TECHNISCHE UNIVERSITÄT
D O R T M U N D
FAKULTÄT FÜR INFORMATIK

Florian Schmoll

ILP-basierte Registerallokation unter Ausnutzung von WCET-Daten

Diplomarbeit

29. September 2008

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl 12 (Eingebettete Systeme)
Fakultät für Informatik
Technische Universität Dortmund

Gutachter:
Prof. Dr. Peter Marwedel
Dr. Heiko Falk

GERMANY · D-44221 DORTMUND

Vorwort

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

Daher möchte ich den Mitgliedern der Forschungsgruppe für eingebettete Systeme am Lehrstuhl 12 der Fakultät für Informatik der Technischen Universität Dortmund danken. In unzähligen Arbeitsstunden haben sie den WCET-aware C-Compiler entwickelt, und so diese Diplomarbeit erst möglich gemacht.

Ganz besonders will ich meinem Betreuer Dr. Heiko Falk danken. Durch seinen außerordentlichen Einsatz und seine stetige Hilfsbereitschaft hat er mich immer wieder motiviert und in hohem Maße bei der Erstellung dieser Diplomarbeit unterstützt.

Schließlich will ich meiner Familie für ihre Unterstützung und ihren Rückhalt danken.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Diese Diplomarbeit behandelt ein Thema aus dem Bereich der eingebetteten Echtzeitsysteme. „Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres System integriert sind und die normalerweise nicht direkt vom Benutzer wahrgenommen werden.“ [Mar07, Seite 1]. Oftmals ist dem Nutzer eines Produkts deshalb gar nicht bewusst, dass es ein oder mehrere eingebettete Systeme enthält. Sie werden kaum wahrgenommen, obwohl sie weit verbreitet sind. Weltweit übersteigt die Anzahl der eingebetteten Systeme die Anzahl der PC-Systeme um ein Vielfaches. Mobiltelefone, Fernseher, Stereo-Anlagen, Kaffeeautomaten, Waschmaschinen - ein Großteil der strombetriebenen Geräte enthält eingebettete Systeme. Auch moderne Autos, Busse, Bahnen und Flugzeuge enthalten eingebettete Systeme und sogar gleich mehrere auf einmal. Die primäre Funktion dieser Produkte ist nicht die Informationsverarbeitung, die das eingebettete System durchführt. Die Funktionsweise basiert aber so stark auf der Informationsverarbeitung, dass diese unverzichtbar wird.

Da ein eingebettetes System ein wichtiger Teil des umgebenden Produkts ist, muss es bestimmte Anforderungen erfüllen. Ein eingebettetes System muss natürlich zuverlässig sein. Von seiner Zuverlässigkeit hängt die Funktionsweise des umgebenden Produkts ab. Zusätzlich wird oft gefordert, dass das System eine gewisse Reaktionszeit aufweist. In diesem Fall ist das eingebettete System ein Echtzeitsystem. Für jede Aufgabe, die das eingebettete System erfüllen muss, gibt es dann eine festgelegte Zeitspanne. Innerhalb dieser Zeitspanne muss das Ergebnis der Aufgabe berechnet sein. Der letzte Zeitpunkt, zu dem das Ergebnis vorliegen muss, wird als *deadline* bezeichnet. Je nachdem, welche Konsequenzen eintreten können, wenn die vorgegebene Zeitspanne nicht eingehalten wird, wird zwischen weichen und harten *deadlines* unterschieden.

Wird eine weiche *deadline* überschritten, ist dies für den Nutzer des Produktes in der Regel ärgerlich, führt aber zu keinem Schaden. Eingebettete Systeme, die in Unterhaltungselektronik eingesetzt werden, sollen häufig weiche *deadlines* einhalten. Z. B. muss für die Anzeige eines Films jedes Einzelbild, aus dem sich der Film zusammensetzt, innerhalb von Bruchteilen von Sekunden verarbeitet werden. Kann die Verarbeitung von wenigen aufeinander folgenden Bildern nicht rechtzeitig abgeschlossen werden, ist für den Nutzer beim Betrachten des Films ein Ruckeln wahrnehmbar.

Harte *deadlines* müssen in jedem Fall eingehalten werden, da es sonst zu einer mehr oder weniger großen Katastrophe kommen kann. Jedes eingebettetes System, das in einem sicherheitskritischen Bereich eine steuernde Funktion erfüllt, muss innerhalb der vorgegeben Zeitspanne richtig reagieren. Z. B. kann ein Airbag, der bei einem Unfall zu spät gezündet wird, nicht seine volle Wirkung entfalten.

Ein weiterer wichtiger Aspekt eines eingebetteten Systems ist seine Effizienz. Er ist bei eingebetteten Systemen bedeutender als im Bereich der PC-Systeme. Die Hardware und die Software von PC-Systemen wird in der Regel von verschiedenen Herstellern getrennt entwickelt, produziert und vertrieben. Bei der Entwicklung der Software wird die Annahme getroffen, dass das System gewisse Mindestanforderungen erfüllt und daher bei der Ausführung der Software ausreichend Ressourcen zur Verfügung stehen. Bei der Zusammenstellung der Hardware wird dann darauf geachtet, dass für jede Software, die möglicherweise auf dem System ausgeführt wird, die Anforderungen erfüllt sind. Wesentliche Größen bei der Wahl der Hardwarekomponenten sind die Rechenleistung, Speichergrößen und der Preis bei der Anschaffung. Dagegen haben Größe, Gewicht und Energieverbrauch kaum einen Einfluss auf die Wahl der Zusammenstellung. Überflüssige Ressourcen werden nicht immer als Nachteil angesehen, sondern auch als Reserve für den Fall, dass andere Anwendungen auf der Hardware ausgeführt werden sollen.

Im Gegensatz dazu soll ein eingebettetes System mit möglichst wenigen Ressourcen ausgestattet sein. Da ein eingebettetes System ein fester Bestandteil eines Produktes ist, ist nicht zu erwarten, dass es irgendwann für eine andere Aufgabe eingesetzt wird als für die Informationsverarbeitung innerhalb des Produkts. Die Herstellungs- und Betriebskosten für die Bereitstellung der Ressourcen sollten deshalb minimiert werden. Oftmals sind eingebettete Systeme Teil mobiler Systeme wie Mobiltelefone und Digitalkameras. Dann darf das eingebettete System nur eine geringe Größe und ein geringes Gewicht aufweisen. Da die Geräte mit Energie aus Batterien betrieben werden, sollte für eine lange Gebrauchsdauer ohne Batteriewechsel der Energiebedarf auf ein Minimum reduziert werden.

In einem eingebetteten Echtzeitsystem sind der Einsparung von Ressourcen aber Grenzen gesetzt. Es müssen immer so viele Ressourcen vorliegen, dass garantiert werden kann, dass das eingebettete System die vorgegebenen Zeitspannen einhält. Wie viele Ressourcen zur Verfügung stehen müssen, wird durch die maximale Ausführungszeit eines Programms bestimmt. Diese wird auch als (*worst-case execution time* (WCET)) bezeichnet. Sie gibt die größte Ausführungsdauer an, die im schlimmsten Fall auftreten kann, und darf eine fest vorgegebene Zeitspanne aus den zuvor genannten Gründen nicht überschreiten. Für Echtzeitsysteme hat die WCET eines Programms daher eine besondere Bedeutung.

