützt das Erzeugen von Regionen, die sowohl einzelne Basisblöcke enthalten, als auch sogenannte Treegions, welche eine Form des globalen Schedulings darstellen. Auf die genaue Definition dieser wird in Kapitel 4 eingegangen. Die Klasse *LLIR_SchedulingStatistics* wird zum Führen einer Schedule-Statistik verwendet. Diese enthält die Anzahl der Gesamtzahl an Instruktionen im geschedulten Programm, eine Aufschlüsselung dieser in IP- sowie LS-Instruktionen sowie die Anzahl an Bündeln, die im vom WCC erzeugten LLIR-Code enthalten waren und dazu zum Vergleich, die Anzahl an Bündeln die der Scheduler ermöglicht hat. Durch das Führen der Statistik können Vergleiche zwischen lokalem und globalem Scheduling sowie dem optimalen Scheduling gemacht werden. Die Klasse *LLIR_PriorityFinder* schliesslich stellt Funktionen bereit, um Instruktionen zu klassifizieren und Instruktionstypen, wie Sprunginstruktionen, zu identifizieren. Die wichtigste Aufgabe dieser Klasse ist aber das Bereitstellen einer Funktion zur Bestimmung eines bestmöglichen Bündels an Instruktionen zu einem bestimmten Ausführungszyklus. Dazu sind die zuvor in 3.5 vorgestellten Heuristiken in dieser Klasse implementiert. Die Klasse *LLIR_PriorityFinder* wird durch die List Scheduler-Klasse *LLIR_TCListScheduler* instanziiert und verwendet.

Einige der Klassen erhalten beim Instanziieren über den Konstruktor zudem die gleichen Objekte von Klassen übergeben, die Scheduler-übergreifend gelten. Zum einen ist dies die Klasse *Configuration*, welche die globalen Konfigurationsdaten des WCC enthält. Zudem wird allen Klassen die eine Statistik führen, ein Objekt der *LLIR_ScheduleStatistics*-Klasse übergeben. Dieses wird in der Klasse *LLIR_TCScheduler* erzeugt und initialisiert. Eine weitere Gemeinsamkeit ist der Parameter des Typs *Configuration::ScheduleTime*, welcher in der *Configuration*-Klasse definiert ist und zwei Werte annehmen kann. Diese sind *ST_PRERA* und *ST_POSTRA*. Dadurch wird den betreffenden Klassen mitgeteilt, ob das Scheduling, an dem sie beteiligt sind, vor der Registerallokation oder danach stattfindet. So können die Klassen auf unterschiedliche Konfigurationseinstellungen, je nach Situation, zugreifen. Nachfolgend sollen alle Klassen einzeln vorgestellt werden und auf die wichtigsten Funktionen und Attribute eingegangen werden.

3.8.1 LLIR_TCScheduler

Das Klassendiagramm der Klasse *LLIR_TCScheduler* ist in Abbildung 3.10 zu sehen.

Diese Klasse wird innerhalb des zentralen endlichen Automaten des WCC, welcher den Ablauf des Übersetzvorgangs steuert, instanziiert. Dies geschieht dabei, je nach übergebener Kommandozeilenoption vor der Registerallokation oder nachdem die Registerallokation und weitere Optimierungen auf der LLIR stattgefunden haben. Dem Konstruktor wird dabei ein weiterer Parameter übergeben, *LLIR_MemoryLayout*. Dieses wird benötigt, um ein Objekt der Klasse *DataobjectLink* zu erzeugen. Dieses wird innerhalb der Klasse *LLIR_Region* benötigt, um Informationen der Werte-Analyse zur

Erzeugung nur benötigter Abhängigkeiten zwischen Lade- sowie Speicherinstruktionen zu modellieren.

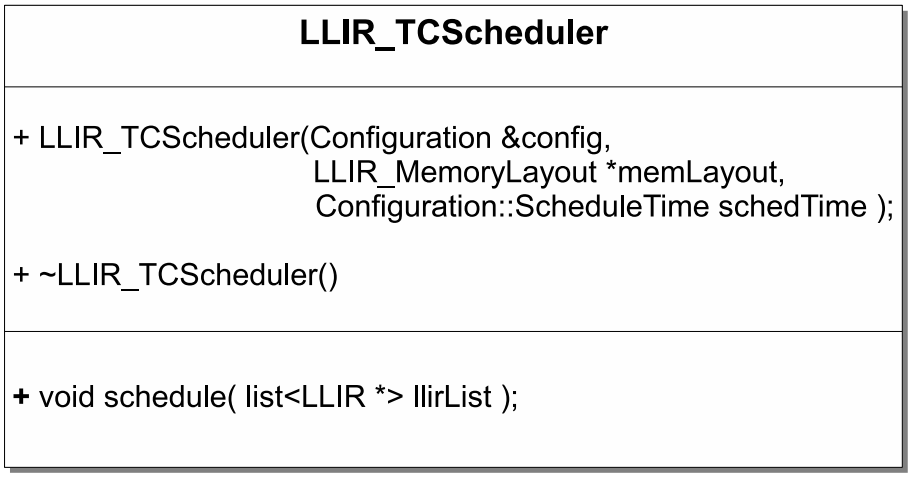

Abbildung 3.10: *Klasse LLIR_TCScheduler*

Innerhalb der *LLIR_TCScheduler*-Klasse werden Objekte der Klassen *LLIR_Treegion*, *LLIR_TCListScheduler* und *LLIR_TCOptScheduler* erzeugt. Durch Aufrufen der Methode *schedule* wird der *LLIR_TCScheduler*-Klasse eine Liste von LLIR-Objekten übergeben. Über diese wird intern iteriert und jede einzelne LLIR an die LLIR_Treegion-Klasse übergeben, welche daraufhin aus jeder LLIR_Function innerhalb der LLIR eine Liste einzelner *LLIR_Region*-Klassen erzeugt. Abbildung 3.11 zeigt eine solche mögliche Aufteilung. Über diese Liste von Regionen wird daraufhin iteriert und jede Region einzeln zuerst der *LLIR_TCListScheduler*-Klasse übergeben, welche ein List Scheduling auf der übergebenen Region durchführt. Der List Scheduler bestimmt dabei eine obere Schranke des erzeugten Schedules, welches zusammen, sollte ein optimales Scheduling über die Kommandozeile eingeschaltet worden sein, mit der Region an die *LLIR_TCOptScheduler*-Klasse übergeben wird. Diese stellt ein ILP-Gleichungssystem für die übergebene Region auf und übergibt diese an den Solver. Dabei werden sowohl durch dieser Klasse, als auch durch die *LLIR_TCListScheduler*-Klasse Instruktionen innerhalb der LLIR äquivalent zu dem erzeugten Schedule in ihrer Reihenfolge umgestellt.

3.8.2 LLIR_Treegions

Das Klassendiagramm zur *LLIR_Treegions*-Klasse ist in Abbildung 3.12 ersichtlich.

Diese Klasse erbt von der deque ⟨ Region * ⟩-Klasse, welche eine doppelt verkettete Liste implementiert. Nachdem die Klasse innerhalb der *LLIR_TCScheduler*-Klasse instanziiert wurde, wird ihr mittels der *processLLIR*-Methode eine einzelne LLIR übergeben. Diese LLIR wird in einzelne Regionen unterteilt. Eine Region enthält dabei, wie schon erwähnt, einen oder mehrere Basisblöcke der LLIR. Die Methode *absorbIntoTree* spielt bei der Aufstellung von Treegions eine Rolle, auf die näher in Kapitel 4 eingegangen wird. Durch diese Methode wird einer Region ein weiterer Basisblock hin-

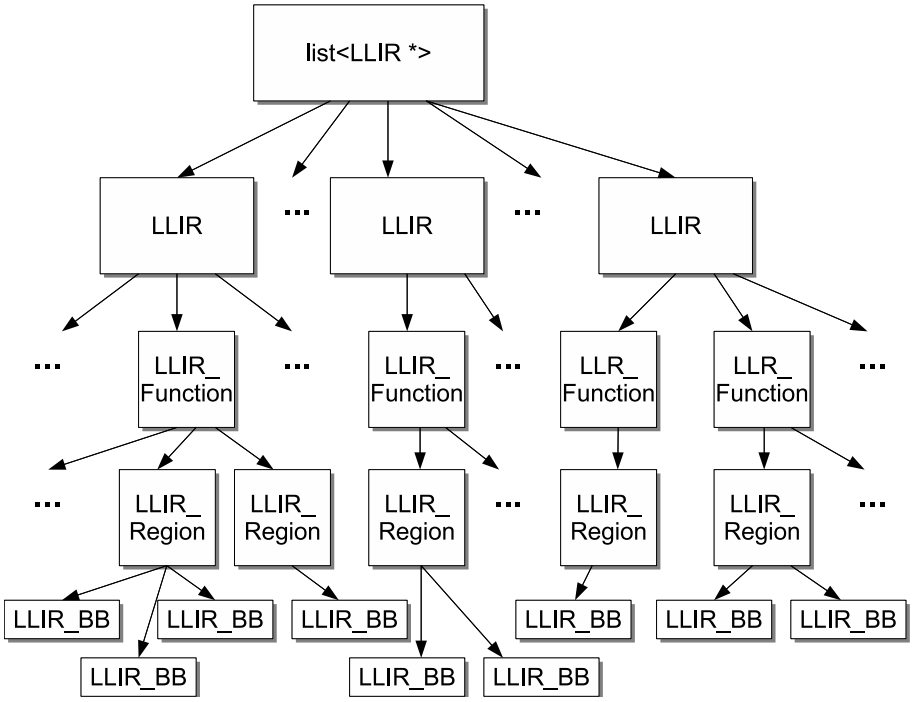

Abbildung 3.11: *Repräsentation eines zu übersetzenden Programms innerhalb des TriCore-Schedulers*

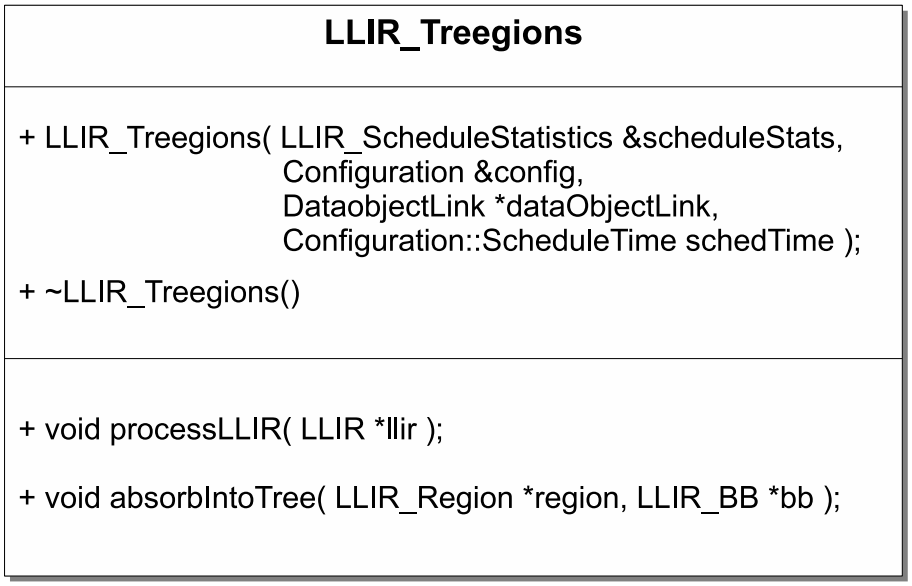

Abbildung 3.12: *Klasse LLIR_Treegions*

zugefügt. Es werden zudem Methoden der Basisklasse zur Iteration durch die Liste an Regions zur Vefügung gestellt, um auf diese zugreifen zu können.

3.8.3 LLIR_Region

Das Klassendiagramm der *LLIR_Region*-Klasse ist in Abbildung 3.13 zu sehen.

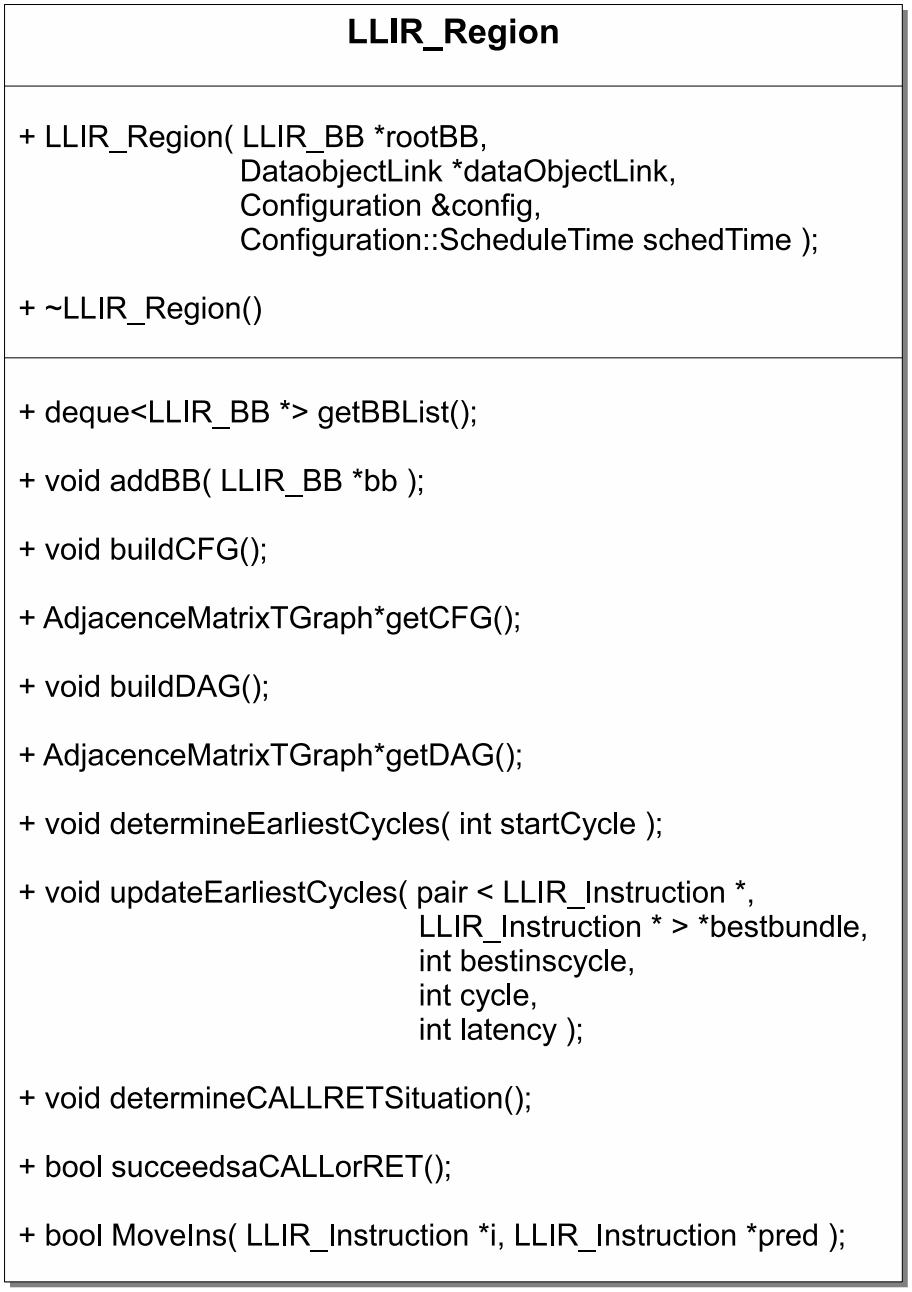

Abbildung 3.13: *Klasse Region*

Diese Klasse ist die umfangreichste innerhalb des TriCore-Schedulers und implementiert zum einen das Aufstellen eines Kontrollflußgraphen (CFG) und eines DAG für den innerhalb dieser Region enthaltenen Basisblocks oder mehrerer Basisblöcke, zum anderen kann daraus eine Liste von ASAP- und ALAP-Zeiten für jede darin enthaltene Instruktion aus dem DAG erzeugt werden. Ein Kontrollflußgraph ist dabei ein Graph, welcher Knoten für Basisblöcke und Kanten für zwischen Ba-

sisblöcken bestehende Kontrollflüsse enthält.

Um eine Region aufzustellen, wird innerhalb des Konstruktors ein Basisblock *LLIR_BB *rootBB* als Wurzel der gesamten Region angegeben. Zusätzlich muss dieser Basisblock mittels *void addBB(LLIR_BB *bb)* zur internen Liste an Basisblöcken hinzugefügt werden. Sind alle Basisblöcke hinzugefügt, kann durch *buildCFG* ein CFG bestimmt werden. Dazu werden vom Wurzel-Basisblock der Region aus innerhalb der LLIR alle Nachfolger-Basisblöcke als Kanten zum CFG hinzugefügt, welche durch *addBB* zur Region hinzugefügt wurden. Zwischen Vorgänger- und dem Nachfolger-Basisblock wird eine Kante zum CFG hinzugefügt. Durch Aufruf der Methode *buildDAG* wird ein DAG der Region bestimmt. Auf beide Graphen kann mittels der *getCFG* und *getDAG* Methoden zugegriffen werden.

Um aus dem DAG die ASAP- und ALAP- sowie Ready-List zu generieren, ist die Methode *determineEarliestCycles* verfügbar. Als Parameter kann ein Integer-Wert übergeben werden, der bestimmt, welchen Ausführungszyklus den ersten ausführbereiten Instruktionen innerhalb der Listen zugewiesen werden soll. Mittels der *updateEarliestCycles* Methode, wird die Ready-List, wie in 3.4.1 beschrieben, aktualisiert. Der Parameter *bestbundle* ist hierbei das zuletzt geschedulte Bündel an Instruktionen, *bestinscycle* der alte Ausführungszyklus vor Inkrementierung des Ausführungszyklus-Zählers des List Schedulers, *cycle* der neue Ausführungzyklus und *latency* die höchste Ausführungslatenz, die eine der beiden geschedulten Instruktionen aufweist. Auf die internen Listen kann mittels Methoden zugegriffen werden. Diese sind aus Gründen der Übersichtlichkeit nicht einzeln im Klassendiagramm aufgeführt.

Durch *determineCALLRETSituation* wird festgestellt, ob die Region ein Nachfolger einer Region ist, die mit einer CALL- oder RET-Instruktion endet. Wird diese Methode vor Aufstellen der Ready-List aufgerufen, so wird beim Aufstellen dieser automatisch den zuerst ausführbereiten LS-Instruktionen ein frühestmöglicher Ausführungszyklus von 2 zugewiesen, wie in 3.4.1 beschrieben. Dieses Flag kann durch die *succeedsaCALLorRET* Methode abgefragt werden.

Um eine Instruktion innerhalb der Region zu verschieben, sei es innerhalb eines einzelnen oder zwischen mehreren Basisblöcken, wird die Methode *MoveIns* verwendet. Diese sorgt beim Verschieben zwischen mehreren Basisblöcken zusätzlich dafür, dass durch das Verschieben die Korrektheit des Programms erhalten bleibt. Die genaue Vorgehensweise wird in Kapitel 4 erläutert.

3.8.4 LLIR_TCListScheduler und LLIR_PriorityFinder

Das Klassendiagramm der *LLIR_TCListScheduler*-Klasse ist in Abbildung 3.14 zu sehen.