Sollte beim Entwurf eines eingebetteten Systems festgestellt werden, dass die WCET eines Programms größer als die vorgegebene Zeitspanne ist, können zwei Ansätze verfolgt werden, um sie zu verringern. Zum einen kann die maximale Ausführungsdauer eines Programms verkürzt werden, indem die Programmbefehle schneller ausgeführt werden. Dazu muss die Rechenleistung des Systems erhöht werden. Dies erfordert einen leistungsstärkeren Prozessor, der mehr Programmbefehle pro Zeiteinheit ausführen kann. Ein solcher Prozessor benötigt in der Regel aber auch mehr Energie. Damit Speicherzugriffe die Ausführung des Programms nicht übermäßig aufhalten, müssen auch die Speicher des Systems dem Prozessor angepasst werden. Ihre Geschwindigkeit und gegebenenfalls auch ihre Kapazität muss ebenfalls erhöht werden. Insgesamt werden also mehr Ressourcen benötigt. Wird daher dieser Ansatz zur Verringerung der WCET gewählt, sinkt die Effizienz des eingebetteten Systems.

Zum anderen kann die maximale Ausführungsdauer eines Programms aber auch verkürzt werden, indem das Programm so verändert wird, dass weniger Programmbefehle für die Berechnung eines Ergebnisses ausgeführt werden müssen. Auch können die Eigenschaften der Hardware vorteilhaft genutzt werden und Befehle so angeordnet werden, dass weniger Wartezeiten z. B. durch Speicherzugriffe entstehen. Da ein eingebettetes System

für genau einen Anwendungszweck vorgesehen ist, können Software und Hardware genau auf einander abgestimmt werden. Kann die WCET eines Programms durch Optimierungen verkleinert werden, wird weniger Rechenleistung benötigt, um das Ergebnis innerhalb der vorgegebenen Zeitspanne zu berechnen. Durch Optimierungen der Software kann also erreicht werden, dass weniger Ressourcen benötigt werden.

Als eine der bedeutendsten, wenn nicht die bedeutendste Optimierung, bei der Erstellung von Software bezeichnen Hennessy und Patterson die Registerallokation [HP03]. In einem Prozessorkern befinden sich in geringer Anzahl Register. Dies sind schnelle energieeffiziente Speicher, in denen Daten gespeichert werden. Die Aufgabe der Registerallokation besteht darin, die Software so zu optimieren, dass sie die Register effizient ausnutzt. Letztendlich sollen bei der Programmausführung möglichst wenige Zugriffe auf andere größere Speicher nötig sein, die energieintensiv und zugleich langsamer sind.

Die Optimierung der Software, die von der Registerallokation durchgeführt wird, stellt ein Optimierungsproblem dar. Alle verbreiteten Registerallokationsverfahren lösen das Optimierungsproblem nicht direkt, sondern abstrahieren erst und überführen das Problem in eine einfachere und strukturiertere Darstellung. Je nach Art der Darstellung können bei der Überführung Informationen verloren gehen. Die Ausgabe des Registerallokators stellt dann keine optimale Lösung dar. Eine sehr genaue Beschreibung von Optimierungsproblemen erlauben lineare ganzzahlige Programme. Eine ILP-basierte Registerallokation nutzt ein lineares ganzzahliges Programm bei der Optimierung der Software und kann so bessere Ergebnisse erzielen als andere Allokationsverfahren.

1.2 Ziele der Diplomarbeit

Die Verkleinerung der WCET eines Programms wirkt sich positiv auf den Ressourcenbedarf aus, der für eine ausreichend schnelle Ausführung des Programms innerhalb einer Zeitschranke nötig ist. Daher ist es sinnvoll, eine Registerallokation vorzunehmen, die das Programm mit dem primären Ziel optimiert, die WCET des Programms zu verringern. Bislang wurde ein solches Allokationsverfahren aber noch nicht vorgestellt. In dieser Arbeit soll deshalb ein Allokationsverfahren entwickelt und implementiert werden, das primär die WCET eines Programms optimieren soll.

Die Überführung des Optimierungsproblems der Registerallokation in ein ILP erzeugt eine strukturierte Beschreibung des Problems. Außerdem können einem ILP leicht zusätzliche Anforderungen hinzugefügt werden. Dies scheint geeignet, um die Forderung nach einer geringen WCET zu formulieren. Das Registerallokationsverfahren soll daher ein ILP nutzen, um das Optimierungsproblem zu lösen.

An dem Lehrstuhl 12 der Fakultät für Informatik der Technischen Universität Dortmund wird von der Forschungsgruppe für eingebettete Systeme ein Compiler namens WCET-aware C-Compiler (WCC) [FLT06] entwickelt. Der WCC ist ein Compiler für C-Programme mit einem Fokus auf WCET-Optimierungen. Für die Registerallokation wird in dem WCC zur Zeit ein Standardverfahren verwendet. Dieses nimmt keine gezielte Verringerung der WCET eines Programms vor.

Das Registerallokationsverfahren, das in dieser Diplomarbeit erstellt werden soll, soll als Alternative zu dem Standardverfahren ausgeführt werden können. Der Registerallokator kann dabei die Infrastruktur des WCC für die Ermittlung von WCET-Daten nutzen. Da der WCC C-Programme in Maschinenprogramme für den Infineon Mikrocontroller TC1796 [Inf08] mit einem Infineon TriCore Prozessor [Inf05a] übersetzt, muss der Regis-

terallokator auf die Architekturmerkmale dieses Prozessors abgestimmt sein. Im einzelnen sind die Ziele, die in dieser Diplomarbeit erreicht werden sollen:

- Es soll ein ILP-basierter Registerallokator erstellt werden.
- Der Registerallokator soll WCET-Daten nutzen, um Entscheidungen so zu treffen, dass die WCET eines Programms minimiert wird.
- Der Registerallokator soll innerhalb des WCC ausgeführt werden können.

1.3 Aufbau der Diplomarbeit

In diesem Abschnitt wird der Aufbau dieser Diplomarbeit beschrieben. Dazu werden kurz die Themen der einzelnen Kapitel vorgestellt.

In Kapitel 2 werden die Grundlagen zu den zwei zentralen Themenbereichen dieser Diplomarbeit behandelt. Zum einen wird auf das Problem der Registerallokation eingegangen und bestehende Lösungsverfahren vorgestellt. Zum anderen wird definiert, was die WCET eines Programms ist, und beschrieben, welche Schwierigkeiten bei der Bestimmung der WCET eines Programms und der Optimierung eines Programms bezüglich der WCET bestehen.

Kapitel 3 handelt von dem WCET-aware C-Compiler (WCC), in den die Registerallokation, die in dieser Diplomarbeit erstellt wurde, integriert ist. In diesem Kapitel wird der Aufbau des WCC beschrieben und die Registerallokation in diesem Aufbau eingeordnet. Des weiteren wird die interne Zwischendarstellung für Programme beschrieben, auf die die Registerallokation angewendet wird. Der WCC ist in der Lage, für diese Zwischendarstellung eine WCET-Analyse mit Hilfe eines professionellen Analysewerkzeugs durchzuführen. Dieses WCET-Analysewerkzeug wird ebenfalls in diesem Kapitel vorgestellt. Da die Registerallokation eine hardwarenahe Optimierung ist, wird abschließend noch die Zielarchitektur des WCC beschrieben.

In Kapitel 4 wird das ILP-Modell von Goodwin und Wilken [GW96] für einen optimalen Registerallokator vorgestellt. Auf diesem Modell basiert der Registerallokator, der in dieser Diplomarbeit entwickelt und implementiert wurde.

Kapitel 5 zeigt auf, wie die WCET einer Funktion durch ein ILP-Modell beschrieben werden kann. Es wird erläutert auf welche Weise WCET-Daten genutzt und in dem ILP-Modell des Registerallokators mit dem Ziel berücksichtigt werden, die WCET eines Programms zu verringern.

Kapitel 6 befasst sich mit der Bestimmung von WCET-Daten. Es wird zunächst erklärt, welche Schwierigkeiten bei der Ermittlung von WCET-Daten bewältigt werden müssen. Anschließend wird die Vorgehensweise vorgestellt, die in dieser Diplomarbeit gewählt wurde, um die WCET-Daten zu bestimmen, die in das ILP-Modell einfließen.

Der in dieser Diplomarbeit erstellte Registerallokator wird mit einem bestehenden Registerallokator verglichen, der ein Standardverfahren implementiert. In Kapitel 7 werden die Ergebnisse dieses Vergleichs vorgestellt.

Kapitel 8 schließt diese Diplomarbeit mit einer Zusammenfassung ab. Die Erkenntnisse, die durch diese Diplomarbeit gewonnen wurden, werden noch einmal kurz wiederholt. Rückblickend wird bewertet, ob und inwiefern die Ziele dieser Diplomarbeit erreicht wurden. Zu allerletzt wird ein Ausblick auf mögliche Verbesserungen gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden Begriffe und Algorithmen vorgestellt, deren Kenntnis für das Verständnis der folgenden Kapitel nützlich sind. Zunächst wird in Abschnitt 2.1 der Kontrollflussgraph vorgestellt, der eine zentrale Datenstruktur für hardwarenahe Übersetzungsschritte ist, zu denen auch die Registerallokation zählt. Das Problem der Registerallokation und einige seiner Lösungsverfahren werden in Abschnitt 2.2 beschrieben. Die Güte der von den Lösungsverfahren erzeugten Ausgaben kann an Hand von verschiedenen Kriterien beurteilt werden. Häufig wird die Codegröße oder die durchschnittliche Ausführungszeit der resultierenden Programme untersucht. Fast gar nicht wurde bislang die maximale Ausführungszeit als Kriterium verwendet. Dieses Kriterium wird in Abschnitt 2.3 vorgestellt.

2.1 Kontrollflussgraph

Der Kontollflussgraph (*control flow graph* (CFG)) einer Funktion ist eine Darstellung, die genutzt wird, um Aussagen über die Ausführungsreihenfolge von Anweisungen in der Funktion zu treffen. Er wird aus dem Programm abgeleitet und spiegelt die Programmstruktur wieder.

Definition 2.1 *Ein Flussgraph $G = (V, E, i)$ ist ein gerichteter Graph (V, E) mit einem initialen Knoten $i \in V$. Für jeden Knoten $v \in V$ gibt es einen Pfad in G von i nach v [Tar73].*

Definition 2.2 *Ein Kontrollflussgraph $G = (V, E, i)$ zu einer Funktion F ist ein Flussgraph (V, E, i), der für jede Anweisung in der Funktion F genau einen Knoten enthält. Ferner enthält er eine Kante zwischen zwei Knoten x und y, wenn die Anweisung, die durch den Knoten y repräsentiert wird, unmittelbar auf die Anweisung folgen kann, die durch den Knoten x repräsentiert wird [App98]. Der initiale Knoten i repräsentiert die erste Anweisung der Funktion.*

Hat ein Sprungbefehl mehrere Ziele, wird angenommen, dass alle Anweisungen, die ein Ziel des Sprunges sind, auch auf den Befehl folgen können. Sprungbedingungen werden also nicht ausgewertet. Der Kontollflussgraph repräsentiert somit die Menge aller Kontrollpfade einer Funktion, aber auch Kontrollpfade, die bei der Programmausführung niemals beobachtet werden können, da z. B. eine Sprungbedingung niemals erfüllt sein kann, oder Abhängigkeiten zwischen Sprungbedingungen bestehen. Abbildung 2.1 zeigt hierzu ein Beispiel. Dargestellt ist zweimal der gleiche Kontrollflussgraph. Die Instruktionen A und C sind bedingte Sprungbefehle. Während von der Instruktion A der Sprung zur Instruktion C genau dann ausgeführt wird, wenn die Werte der Variablen $v1$ und $v2$ identisch

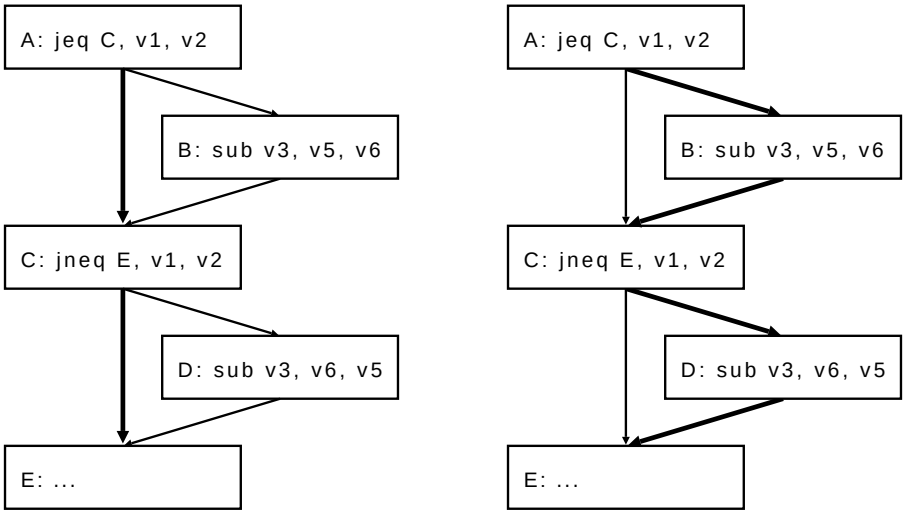

Abbildung 2.1: Kontrollflussgraph mit zwei *infeasible paths*.

sind, ist die Bedingung für den Sprungbefehl C das Gegenteil. Da von Instruktion B der Wert der Variablen $v1$ und $v2$ nicht geändert wird, können nicht beide Sprünge hintereinander erfolgen. Auch können auf keinem Ausführungspfad die Instruktionen B, C und D in unmittelbarer Folge vorkommen. Solche Pfade, die zwar in dem Kontrollflussgraphen enthalten sind, aber keine mögliche Ausführungsreihenfolge der Programmbefehle darstellen, werden *infeasible paths* genannnt. In der Abbildung 2.1 sind jeweils die Kanten, die *infeasible paths* repräsentieren mit größerer Linienbreite eingezeichnt.

Oftmals werden bei der Untersuchung des Programmflusses aufeinanderfolgende Anweisungen, die immer nacheinander ausgeführt werden, sobald der Kontrollfluss die erste Anweisung erreicht, zu größeren Einheiten zusammengefasst. Nur die letzte Anweisung der Anweisungsfolge kann ein Sprungbefehl sein und den Kontrollfluss verzweigen.