Die Klasse implementiert den in 3.4 beschriebenen List Scheduler. Von Interesse ist hier die Methode *schedule*. Dieser wird ein Objekt der *LLIR_Region*-Klasse übergeben. Die Region wird dann gemäß dem Algorithmus im Abschnitt über das List Scheduling beschrieben geschedult. Dazu verwendet die Klasse in der *LLIR_Region*-Klasse bereitgestellten Methoden zum aufstellen und aktua-

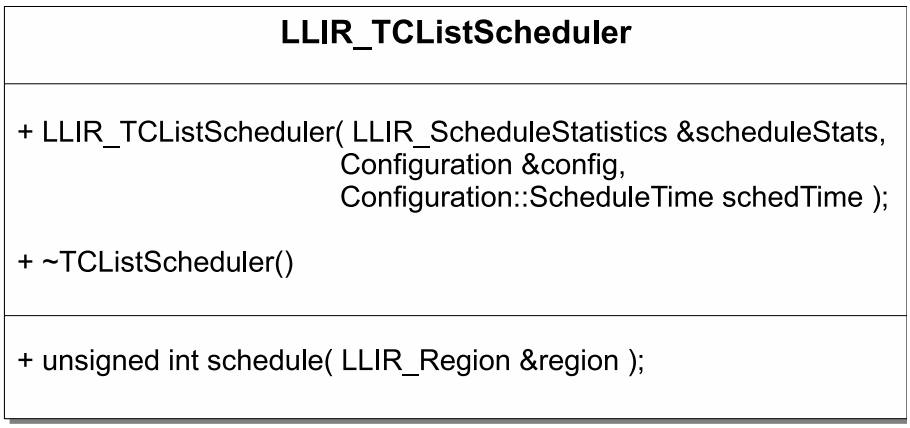

Abbildung 3.14: *Klasse LLIR_TCListScheduler*

lisieren der Ready-Liste.

Zur Bestimmung der zum aktuellen Ausführungszyklus bestmöglichen zu scheduleden Instruktionen wird die Methode *findBestBundle* der Klasse *LLIR_PriorityFinder* verwendet. Diese Methode gibt eine bestmögliche Instruktion oder ein Bündel an IP- und LS-Instruktionen zurück, welche im derzeitigen Ausführungszyklus unter Verwendung der aktivierten Heuristiken die höchste Priorität aufweist oder aufweisen. Innerhalb der *LLIR_PriorityFinder*-Klasse sind auch die Prioritätsheuristiken implementiert, welche durch *findBestBundle* verwendet werden. Das Klassendiagramm dieser Klasse ist in Abbildung 3.15 ersichtlich. Die Methode *schedule* gibt nach beendetem Scheduling die normalisierte obere Ausführungsschranke für das Schedule zurück.

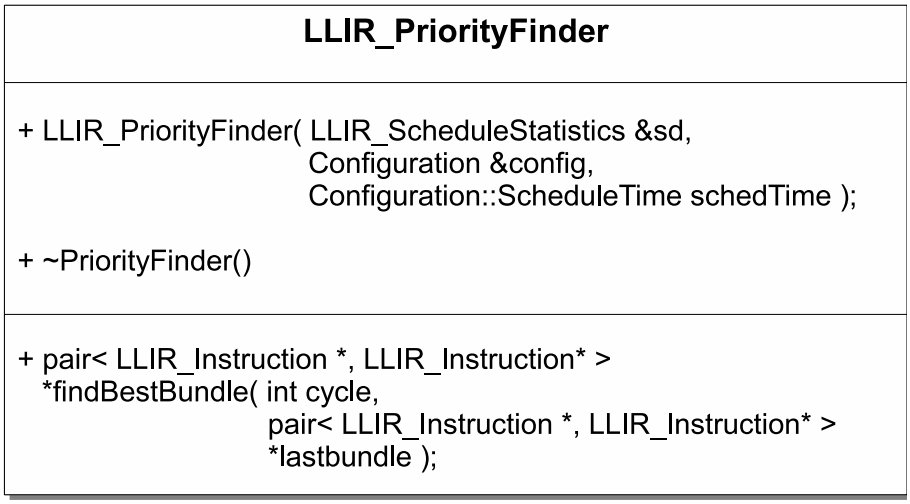

Abbildung 3.15: *Klasse PriorityFinder*

3.8.5 LLIR_TCOptScheduler

Das Klassendiagramm der *LLIR_TCOptScheduler*-Klasse ist in Abbildung 3.16 zu sehen.

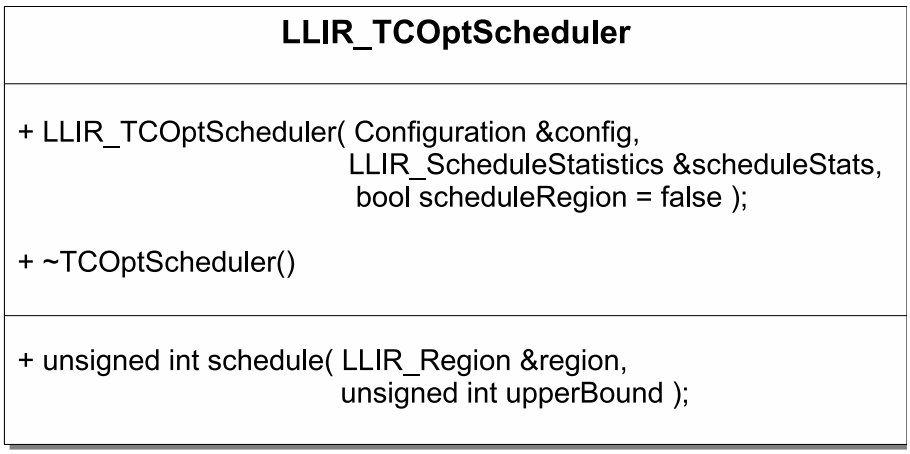

Abbildung 3.16: *Klasse LLIR_TCOptScheduler*

Die Klasse implementiert das in 3.7 beschriebene optimale Scheduling. Durch den Parameter *bool scheduleRegion = false* des Konstruktors kann bestimmt werden, ob das optimale Scheduling Auswirkungen auf die LLIR haben soll, ob also die Instruktionsabfolge innerhalb der LLIR analog des gefundenen optimalen Schedules umgestellt werden soll. Default-Einstellung der Klasse ist dies nicht zu tun, diese Voreinstellung wird aber im TriCore-Scheduler auf *true* gesetzt.

Um ein optimales Scheduling durchzuführen wird dem instanziierten Objekt der Klasse analog zur *LLIR_TCListScheduler*-Klasse durch die *schedule*-Methode ein Objekt der Region-Klasse übergeben sowie die durch den List Scheduler bestimmte normalisierte obere Ausführungsschranke in Parameter *upperBound*.

3.8.6 Kommandozeilenparameter des Schedulers innerhalb des WCC

Um den Scheduler von Ausserhalb des WCC steuern und verschiedene Heuristiken wählen zu können, wurden dem WCC eine Reihe von Kommandozeilenparametern hinzugefügt. Diese richten sich nach der Konvention der Kommandozeilenparameter der GNU Compiler Collection (GCC). Die möglichen Kommandozeilenparameter sind wie folgt:

- **-fschedule-insns**
  Lokales Scheduling vor der Registerallokation.

  -fschedule-insns2
  Lokales Scheduling nach der Registerallokation.

  -fsched-use-treegions
  Wenn zusammen mit **-fschedule-insns** angegeben, wird ein Treegion Scheduling vor der Registerallokation durchgeführt.

  -fsched2-use-treegions
  Wenn zusammen mit **-fschedule-insns2** angegeben, wird ein Treegion Scheduling vor der

Registerallokation durchgeführt.

Um verschiedene Heuristiken oder den optimalen Scheduler auszuwählen ist zudem nach einer der beiden **-fschedule-insns** oder **-fschedule-insns2** Kommandozeilenparameter ein zusätzlicher Parameter **–param scheduling-heuristics=** anfügbar. Nach diesem sind alle verfügbaren Heuristiken, durch Kommata getrennt, angebbar. Die genauen Optionen sind wie folgt:

- **asap**
  Hierdurch wird der ASAP-Algorithmus zur Bestimmung der Ready-List ausgewählt.

  alap
  Hierdurch wird der ALAP-Algorithmus zur Bestimmung der Ready-List ausgewählt.
- **maxdelay**
  Hierdurch wird die Maximum Delay-Heuristik ausgewählt.
- **inspriority**
  Hierdurch wird die Instruction Priority-Heuristik ausgewählt.
- **noofchildins**
  Hierdurch wird die Number of Child Instructions-Heuristik ausgewählt.
- **mobility**
  Hierdurch wird die Mobility-Heuristik ausgewählt.
- **minregs**
  Hierdurch wird die Heuristik zur Registerzahl-Minimierung ausgewählt.
- **impstlddep**
  Hierdurch wird die Heuristik zur verbesserten Bestimmung von Lade-/Speicher-Abhängigkeiten auf Basis der Werteanalyse ausgewählt.
- **depheight**
  Hierdurch wird die Dependence Height-Heuristik des Treegion Schedulings ausgewählt.
- **exitcount**
  Hierdurch wird die Exitcount-Heuristik des Treegion Schedulings ausgewählt.

  optimal
  Hierdurch wird zusätzlich zum List Scheduling noch ein optimales Scheduling durchgeführt.

Ohne expliziter Angabe von Heuristiken sind per Default der ASAP-Algorithmus sowie die Maximum Delay-, Mobility- sowie Instruction Priority-Heuristiken aktiv.

Kapitel 4

Globales Scheduling

In folgendem Abschnitt soll das innerhalb der Diplomarbeit erarbeitete globale Scheduling vorgestellt werden. Von einem globalen Scheduling spricht man, wenn das Scheduling über Basisblock-Grenzen hinaus stattfindet und Instruktionen zwischen unterschiedlichen Basisblöcken verschoben werden.

Zuerst sollen in Abschnitt 4.1 verwandte Arbeiten vorgestellt werden. Danach wird in Abschnitt 4.2 das Treegion-Scheduling vorgestellt, welches den im vorliegenden Scheduler implementierten Typ an globalem Scheduling darstellt. In Abschnitt 4.2.1 wird dann erklärt, wie die Bildung von Treegions vonstatten geht. Abschnitt 4.3 erläutert die Bestimmung und den Zweck eines Kontrollflußgraphen für das globale Scheduling, um im Abschnitt 4.4 auf die Besonderheiten und zusätzlich benötigte Informationen innerhalb des Datenabhängigkeitsgraphen innerhalb des globalen Schedulings einzugehen. Abschnitt 4.5 stellt danach Heuristiken vor, die zusätzlich zu denen des lokalen Scheduling für das Treegion-Scheduling implementiert wurden. Abschnitt 4.6 soll dann in die Problematik der Prioritätsbildung beim vorliegenden globalen Scheduling eingehen. Um korrekt Instruktionen innerhalb von Treegions verschieben zu können, werden dann in Abschnitt 4.7 Lösungen dazu vorgestellt.

4.1 Verwandte Arbeiten

In diesem Abschnitt soll ein Überblick über die bisher veröffentlichten Forschungsarbeiten gegeben werden, die sich mit dem Thema des globalen Schedulings befasst haben.

Innerhalb globaler Scheduling-Verfahren gab es viele Arbeiten, die sich auf die Bildung linearer Regionen unter Zuhilfenahme von Profiling-Informationen konzentriert haben. *Trace Scheduling* [20] von Fisher versucht durch die gegebenen Profiling-Informationen den am häufigsten ausgeführten Pfad innerhalb eines Programms zu finden. Basisblöcke auf diesem Pfad bilden dann eine Region (siehe auch 3.8, welche wie ein einzelner zusammenhängender Basisblock mittels List Scheduling geschedult wird. Um dabei ein spekulatives Verschieben von Instruktionen über Verzweigungspunk-

te hinaus korrekt durchzuführen, wird Kompensationscode eingefügt. Der wesentliche Nachteil dieses Verfahrens besteht in der Verwaltung (Bookkeeping) dieses Codes.

Hwu et al. haben eine Variante des Trace-Scheduling, das sogenannte *Superblock Scheduling* [21] vorgestellt, die die Notwendigkeit dieses Kompensationscodes überflüssig macht. Dieses Scheduling-Verfahren arbeitet in zwei Schritten. Im ersten Schritt werden Traces mittels Profiling-Informationen identifiziert, um im zweiten Schritt mittels sogenannter *Tail-Duplication* Verzweigungspunkte zu vermeiden. Tail-Duplication erzeugt dabei Duplikate von Basisblöcken, welche als Zusammenschluss zweier Kontrollflüsse fungieren. Diese erweiterten Basisblöcke werden Superblöcke genannt.

Beide Scheduling-Verfahren profitieren im Wesentlichen vom Vorhandensein eines einzelnen Kontrollflusses, welcher am häufigsten ausgeführt wird. Innerhalb kontrollfluß-intensiver Programme ist dies nicht immer der Fall. Mahlke et al. entwickelten ein Scheduling-Verfahren, das sogenannte Hyperblock-Scheduling [22], welches den Kontrollflußgraphen eines Programms transformiert. Diese Transformation ist dabei eine Umwandlung von Sprung-Instruktionen hin zu Vergleich-Instruktionen. Innerhalb der dem Sprung folgenden Basisblöcken werden Instruktionen in bedingte Instruktionen umgewandelt. Diese werden dann abhängig der Vergleich-Instruktion ausgeführt oder verworfen. Hierdurch können mehrere Kontrollflüsse einen Hyperblock bilden, der wie ein einzelner Basisblock geschedult werden kann.

Ein weiteres globales Scheduling-Verfahren ist das sogennante *Region-Scheduling* [23]. Dieses von Gupta et al. vorgestellte Verfahren basiert auf einem kombinierten Kontroll- und Datenabhängigkeitsgraphen welcher ein Verschieben von Code zwischen Regionen ermöglicht. Durch verschiedene Code-Transformationen wird dabei eine verbesserte Balance von Parallelität innerhalb der Regionen erreicht.

Eine Kombination der Ideen des Hyperblock-Schedulings und des Region-Scheduling wurden dann durch Havanki et al. innerhalb des *Treegion-Scheduling* [24] realisiert. Treegion-Scheduling partitioniert den Kontrollflußgraphen eines Programms in baumartige Regionen, auch *Treegions* genannt. Die darin enthaltenen nebenläufigen Pfade werden mittels bedingter Instruktionen gleichzeitig ausgeführt. Der Vorteil dieses Verfahrens gegenüber den zuvor vorgestellten Verfahren liegt, zusätzlich zu den größeren Möglichkeiten der Ausnutzung der ILP eines Prozessors, darin dass keine Profiling-Informationen zur Aufstellung einer Treegion notwendig sind. Eine Abwandlung dieses Verfahrens wird innerhalb des vorgestellten Schedulers implementiert. Auf Details wird später innerhalb dieses Kapitels eingegangen.

4.2 Treegion-Scheduling

Eine Treegion ist ein Baum (Tree) von Basisblöcken innerhalb einer Region, welcher einen Teilbaum des Kontrollflußgraphen (CFG) eines Programms darstellt. Eine Treegion kann dabei mehrere, ne-

benläufige Ausführungspfade enthalten. Durch die Beschränkung auf eine Baumstruktur ist eine Treegion azyklisch, wodurch ausser im Wurzel-Knoten keine sog. *Merge-Points*, in denen die Zusammenführung von Ausführungspfaden stattfindet, auftreten können. Abbildung 4.1 zeigt einen Kontrollflußgraphen, welcher in Treegions partitioniert wurde. Die mittels gestrichelten Rechtecken umrandeten Basisblöcke bilden dabei jeweils eine Treegion. Diese Abbildung wurde [25] entnommen.

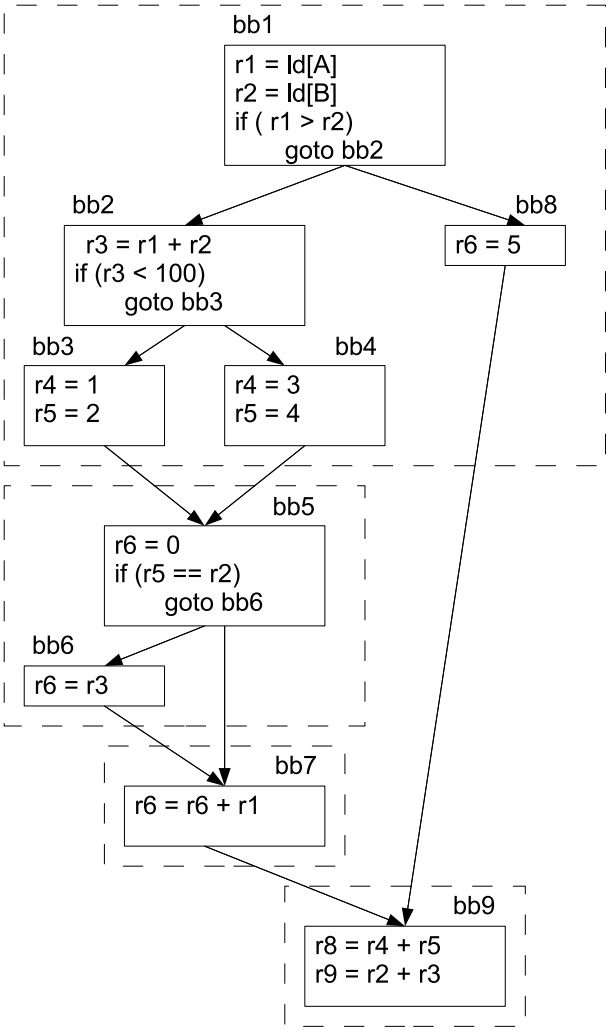

Abbildung 4.1: *Partitionierung eines CFG durch Treegions*

In der Abbildung wurde absichtlich Pseudo-Code verwendet, um ein besseres Verständnis der Struktur des Programms zu vermitteln.

In dieser Diplomarbeit wurde das Treegion-Scheduling aus Gründen, die nachfolgend genannt werden gewählt. Dies ist zum einen die Eigenschaft von Treegions, keine Merge-Points ausserhalb des Wurzelknotens zu enthalten. Dadurch ist ein spekulatives Scheduling von Instruktionen ohne Kompensations-Code im Gegensatz zum Trace-Scheduling möglich. Dies hilft die Komplexität des Scheduling-Verfahrens niedrig zu halten. Ein weiterer Vorteil des Treegion-Schedulings zu anderen globalen Scheduling-Verfahren wie Trace-, Superblock- und Hyperblock-Scheduling ist, dass keine

Profiling-Informationen benötigt werden, um Basisblöcke zu Regionen zusammenzufassen, welche ein Basisblock-übergreifendes Scheduling ermöglichen.

Die vorliegende Implementation des Treegion Schedulings adaptiert das Treegion-Scheduling-Verfahren an den Infineon TriCore und die LLIR des WCC an. Grundsätzlich ist das Treegion-Scheduling, wie es in [25] vorgestellt wird, für Prozessoren mit vielen Ausführungseinheiten konzipiert. Um Instruktionen aus nebenläufigen Ausführungspfaden zu schedulen, werden bedingte Instruktionen verwendet. Diese ermöglichen es, Berechnungen innerhalb einer Ausführungseinheit durchzuführen, welche abhängig von einem Prädikatregister die Zielregister beschreiben oder aber die Berechnung verworfen wird. Da der TriCore Prozessor keine bedingten Instruktionen kennt, werden Treegions innerhalb des entwickelten Schedulers dazu verwendet, Möglichkeiten des Verschiebens von Instruktionen zwischen Basisblöcken zu erkennen. Falls das Verschieben dieser Instruktionen keine Datenabhängigkeiten verletzt und zu einer höheren Auslastung der Ausführungseinheiten beiträgt, werden diese Instruktionen spekulativ verschoben. Ein weiterer Vorteil des Treegion-Schedulings besteht darin, erweiterte Basisblöcke zu bilden. Diese umfassen mehrere Basisblöcke, die mit einem Funktionsaufruf enden. Somit wird eine Einschränkung des Schedulings, die durch die Definition eines Basisblocks innerhalb der LLIR (siehe 2.2.1) entsteht, aufgehoben. Es ergeben sich dadurch neue Möglichkeiten des Verschiebens und der Bündelung von Instruktionen. Ein Treegion-Scheduling umfasst im Vergleich zum lokalen Scheduling zwei weitere Schritte. Der Gesamtablauf eines globalen Schedules ist in 4.2 ersichtlich.

Abbildung 4.2: *Beispielhafter Ablauf eines globalen Schedules*

Der erste Schritt besteht darin, die in der LLIR enthaltenen Basisblöcke in Treegions zusammengefasst. Nach der Partitionierung wird für jede Treegion ein Kontrollflußgraph aufgestellt. Danach wird, wie beim lokalen Scheduling, ein Datenabhängigkeitsgraph aufgestellt. Dieser enthält Abhängigkeiten zwischen allen innerhalb der Treegion enthaltenen Instruktionen. Die verbleibenden Schritte gleichen denen des lokalen Schedulings, es werden aber weitere Heuristiken eingeführt. Zudem muss beim Verschieben von Instruktionen über Basisblock-Grenzen hinaus beachtet werden, dass der Kontrollfluß innerhalb der LLIR gewahrt bleibt. Auf diese vom lokalen Scheduling abweichenden Details soll nachfolgend genauer eingegangen werden.