Definition 2.3 *„Ein Basisblock $B = (I_1, \ldots, I_n)$ ist eine Sequenz von Instruktionen maximaler Länge, so dass gilt:*

- *B wird nur über I_1 betreten, und*
- *B wird nur über I_n verlassen.“[Muc97, S. 173]*

2.2 Registerallokation

Ein Register ist ein kleiner Speicher innerhalb eines informationsverarbeitenden Systems, der durch den Zusammenschluss von 1-Bit-Speichern (Flip-Flops und Latches) entsteht. Ein moderner Prozessor verfügt über eine geringe Anzahl dieser Register, in denen Operanden und Ergebnisse von Operationen gespeichert werden. Eine Aufgabe eines Compilers besteht daher darin, die Werte von Programmvariablen, Konstanten und Zwischenergebnisse von Berechnungen in den Registern zu platzieren. Da die Anzahl der Register begrenzt ist, kann immer nur eine beschränkte Anzahl an Werten gleichzeitig in den Registern gespeichert werden. Zeitweise müssen die Inhalte in andere größere Speicher

ausgelagert werden, damit andere gerade benötigte Operanden und neue Ergebnisse den Registern zugewiesen werden können.

Die Verwendung anderer Speicher ist in diesem Fall zwar notwendig, hat aber Nachteile und sollte sonst nach Möglichkeit vermieden werden. Register sind als Teil des Prozessors auf dem Prozessor-Chip integriert. Zugriffe auf sie benötigen weniger Energie als auf andere Speicher und können auf Grund der Lokalität am schnellsten ausgeführt werden. Somit sind Register die effizientesten Speicher, auf die der Prozessor zugreifen kann.

Bei Load/Store-Architekturen ist erforderlich, dass sich die Operanden in Registern befinden. Zusätzliche Instruktionen müssen in dem Programmcode eingefügt werden, um die Inhalte zwischen den Registern und anderen Speichern zu transferieren. Dies wirkt sich negativ sowohl auf den Energieverbrauch, die Codegröße als auch auf die Laufzeit des Programms aus.

Um nicht bei jedem Übersetzungsschritt das Problem einer guten Registernutzung bewältigen zu müssen, wird dieses Problem erst in einem separaten Übersetzungschritt gelöst, der Registerallokation. Vorherige Übersetzungschritte nehmen zur Vereinfachung an, dass es eine unendliche Anzahl an Registern gäbe. Wann immer eine neue Konstante, eine Variable oder ein Ergebnis gespeichert werden muss, wird einfach ein neues Register verwendet. Um diese Register von den physikalisch existierenden Registern zu unterscheiden, werden sie als virtuelle Register bezeichnet. Die Aufgabe der Registerallokation besteht nun darin, die virtuellen Register effizient auf die physikalischen Register abzubilden.

Der Registerallokation geht im Allgemeinem eine Lebendigkeitsanalyse voraus. Diese wird in Abschnitt 2.2.1 kurz charakterisiert. In Abschnitt 2.2.2 wird das Problem der Registerallokation präzisiert. Einige Lösungsverfahren für die Registerallokation werden in Abschnitt 2.2.3 vorgestellt.

2.2.1 Lebendigkeitsanalyse

Die Lebendigkeitsanalyse (*life-time analysis* (LTA)) ermittelt, an welchen Stellen im Kontrollflussgraphen ein (virtuelles) Register lebendig ist. Die Lebendigkeit eines Registers gibt Auskunft darüber, wann das Register einen Wert enthält, der im Programm möglicherweise noch genutzt wird. Hierzu wird untersucht, wie das Register von den Instruktionen im Programm verwendet wird. Ein Register kann auf zwei Arten von einer Instruktion verwendet werden:

Definition 2.4 *Eine Instruktion* definiert *ein Register genau dann, wenn sie den im Register gespeicherten Wert ändern kann.*

Definition 2.5 *Eine Instruktion* nutzt *ein Register genau dann, wenn sie den im Register gespeicherten Wert ausliest*

Mit diesen zwei Begriffen lässt sich die Lebendigkeit eines Registers präzise definieren:

Definition 2.6 *Ein Register ist genau dann nach einer Instruktion lebendig, wenn es einen Pfad im Kontrollflussgraphen gibt, der bei der Instruktion beginnt und mit einer Instruktion endet, die das Register nutzt, und keine Instruktion auf dem Pfad das Register definiert.*

Definition 2.7 *Ein Register ist genau dann vor einer Instruktion lebendig, wenn es von der Instruktion genutzt wird, oder wenn es nach der Instruktion lebendig ist, und nicht von der Instruktion definiert wird.*

Das Ergebnis der Lebendigkeitsanalyse sind somit zwei Relationen auf der Menge der Instruktionen und der Register: LiveIn und LiveOut: Instruktion × Register. Sie werden üblicherweise durch Abbildungen Instruktion $\mapsto \mathcal{P}\{\text{Register}\}$ dargestellt, wobei $\mathcal{P}\{\text{Register}\}$ die Potenzmenge der Menge aller Register ist. Die Menge LiveIn einer Instruktion enthält all die Register, die vor der Instruktion lebendig sind, LiveOut ist die Menge der Register, die unmitelbar nach der Instruktion lebendig sind. Effiziente Algorithmen zur Berechnung der Mengen gibt Appel an [App98].

2.2.2 Registerallokationsproblem

Das Problem der Registerallokation tritt auf, wenn ein Programm aus einer Hochsprache in Maschinencode überführt wird. Überlicherweise erfolgt diese Übersetzung des Programms durch die sequentielle Ausführung von Übersetzungsschritten, die jeweils ein Teilproblem oder eine kleine Menge verwandter Teilprobleme der Übersetzung lösen. Eines dieser Teilprobleme ist das Registerallokationsproblem, das von der Registerallokation gelöst wird. Übersetzungschritte, die vor der Registerallokation ausgeführt werden, nehmen zur Vereinfachung an, dass es eine unendlich große Anzahl an virtuellen Registern gäbe. Die Eingabe für die Registerallokation ist daher eine Zwischendarstellung (*intermediate representation* (IR)) des Programms, in der virtuelle Register vorkommen.

Das Problem der Registerallokation besteht darin, jedem virtuellen Register ein physikalisches Register so zuzuordnen, dass zwei virtuelle Register, die gleichzeitig lebendig sind, nicht demselben physikalischen Register zugeordnet sind. In der Umkehrung bedeutet dies, dass zwei Register, die nicht gleichzeitig lebendig sind, dem gleichem Register zugeordnet werden können. So kann es trotz der großen Anzahl an virtuellen Registern und der geringen Anzahl an physikalischen Registern eine Abbildung der virtuellen Register auf die physikalischen Register geben, ohne dass sich die Werte der virtuellen Register in den physikalischen Registern gegenseitig überschreiben. Hat die Registerallokation das Allokationsproblem gelöst, muss gemäß der Abbildung jedes Vorkommen der virtuellen Register in der Zwischendarstellung durch die Bezeichung des physikalischen Registers ersetzt werden, dem das virtuelle Register zugeordnet ist.

Die Problemstellung ist häufig dadurch erweitert, dass für eine Teilmenge der virtuellen Register vorgeschrieben ist, auf welche physikalischen Registern sie abgebildet werden müssen. Der Grund hierfür kann z. B. sein, dass auf diesen Registern Operationen ausgeführt werden, die ihre Operanden in festgelegten physikalischen Registern erwarten. Solche Vorgaben werden als Vorfärbungen bezeichnet. Dieser Begriff wurde durch die Allokationsverfahren geprägt, die auf der Graphfärbung basieren. Dort werden Register durch Farben repräsentiert.