4.2.1 Partitionierung in Treegions

Die Partitionierung der LLIR in Treegions startet innerhalb des Anfangs-Basisblocks jeder LLIR_Function. Dies hat den Beweggrund, dass Basisblöcke zwischen zwei LLIR_Function's innerhalb der LLIR keine direkte Verbindung bezüglich des Kontrollflusses besitzen. Die Bildung einer Treegion läuft dabei so ab, dass einer Treegion startend von dem Wurzel-Basisblock an Nachfolge-Basisblöcke innerhalb des CFG hinzugefügt werden. Es werden dabei alle Nachfolger hinzugefügt, solange der Nachfolge-Basisblock nicht mehrere Vorgänger hat. Die so verbleibenden Basisblöcke, die sogenannten *Baumtriebe*, bilden Wurzeln für neue Treegions. Die genaue Bildung einer Treegion soll nachfolgender Algorithmus darstellen, welcher [25] entnommen wurde:

```
1  treeform (CFG)
2  {
3     Füge Anfangsknoten des CFG der unprocessedQueue hinzu;
4    while( unprocessedQueue nicht leer ist ) {
5     Entnehme ersten Knoten aus unprocessedQueue;
6     if ( Knoten schon in einer Treegion )
7      continue;
8
9     Erzeuge eine neue Treegion;
10    absorb-into-tree( treegion, knoten );
11
12    for jeden Baumtrieb
13       if ( Baumtrieb noch nicht in einer Treegion )
14          Füge Baumtrieb der unprocessedQueue hinzu.
15   }
16 }
17
18 absorb-into-tree( treegion, knoten )
19 {
20    Füge knoten der Kandidatenliste hinzu.
21    while( Kandidatenliste nicht leer ) {
22       Entnehme ersten Knoten der Kandidatenliste.
23       if ( Knoten schon in einer Treegion )
24          continue;
25       if ( Knoten ist ein Merge-Point und nicht der Wurzelknoten )
26          continue;
27
28       Füge Knoten treegion hinzu.
29       Füge jeden Nachfolgeknoten von knoten der Kandidatenliste hinzu;
30    }
31 }
```

Ist die LLIR in Treegions partitioniert, so wird für jede Treegion ein Kontrollflußgraph aufgestellt.

4.3 Kontrollflußgraph

Besteht eine Treegion aus mehreren Basisblöcken, so ist im weiteren Verlauf des Schedulings wichtig zu wissen, ob Basisblöcke innerhalb des Kontrollflusses des Programms voneinander aus erreichbar sind, oder ob zwei Basisblöcke nebenläufig im Sinne des Kontrollflusses sind, also einen gemeinsamen Vorgängerbasisblock haben, an welchem sich der Kontrollfluss in beide verzweigt. Solche Abfragen sind effizient mittels eines CFG durchführbar. Dieser wird aufgebaut, indem von einem Startbasisblock der Region, z.B. dem Anfangsbasisblock einer LLIR_Function aus rekursiv die Nachfolgebasisblöcke besucht werden. Für diese Basisblöcke werden Knoten im CFG erzeugt, sowie Kanten dazwischen, die den Kontrollfluß repräsentieren. Hierbei muss darauf geachtet werden, dass Zyklen, die durch eine Schleife induziert werden, nicht zu Rückkanten im CFG führen. Ein Zyklus in einem CFG ist zwar nicht per Definition ausgeschlossen, würde aber im Falle des Treegion-Schedulings keine nützlichen Information darstellen. In Abbildung 4.3 ist ein einfacher beispielhafter Kontrollflussgraph zu sehen.

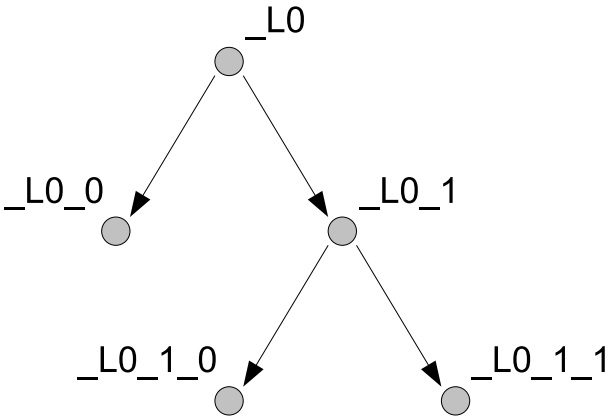

Abbildung 4.3: *Beispiel eines Kontrollflussgraphen*

4.4 Datenabhängigkeitsgraph

Unter Verwendung des aufgestellten Kontrollflußgraphen wird ein erweiterter Datenabhängigkeitsgraph bestimmt. Dieser enthält nun Abhängigkeiten zwischen allen Instruktionen, die voneinander aus erreichbar sind. Um das Verschieben von Instruktionen über CALL-Grenzen hinaus zu ermöglichen, werden beim Treegion Scheduling keine Kontrollflußkanten zwischen jeder anderen Instruktion vor dem CALL hin zu der CALL-Instruktion erzeugt und auch von Instruktionen nach dem CALL hin zu der betreffenden Instruktion, solange diese Instruktionen nur die oberen Kontext-Register verwenden (siehe 2.1.2). Um dies zu erreichen, werden zwischen Instruktionen, die die

unteren Kontext-Register verwenden und einer vorhergehenden CALL-Instruktionen Ausgabeabhängigkeits-Kanten eingefügt. Zudem wird zwischen CALL-Instruktionen und vorhergehenden Instruktionen, welche die unteren Kontext-Register verwenden, Kontrollfluß-Instruktionen eingefügt.

Um zudem spekulativ Instruktionen über Verzweigungen hinaus schedulen zu können, werden Kontrollflußabhängigkeiten zwischen Sprung-Instruktionen und nachfolgenden Instruktionen einzeln auf mögliches Entfernen dieser überprüft. Da hier eine Überprüfung zwischen jeder vorhergehenden Sprung-Instruktion innerhalb der Treegion und der betreffenden Instruktion stattfindet, kann diese Instruktion im optimalen Fall bis in den Wurzel-Basisblock aufsteigen. Dies unter der Voraussetzung, dass keine weiteren Abhängigkeiten dies verhindern.

Kriterien für das spekulative Verschieben von Instruktionen

Die Überprüfung wird mittels eines Kriteriums vorgenommen, das aus der Loop Invariant Code Motion Optimierung des WCC entliehen wurde. Es müssen folgende Einzelkriterien gelten, um einer Instruktion i zu erlauben, spekulativ über Verzweigungen hinaus in einen Basisblock $PRED$ geschedult zu werden:

1. Erzeuge eine temporäre Kopie von i hinter der letzten Instruktion j von $PRED$. i muss j dominieren, d.h. i muss vor j ausführbar sein.

2. Die zu verschiebende Instruktion i definiert Register, welche nicht vorgefärbt sind, d.h. es darf kein physikalisches Register vor der Registerallokation diesen Zielregistern zugewiesen sein. Dies ist beim Scheduling vor der Registerallokation wichtig. Ist ein Register vorgefärbt, so ist es vor der Registerallokation nicht möglich zu erkennen, ob vorhergehenden Instruktionen das gleiche physikalische Register zugewiesen wird, oder nicht.

3. Überprüfe, wie viele andere Instruktionen innerhalb des Basisblocks von i die gleichen Register wie i definieren. Wenn es solche Instruktionen gibt, ist eine spekulatives Scheduling über Basisblock-Grenzen hinaus schon durch dazwischen gegebene Ausgabeabhängigkeiten nicht möglich. Überprüfe zudem für jede Instruktion k, welche die von i definierten Register verwenden, die Anzahl an Definitionen dieser Register. Diese Anzahl kann pro definiertem Register wegen dem vorhergehenden Grund nur 1 sein. Verschiebe i temporär in $PRED$ hinein. Überprüfe nun erneut die Anzahl an registrierten Definitionen für jede Instruktion k. Die Anzahl muss konstant bleiben, andernfalls ist ein Verschieben nicht möglich. Würde ein Verschieben in diesem Fall passieren, wären die von jeder Instruktion k verwendeten Register nicht korrekt definiert.

4. Überprüfe, ob die durch i definierten Register nicht lebendig am Ende von $PRED$ sind. Register, die lebendig sind, werden vor oder innerhalb von $PRED$ definiert und in Nachfolgern von $PRED$ verwendet. Ein Verschieben von i in $PRED$ würde diese Definitionen invalidieren.

Lade- und Speicher-Instruktionen sind hiervon ausgeschlossen. Ein Verschieben dieser könnte den Datenfluß innerhalb des Programms und somit die Semantik verändern.

Es wird zudem ein weiterer Datenabhängigkeitsgraph geführt, *exclusionDAG* genannt. Innerhalb dessen werden Abhängigkeiten zwischen Instruktionen von sich ausschliessenden nebenläufigen Pfaden geführt. Ein beispielhafter exclusionDAG ist in Abbildung 4.4 zu sehen. Die Richtung der Kanten ist im Falle des exclusionDAG nicht von Belang. Ob zwei Basisblöcke in nebenläufigen Pfaden liegen wird durch eine Abfrage innerhalb des CFG bewerkstelligt. Durch die im exclusionDAG enthaltenen Abhängigkeiten ist es möglich effizient abzufragen, ob zwei Instruktionen aus sich ausschliessenden nebenläufigen Pfaden gleichzeitig in einem gemeinsamen Vorgänger-Basisblock verschoben werden dürfen.

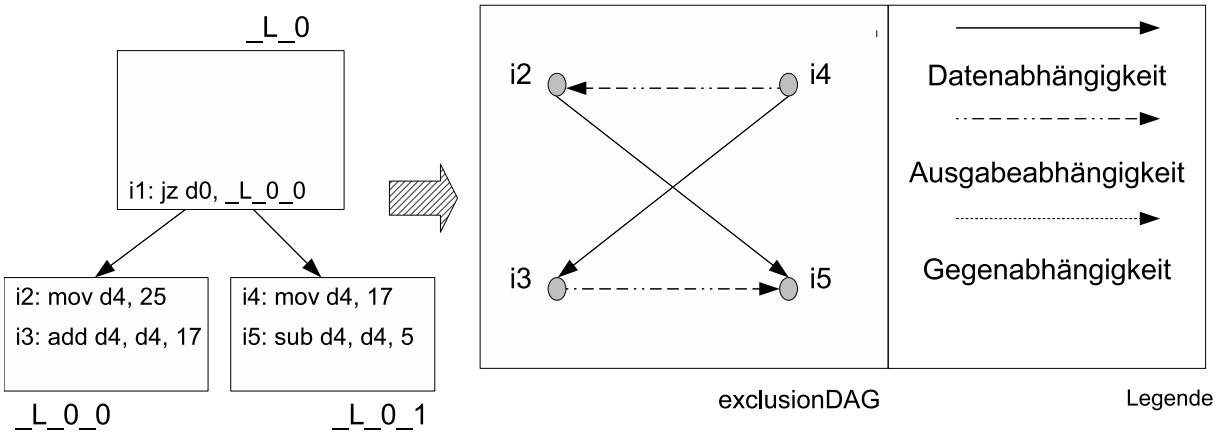

Abbildung 4.4: *Beispielhafter exclusionDAG*

Eine Abhängigkeit zwischen zwei Instruktionen innerhalb des exclusionDAG bedeutet, dass mindestens eine Instruktion ein Register definiert, welches durch die andere verwendet wird, oder beide das gleiche Register definieren. Ein Verschieben beider Instruktionen hin zu einem gemeinsamen Vorgänger-Basisblock würde die Semantik des Programms verändern. Im Falle des Beispieles sind das z.B. die Instruktion i2 zusammen mit Instruktionen i4 und/oder i5. Dies bedeutet, dass beim spekulativen Verschieben der Instruktion i2 in Basisblock _L_0 Instruktionen i4 und i5 nicht mehr nach _L_0 verschoben werden dürfen. Instruktion i3 kann dagegen hinter i2 in Basisblock _L_0 verschoben werden. Die Abhängigkeit zwischen i2 und i3 wird dabei weiterhin im regulären DAG geführt.

4.5 Heuristiken

Zusätzlich zu den Heuristiken des lokalen Schedulings wurden zwei weitere Heuristiken für das Treegion-Scheduling implementiert. Diese sind angelehnt an die Heuristiken, welche in [25] vorgestellt werden.

4.5.1 Dependence Height

Die Dependence Height Heuristik ist eine Variation der Maximum Delay-Heuristik. Da innerhalb einer Treegion alle Instruktionen innerhalb einer Treegion einen DAG aufspannen, werden Instruktionen nahe des Wurzel-Basisblocks einen sehr hohen Maximum Delay-Wert zugewiesen, welcher von der Senke der Treegion aus nach oben über die Abhängigkeitskanten bestimmt wird. Dies führt dazu, dass Instruktionen nahe des Wurzel-Basisblocks bevorzugt werden und Instruktionen aus tiefer liegenden Basisblöcken nur innerhalb eines höher liegenden Basisblocks geschedult werden, wenn innerhalb eines Instruktionsbündels ein Bündelpartner fehlt. Um Instruktionen aus tiefer liegenden Basisblöcken eine größere Chance des aufsteigens zu geben, werden bei der Dependence Height-Heuristik Maximum Delay-Werte nicht Basisblock-übergreifend bestimmt. Jeder Instruktion eines Basisblocks werden Maximum Delay-Werte zugewiesen, wie sie innerhalb des lokalen Schedulings zugewiesen worden wären.

4.5.2 Exitcount

In einigen Fällen kann die Prioritisierung mittels Depencence Height-Heuristik zu viele Instruktionen in einen Vorgänger-Basisblock schedulen. Die Exit Count-Heuristik prioritisiert daher Instruktionen von Basisblöcken gemäß der Anzahl der Nachfolge-Basisblöcken innerhalb der Treegion. Ein Basisblock und die darin enthaltenen Instruktionen werden höher prioritisiert wenn dieser Basisblock eine hohe Anzahl an direkten Nachfolgern innerhalb der Treegion besitzt. Zusätzlich werden Instruktionen sekundär nach ihrer Dependence Height prioritisiert.

4.6 Erweiterte Prioritätsvergabe

Durch die Baum-Struktur einer Treegion geschieht das Scheduling von nebenläufigen Pfaden, indem nach der Verzweigung zuerst ein Pfad geschedult wird, um danach weitere nebenläufige Pfade zu schedulen. Dabei kann sich dies in komplexeren Treegions beliebig weit verzweigen. Um diese Fälle korrekt zu behandeln, müssen innerhalb der Prioritätsvergabe Regeln eingebaut werden. Alle diese Regeln werden nur dann aktiv, wenn eine vorhergehende Instruktion geschedult wurde. Ist dies nicht der Fall, so wird gerade die erste Instruktion im Wurzel-Basisblock der Treegion geschedult und es findet keine Einschränkung statt. Ist dagegen eine Vorgänger-Instruktion bekannt, so wird überprüft, ob die zu prioritisierende Instruktion und die Vorgänger-Instruktion innerhalb des gleichen Basisblocks liegen. Ist dies der Fall, so werden auch hier keine Einschränkungen vorgenommen.

Ist dagegen die Vorgänger-Instruktion i in einem Basisblock B, von welchem aus der Basisblock C der zu scheduleden Instruktion j nicht erreichbar ist, so wird überprüft, ob sich noch weitere Instruktionen innerhalb der Ready-List befinden, die auf einem von B erreichbaren Pfad liegen. Ist dies der Fall, so wird ein Scheduling von j unterbunden, um zuerst den gesamten Pfad, auf

dem B sich befindet, zu schedulen. Sobald dieser Pfad an Basisblöcken geschedult wurde, wird ein Scheduling von j erlaubt. Dies dient dazu die schon erwähnte Vorgehensweise, zuerst einen Pfad zu schedulen, einzuhalten. Es werden somit zuerst alle Instruktionen eines Pfades geschedult auch wenn Instruktionen mit einer höheren Priorität aus einem weiteren nebenläufigen Pfad ausführbereit sind.

Wird eine Instruktion j aus einem Basisblock C in einen Basisblock B geschedult und beide Basisblöcke liegen auf einem Pfad, so wird überprüft, ob Instruktionen in B geschedult wurden, die innerhalb des exclusionDAG eine Abhängigkeit zu j besitzen. Ist dies der Fall, so wird ein Scheduling von j unterbunden. Hierdurch wird ein Verschieben von sich ausschliessenden Instruktionen in einen gemeinsamen Basisblock unterbunden.

Zusätzlich sind weitere Fälle zu beachten, wenn Instruktionen gebündelt werden. Angenommen es gibt eine direkt vorher geschedulte Instruktion i und zwei zu bündelnde Instruktionen x und y. Dann können folgende Fälle auftreten, die gesondert zu beachten sind:

a) **Bedingung:** *Instruktionen x aus Basisblock B und y aus C sind aus verschiedenen, nicht auf einem gemeinsamen Pfad innerhalb des CFG liegenden Basisblöcken und sollen hinter i in A verschoben werden. Sowohl B als auch C ist von A aus erreichbar.*
Vorgehen: Ist i eine Sprung-Instruktion, so ist eine Bündelung von x und y nicht erlaubt. Dadurch wird ein Verschieben mindestens einer der beiden Instruktion aus ihrem Kontrollfluß-Pfad verhindert.
Ist i keine Sprung-Instruktion, so prüfe ob x und y keine Abhängigkeiten innerhalb des exclusionDAG aufweisen. Ist dies der Fall, so dürfen x und y nicht gemeinsam innerhalb eines Pfades ausgeführt werden. Eine Bündelung von x und y ist in diesem Fall nicht erlaubt.

b) **Bedingung:** *Instruktionen i und x liegen in B. Basisblock C ist nicht von B aus erreichbar.*
Vorgehen: Eine Bündelung von x und y is ausgeschlossen, da hierdurch eine Veränderung der Semantik des Programms entstehen würde.

c) **Bedingung:** *Instruktionen i und x liegen in A. Basisblock C ist von A aus erreichbar.*
Vorgehen: Prüfe, ob Instruktionen in A verschoben wurden, die eine Abhängigkeit zu x innerhalb des exclusionDAG aufweisen. Ist dies der Fall, so ist eine Bündelung ausgeschlossen.

Diese Fälle sind nochmals in Abbildung 4.5 zu sehen. Alle weiteren Fälle, die auftreten können, erlauben ohne Einschränkungen eine Prioritisierung und somit ein Scheduling des Bündels.