Ist das Registerallokationsproblem nicht lösbar, muss ein erweitertes Allokationsproblem gelöst werden. Dies ist vielfach der Fall, weil die Anzahl der Register, die gleichzeitig lebendig sind, größer ist als die Anzahl der physikalischen Register, die zur Verfügung stehen. Neben den Registern steht dann noch ein weiterer Speicher zur Verfügung, in den virtuelle Register ausgelagert werden können, und von dem sie auch wieder in ein physikalischen Register geladen werden können. Üblicherweise wird hierfür der Laufzeitstack des

Programms verwendet. Jedem Register, das ausgelagert wird, ist ein eigener Speicherbereich zugeordnet, so dass sich die Werte der Register im Speicher nicht überschreiben. Es wird angenommen, dass der Speicher groß genug ist, um alle virtuellen Register speichern zu können.

Bei einigen Architekturen können Instruktionen virtuelle Register als Operanden verwenden, obwohl diese sich im Speicher befinden. Liegt der Registerallokation aber eine Load-Store Architektur zu Grunde, gibt es keine solche sogenannten *Memory Operanden*. Dann muss ein virtuelles Register bei einer Definition oder Nutzung wieder dem physikalischen Register zugeordnet sein. An den Stellen in der Zwischendarstellung, an denen ein Transfer zwischen Registern und dem Speicher stattfinden soll, müssen Speicher- und Lade-Befehle in die Zwischendarstellung eingefügt werden. Diese Programmbefehle zum Speichern und Laden der Register werden als Spillinstruktionen oder Spillcode bezeichnet.

Durch die Möglichkeit, Register zeitweise in den Speicher zu verschieben, kann die Lebendigkeit der Register verkürzt werden. Wird durch das Einfügen der Spillinstruktionen erreicht, dass sich immer eins von zwei Registern, die anfangs gleichzeitig lebendig waren, im Speicher befindet, so sind diese beiden Register nicht mehr gleichzeitig lebendig und können ein und demselben Register zugeordnet werden.

Das erweiterte Allokationsproblem besteht dann darin, Spillcode so in die Zwischendarstellung einzufügen, dass das Registerallokationsproblem lösbar ist und eine Zielfunktion minimiert wird. Prinzipiell kann auf naive Weise jedes Register nach einer Definiton in den Speicher geschrieben und vor jeder Nutzung wieder geladen werden. Die Zielfunktion soll daher sicherstellen, dass nicht unnötig viel Spillcode in die Zwischendarstellung eingefügt wird. Der Wert der Zielfunktion ist häufig die zusätzliche Codegröße, die Verlängerung der minimalen, durchschnittlichen oder maximalen Ausführungsdauer oder der zusätzliche Energieverbrauch, der durch den Spillcode entsteht.

In aller Regel ist gefordert, dass ein Registerallokator bis auf das Ersetzen der Registerbezeichner und das Einfügen von Spillcode die Registerallokation keine Veränderungen an der Zwischendarstellung vornehmen darf. Nur wenn der Allokator zusätzliche Optimierungen durchführen soll, wie sie z. B. in Abschnitt 2.2.2.3 beschrieben werden, gibt es von der Anforderung Abweichungen. In keinem Fall aber darf die Registerallokation die Reihenfolge der Instruktionen ändern, um die Lebenszeiten der Register zu beeinflussen.

Da die Suche nach der Lösung für das Registerallokationsproblem sehr aufwendig sein kann, versuchen viele Allokatoren nicht, es optimal zu lösen. In Folge dessen fügen Allokatoren Spillcode ein, obwohl bei einer optimalen Lösung des Registerallokationsproblems kein Spillcode benötigt würde. Oft wird daher zwischen dem Registerallokationsproblem und dem erweiterten Allokationsproblem nicht unterschieden. Ein Registeralloktor muss sowieso in der Lage sein, das erweiterte Allokationsproblems zu lösen. Auch in dieser Diplomarbeit wird im Folgenden das erweiterte Allokationsproblem ebenfalls als Registerallokationsproblem bezeichnet.

2.2.2.1 Calling Conventions

Zu Beginn einer Funktion könnten physikalische Register Werte von virtuellen Registern der Funktion enthalten, die die Funktion aufgerufen hat. Damit nicht angenommen werden muss, dass jedes Register einen Wert enthält, der von der aufrufenden Funktion noch benötigt wird, werden die physikalischen Register drei Klassen zugeordnet. Diese Zuordnung wird als *calling convention* bezeichnet.

Ein Teil der Register sind *callee-saved register*. Diese Register können durch die aufrufende Funktion belegt sein. Bevor diesen Registern ein virtuelles Register zugeordnet wird, muss in der aufgerufenen Funktion der Inhalt des Registers in dem Speicher gesichert werden. Bevor die aufgerufene Funktion wieder verlassen wird, muss der Wert des Registers wiederhergestellt werden. *Caller-saved register* sind die Register, die vor einem Funktionsaufruf in der aufrufenden Funktion gesichert werden müssen. In der aufgerufenden Funktion können die physikalischen Register sofort mit virtuellen Registern belegt werden.

Die Register, die der dritten Klasse angehören, werden für die Übergabe von Parametern an die aufgerufene Funktion und zur Rückgabe von Ergebnissen an die aufrufende Funktion verwendet. Durch Vorfärbungen wird erreicht, dass sich der Wert des virtuellen Registers, das den Parameter für die aufgerufene Funktion enthält, in dem physikalischen Register befindet, in dem die aufgerufene Funktion den Wert auch erwartet. Ähnliches gilt für die Rückgabe von Funktionsergebnissen.

2.2.2.2 Komplexität

Die Registerallokation kann verschiedene Bereiche einer Zwischendarstellung auf einmal berücksichtigen. Globale Allokatoren arbeiten auf ganzen Funktionen. Interprozedurale Effekte, die über die *calling conventions* hinaus gehen, werden aber nicht berücksichtigt. Lokale Allokatoren würden eine Allokation für jeden Basisblock getrennt vornehmen.

Unabhängig davon ist das Problem der Registerallokation NP-schwer. Sethi [Set73] zeigt, dass das Problem der globalen Registerallokation NP-schwer ist. Dass auch lokale Allokatoren, die jeden Basisblock isoliert betrachten, ein NP-schweres Problem lösen müssen, zeigt Farach et al. [FL98].

2.2.2.3 Erweiterungen der Registerallokation

Die Registerallokation bestimmt eine Abbildung der virtuellen Register auf die physikalischen Register. Dadurch ist jedem virtuellen Register genau ein physikalisches Register zugeordnet. Wenn das Register in den Speicher ausgelagert wird und dann wieder geladen werden soll, gilt diese Zuordnung weiterhin. Das Register muss dann wieder in das gleiche physikalische Register geladen werden. Mit *live range splitting* wird die Unterteilung der Lebendigkeit eines Registers in kleinere Abschnitte bezeichnet. Nach erfolgter Aufteilung kann das Register für jeden Teilabschnitt einem anderen physikalischen Register zugeordnet sein. Zunächst kann ein virtuelles Register einem physikalischem Register zugeordnet sein. Nachdem das Register im Speicher gesichert wurde, kann es in ein anderes physikalisches Register geladen werden. Auf diese Weise können Konflikte zwischen gleichzeitig lebendigen Registern besser vermieden werden. Registerallokatoren, die ein *live range splitting* durchführen, brauchen daher in der Regel weniger Spillcode einzufügen.