4.7 Erhaltung der Korrektheit beim Verschieben von Instruktionen innerhalb der LLIR

Anders als im Falle des lokalen Schedulings werden Instruktionen beim globalen Scheduling nicht nur innerhalb eines Basisblocks verschoben sondern, falls ein spekulatives Scheduling von Instruk-

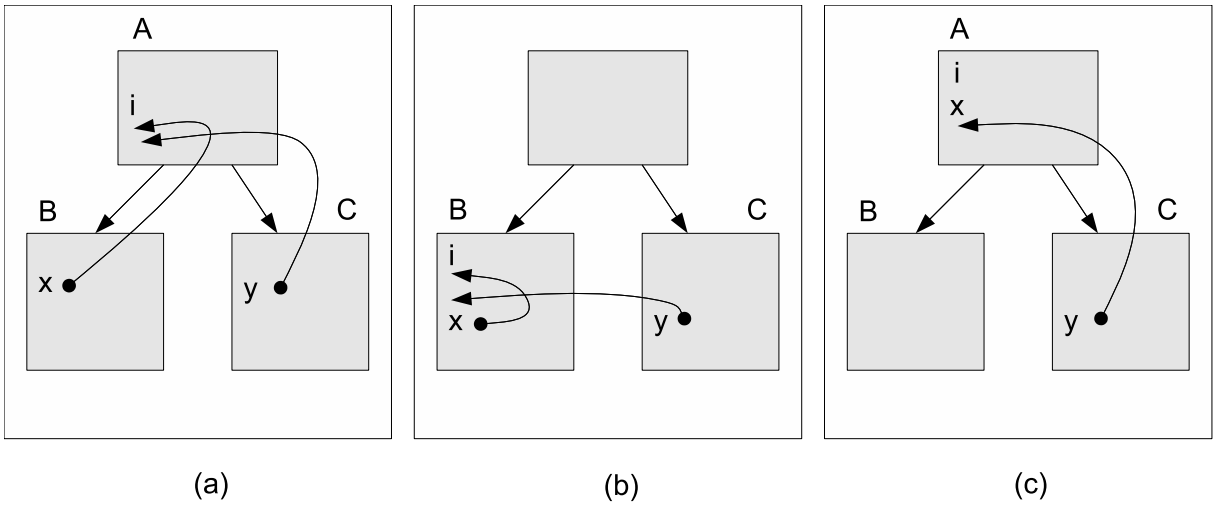

Abbildung 4.5: *Mögliche Fälle beim Bündeln von Instruktionen*

tionen stattfindet, auch über Basisblock-Grenzen hinaus. Ein weiteres Problem besteht darin, dass der Kontrollfluß des Programms zu keiner Zeit verändert werden darf. Da innerhalb des implementierten Schedulers jedes Verschieben einer Instruktion j generell an eine Vorgänger-Instruktion i gekoppelt ist, müssen beim Verschieben von Instruktion j nach i verschiedene Regeln beachtet werden. Eine wichtige Information beim globalen Scheduling von Instruktionen ohne Veränderung der Verzweigungsstruktur der LLIR spielt dabei die Speicherung der letzten geschedulten Instruktion innerhalb jedes Basisblocks der gerade betrachteten Treegion. Somit kann beim Scheduling einer Instruktion j in Block B diese hinter die letzte geschedulte Instruktion in B verschoben werden. Die nachfolgend vorgestellten Regeln dienen zur Bestimmung des korrekten Basisblocks, in den j unter Annahme des Basisblocks von i verschoben werden soll.

Die wichtigste Regel ist, dass Sprung-Instruktionen generell nicht verschoben werden. Würde dies geschehen, wäre der Kontrollfluß innerhalb der LLIR nicht mehr korrekt. Alle weiteren Regeln sind von einer Anfangsbedingung abhängig. Diese sollen nun durch diese gegliedert vorgestellt werden:

Anfangsbedingung: *Scheduling von j findet statt, nachdem eine Instruktion i geschedult wurde.*

Hier ist eine weitere Unterteilung zwischen der Situation, ob die vorher geschedulte Instruktion i eine Sprung-Instruktion ist, also den Basisblock, in dem diese sich befindet, abschliesst, oder aber eine andere Instruktion ist, notwendig. CALL-Instruktionen sind hier nicht separat zu beachten, obwohl diese innerhalb der LLIR auch Basisblöcke abschliessen. Diese erzeugen aber keine Verzweigung des Kontrollflusses. Die Gewährleistung der Korrektheit ist in diesem Fall durch die Datenabhängigkeiten gegeben.

Bedingung: *i ist eine Sprung-Instruktion.*

Hier sind drei Fälle zu beachten, die auftreten können.

a) **Bedingung:** *Der Basisblock C, aus dem j stammt, ist direkter Nachfolger des Basisblocks B, aus dem die Sprung-Instruktion i stammt.*
Vorgehen: Ist dies der Fall, so verschiebe j hinter die zuletzt in C geschedulte Instruktion.

b) **Bedingung:** *Der Basisblock C, aus dem j stammt, liegt innerhalb des CFG auf einem Pfad mit Basisblock B, aus dem die Sprung-Instruktion i stammt.*
Vorgehen: Ist dies der Fall, so bestimme den direkten Nachfolger-Basisblock X auf dem Pfad zu B. Verschiebe j hinter die zuletzt geschedulte Instruktion in X.

c) **Bedingung:** *Der Basisblock C, aus dem j stammt, liegt innerhalb des CFG nicht auf einem Pfad mit Basisblock B, aus dem die Sprung-Instruktion i stammt.*
Vorgehen: Ist dies der Fall, so verschiebe j hinter die zuletzt geschedulte Instruktion innerhalb ihres eigenen Basisblocks C. Dieser Fall tritt ein, wenn alle Instruktionen innerhalb eines nebenläufigen Pfades geschedult wurden und nun ein anderer nebenläufiger Pfad geschedult wird.

Diese Fälle sind nochmals in Abbildung 4.6 zu sehen. Die angedeutete „Explosion“ am Pfeil zwischen Instruktion j und i soll dabei symbolisieren, dass ein solches Verschieben von Instruktion j die Semantik des Programms verändern würde.

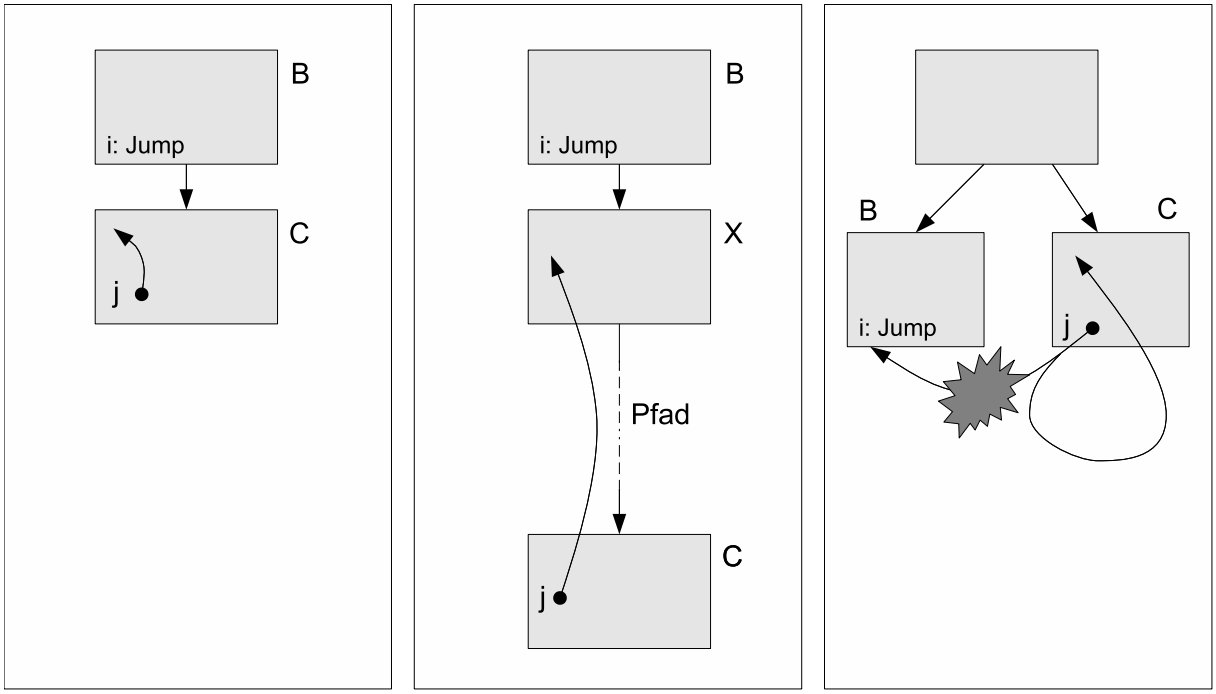

Abbildung 4.6: *Mögliche Fälle beim Verschieben von Instruktionen nach einer Sprung-Instruktion*

Bedingung: *i ist eine Nicht-Sprung-Instruktion.*

Im Falle des Verschiebens einer Instruktion j in Basisblock C hinter eine Nicht-Sprung-Instruktion i

in Basisblock B sind drei Fälle zu beachten. Davon ausgenommen ist der triviale Fall wenn $B = C$ gilt. Dann wird j hinter i verschoben.

a) **Bedingung:** *B und C liegen auf einem Pfad innerhalb des CFG und B liegt auf dem Pfad vor C.*
Vorgehen: In diesem Fall wird j hinter i verschoben.

b) **Bedingung:** *B und C liegen auf einem Pfad innerhalb des CFG und B liegt auf dem Pfad nach C.*
Vorgehen: Es muss überprüft werden, ob es keine Abhängigkeiten innerhalb C zwischen j und weiteren Instruktionen gibt. Ist dies nicht der Fall, wird j hinter i verschoben.

c) **Bedingung:** *B und C liegen nicht auf einem gemeinsamen Pfad.*
Vorgehen: Zuerst ist zu prüfen, ob sich in der Ready-List zusätzlich zu j eine Sprung-Instruktion k befindet, welche zum selben Ausführungszyklus schedulebar ist, wie j. Ist dies der Fall, so verschiebe j vor k. Dieser Fall kann auftreten, wenn ein anderer Pfad komplett geschedult wurde und i in einen Basisblock verschoben werden soll, in dem keine weitere Instruktion ausser der Sprung-Instruktion k verschoben wird.
Ist keine Sprung-Instruktion in der Ready-List, so verschiebe j hinter die zuletzt geschedulte Instruktion in C.

Diese Fälle, ausser des trivialen, sind nochmals in Abbildung 4.7 zu sehen.

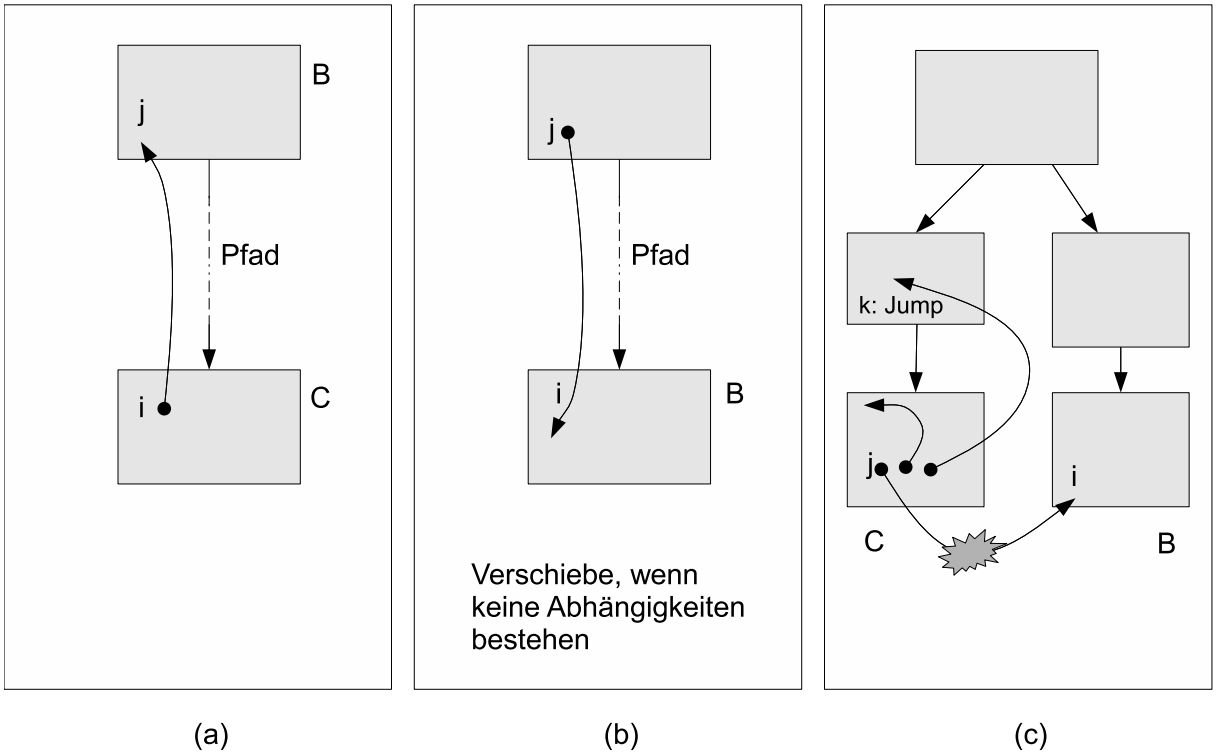

Abbildung 4.7: *Mögliche Fälle beim Verschieben von Instruktionen nach einer Nicht-Sprung-Instruktion*

Anfangsbedingung: *Scheduling von j findet statt, ohne dass eine Instruktion i geschedult wurde.*

In diesem Fall wird j in den Wurzel-Basisblock der Treegion verschoben. Diese Regel ist wichtig, um ein Bündel einer IP- und LS-Instruktion korrekt zu schedulen. So kann es passieren, dass die IP-Instruktion spekulativ aus einem niedrigeren Basisblock innerhalb der LLIR zusammen mit der LS-Instruktion in die Wurzel der Treegion verschoben werden soll. Es steht hier keine weitere Information bezüglich der zuletzt geschedulten Instruktion i bereit, somit muss dieses Bündel explizit in die Wurzel verschoben werden. Durch diese Regel wird auch verhindert, dass die IP-Instruktion fälschlicherweise innerhalb ihres Basisblocks verbleibt und die LS-Instruktion aus der Wurzel zum niedrigeren Basisblock verschoben wird. Diese Situation ist in Abbildung 4.8 zu sehen.

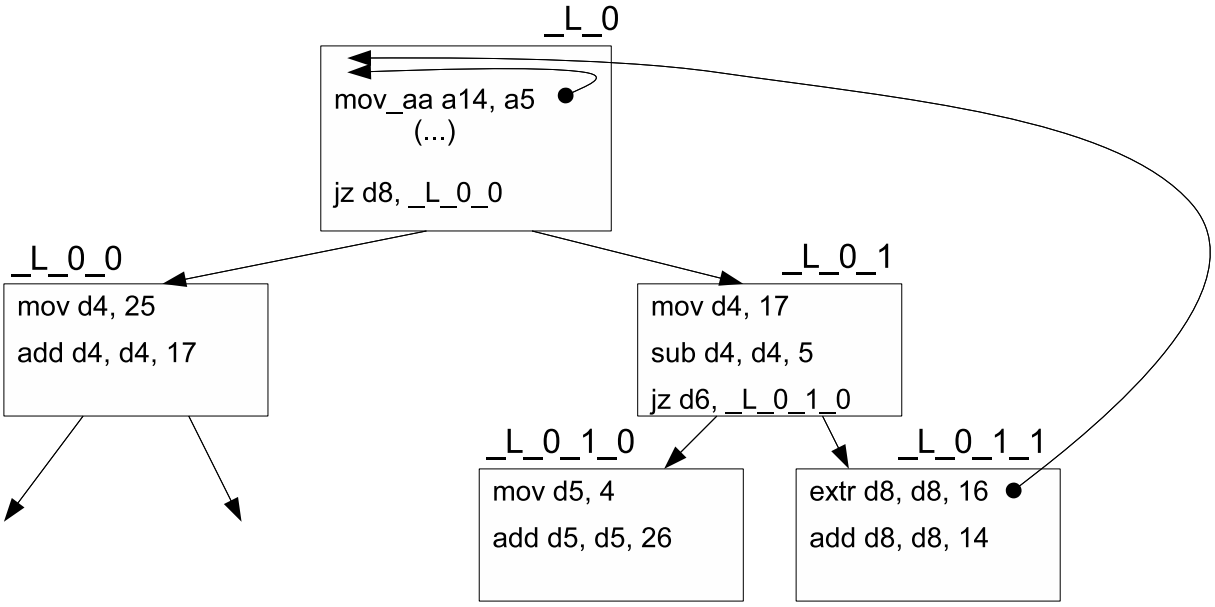

Abbildung 4.8: *Korrektes Verschieben von Instruktionen zur Treegion-Wurzel*

Kapitel 5

Ergebnisse

Das folgende Kapitel soll den in dieser Diplomarbeit erarbeiteten Instruction Scheduler evaluieren. Es sollen ACET-Werte zwischen dem ungeschedulten und dem geschedulten Code unter Verwendung unterschiedlicher Heuristiken verglichen werden. Dazu wurden, um aussagekräftige Ergebnisse zu erhalten, 69 repräsentative Benchmarks aus der *WCETBENCH*-Benchmarksammlung des WCC ausgewählt. Diese bestehen aus Benchmarks der Benchsuiten *DSPstone* [26], *MediaBench* [27], *MiBench* [28], *misc*, *MRTC* [29], *NetBench* [30], *Bitonic* sowie *UTDSP* [31]. Die Benchmarks innerhalb der *misc* Benchsuite stammen dabei aus der ICD-C eigenen Testbench-Suite. Die Benchmarks wurden dabei unter folgenden Kriterien ausgewählt:

- Alle Benchmarks, die ACET-Werte aufweisen, welche extrem vom ungeschedulten Fall abweichen und nachweislich auf einen Fehler in der ACET-Bestimmung zurückzuführen sind, wurden entfernt.
- Benchmarks, welche die selbe Funktionalität implementieren, wurden entfernt. Beispiele sind Programme aus der DSPStone-Suite, deren einziger Unterschied darin besteht, dass auf unterschiedlich großen Arrays gearbeitet wird. Dabei wurden die Benchmarks entfernt, die eine geringere Änderung der ACET hin zum positiven oder negativen zeigen.
- Benchmarks, die eine relativ geringe ACET aufweisen und die Veränderung der ACET im geschedulten Fall im Vergleich zum ungeschedulten Fall im prozentualem Nachkommabereich lag, wurden entfernt. Hierbei handelt es sich um nicht repräsentative Benchmarks, die kein aussagekräftiges Evaluieren des Schedulers erlauben.

Eine genaue Auflistung aller ausgewählten Benchmarks kann in Anhang A eingesehen werden.

Alle Benchmark-Werte wurden gesondert für jede der drei Optimierungsstufen des WCC bestimmt. Mit steigender Optimierungsstufe werden dabei innerhalb des WCC eine größere Zahl an Optimierungen eingeschaltet. Durch die unterschiedliche Anzahl an Optimierungen vor und nach dem Scheduling ergeben sich verschiedene ACET-Werte. Deshalb ist ein Vergleich zwischen unterschiedli-

chen Optimierungsstufen nicht möglich. Es soll daher ein Vergleich getrennt nach der jeweiligen Optimierungsstufe erfolgen. Alle Resultate basieren auf dem Code der aus dem gecachten Flash-Speichers des TriCore Prozessors ausgeführt wurde. Die Wahl des Speichers ist dadurch motiviert, dass heutige High-Performance-Systeme auf den Cache angewiesen sind und daher typischerweise diesen Speicher ausnutzen würden. Leider war es aufgrund einer Limitierung des derzeitigen ACET-Simulators nicht möglich Werte für den Fall zu bestimmen, wenn der Programmcode innerhalb des schnellen SRAM Speichers (Zugriffszeit von 1 Taktzyklus) liegen würde.

5.1 Lokales Scheduling

Innerhalb dieses Abschnittes sollen Ergebnisse bezüglich des lokalen Schedulings vorgestellt werden.

Die meisten Ergebnisse wurden für das lokale Scheduling nach der Registerallokation gesammelt. Dies liegt zum einen daran, dass dies das erste Ziel der Diplomarbeit war und zum anderen sich gut dazu eignet, die entwickelten Heuristiken zu evaluieren, ohne dass zusätzlicher Spill-Code im Falle des Schedulings vor der Registerallokation die Ergebnisse verändert. Diese Ergebnisse sollen nachfolgend vorgestellt werden.

5.1.1 Scheduling nach der Registerallokation

Es sollen nun sowohl die Maximum Delay- wie auch die Number of Child-Instructions-Heuristik einzeln und in Kombination mit den verschiedenen zusätzlichen Heuristiken verglichen werden. Es sollen dazu sowohl Werte für den ASAP- als auch ALAP-Algorithmus vorgestellt werden. Innerhalb der folgenden Diagramme wird dabei der Mittelwert der Verringerung der ACET-Zeit über alle ausgewählten Benchmarks aufgezeigt. Der Vergleichswert von 100% ist hierbei immer der Mittelwert der ACET-Werte aller Benchmarks bei selber Optimierungsstufe aber ohne Scheduling. Um die Größe der Diagramme zu Verringern wurden Abkürzungen für die verschiedenen Heuristiken eingeführt. Diese wären: **MD** = Maximum Delay, **NC** = Number of Child-Instructions, **IP** = Instruction Priority, **MO** = Mobility. Die dunklen Diagrammbalken stehen für den ASAP-Algorithmus, die hellen für den ALAP-Algorithmus.

Maximum Delay-Heuristik

Diese Heuristik weist einer Instruktion die Länge des Pfades von dieser Instruktion hin zur Senke der diese Instruktion enthaltenden Region zu. Siehe dazu auch 3.5.

Wie in Abbildung 5.1 für die Optimierungsstufe O0 zu sehen ist, ist das Scheduling durch den ASAP-Algorithmus von Vorteil. Dies liegt zum einen darin begründet, dass Delay-Zyklen eher mit

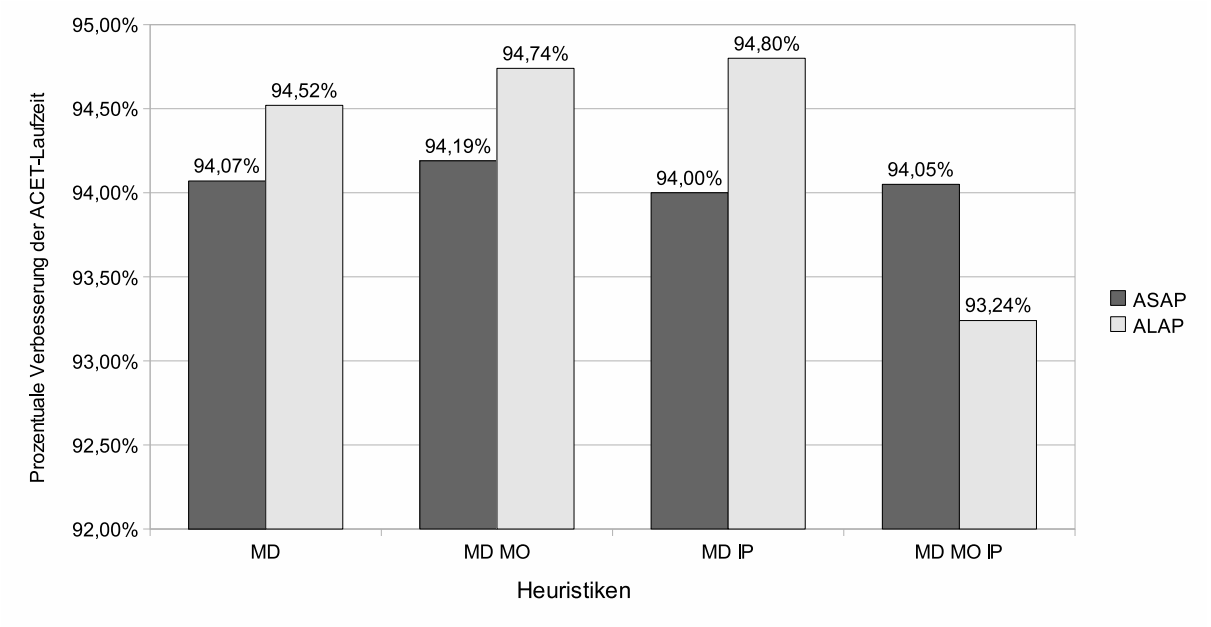

Abbildung 5.1: *Vergleich der möglichen Fälle mit der Maximum Delay-Heuristik bei Optimierungsstufe O0*

unabhängigen Instruktionen füllbar sind, da eine Kombination aus Instruktionen, welche diese verursacht, nicht hinausgezögert wird. Zum anderen liegt dies auch darin begründet, dass in einigen Fällen das Scheduling von IP- oder LS-Instruktionen, welche nicht abhängig von anderen Instruktionen innerhalb eines Basisblöcks sind, bis zum Ende dieses Basisblocks hinausgezögert wird. Es ist dann nicht immer möglich geeignete Instruktionen als Partner zum Bündeln zu finden. Diesbezüglich gibt es aber auch gegenteilige Fälle, in denen durch den ALAP-Algorithmus mehr Bündel ermöglicht werden. Dies liegt darin begründet, dass Instruktionen, welche eine hohe Priorität aufweisen, aber im derzeitigen Ausführungszyklus des Schedulers keine Bündelpartner innerhalb der Ready-List vorhanden sind, diese Instruktion alleine geschedult wird. Durch die Verzögerung des Schedulings der Instruktion zu einem späteren Zeitpunkt sind möglicherweise Bündelpartner ausführbereit. Zudem werden auch in einigen Fällen Stalls und Delay-Zyklen stärker vermieden, denn die Verzögerung des Schedulings von Instruktionen durch den ALAP-Algorithmus kann zum Füllen solcher Issue-Slots führen.

Zu sehen ist auch, dass die Instruction Priority-Heuristik in den meisten Kombinationen Vorteile bringt, da hier Dual-Pipeline Instruktionen so früh wie möglich geschedult werden und somit potentielle Nachfolge-Instruktionen als Bündelpartner zur Verfügung stehen. Dem gegenüber steht die Mobility-Heuristik, welche offensichtlich eher nachteilig scheint. Dies konnte allerdings für die Fälle, die zu dieser Verschlechterung führen, nicht grundsätzlich verifiziert werden. So weist der WCET-Analyser aiT keine Verschlechterung der WCET in diesen Fällen auf und zeigt in der Pipeline-Visualisierung eine gleiche Anzahl von Zyklen für alle Basisblöcke auf, wenn eine Heuristik-Kombination mit Mobility sowie ohne verglichen wird. Somit ist anzunehmen, dass im CoMET ACET-Simulator weitere Annahmen getroffen werden.

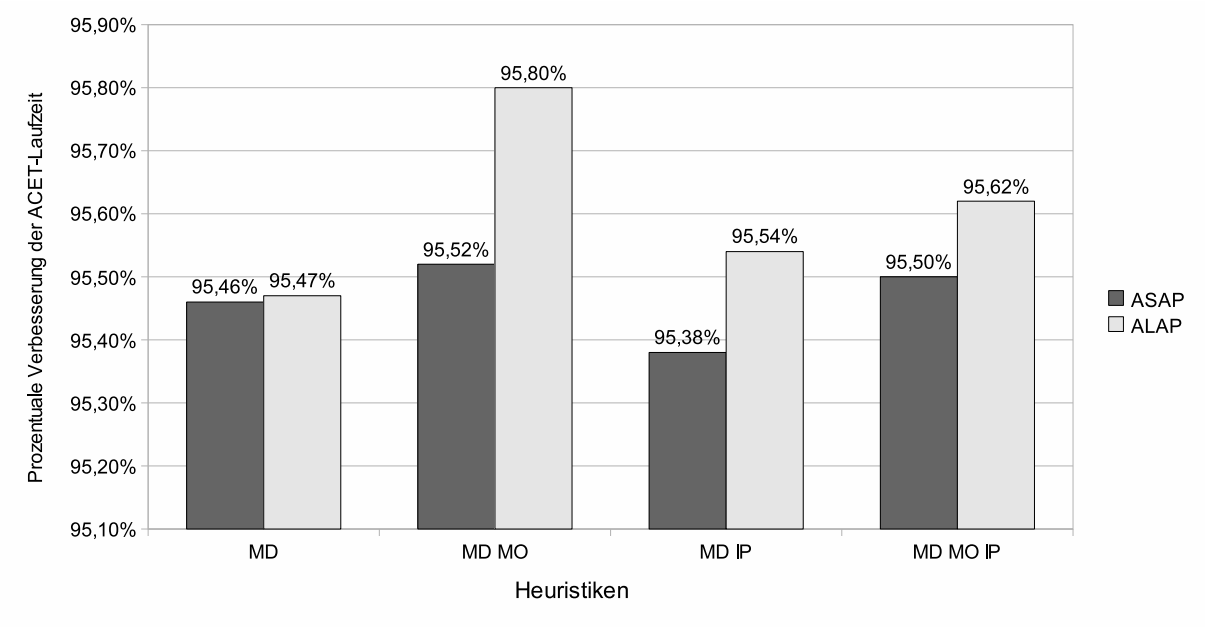

Abbildung 5.2: *Vergleich der möglichen Fälle mit der Maximum Delay-Heuristik bei Optimierungsstufe O1*

Im Fall der Optimierungsstufe O1 gelten die gleichen Ausführungen wie im Fall der Optimierungsstufe O0. Im direkten Vergleich fällt auf, dass die Verbesserung der Laufzeiten hier nicht so groß ist. Auf Gründe hierfür wird nach der Evaluation für die Optimierungsstufe O3 eingegangen.

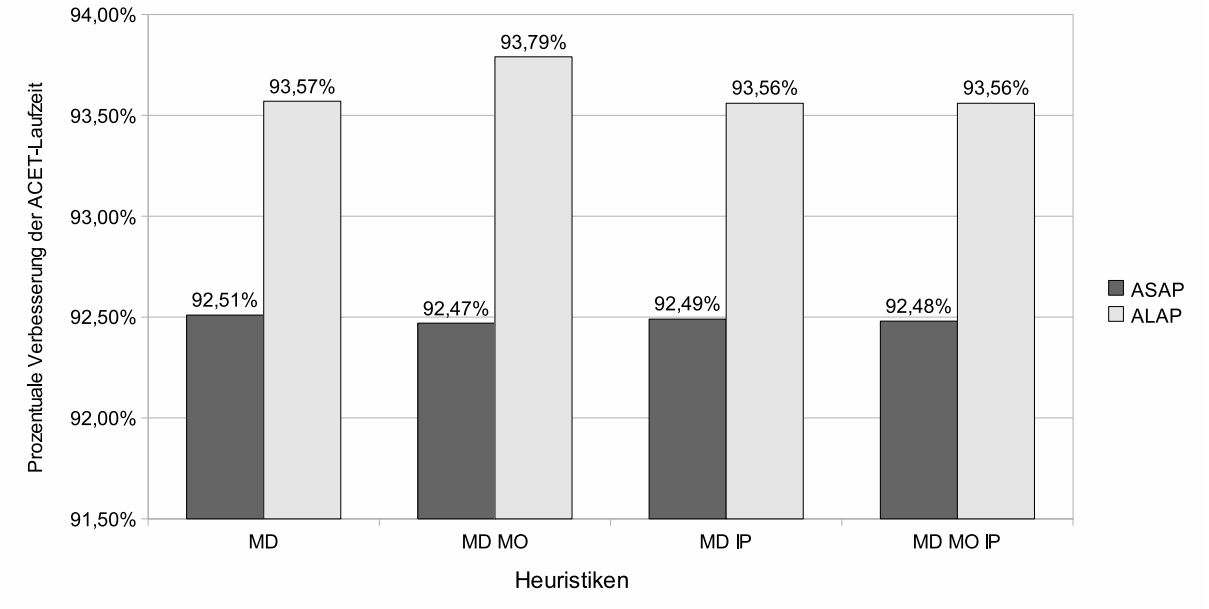

Abbildung 5.3: *Vergleich der möglichen Fälle mit der Maximum Delay-Heuristik bei Optimierungsstufe O2*

Wie an den Diagrammen ersichtlich, ist das Potential zum Scheduling innerhalb der Optimierungsstufe O1 am geringsten, gefolgt von O0. Optimierungsstufe O2 zeigt das größten Optimierungspotential für den Scheduler. O3 liegt ein wenig darunter. Dies liegt an den unterschiedlichen Optimie-

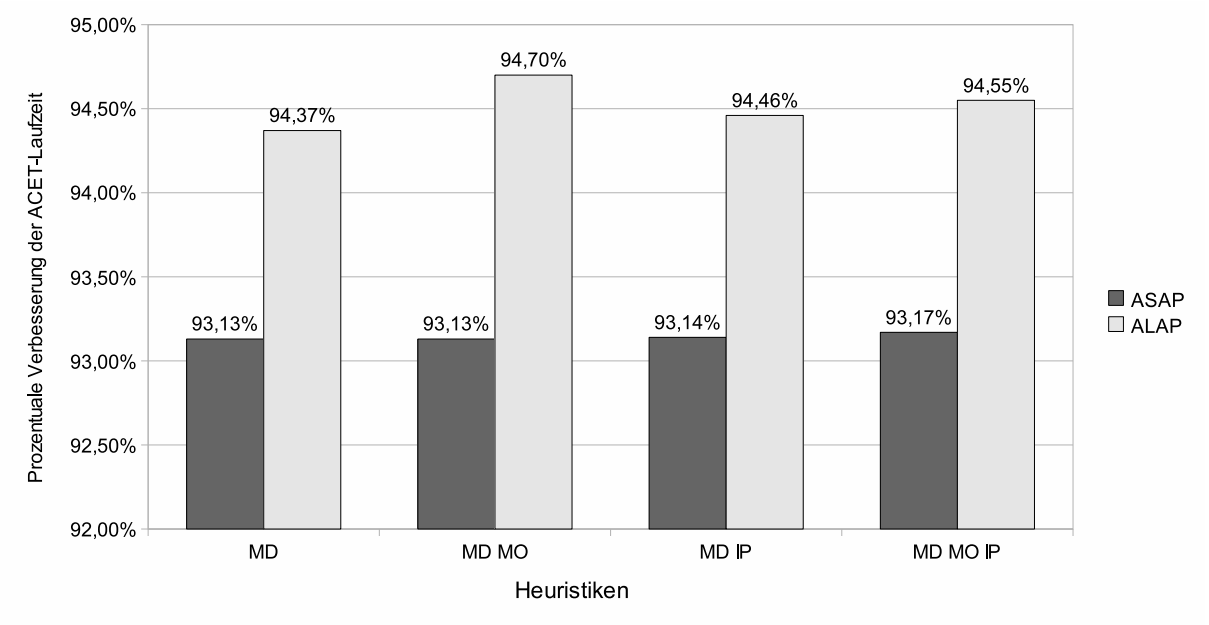

Abbildung 5.4: *Vergleich der möglichen Fälle mit der Maximum Delay-Heuristik bei Optimierungsstufe O3*

rungen, die innerhalb des WCC durch die Optimierungsstufen aktiviert werden. Es soll exemplarisch für den Benchmark *lms* aus der DSPstone_floating_point Benchsuite gezeigt werden, welche Auswirkungen diese Optimierungen auf den Code und somit auf den nachfolgend eingesetzten Scheduler haben. Die nachfolgende Tabelle 5.1 führt dazu verschiedene statistische Informationen bezüglich des Benchmarks auf. Bei der Bestimmung dieser wurden die Default-Heuristiken des Schedulers verwendet. Wie dabei zu sehen ist, wurde sowohl im Falle von O0 als auch O2 eine Verbesserung

Tabelle 5.1: *Statistiken für Benchmark lms unter verschiedenen WCC-Optimierungsstufen*

	O0	O1	O2	O3
Instruktionen (total)	153	148	145	164
IP-Instruktionen	36	23	23	27
LS-Instruktionen	107	118	115	130
IP-LS Ratio	33,00%	19,00%	20,00%	20,00%
Bündel vor Scheduling	13	12	8	11
Bündel nach Scheduling	26	17	20	24
Verbesserungsfaktor	2	1,42	2,5	2,18
Anzahl Basisblöcke	13	13	10	10
ACET vor Scheduling	1391	1289	1160	1070
ACET nach Scheduling	1216	1274	1015	981
Verbesserung	12,58%	1,16%	12,50%	8,32%

von mindestens 12,5% erzielt. Dies liegt daran, dass im Vergleich zu der Anzahl der Bündel vor dem Scheduling der Scheduler die Anzahl an Bündeln verdoppeln konnte. Dies war auch im Falle von O3 möglich, hier ist allerdings auch die Gesamtzahl der Instruktionen am höchsten. Somit ist

der prozentuale Anteil an Bündeln im Vergleich zur Gesamtzahl der Instruktionen geringer wodurch auch die ACET-Verbesserung kleiner ausfällt. Ein weiterer Grund für das gute Abschneiden der Optimierungsstufen O2 und O3 ist, dass hier Basisblöcke zusammengefasst wurden. So weisen O0 und O1 13 Basisblöcke auf, wohingegen O2 und O3 3 Basisblöcke weniger aufweisen. Dies ist auch der Grund für die kaum vorhandene Verbesserung bei O1. Hier wurden im Vergleich zu O0 viele IP-Instruktionen wegoptimiert. Somit besteht kein großes Potential der Bündelung. Zwar ist diese Optimierung auch in O2 und O3 vorhanden, aber hier wurde durch das Verschmelzen von Basisblöcken neues Optimierungspotential für den lokalen Scheduler geschaffen. Das geringe Potential im Falle von O1 ist auch in der beispielhaften Statistik des Basisblocks _L5 in Tabelle 5.2 zu sehen.

Tabelle 5.2: *Statistiken des _L5 Basisblocks beim lms Benchmark*

	O0	O1
IP-Instruktionen	11	2
LS-Instruktionen	7	7
Bündel vor Scheduling	1	1
Bündel nach Scheduling	7	2

Diese Tabelle zeigt, dass in _L5 die meisten Bündel innerhalb des Programms für Optimierungsstufe O0 gebildet werden. Wie auch zu sehen ist, werden dabei sowohl für O0 als auch O1 die größtmögliche Anzahl an Bündeln durch den Scheduler gebildet. Durch die Optimierungen innerhalb der Optimierungsstufe O1 wurden 9 IP-Instruktionen eliminiert, wodurch nur noch 2 Bündel erzeugbar waren.

Number of Child-Instructions-Heuristik

Diese Heuristik weist einer Instruktion die Anzahl der von dieser Instruktion abhängigen Nachfolge-Instruktionen zu. Siehe dazu auch 3.5.

Generell stellte sich beim Vergleich der Ergebnisse der Number of Child-Instructions-Heuristik und der Maximum Delay-Heuristik heraus, dass diese der Maximum Delay-Heuristik unterlegen ist. Dabei zeigten die Ergebnisse, bis auf zwei Fälle die selben Unterschiede zwischen den verschiedenen Kombinationen an Heuristiken. Diese zwei Fälle treten bei der Optimierungsstufe O1 auf, wie in Abbildung 5.5 zu sehen. Diese Fälle sind zum einen die Kombination der Heuristiken Mobility und Instruction Priority und zum anderen nur Instruction Priority. Grund hierfür ist die geringere Differenzierung zwischen Instruktionen, die die selbe Anzahl an Kindern haben. So werden Instruktionen auf dem kritischen Pfad, welcher der längste Pfad innerhalb eines Basisblocks ist, nicht bevorzugt. Eine unnötigte Verzögerung des längsten Pfades kann dadurch auftreten, was eine längere Ausführungszeit des Basisblocks und somit eine Erhöhung der ACET zur Folge hat.

Die zwei Fälle, in denen die Number of Child-Instructions-Heuristik einen höheren Mittelwert auf-

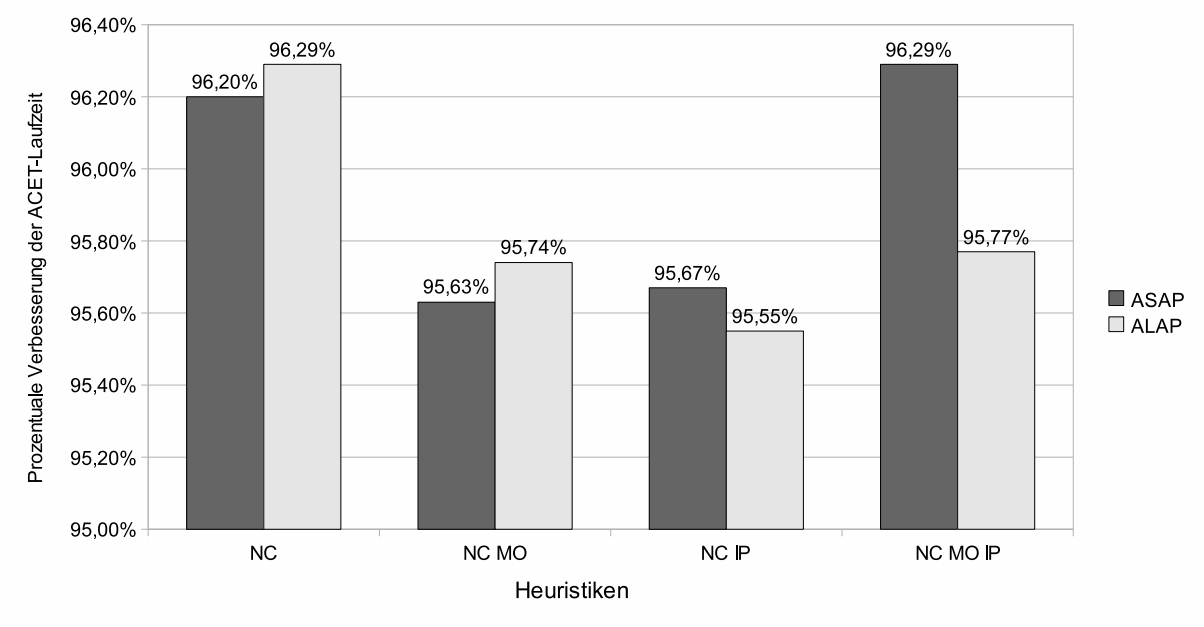

Abbildung 5.5: *Vergleich der möglichen Fälle mit der Number of Child-Instructions-Heuristik bei Optimierungsstufe O1*

weist, sind auch die einzigen Fälle in denen der ALAP-Algorithmus einen höheren Durchschnitt erreicht wie der ASAP-Algorithmus. Als Beispiel sei hier der *matrix1* Benchmark aus der floating_point *DSPStone*-Benchsuite genannt. Die folgende Tabelle 5.3 enthält Werte für die vier verschiedenen Fälle.