Eine Kopierinstruktion ist eine Registertransferinstruktion, die den Wert eines virtuellen Registers A unverändert einem virtuellen Register B zuweist. Kopierinstruktionen, bei denen sich das virtuelle Register A und das virtuelle Register B in dem gleichen physikalischen Register befinden, sind redundant und können aus der Zwischendarstellung entfernt werden. Bei der Optimierung *coalescing* sollen virtuelle Register so den physikalischen Register zugeordnet werden, dass möglichst viele solcher redundanten Kopierinstruktionen entstehen und entfernt werden können.

Enthält ein Register einen Wert, der durch eine einzelne Instruktion in einem Register erzeugt werden kann, z. B. eine Konstante, ist es nicht sinnvoll, diesen Wert in den Speicher auszulagern und anschließend wieder in das Register zu laden. Statt dessen kann auf die Spillinstruktionen verzichtet werden, und jedesmal wenn das Register geladen werden müsste, der Wert durch die einzelne Instruktion wiederhergestellt werden. Diese Vermeidung von Spillcode wird als *rematerialization* bezeichnet.

2.2.3 Registerallokationsverfahren

Ein Allokationsverfahren, das unter allen Gesichtspunkten das beste ist, gibt es nicht. Abhängig davon, welche Anforderungen an die Allokation gestellt werden, muss zwischen der Güte der erzeugten Ausgaben und dem benötigten Ressourcenbedarf abgewogen werden. Für die Registerallokation gibt es daher zahlreiche Lösungsansätze und -verfahren mit unterschiedlichen Eigenschaften. An dieser Stelle beschränken wir uns auf die Betrachtung von globalen Allokationsverfahren.

Das Graphfärbungsverfahren ist ein weit verbreitetes Verfahren, das sich als das Standardallokationsverfahren etabliert hat. Es stellt einen guten Kompromiss dar, der sowohl gute Lösungen erzeugt als auch eine moderate Laufzeit aufweist. Die Güte eines neuen Verfahrens wird fast immer durch Vergleiche mit dem Graphfärbungsverfahren beurteilt. Eine Beschreibung dieses Verfahren erfolgt in Abschnitt 2.2.3.1.

Linear Scan Allokatoren zeichnen sich durch eine geringe Laufzeit aus. Die Allokation erfolgt mittels einer Heuristik, die wenig Aufwand erfordert und nur einen lokalen Kontext berücksichtigt. Dadurch gelingt es nicht, die physikalischen Register so effektiv zu nutzen wie ein Graphfärbungsallokator. Folglich müssen Linear Scan Allokatoren mehr Spill Code einfügen. Sie werden daher dann eingesetzt, wenn der Laufzeit des Allokators eine größere Bedeutung als der Güte des resultierenden Codes zukommt. Stellvertretend für eine ganze Reihe von Linear Scan Allokatoren, wird in Abschnitt 2.2.3.2 der Linear Scan Allokator von Poletto und Sarkar [PS99] vorgestellt.

In Abschnitt 2.2.3.3 wird das Prinzip von ILP-basierten Allokatoren beschrieben. Die Güte dieser Allokatoren ist bezüglich des verwendeten Kostenmodells optimal. Ihr Nachteil ist die vergleichsweise große Laufzeit. Weitere Allokationsverfahren werden in Abschnitt 2.2.3.4 genannt.

2.2.3.1 Graphfärbungverfahren

Bei den Graphfärbungsverfahren wird das Problem der Registerallokation auf das k-Graphfärbeproblem zurückgeführt. Das k-Graphfärbeproblem besteht darin, jedem Knoten in dem Graphen eine Farbe von k Farben so zuzuordnen, dass zwei Knoten, die durch eine Kante verbunden sind, unterschiedlich gefärbt sind. Dies abstrahiert in angemessener Weise von dem Problem der Registerallokation, wo jedem virtuellen Register ein physikalisches Register zugeordnet werden muss.

Die k Farben repräsentieren die zur Verfügung stehenden physikalischen Register. Der Graph, der gefärbt wird, ist ein Interferenzgraph. Er enthält für jedes virtuelle Register einen Knoten. Die Knoten von zwei Registern, deren Lebenszeiten interferieren, d. h. die gleichzeitig lebendig sind, sind mit einer Kante verbunden. Auf diese Weise wird sichergestellt, dass solche Register nicht dem selbem physikalischen Register zu geordnet werden. Auch alle anderen Bedingungen, die eine gültige Registerallokation erfüllen muss, müssen

durch den Interferenzgraphen modelliert werden. So kann durch das Einfügen weiterer Knoten und Kanten erzwungen werden, dass ein virtuelles Register einem physikalischem Register zugeordnet werden muss, oder dass ein virtuelles Register einem bestimmten Register nicht zugeordnet werden darf.

Kann der Interferenzgraph mit den k Farben gefärbt werden, kann daraus unmittelbar eine Lösung für die Registerallokation abgeleitet werden, bei der jedes virtuelle Register einem physikalischen Register zugeordnet ist. Für das Färben des Graphen wird aus zwei Gründen eine Heuristik eingesetzt. Zum einem ist das k-Graphfärbeproblem für ein festes $k \geq 3$ wie das Problem der Registerallokation NP-schwer (siehe [Weg03]). Zum anderen ist der Interferenzgraph möglicherweise gar nicht mit den k Farben färbbar. In diesem Fall soll die Heuristik bereits Register ermitteln, die auf den Laufzeitstack ausgelagert werden sollen.

Die wohl bekanntesten Graphfärbungsallokatoren stammen von Chaitin [Cha82] und Briggs [Bri92]. Sie setzen eine Heuristik ein, deren Arbeitsweise sich in zwei Phasen unterteilt. In der ersten Phase, die *simplify* genannt wird, werden nacheinander alle Knoten zusammen mit ihren inzidenten Kanten aus dem Interferenzgraphen entfernt und auf einem Stack gespeichert. Wann immer es einen Knoten mit weniger als k Nachbarn gibt, wird dieser aus dem Graphen entfernt. Da er weniger als k Nachbarn hat, kann diesem Knoten in jedem Fall eine Farbe zugeordnet werden, die keinem seiner Nachbarn zugeordnet ist. Tritt der Fall ein, dass alle Knoten in dem Graphen mindestens k Nachbarn aufweisen, wird ein Knoten nach einem Kostenkriterium ausgewählt. Das Kostenkriterium kann z. B. berücksichtigen, wie viel Spill Code eingefügt werden müsste, wenn das Register auf den Stack ausgelagert werden würde. Auch ob der Spill Code in einer Schleife liegen würde, also potenziell öfter ausgeführt werden würde, kann in das Kostenkriterium einfließen. Wenn ein Knoten aus dem Graphen entfernt wird, sinkt der Grad, d. h. die Anzahl der Nachbarn, für die Nachbarknoten um eins. In der Hoffnung, dass dadurch viele Knoten einen Grad kleiner k erhalten, werden bei der Auswahl ebenfalls Knoten mit vielen Nachbarn bevorzugt.