Tabelle 5.3: *Statistiken des matrix1 (floating point) Benchmarks für die Number Of Child-Instructions-Heuristik*

	Mobility Instruction Priority	Instruction Priority
ASAP		
ACET	23065	23668
Bündel vor Scheduling	6	6
Bündel nach Scheduling	9	9
ALAP		
ACET	22656	22656
Bündel vor Scheduling	6	6
Bündel nach Scheduling	9	9

Wie zu sehen ist, wurden in allen Fällen gleich viele Bündel gebildet. Trotzdem sind die ACET-Werte im Falle des ALAP-Algorithmus niedriger. Dies liegt daran, dass Stall-Zyklen zwischen Store- und nachfolgenden abhängigen Load-Instruktionen durch Einfügen von unabhängigen Instruktionen im ALAP-Fall vermieden wurden. Die dafür verwendeten Instruktionen wurden im ASAP-Fall vor dem Auftreten dieser Load- und Store-Instruktionen geschedult.

Verbesserte Bestimmung von Lade-/Speicher-Abhängigkeiten-Heuristik

Diese Heuristik minimiert mittels Werteanalyse die Anzahl an Abhängigkeiten zwischen Lade- und Speicher-Instruktionen. Es werden nur tatsächliche Abhängigkeiten modelliert. Siehe dazu auch 3.5.

Abbildung 5.6 zeigt die bestimmten Mittelwerte der Benchsuiten für die vier Optimierungsstufen. Es wurden dabei jeweils die Default-Heuristiken des Schedulers verwendet, sowie zusätzlich die genannte Heuristik angewandt, hier ST/LD-Heuristik genannt. Als Vergleichswert von 100% wird der Mittelwert der ACET-Werte des ungeschedulten Codes der Benchsuiten je Optimierungsstufe angenommen.

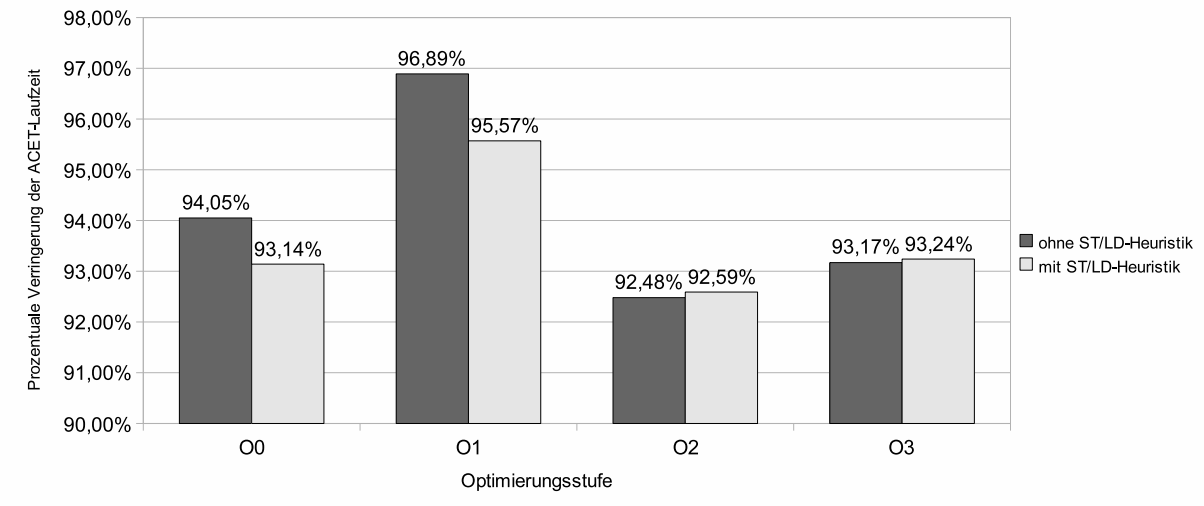

Abbildung 5.6: *Resultate der 'Verbesserte Bestimmung von Lade-/Speicher-Abhängigkeiten'-Heuristik*

Wie zu erkennen ist, bringt die Verringerung von Abhängigkeiten und somit die Erweiterung der Möglichkeiten des Schedulers weitere Vorteile bei den Optimierungsstufen O0 sowie O1. Die Werte für O2 und O3 sind bezüglich der ACET geringfügig schlechter, diese Verschlechterung tritt bei einigen ausgewählten Benchmarks auf. Als Beispiel sei der *ludcmp* Benchmark aus der *MRTC* Benchsuite zu nennen. Tabelle 5.4 zeigt ausgewählte Werte für die Default-Einstellungen des Schedulers ohne sowie mit ST/LD-Heuristik bei Optimierungsstufe O2.

Tabelle 5.4: *Statistiken des ludcmp Benchmarks für die ST/LD-Heuristik für Optimierungsstufe O2*

	ohne ST/LD-Heuristik	mit ST/LD-Heuristik
ACET	7450	7602
WCET	7173	7128
Bündel vor Scheduling	31	31
Bündel nach Scheduling	56	56

Wie daraus zu erkennen ist, werden hier die gleiche Anzahl an Bündeln erzeugt. Dabei zeigen die

ACET- sowie WCET-Werte eine unterschiedliche Tendenz. Während die ACET-Werte mit ST/LD-Heuristik hier sich verschlechtern, verbessern sich die WCET-Werte im Falle dieses Benchmarks. Grund sind unterschiedliche Annahmen bei der Simulation des Codes innerhalb des CoMET Simulators und des aiT WCET-Analyzers.

Generell sind die verbuchten Verbesserungen oder Verschlechterungen in den meisten Fällen bis auf sehr große Benchmarks nicht auf ein höheres Potential an Bündelpartnern durch verringerte Abhängigkeiten zurückzuführen, sondern die daraus resultierende veränderte Abfolge an Instruktionen. Dadurch werden Stalls wie z.B. Linecrossing-Effekte (siehe 2.1.4) vermieden oder Delay-Zyklen ausgenutzt. In den meisten Fällen ergibt sich kein vergrößertes Optimierungspotential durch diese Heuristik. Dies liegt darin begründet, dass es unerheblich ist, ob eine IP-Instruktion mit einer Lade- oder einer Speicher-Instruktion kombiniert wird. Die daraus resultierende Parallelisierung bleibt erhalten. Die zusätzlich eingefügten falschen Abhängigkeiten beim Aufstellen des DAG ohne diese Heuristik (siehe 3.3) weisen den Lade- und Speicher-Instruktionen nur eine feste Reihenfolge des Schedulings zu, da die eingefügte Latenz zwischen diesen meist 1 ist. Eine Ausnahme ist, wenn ein Stall zwischen einer Speicher- und einer nachfolgenden Lade-Instruktion erkannt wird, siehe 2.1.4. Die Anzahl der Bündelpartner in Form von IP-Instruktionen bleibt konstant.
Festzuhalten ist, dass diese Heuristik optional zuschaltbar bleiben soll und die Anwendung vom Entwickler je nach Optimierungsziel zu evaluieren ist.

Der Vollständigkeit halber sollen nun in Abbildung 5.7 auch Mittelwerte der WCET-Werte für die möglichen Fälle vorgestellt werden.

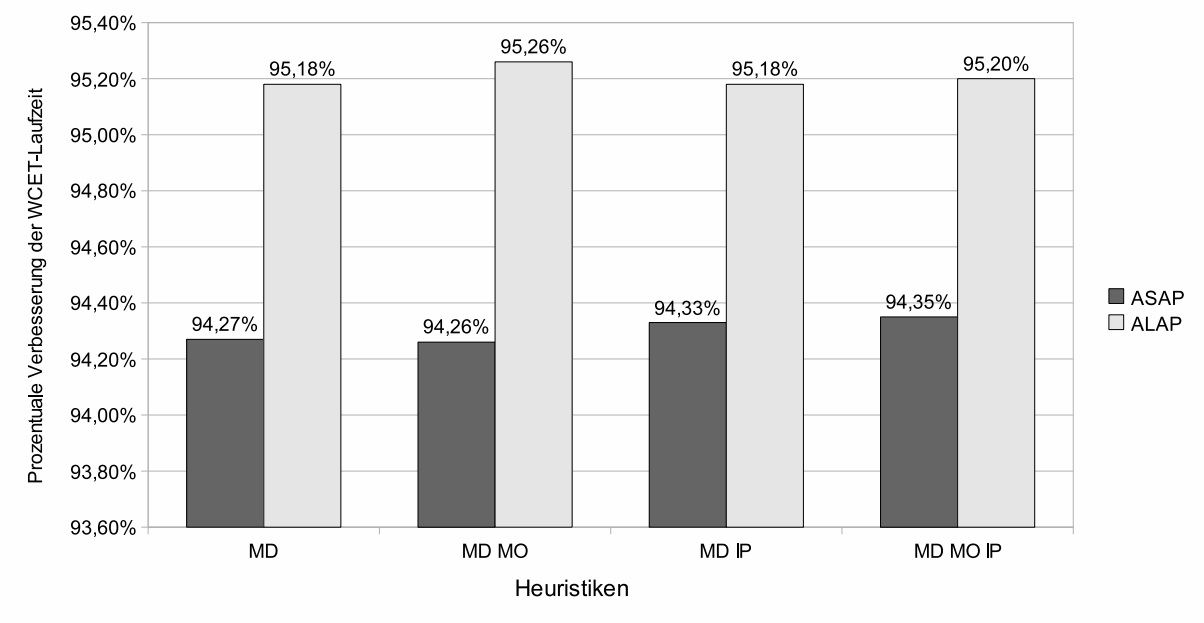

Abbildung 5.7: *Vergleich der WCET-Werte für die möglichen Fälle mit der Maximum Delay-Heuristik bei Optimierungsstufe O2*

Der Vergleich mit Abbildung 5.3 zeigt bis auf kleinere Abweichungen die selben Resultate. Die Fälle in denen die Instruction Priority-Heuristik angewandt wird, zeigen bei Anwendung des ASAP-

Algorithmus dabei eine Verschlechterung der Werte gegenüber den restlichen Fällen auf.

Fazit: Da in den meisten Fällen der ASAP-Algorithmus dem ALAP-Algorithmus überlegen ist, ist dieser der Default-mässige Algorithmus innerhalb des WCCs. Es zeigte sich ausserdem, dass die Kombination aus Maximum Delay- und der Instruction Priority-Heuristik ohne Mobility-Heuristik in zwei aus vier Fällen die besten Ergebnisse liefert, in einem Fall sehr nahe am besten Ergebnis liegt. Diese Kombination soll nach Abschluss der Diplomarbeit die neue Default-Einstellung des Schedulers werden.

5.1.2 Scheduling vor der Registerallokation

Das Scheduling vor der Registerallokation soll exemplarisch für die Optimierungsstufe O2 evaluiert werden. Hier ist, wie schon durch die Diagramme der Maximum Delay-Heuristik beim Scheduling nach der Registerallokation gezeigt wurde, das höchste Potential für den Scheduler gegeben. Durch eine nicht geschickt gewählte hohe Parallelisierung von Instruktionen und damit eine hohe Zahl an gleichzeitig lebendiger Register kann dieses Potential durch viel Spill-Code, welches während der Registerallokation hinzugefügt wird, wieder zunichte gemacht werden. Die nachfolgenden ACET-Werte für das lokale Scheduling wurden dabei für die Default-Optionen des Schedulers bestimmt. Dies wären der ASAP-Algorithmus sowie die Maximum Delay-, Mobility- sowie Instruction Priority-Heuristiken. Es soll evaluiert werden, welche Vorteile ein Scheduling vor der Registerallokation im Vergleich zum Scheduling nur nach der Registerallokation bringt. Zusätzlich soll verglichen werden, in welcher Größenordnung die Verbesserung liegt, wenn das Scheduling vor der Registerallokation mit dem Scheduling nach der Registerallokation kombiniert wird, um eventuell aufgetretenen Spill-Code effizient innerhalb des Codes zu verteilen.

Da innerhalb des WCC Compilers eine Wahl zwischen einem heuristischen Registerallokator, welcher die Default-Einstellung darstellt, und einem ILP-basierten optimalen Registerallokator [32] besteht, soll auch der Einfluss des Registerallokators auf das Scheduling vor der Registerallokation untersucht werden. Die ILP-basierte Registerallokation bietet dabei zusätzlich die Möglichkeit der Verwendung des unteren Kontexts der physikalischen Register (siehe 2.1.2). Es soll untersucht werden, ob diese zusätzlich verfügbaren Register zu einer Minimierung des Spill-Codes und somit eine bessere Ausnutzung der Parallelität ermöglichen. Der ILP-basierte optimale Registerallokator soll nachfolgend auch ILP-RA genannt werden.

Heuristische Registerallokation

Auch hier sollen vor den Diagrammen Abkürzungen eingeführt werden, um die Diagrammfläche nicht unnötig zu verkleinern. **preRA** soll für das Scheduling vor der Registerallokation stehen, bei der die Heuristik der Minimierung der Registerzahl 3.5.1 nicht angewandt wird. **preRA+** soll für das Scheduling vor der Registerallokation mit dieser Heuristik stehen. **postRA** ist das Scheduling nach der Registerallokation. Als Vergleichswert mit 100% wird, äquivalent zu den vorhergehenden Dia-

grammen, der Mittelwert der ACET-Werte aller Benchmarks im ungeschedulten Fall angenommen.

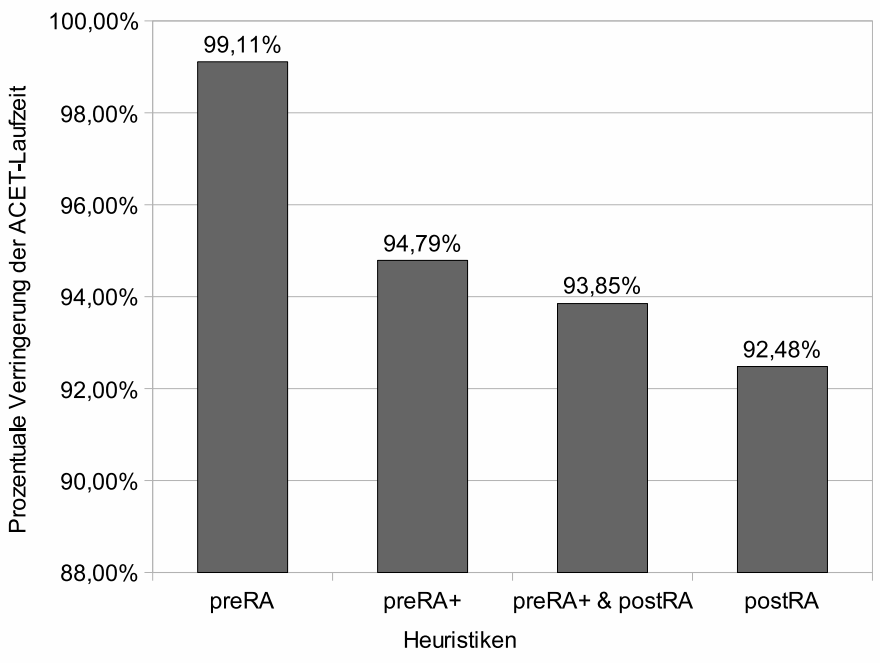

Abbildung 5.8: *Vergleich der ACET-Zeiten mit Augenmerk auf das preRA-Scheduling bei Optimierungsstufe O2*

Abbildung 5.8 zeigt den Vergleich zwischen dem Scheduling vor der Registerallokation ohne und mit Heuristik der Minimierung der Registerzahl, sowie dieses kombiniert mit nachfolgendem postRA-Scheduling. Zum Vergleich ist auch der Mittelwert aller Benchmarks für den Fall des Schedulings nur nach der Registerallokation eingezeichnet. Wie dabei zu sehen ist, ergeben sich kaum Vorteile für das preRA-Scheduling im Vergleich zum ungeschedulten Fall. Hier werden zu viele Spill-Instruktionen eingefügt, welche eine höhere Parallelisierung zunichte machen. Das preRA+-Scheduling, welches die Anzahl der Registerzahl zu minimieren versucht, liefert ein erfreulicheres Bild. Die Verbesserung ist allerdings trotzdem niedriger als beim postRA-Scheduling. Wie zudem zu sehen ist, ermöglicht ein nachfolgendes postRA-Scheduling nach dem preRA-Scheduling eine weitere Optimierung des Codes, indem eingefügte Spill-Code Instruktionen effizient innerhalb des Codes verteilt werden. Diese Übersicht soll nun innerhalb einer Tabelle, welche einen Ausschnitt der Benchmarks und der Werte bietet, verfeinert dargestellt werden.

Die in Tabelle 5.5 zu findenden Benchmarks sind eine Untermenge der MRTC Benchsuite.

Wie an den Werten zu sehen ist, sind die Ergebnisse sehr unterschiedlich und haben eine große Bandbreite. Die Benchmarks *fir* sowie *insertsort* werden durch keines der preRA-Scheduling-Varianten effektiver geschedult als durch das postRA-Scheduling. Dies liegt zum einen daran, dass der Code Selector des WCC generell Berechnungen, wie sie im Quelltext zu finden sind, linear der Reihe nach in Instruktionen abbildet. Somit werden in den meisten Fall nur so viele Register verwendet, wie durch die unmittelbar stattfindende Berechnung benötigt. Durch zu aggressives Scheduling in beiden Benchmarks durch die preRA-Variante wird in *fir* geringfügig mehr Spill-Code erzeugt, als im

Tabelle 5.5: *Vergleich verschiedener Ansätze des Schedulings vor der standardmäßigen Registerallokation*

	unsche-dult	preRA	Unter-scheid	preRA+	Unter-schied	preRA+ postRA	Unter-schied	postRA	Unter-schied
fir	7159	6749	5,73%	7206	-0,66%	7001	2,21%	6727	6,03%
insertsort	1209	2536	-109,76%	1031	14,72%	1032	14,64%	1028	14,97%
jfdctint	5173	5035	2,67%	4432	14,32%	4392	15,10%	5012	3,11%
ludcmp	7889	7899	-0,13%	7751	1,75%	7618	3,44%	7450	5,56%
matmult	454710	311921	31,40%	255918	43,72%	247913	45,48%	435912	4,13%
lms	647021	603684	6,70%	603206	6,77%	607398	6,12%	614985	4,95%
minver	3866	4363	-12,86%	3989	-3,18%	3956	-2,33%	3767	2,56%
select	1871	2078	-11,06%	2057	-9,94%	2048	-9,46%	1804	3,58%

ungeschedulten Fall. Im Falle von *insertsort* führt das aggressive Scheduling zu einer sehr hohen Anzahl an gleichzeitig lebendigen Registern und somit zu sehr viel Spill-Code, welches in einer doppelt so großen ACET im Vergleich zum ungeschedulten Fall resultiert. Hier ist die preRA+-Variante effektiver, indem diese durch Minimierung der gleichzeitig lebendigen Register auch den Spill-Code minimiert. Trotzdem wird durch eine zu hohe Parallelisierung im Vergleich zum postRA-Scheduling ein zu hohes Maß an gleichzeitig lebendigen Registern erzeugt, welches das Ergebnis im Vergleich zum postRA-Scheduling verschlechtert.

Der Benchmark *jfdctint* dagegen zeigt die mögliche Effektivität des Schedulings vor der Registerallokation. Allerdings kann auch hier eine übermäßige Parallelisierung, wie im preRA-Fall zu sehen, zu einem Übermaß an Spill-Code führen. Durch versuchtes Reduzieren der gleichzeitig lebendigen Register ist es der preRA+-Variante möglich, 14,32% Verbesserung in der ACET im Vergleich zum ungeschedulten Fall zu ermöglichen, mit nachgeschaltetem postRA-Scheduling sogar 15,1%.

Ein weiterer Benchmark, der stark von einem Scheduling vor der Registerallokation profitiert, ist der *matmult* Benchmark. Hier ist bis zu 45,48% an Verbesserung der ACET zu verzeichnen. So wurden durch die preRA+-Variante 13 Bündel mehr erzeugt, also 13 Ausführungstakte gespart. Zusätzlich konnten durch die Minimierung der gleichzeitig lebendigen Register 15 Spill-Code-Instruktionen eingespart werden. Dies kann innerhalb einer Schleife zu hohen Gewinnen an Ausführungszeit führen, wie hier zu sehen.

Im Falle des *lms* Benchmarks ist das nachfolgende postRA-Scheduling dagegen von Nachteil. Dies liegt darin begründet, dass innerhalb einer Schleife der gaussian() Funktion dieses Benchmarks die Instruktionsabfolge durch das postRA-Scheduling verändert wird, ohne dabei die Anzahl an Bündeln zu verringern. Durch diese Veränderung der Instruktionsabfolge treten Effekte des Flash-Speichers zu Tage, welche zu zusätzlichen Wartezyklen innerhalb der Speicheranbindung führen. Dies führt zu zusätzlichen Taktzyklen.

Optimale Registerallokation ohne untere-Kontext-Register

Abbildung 5.9 zeigt den Vergleich zwischen dem Scheduling vor der Registerallokation ohne und mit Heuristik der Minimierung der Registerzahl, sowie dieses kombiniert mit nachfolgendem postRA-Scheduling unter Verwendung der ILP-RA. Es wurde als Vergleichswert der Mittelwert des ungeschedulten Falls unter Verwendung der ILP-RA bestimmt.

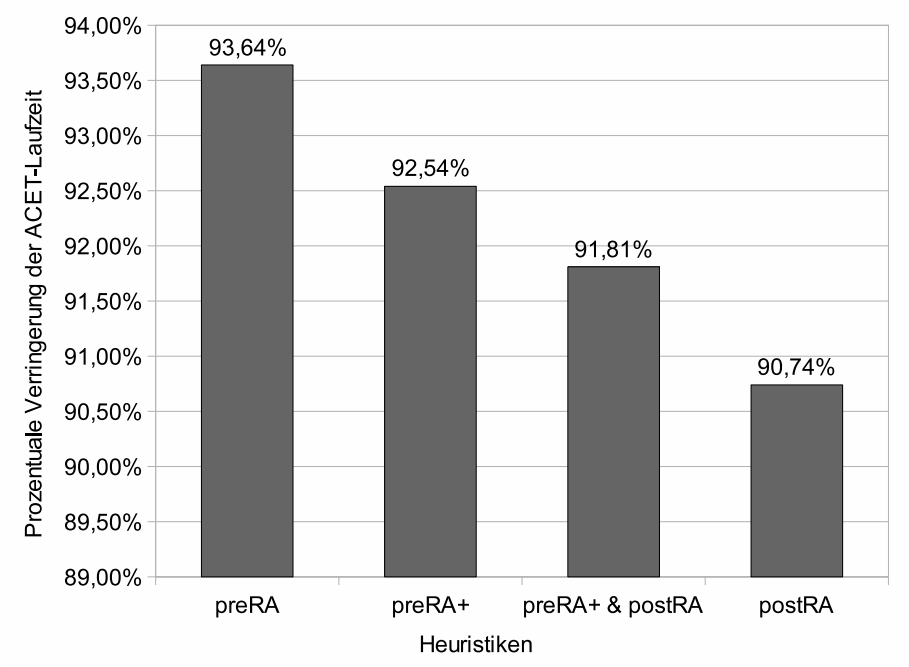

Abbildung 5.9: *Vergleich der ACET-Zeiten mit Augenmerk auf das preRA-Scheduling bei Optimierungsstufe O2 mit ILP-RA*

Wie im Vergleich zu Abbildung 5.8 zu sehen ist, haben sich die Werte für die preRA-Heuristiken deutlich verbessert. Dies liegt darin begründet, dass die ILP-RA eine bessere Ausnutzung der verfügbaren Register unter Minimierung des Spill-Codes erreicht. Trotzdem ist der post-RA Fall im Mittel allen pre-RA Heuristiken überlegen. In Einzelfällen kann die preRA+-Heuristik aber über 20% Gewinn im Vergleich zum postRA-Fall erreichen.

Optimale Registerallokation mit untere-Kontext-Register

Abbildung 5.10 zeigt den Vergleich zwischen dem Scheduling vor der Registerallokation ohne und mit Heuristik der Minimierung der Registerzahl, sowie dieses kombiniert mit nachfolgendem postRA-Scheduling unter Verwendung der ILP-RA. Es wurde als Vergleichswert der Mittelwert des ungeschedulten Falls unter Verwendung der ILP-RA bestimmt.

Im Vergleich mit Abbildung 5.9 ist zu erkennen, dass sich die Werte der preRA-Heuristik deutlich gebessert haben. Dies liegt daran, dass durch die höhere Anzahl an Registern durch Verwendung des unteren Kontextes, die der Registerallokation zur Verfügung stehen, eine höhere Parallelisie-

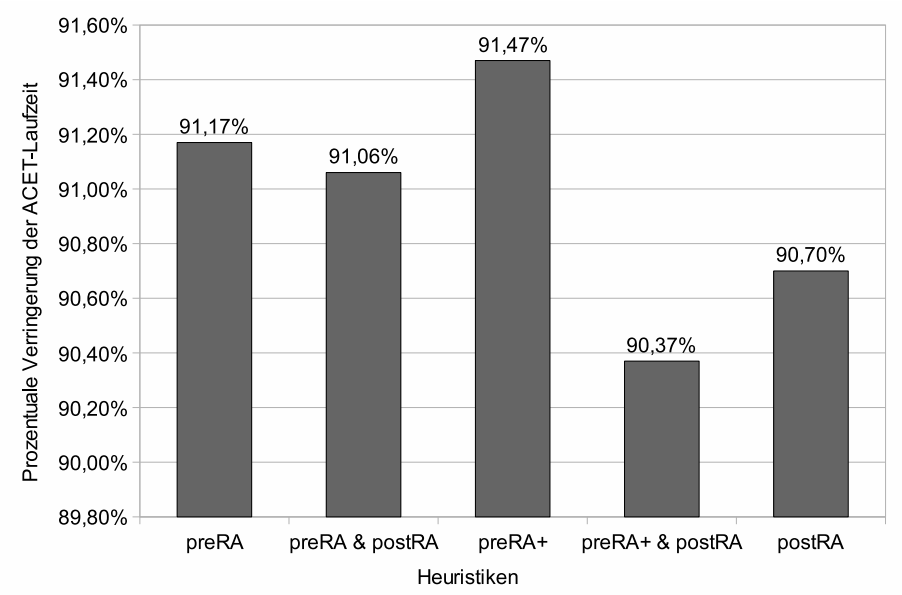

Abbildung 5.10: *Vergleich der ACET-Zeiten mit Augenmerk auf das preRA-Scheduling bei Optimierungsstufe O2 mit ILP-RA und unterem Kontext*

rung durch die preRA-Heuristik mit weniger Spill-Code einhergeht. Da hier die preRA-Heuristik zum ersten Mal der preRA+-Heuristik überlegen ist, wurden auch Werte der preRA-Heuristik mit nachgeschaltetem postRA-Scheduling bestimmt. Im Vergleich zu dem Ergebnis der preRA+-Heuristik mit nachgeschaltetem postRA-Scheduling zeigt sich aber, dass hier kein großer Gewinn erzielbar ist. Dies liegt daran, dass durch die aggressive Bündelung der preRA-Heuristik kaum Bündelpartner für den erzeugten Spill-Code innerhalb der RA vorhanden sind, welche mittels postRA-Scheduling bündelbar wären. Wie zu sehen ist, ist der postRA+-Heuristik durch eine weniger aggressive Bündelung möglich weniger Spill-Code zu erzeugen um beim nachgeschaltetem postRA-Scheduling einen größeren Gewinn durch Bündelung zu erzielen. In diesem Fall ist das Ergebnis sogar besser als das des postRA-Schedulings alleine.

Fazit: Durch die hohen Schwankungen der bestimmten Werte sollte ein Scheduling vor der Registerallokation im Einzelfall evaluiert werden. Hierbei sollte ein Vergleich zwischen jeder möglichen Kombination gemacht werden um zu bestimmen, welche Kombination den meisten Gewinn für den zu scheduleden Code bringt.

5.1.3 optimales Scheduling

Um die Optimalität des lokalen Schedulers zu bestimmen, wurde ein optimaler Scheduler für den TriCore Prozessor implementiert. Dieser wurde als eine zweite Schedule-Phase dem List Scheduling nachgeschaltet. Tabelle 5.6 zeigt die gesammelten Ergebnisse eines Schedule Durchlaufs für die ausgewählten Benchmarks. Dabei wurde jeder Benchmarks für jede Optimierungsstufe einzeln übersetzt, d.h. jeder Benchmark geht viermal in diese Statistik ein.

Tabelle 5.6: *Statistik des optimalen Schedulers bei Default-Einstellungen*

Anzahl Basisblöcke	optimale Basisblöcke	Basisblöcke an ILP übergeben	Basisblöcke optimiert	Gesparte Zyklen	Zeitüber-schreitungen	suboptimale Lösungen
18416	18279	2276	137	147	21	0

Wie aus der Tabelle zu erkennen ist, waren von betrachteten 18416 Basisblöcken 18279 optimal durch den List Scheduler geschedult worden. 2276 Basisblöcke wiesen eine berechnete untere Ausführungsschranke auf, die niedriger war als die durch den List Scheduler für diesen Basisblock bestimmte obere Ausführungschranke. Diese Basisblöckes wurden an den optimalen Scheduler übergeben, von diesen konnten aber nur 137 optimiert werden. Die Mehrheit der 2276 Basisblöcke konnte wegen Instruktionsabhängigkeiten, welche zusätzlich einschränkend auf die Parallelisierbarkeit des Codes auswirken, nicht weiter optimiert werden. Innerhalb der 137 optimierten Basisblöcke wurden dabei 147 Ausführungszyklen (ohne Berücksichtigung evtl. Schleifeniterationen) eingespart.

Es wurde 21 Mal die Zeitschranke von 30 CPU-Minuten überschritten, innerhalb der der Solver keine bessere Lösung finden konnte (wie in 3.7.1 beschrieben). In diesen 21 Fällen wurde der Versuch eine niedrigere Schranke für den betreffenden Basisblock zu finden abgebrochen. Innerhalb der gesamten Benchsuite sind keine suboptimalen Lösungen durch die Solver aufgetreten. Dies bedeutet, jede innerhalb der Zeitschranke gefundene Lösung war für die gegebene Ausführungsschranke optimal.

Somit wurden innerhalb der ausgewählten Benchmarks nur 0,74% der Basisblöcke nicht optimal im Sinne der Bündelung und Verhinderung von Stall- oder Delay-Zyklen durch den List Scheduler geschedult. Dies ist darin begründet, dass das List Scheduling-Verfahren ein Online-Algorithmus ist, welcher mangels Wissen über die zukünftigen Möglichkeiten greedy eine Entscheidung auf Basis der eingesetzten Prioritätsheuristiken trifft. Hierdurch kann nicht in jedem Fall vorausgesehen werden, dass eine eventuelle Verzögerung des Schedulings einer Instruktion in der Zukunft einen Delay-Slot füllen kann, was zu einem gesparten Ausführungszyklus führen wird. Es zeigt sich aber, dass dies nur in sehr wenigen Fällen auftritt. Der Wert von 0,74% suboptimalen Basisblöcken bestätigt dabei den von Wilken et. al in [15] bestimmten Wert von 0,3% suboptimalen Basisblöcken für einen List Scheduler. Somit ist der für den TriCore Prozessor entwickelte Scheduler nahe am möglichen Optimum bezüglich des lokalen Schedulings.

Von Interesse ist außerdem die zusätzlich durch den optimalen Scheduler verursachte Laufzeit. So benötigte der List Scheduler für einen Durchlauf aller 69 Benchmarks bei Bestimmung der ACET-Werte jeder Optimierungsstufe pro Benchmark 7 Stunden, 11 Minuten und 5 Sekunden. Ein Durchlauf mit zugeschaltetem optimalen Scheduling benötigt dagegen 23 Stunden, 3 Minuten und 5 Sekunden. Dies ergibt einen zusätzlichen Overhead von 15 Stunden und 52 Minuten. Das bedeutet eine Erhöhung der Laufzeit um den Faktor 3,21. Hier gehen vor allem sehr große Basisblöcke ein, die über 1000 Instruktionen aufweisen sowie eine hohe Anzahl an Abhängigkeiten. Das Lösen der

ILP-Modelle dieser Basisblöcke benötigt trotz der Vereinfachungen des ILP-Modells (siehe 3.7.1) sehr viel Rechenzeit.

Fazit: Wegen der nicht unwesentliche Verlängerung der Laufzeit des Schedulers durch den Einsatz des optimalen Schedulings und des im Vergleich dazu minimalen Gewinns an gesparten Ausführungszyklen innerhalb des geschedulten Codes ist der Einsatz des optimalen Schedulers im produktiven Umfeld nur bei Code mit sehr großen Basisblöcken von Interesse. Bei Code mit Basisblockgrößen unter 1000 Instruktionen ergibt die Anwendung des optimalen Schedulers wegen der guten Werte des List Schedulings keinen Vorteil.

5.2 Globales Scheduling

Auch die Werte des Treegion Schedulings wurden unter Verwendung des ASAP-Algorithmus und der Verwendung der Prioritätsheuristiken Mobility sowie Instruction Priority bestimmt. Anstatt der Maximum Delay-Heuristik wurden die ans Treegion Scheduling angepassten Heuristiken Exitcount sowie Dependence Height verwendet. Die bestimmten Werte sollen nun für die Optimierungsstufe O2 vorgestellt werden. Da das Treegion Scheduling nicht auf Scheduling nach der Registerallokation limitiert ist, wurden ACET-Werte für beide Fälle bestimmt.

5.2.1 Scheduling nach der Registerallokation

Abbildung 5.11 zeigt die Mittelwerte für die Exitcount- sowie Dependence Height-Heuristik des Treegion Schedulings sowie zum Vergleich den Mittelwert, welcher durch das lokale Scheduling unter Verwendung der Maximum Delay-Heuristik sowie der weiteren hier verwendeten Heuristiken ermöglicht wurde. Der helle Balken innerhalb des Diagramms soll dabei für das lokale Scheduling stehen, die beiden dunklen Balken für das Treegion Scheduling.

Wie in der Abbildung zu sehen, ist das Treegion Scheduling für beide Heuristiken dem lokalen Scheduling unterlegen. Da das Treegion Scheduling gewählt wurde, um die Bündelung von Instruktionen über Basisblockgrenzen hinaus zu maximieren, soll in Tabelle 5.7 ein Vergleich zwischen dem lokalen Scheduling und den beiden Heuristiken des Treegion Schedulings bezüglich der Bündelung gemacht werden. Diese Werte spiegeln die Summe aller Bündel dar innerhalb der 69 ausgewählten Benchmarks für alle vier Optimierungsstufen des WCCs.

Dabei ist zu sehen, dass das lokale Scheduling es ermöglicht, die Anzahl der Bündel um den Faktor 1,62 zu steigern. Das Treegion Scheduling unter Verwendung der Exitcount-Heuristik ermöglicht es weitere 651 Bündel zu bilden. Die meisten dieser Bündel werden innerhalb größerer Benchmarks ermöglicht, die Anzahl an Bündeln innerhalb vieler kleinerer Benchmarks bleibt dagegen konstant. Dies liegt darin begründet, dass zum einen Instruktionen benötigt werden, wie über Sprunggrenzen hinaus verschiebbar sind, zum anderen sind nicht alle dieser Instruktionen gleichzeitig in den

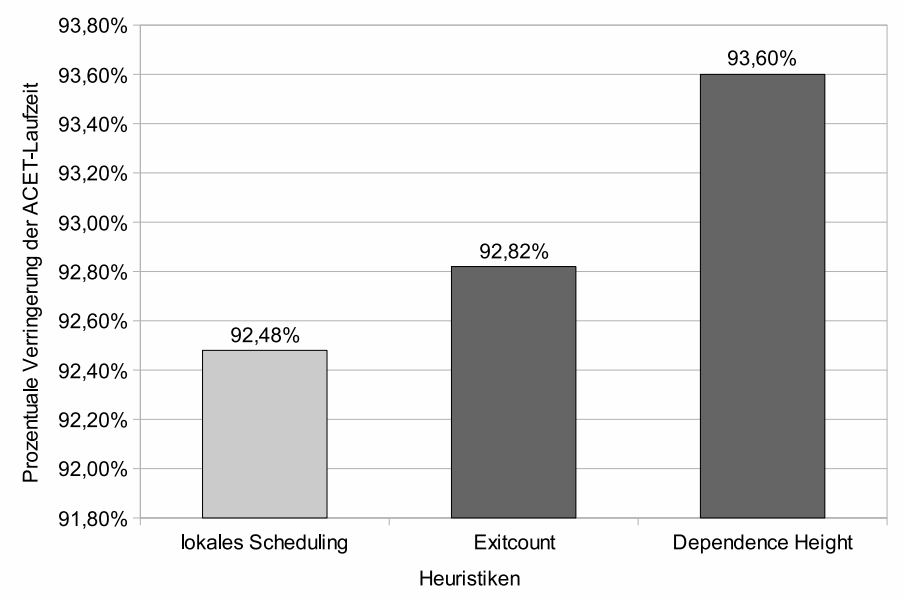

Abbildung 5.11: *Vergleich der ACET-Zeiten des Treegion Schedulings für Optimierungsstufe O2*

Tabelle 5.7: *Vergleich der Anzahl an Instruktionsbündeln zwischen den unterschiedlichen Scheduling-Verfahren.*

# Bündel vor Scheduling	# Bündel nach List Scheduling	# Bündel nach Treegion Scheduling	
		Exitcount	Dependence Height
16586	27013	27664	27288

Vorgänger-Basisblock verschiebbar, da diese die gleichen Register definieren. Hier würde Register Renaming, wie in der orignalen Implementation des Treegion Schedulings ein mehr an Potential erschliessen. Zudem muss für diese spekulativ ausführbaren Instruktionen ein Bündelpartner innerhalb des vorhergehenden Basisblocks gefunden werden. Die Exitcount-Heuristik konnte aber in 262 von 276 Fällen die Anzahl der Bündel erhalten und innerhalb dieser Fälle vielfach sogar steigern. Es zeigt sich also, dass die Verschlechterung des Mittelwerts der ACET-Werte nicht auf das Konzept des Treegion Schedulings zurückzuführen ist.

Die Dependence Height-Heuristik ermöglicht im Vergleich dazu viel weniger Bündel zu bilden. Dies liegt darin begründet, dass diese Heuristik Maximum Delay-Werte jeder Instruktion basierend auf der Pfadlänge von dieser Instruktion hin zur Senke des Basisblocks bestimmt, in welchem sich diese Instruktion befindet. Somit sind Instruktionen aus Basisblöcken niedriger innerhalb des CFG gleichberechtigt wie Instruktionen aus Basisblöcken, welche höher innerhalb des CFG liegen. Dadurch kann es passieren, dass ein Basisblock, welcher niedriger liegt, zuerst geschedult wird und diese Instruktionen aus der Ready-List entfernt werden. Sodann wird ein höherer Basisblock geschedult und die Instruktionen aus dem geschedulten niederen Basisblock sind nicht mehr als Bündelpartner verfügbar. Dies verringert die Möglichkeiten der Bündelung.