Alle Register, die von Knoten repräsentiert werden, die mindestens k Nachbarn hatten, als sie aus dem Graphen entfernt wurden, werden bei dem Allokator von Chaitin auf dem Laufzeitstack ausgelagert. Für diese Register wird Spill Code in das Programm eingefügt, wodurch sich die Lebenszeiten der Register verkürzen. Da diesen Registern aber noch kein physikalisches Register zugeordnet werden konnte, muss für das geänderte Programm die Allokation beginnend mit der Erstellung des Interferenzgraphen erneut durchgeführt werden.

Sobald aber einmal in der gesamten *simplify*-Phase kein weiteres Register auf den Laufzeitstack ausgelagert wird, können auf einfache Weise den Registern physikalische Register zugeordnet werden. Dies erfolgt in der zweiten Phase, die *select* genannt wird. In dieser Phase werden die Knoten in der umgekehrten Reihenfolge, in der sie entfernt wurden, wieder in den Interferenzgraphen eingefügt und eine freie Farbe zu geordnet, die also noch keinem Nachbarn zugeordnet ist. Dies ist immer möglich, da beim Wiedereinfügen des Knotens genau der Interferenzgraph wieder entsteht, der vorlag, als der Knoten entfernt wurde. Schließlich war für den Knoten in diesem Graphen garantiert, dass für den Knoten eine freie Farbe gefunden werden kann.

Eine Veränderung des Allokators von Briggs gegenüber dem vom Chaitin besteht darin, dass dieser nicht unbedingt jedes Register auf den Laufzeitstack auslagert, das durch einen Knoten repräsentiert wird, der beim Entfernen aus dem Interferenzgraphen mindestens k

Nachbarn hatte. Statt dessen wird ein solches Register nur als ein möglicher Kandidat für das Spilling markiert. Im Gegensatz zum Allokator von Chaitin führt der Allokator auch immer noch die *select*-Phase aus. Nacheinander werden die Knoten wieder in den Graphen eingefügt, aber zunächst nur den Knoten Farben zugewiesen, die keine Kandidaten für das Spilling sind. Erst nachdem der vollständige Interferenzgraph wiederhergestellt ist, wird versucht den Knoten, die Kandidaten für das Spilling sind, auch noch eine Farbe zuzuweisen. Da nicht alle Nachbarn eines Knotens unterschiedlich gefärbt sein müssen, kann dies erfolgreich sein, auch wenn der Knoten mehr Nachbarn hat, als Farben zur Verfügung stehen.

Nur die Knoten, denen nun keine Farbe zugeordnet werden kann, werden auf den Laufzeitstack ausgelagert. Weil der Allokator die optimistische Annahme trifft, den Knoten am Ende der *select*-Phase noch eine Farbe zuordnen zu können, wird diese Graphfärbeheuritisk von Briggs selbst *optimistic spill heuristic* genannt. Sie führt möglicherweise dazu, dass weniger Register auf den Stack ausgelagert werden. Dadurch kann die Güte der Allokation besser sein, als die des Allokators von Chaitin. Gleichzeitig müssen aber vielleicht mehr *simplify*- und gewiss mehr *select*-Phasen ausgeführt werden. Denn zum einen wird nie nach der *simplify*-Phase abgebrochen. Zum andern werden durch das Auslagern der Register auf den Laufzeitstack ihre Lebenszeiten verkürzt, was ein Färben des Interferenzgraphen erleichtern soll. Wird in jedem Schritt nur von einer kleineren Anzahl der Register die Lebenszeit verkürzt, müssen eventuell mehrere Iterationen ausgeführt werden, bis für den Interferenzgraphen eine Färbung gefunden ist.

Briggs berichtet, dass der Einsatz der *optimistic spill heuristic* zu 20-40% besseren Ergebniss führt, die asymptotische Laufzeit aber weiterhin $\mathcal{O}(n \log n)$ beträgt, wobei n die Anzahl der Instruktionen im Programm ist, für das die Allokation ausgeführt wird.

2.2.3.2 Linear Scan Allokatoren

Der Begriff Linear Scan Allokator wurde von Poletto und Sarkar [PS99] eingeführt. In diesem Abschnitt wird ihr Linear Scan Allokator vorgestellt. Dieser setzt voraus, dass es eine lineare Ordnung auf den Basisblöcken gibt. Damit ergibt sich auch für die Instruktionen eine lineare Ordnung. Jeder Instruktion ist dadurch ein eindeutiger Index zugeordnet, der der Position in der linearen Aneinanderreihung der Basisblöcke entspricht. Bezüglich der Ordnung über den Basisblöcken wird keine weitere Annahme getroffen, so dass die Wahl der Ordnung keinen Einfluss auf die Korrektheit des Allokators hat. Allerdings beeinflusst sie die Güte des resultierenden Codes.

Basierend auf der Ordnung der Instruktionen bestimmt der Allokator für jedes virtuelle Register ein Lebensintervall $[i, j]$, so dass es keine Instruktion $i' < i$ und keine Instruktion $j' > j$ gibt, zu der das Register lebendig ist. Außerhalb des abgeschlossenen Intervalls ist das Register somit in jedem Fall nicht lebendig. Innerhalb des gesamten Intervalls muss das Register aber nicht an jeder Stelle lebendig sein. Es kann Teilintervalle geben, zu dem das virtuelle Register nicht lebendig ist. Solche Intervalle werden als Lebenslücken bezeichnet.

Je weniger Lebenslücken ein Lebensinterval enthält, umso bessere Ergebnisse können erzielt werden. Hierauf hat die Wahl der Ordnung auf den Basisblöcken einen unmittelbaren Einfluss. Poletti und Sarkar schlagen vor, als Ordnung die umgekehrte Postorder zu verwenden. Sagonas et al. [SS03] bestätigen, dass in ihren Versuchen diese Ordnung häufig zu besseren Ergebnissen führt, als sieben andere geläufige Ordnungen.

Der Linear Scan Allokator verarbeitet die Lebensintervalle in aufsteigender Reihenfolge ihrer unteren Grenzen. Gibt es ein physikalisches Register, dem zu Beginn des Lebensintervalls kein virtuelles Register zugeordnet ist, so wird es für das virtuelle Register reserviert, auf das sich das Lebensintervall bezieht. Die Reservierung des physikalischen Registers erfolgt dabei für das gesamte Lebensintervall. Sind zu Beginn des Lebensintervalls dagegen alle dem Registerallokator zur Verfügung stehenden physikalischen Register bereits reserviert, wird mit einer einfachen Heuristik entschieden, welches virtuelle Register in den Speicher ausgelagert wird. Aus den virtuellen Registern, für das an der Position ein physikalisches Register reserviert ist, und dem Register, dessen Lebensintervall gerade verarbeitet wird, wird dasjenige ausgewählt, dessen Lebensintervall die größte obere Grenze aufweist.

Diese Heuristik erinnert an das optimale Ersetzungsverfahren für gleichmäßig partitionierte Speicher [Bel66]. Sie führt aber nur dann nachweislich zu optimalem Code, wenn der Kontrollflussgraph keine Verzweigungen aufweist, die Basisblöcke gemäß ihrer Ausführungsreihenfolge geordnet sind und auf jede Definition eines Registers nur genau eine Nutzung erfolgt.