Es sollen nun weitere Gründe aufgezeigt werden, die die Resultate des Treegion Schedulings schmälern. Tabelle 5.8 zeigt zum Vergleich verschiedene Werte des *lms* Benchmarks aus der DSPstone_floating_point Benchsuite bei Verwendung der Exitcount-Heuristik beim Treegion Scheduling.

Tabelle 5.8: *Statistiken für Benchmark lms für Exitcount-Heuristik des Treegion Schedulings.*

	Lokales Scheduling	Treegion Scheduling
IP Instruktionen	15	15
LS Instruktionen	115	115
spekulative Instruktionen	0	9
Bündel vor Scheduling	0	0
Bündel nach Scheduling	0	9
ACET vor Scheduling	1160	1160
ACET nach Scheduling	1015	1106
Veränderung	12,50%	4,66%

Wie aus der Tabelle zu erkennen ist, wurden sowohl beim lokalen als auch beim Treegion Scheduling die gleiche Anzahl an Bündeln gebildet. Dies liegt vor allem an der Einschränkung, dass nur 15 IP-Instruktionen insgesamt innerhalb des Benchmarks existieren. Es wurden im Falle des Treegion Schedulings 9 spekulative Instruktionen ermittelt. Dies sind Instruktionen, die spekulativ über Sprung-Instruktionen hinaus verschoben werden können (siehe 4.4). Dies wurde in einem Fall ausgenutzt, was aus dieser Tabelle nicht zu erkennen ist. So wurde eine IP-Instruktion aus dem _L5 Basisblock hin zum _main Basisblock spekulativ verschoben. Dadurch konnte in _main ein Bündel erzeugt werden, aus Mangel an weiteren IP-Instruktionen innerhalb des _L5 Basisblocks wurde hier aber kein Bündel erzeugt. Die aus der Tabelle zu erkennende Verschlechterung der ACET-Zeit im Falle des Treegion Schedulings ist allerdings nicht durch dieses Verschieben verursacht. Der Grund dafür ist stattdessen, dass innerhalb einer Schleife der main() Funktion Instruktionen in einer anderen Abfolge geschedult wurden, da sich durch das Verschmelzen der Basisblöcke dieser Schleife zu einer Treegion die Prioritäten dieser verändert haben. So ist die Exitcount-Heuristik eine modifizierte Maximum-Delay Heuristik (siehe 4.5.2), welche die Pfadlänge einer Instruktion zur Senke der Treegion bestimmt. Dies kann, bedingt durch Abhängigkeiten innerhalb nachfolgender Basisblöcke innerhalb einer Treegion, den Pfad einer Instruktion innerhalb eines darüberliegenden Basisblocks verändern. Im vorliegenden Beispiel resultiert diese Umstellung der Instruktionsreihenfolge in einem Stall bedingt durch Effekte des Flash-Speichers, wodurch die Schleife einen Taktzyklus mehr zur Ausführung pro Iteration benötigt. Dies resultiert in der Veränderung der gesamten ACET.

Fazit: Das Treegion Scheduling in seiner derzeitigen Form ist zum einen limitiert durch die physikalischen Register, welche ein Verschieben weiterer Instruktionen über Sprunggrenzen hinaus durch falsche Abhängigkeiten (siehe 3.3) verhindern. Dies ermöglicht nur eine geringe Anzahl an zusätzlichen Bündeln. Zum anderen müssen die Heuristiken dahingehend verbessert werden, dass Effekte des Speichersubsystems minimiert werden, welche die minimalen Gewinne durch die zusätzlichen Bündel wieder zunichte machen.

5.2.2 Scheduling vor der Registerallokation

Treegion Scheduling vor der Registerallokation weist, verglichen mit lokalem Scheduling vor der Registerallokation eine niedrigere Verbesserung auf. So liegt die prozentuale Verringerung der ACET des Treegion Schedulings mit Exitcount-Heuristik kombiniert mit der Heuristik zur Verringerung der Registerzahl bei nur 97,79% verglichen mit dem ungeschedulten Fall. Das postRA-Scheduling mit Default-Heuristiken liegt bei 92,48%. Dies liegt an Benchmarks, welche durch die zusätzliche Parallelisierung, ermöglicht durch das Treegion Scheduling, eine hohe Anzahl an Spill-Code erfahren. Dadurch verschlechtert sich der Mittelwert über alle Benchmarks.

Zwei große Benchmarks zeigen aber sehr vielversprechende Zahlen, welche durch Register Renaming auch beim Treegion Scheduling nach der Registerallokation möglich wären. Diese sind in Tabelle 5.9 ersichtlich.

Tabelle 5.9: *ACET ausgewählter Benchmarks mit Gewinn durch fehlende Abhängigkeiten beim Treegion Scheduling*

Benchmark	unge schedult	postRA	Gewinn	preRA+	Gewinn	preRA+ Treegion Scheduling	Gewinn
gsm_encode	13450704	13117228	2,48%	11347692	15,63%	9499821	29,37%
dijkstra	67251029	65631663	2,41%	64433467	4,19%	62954826	6,39%

Wie zu sehen ist, steckt großes Potential innerhalb des Treegion Schedulings, wenn es gelingt falsche Abhängigkeiten zu eliminieren und somit mehr Instruktionen spekulativ über Basisblockgrenzen hinaus zu verschieben.

Fazit: Das Treegion Scheduling hat hohes Potential, welches aber erst beim Auflösen falscher Abhängigkeiten zu Tage tritt. Dies sollte, wie schon erwähnt, beim Scheduling nach der Registerallokation mittels Register Renaming erfolgen. Beim Scheduling vor der Registerallokation ist eine verbesserte Heuristik zur Minimierung der gleichzeitig lebendigen Register vonnöten.

Kapitel 6

Zusammenfassung und Ausblick

Das Ziel der Diplomarbeit war die Evaluation von Scheduling-Verfahren für den Infineon TriCore Prozessor. Dieses Kapitel soll zusammenfassend erläutern, wie dieses Ziel erreicht wurde und welche Beobachtungen während der Evaluation gemacht werden konnten.

Einleitend wurde in Kapitel 1 eine Motivation für Scheduling im Allgemeinen gegeben. Kapitel 2 führte dann in die Architektur des TriCore Prozessors ein und stellte die Möglichkeiten und Eigenheiten dieser Prozessorarchitektur vor. Nachfolgend wurde der Compiler WCC, in den der entwickelte Scheduler implementiert wurde, vorgestellt. Dazu wurden verschiedene Internas beleuchtet und aufgezeigt, an welchen Stellen der Scheduler ansetzen würde. Zudem wurden Entwicklungstools vorgestellt, die für die Implementierung und Evaluation des Schedulers wichtig waren.

In Kapitel 3 wurden dann die Grundlagen des lokalen Schedulings vorgestellt. Dies umfasste ein Vorstellen der benötigten Datenstrukturen sowie die Erläuterung der Funktionsweise des List Scheduling-Verfahrens. Es wurden dabei verschiedene Heuristiken vorgestellt, die sowohl generischer als auch Prozessorspezifischer Natur waren. Danach wurde auf das optimale Scheduling im generellen und die benötigten Änderungen für den TriCore Prozessor im speziellen eingegangen. Anschließend wurde eine Möglichkeit vorgestellt, die Komplexität des resultierende ILP-Modells zu minimieren. Zuletzt wurde der konzeptionelle Aufbau des Schedulers vorgestellt.

Kapitel 4 führte dann in die Thematik des globalen Schedulings ein, indem zuerst auf verschiedene verwandte Arbeiten Bezug genommen wurde. Dann wurde erläutert, wie Treegion-Scheduling in seiner Originalform arbeitet und wie dieses Verfahren an den TriCore Prozessor adaptiert wurde. Dazu wurden die Abweichungen zum lokalen Scheduling vorgestellt und Methoden aufgezeigt, welche die Korrektheit des implementierten Treegion-Schedulings innerhalb des vorgestellten Schedulers und der LLIR gewährleisten.

Abschliessend wurden diese Verfahren in Kapitel 5 evaluiert, indem Vergleiche dieser durch gewählte Benchmarks zum ungeschedulten Fall gemacht wurden. Es hat sich dabei gezeigt, dass das lokale Scheduling sehr gute Werte liefert, vor allem mit der Maximum Delay- sowie Instruction Priority-

Heuristik. Dabei konnte bei der WCC-Optimierungsstufe O2 eine maximale Verringerung der ACET-Zeit von 92,5% gemittelt über alle Benchmarks erreicht werden. Zudem konnte gezeigt werden, dass der entwickelte List Scheduler 99,26% der Basisblöcke innerhalb der Benchsuite optimal schedulte. Weiterhin konnte gezeigt werden, welches Potential im Scheduling vor der Registerallokation steckt. Dieses konnte durch eine Heuristik zur Minimierung der gleichzeitig lebendigen Register noch gesteigert werden.

Leider erwies sich das Treegion Scheduling zu sehr eingeschränkt durch falsche Abhängigkeiten innerhalb des lokalen Schedulings. Zudem traten weitere Effekte des Speichersubsystems auf, welche das Resultat schmälerten, welche nicht durch die wenigen zusätzlich erzeugbaren Instruktionsbündel kompensierbar waren. Es konnte aber für zwei Benchmarks und das Treegion Scheduling vor der Registerallokation gezeigt werden, welches Potential in diesem Verfahren steckt, wenn falsche Abhängigkeiten nicht vorhanden sind.

6.1 Ausblick

Durch die gute Modularität des innerhalb dieser Diplomarbeit entwickelten Schedulers ist eine Vielzahl an Erweiterungen denkbar. So ist es durch Verwendung und Erweiterung der *LLIR_Region*-Klasse möglich, die in Kapitel 4 erwähnten globalen Scheduling-Verfahren wie Trace- und Superblock-Scheduling zu implementieren. Es wäre hier zusätzlich eine Schnittstelle zu einem Profiler notwendig, um den am meisten ausgeführten Pfad zu bestimmen. Zusätzlich wären auch Änderungen an der *LLIR_PriorityFinder*-Klasse nötig, wie dies schon im Falle des Treegion-Schedulings notwendig war. Durch diese beiden Scheduling-Verfahren sollte sich ein nicht unerheblicher Optimierungsvorteil einstellen, da diese Basisblöcke auf einem vom Profiler bestimmten Pfad als einen einzelnen Basisblock betrachten und schedulen. Den Einfluss der Basisblockgröße auf die Möglichkeit der Optimierung konnte dabei durch Tabelle 5.1 gezeigt werden. So konnte zwischen Optimierungsstufe O1 und O2 trotz vergleichbarer Anzahl an Instruktionen eine nicht unerhebliche Vergrößerung des Optimierungspotentials erreicht werden, indem in O2 Basisblöcke verschmolzen wurden. Zu Evaluieren wäre dabei, welchen Einfluss der für die Korrektheit benötigte Kompensationscode (siehe 4.1) auf die ACET des geschedulten Codes hat. Hyperblock-Scheduling ist leider mangels bedingter Instruktionen auf dem TriCore Prozessor nicht implementierbar.

Als weitere Heuristik sowohl für das lokale als auch globale Scheduling wäre eine Low Power-Heuristik für den Infineon TriCore denkbar. Eine in [33] durch Su et al. vorgestellte Low Power-Heuristik wählt, basierend auf der zuletzt geschedulten Instruktion und einer Kostentabelle, welche Instruktion im nächsten Ausführungszyklus geschedult werden soll, die am meisten Potenzial zur Energieeinsparung aufweist. Diese Heuristik wäre ohne größere Anpassungsarbeiten im TriCore Scheduler implementierbar.

Zudem wäre die Heuristik der Registerzahl-Minimierung für den Scheduler vor der Registerallokation dahingehend erweiterbar, dass diese auf weitere Parallelisierung verzichtet, wenn eine fest-

gelegte Anzahl an Daten- oder Adressregistern gleichzeitig lebendig ist. Dies wäre durch die in der LLIR implementierte *LifeTime*-Analyse möglich. Dadurch könnte festgestellt werden, ob eine im Ausführungszyklus *n* ausführbare Instruktion zu viele gleichzeitige Register lebendig werden lässt und diese dann niedriger prioritisieren. Die festgelegte Grenze an gleichzeitig lebendigen Registern sollte sich dabei an der Anzahl an durch die Registerallokation verwendbaren Registern orientieren. Dies soll dazu dienen, dass so wenig Spill-Code wie möglich erzeugt wird. Es sollte dabei aber eine so hoch wie mögliche Parallellität ermöglicht werden. Es wäre dabei zu evaluieren, ob hier auch ein Zusammenstellen aller Ketten von Instruktionen nötig ist und eine Klassifikation dieser Ketten in Parallelisierungsgrad und Anzahl gleichzeitig lebendiger Register vonnöten ist.

Innerhalb des Schedulers im allgemeinen und für das Treegion Scheduling im Speziellen wäre zudem die Implementierung von Register Renaming [34] interessant. Dadurch würden sich mehr Möglichkeiten innerhalb des Schedulings nach der Registerallokation aufzeigen, da sich durch Register Renaming falsche Abhängigkeiten auflösen lassen. Somit könnte ein höheres Maß an Parallelisierung erreicht werden.

Zusätzlich wäre eine Heuristik für das Speicher-Subsystem interessant. Diese sollte zum einen die Stall-Zyklen, verursacht durch den Flash-Speicher minimieren, zum anderen sollte diese auch durch Linecrossings (siehe 2.1.4) verursachte Stall-Zyklen, so weit wie möglich, verhindern.

Anhang A

Benchmarks

Folgende Benchmarks wurden aus den in Kapitel 5 genannten Benchsuiten ausgewählt.

Tabelle A.1: *Innerhalb der Diplomarbeit verwendete Benchmarks*

DSPStone (fixed point)	DSPStone (floating point)	MRTC	UTDSP
adpcm_g721_board_test	complex_update	adpcm_encoder	adpcm
complex_multiply	convolution	binarysearch	compress
convolution	dot_product	bsort100	fft_256
dot_product	fir	compressdata	fir_32_1
fft_16_13	iir_biquad_N_sections	countnegative	g721.marcuslee_decoder
fft_1024_13	lms	crc	histogram
fir	matrix1	duff	iir_1_1
iir_biquad_N_sections	n_real_updates	edn	jpeg
lms	real_update	expint	lmsfir_8_1
matrix1		fdct	lpc
n_real_updates		fir	mult_4_4
real_update		insertsort	qmf_receive
startup		jfdctint	v32.modem_achop
		lms	v32.modem_eglue
		ludcmp	
		matmult	
		minver	
		ndes	
		select	

MediaBench	misc	NetBench	StreamIt
cjpeg_jpeg6b_wrbmp	codecs_codrle1	md5	bitonic
gsm_encode	g721_encode		
h264dec_ldecode_macroblock	g723_encode		
MiBench	h263		
dijkstra	hamming_window		
rijndael_encoder	searchmultiarray		
sha			

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Peter Marwedel. *Eingebettete Systeme*. Springer, 2007.

[2] Infineon. *TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1*. Infineon, 2008. Available from: http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b8abb0882.

[3] Infineon. *TC1796 Data Sheet V1.0*. Infineon, 2008. Available from: http://www.infineon.com/cms/de/product/channel.html?channel=ff80808112ab681d0112ab6b6aaf0819&tab=2.

[4] Infineon. *TriCore® 1 DSP Optimization Guide Part 1: Instruction Set*. Infineon, 2003. Available from: http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b8abb0882.

[5] Infineon. *TriCore® Pipeline Behaviour & Instruction Execution Timing (AP32071)*. Infineon, 2000. Available from: http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b8abb0882.

[6] Infineon. *TriCore® Compiler Writer's Guide*. Infineon, 2002. Available from: http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b8abb0882.

[7] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a WCET-Aware C Compiler. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, Seoul/Korea, October 2006.

[8] Informatik Centrum Dortmund. *ICD-C Compiler framework Developer Manual - Confidential*. Informatik Centrum Dortmund, 2007.

[9] Informatik Centrum Dortmund. *ICD Low Level Intermediate Representation backend infrastructure (LLIR) Developer Manual - Confidential*. Informatik Centrum Dortmund, 2007.

[10] aiT Worst-Case Execution Time Analyzers. Website, 2009. http://www.absint.com/ait/; visited on May 1st 2009.

[11] Reinhold Heckmann and Christian Ferdinand. *Worst-Case Execution Time Prediction by Static Program Analysis*. AbsInt, 2006. Available from: http://www.absint.com/aiT_WCET.pdf.

[12] VaST Systems. *CoMET System Engineering Tool*. VaST Systems, 2008. Available from: http://www.vastsystems.com/docs/CoMET_dec2008.pdf.

[13] John L. Hennessy and Thomas Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, 1983. doi:http://doi.acm.org/10.1145/2166.357217.

[14] Steven S. Muchnick and Phillip B. Gibbons. Efficient Instruction Scheduling for a Pipelined Architecture. *SIGPLAN Not.*, 39(4):167–174, 2004. doi:http://doi.acm.org/10.1145/989393.989413.

[15] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. *SIGPLAN Not.*, 35(5):121–133, 2000. doi:http://doi.acm.org/10.1145/358438.349318.

[16] ILOG CPLEX. Website, 2009. Available online at http://www.ilog.de/products/cplex/; visited on May 1st 2009.

[17] Mixed Integer Linear Programming (MILP) solver lp_solve. Website, 2009. Available online at http://sourceforge.net/projects/lpsolve; visited on May 1st 2009.

[18] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal Instruciton Scheduling Using Integer Programming. Technical Report ACM 1-58113-199-2/00/0006, Department of Electrical and Computer Engineering, University of California, 2000.

[19] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Trans. Program. Lang. Syst.*, 11(1):57–66, 1989. doi:http://doi.acm.org/10.1145/59287.59291.

[20] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981. doi:http://doi.ieeecomputersociety.org/10.1109/TC.1981.1675827.

[21] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *J. Supercomput.*, 7(1-2):229–248, 1993. doi:http://dx.doi.org/10.1007/BF01205185.

[22] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *In Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, 1992.

[23] Rajiv Gupta and Mary Lou Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Trans. Softw. Eng.*, 16(4):421–431, 1990. doi:http://dx.doi.org/10.1109/32.54294.

[24] William A. Havanki. Treegion Scheduling for VLIW Processors. Master's thesis, North Carolina State University, July 1997.

[25] William A. Havanki, Sanjeev Banerjia, and Thomas M. Conte. Treegion Scheduling for Wide Issue Processors. Technical report, Department of Electrical and Computer Engineering, North Carolina State University, February 1998.

[26] Vojin živojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994.

[27] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative Embedded Benchmark Suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. doi:http://dx.doi.org/10.1109/WWC.2001.15.

[29] Mälardalen Real-Time Research Centre: WCET project / Benchmarks. Website, 2008. Available online at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html; visited on May 1st 2009.

[30] Processors Gokhan Memik, B. Mangione-smith, W. Hu, Gokhan Memik, Gokhan Memik, William H. Mangione-smith, and Wendong Hu. NetBench: A Benchmarking Suite for Network. In *In ICCAD*, pages 39–42, 2001.

[31] UTDSP Benchmark Suite. Website, 1992. Available online at http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html; visited on May 1st 2009.

[32] Florian Schmoll. ILP-basierte Registerallokation unter Ausnutzung von WCET-Daten. Master's thesis, Technische Universit¨at Dortmund, september 2008.

[33] Ching-Long Su, Chi-Ying Tsui, and Alvin M. Despain. Saving Power in the Control Path of Embedded Processors. *IEEE Design and Test of Computers*, 11(4):24–30, 1994. doi:http://doi.ieeecomputersociety.org/10.1109/54.329448.

[34] R. Cytron and J. Ferrante. What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proc. Int'l Conf. Parallel Processing*, pages 19–27, 1987.