Nur die Register, die nie in diesem Schritt ausgewählt werden, werden einem physikalischen Register zugeordnet. Da diese Zuordnung für das gesamte Lebensintervall erfolgt, bleibt das physikalische Register während der Lebenslücken im Intervall ungenutzt. Traub et al. [THS98] sowie Wimmer und Mössenböck [WM05] stellen Ansätze vor, bei denen ein *live range splitting* durchgeführt wird. Sie unterteilen die Lebendigkeiten der Register in kleinere Abschnitte, so dass die Lebenslücken von Registern genutzt werden können, die einen Abschnitt haben, der in die Lücke passt.

Der Vorteil von Linear Scan Allokatoren ist ihre geringe Laufzeit. Obwohl für eine genaue Bestimmung der Lebensintervalle eine Datenflussanalyse erforderlich ist, die eine quadratische Laufzeit haben kann, hat der Algorithmus eine fast lineare beobachte Laufzeit bezogen auf die Anzahl der Instruktionen in der Zwischendarstellung.

2.2.3.3 ILP-basierte Allokatoren

ILP-basierte Allokatoren transformieren das Problem der Registerallokation in ein ganzzahliges lineares Programm (*integer linear program* (ILP)). Die Überführung eines Ausgangsproblems in ein ILP hat den Vorteil, dass diese Transformation oft schneller implementiert werden kann als ein Lösungsverfahren, das genau auf das Ausgangsproblem zugeschnitten ist. Lösungsverfahren für ILP müssen nicht selbst implementiert werden. Es kann auf bestehende Werkzeuge, sogenannte ILP-Solver wie lp_solve [BEN08] oder CPLEX [ILO08], zurückgegriffen werden. Diese berechnen eine optimale Lösung für das ILP. Daraus kann dann eine optimale Lösung für das Ausgangsproblem abgeleitet werden.

Ein ILP [NM93] besteht aus drei Komponenten: Einer Menge von Entscheidungsvariablen, einer Zielfunktion und einer Menge von Nebenbedingungen.

Die Entscheidungsvariablen $x_1, \ldots, x_n \in \mathbb{Z}$ in einem ILP sind ganzzahlige Entscheidungsvariablen. Sie dürfen nur ganzzahlige nicht negative Werte annehmen. Variablen, deren Wertebereich nur auf die Werte Null und Eins beschränkt ist, werden als binäre Entscheidungsvariablen bezeichnet. Muss nur ein Teil der Entscheidungsvariablen ganzzahlige Werte annehmen $x_1, \ldots, x_r \in \mathbb{Z}$ und $x_{r+1}, \ldots, x_n \in \mathbb{R}$, so liegt ein *mixed integer linear program* (mILP) vor.

Auf der Menge der Entscheidungsvariablen ist die Zielfunktion definiert. Sie ist eine lineare Funktion $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} c_i\, x_i$. Die Werte der einzelnen Entscheidungsvariablen sind in der Zielfunktion mit Koeffizienten $c_1, \ldots, c_n \in \mathbb{R}$ gewichtet. Das Ziel bei der Lösung des ILP ist, Werte für alle Entscheidungsvariablen so zu bestimmen, dass der Wert der Zielfunktion minimiert wird. Die Werte, die die Entscheidungsvariablen annehmen, können aber nicht vollkommen frei gewählt werden. Sie müssen Nebenbedingungen der Form $\sum_{j=1}^{n} a_{i,j}\, x_j = b_i, (i = 1, \ldots, m)$ erfüllen. Diese Beschränkungen werden auch als Constraints bezeichnet. Die Koeffizienten sind ebenso wie in der Zielfunktion Konstanten aus dem Bereich der reelen Zahlen. Gleiches gilt für die Konstanten $b_1, \ldots, b_m$ auf den rechten Seiten der Gleichungen.

Für eine bessere Lesbarkeit wird in dieser Diplomarbeit bei der Angabe von Constraints jede Variable weggelassen, die in dem Constraint eine Null als Koeffizienten hat. Zudem werden die Constraints nicht immer als Gleichungen angegeben sein, sondern oftmals auch als Ungleichungen. Die Ungleichungen können in Gleichungen überführt werden und stellen keine Erweiterung des ILP-Modells dar.

$$\sum_{j=1}^{n} a_{i,j}\, x_j \qquad\quad \leq b_i \tag{2.1}$$

$$\sum_{j=1}^{n} a_{i,j}\, x_j \quad + y_j \quad = b_i \tag{2.2}$$

Die Ungleichung 2.1 ist äquivalent zu der Gleichung 2.2, bei der sich die linke Seite durch eine zusätzliche Variable von der linken Seite der Ungleichung unterscheidet. Diese neue Variable wird als Schlupfvariable bezeichnet. Ihr Wert ist der Schlupf der Ungleichung. Dies ist der Abstand, den die linke Seite der Ungleichung zum Wert auf der rechten Seite hat. Für Ungleichungen mit der Relation $\geq$ muss nur das Vorzeichen der Schlupfvariablen umgekehrt werden. Ebenso kann durch das Umkehren der Vorzeichen alle Koeffizienten in der Zielfunktion erreicht werden, dass die eigentliche Zielfunktion maximiert wird.

$$min\ -f(x_1, \ldots, x_n) \equiv max\ f(x_1, \ldots, x_n)$$

Das Lösen von ILP ist ein NP-schweres Problem. Die Laufzeit von Lösungsverfahren kann also exponentiell bezüglich der Anzahl der Variablen und Constraints sein. Für viele Anwendungsfälle verhält sich die Laufzeit aber gutartig. Meist werden Lösungsverfahren eingesetzt, die auf dem Simplex-Algorithmus und dem Branch-Bound-Prinzip beruhen. Sie können notfalls nach einer Zeitspanne abgebrochen werden. Dann wurde zwar vielleicht noch nicht die optimale Lösung gefunden, aber vielleicht schon eine gute gültige Lösung, die alle Nebenbedingungen erfüllt.

Wird das Registerallokationsproblem durch ein ILP beschrieben, werden die Bedingungen, die eine korrekte Registerallokation erfüllen muss, mit Hilfe der Constraints beschrieben. Das Gütekriterium der Allokation, wie z. B. Codegröße oder durchschnittliche Laufzeit, wird durch die Zielfunktion formuliert. Aus der Lösung des ILP kann dann unmittelbar die Lösung für die Registerallokation abgeleitet werden.

Der erste ILP-basierte Registerallokator wurde von Goodwin und Wilken [GW96] vorgestellt. Auf Grund der Güte des Allokators bezeichnen sie ihn als optimalen Registerallokator (ORA). Ihre Modellierung des Registerallokationsproblems ist Grundlage des

in dieser Diplomarbeit entwickelten Allokators und wird daher in Kapitel 4 ausführlich vorgestellt.

Kong und Wilken [KW98] erweitern das Modell, so dass auch die besonderen Merkmale von irregulären Architekturen wie z. B. überlappende Register und verkürzte Befehlscodes berücksichtigt werden können.

Fu und Wilken [FW02] stellen fest, dass bei der Allokation einige Entscheidungen getroffen werden, die keinen Einfluss auf die Güte des resultierenden Codes haben. Werden die dazugehörigen Entscheidungsvariablen weggelassen, kann das ILP deutlich schneller gelöst werden. Sie nennen ihren besserten Lösungsansatz daher *Fast ORA*. Ist die beobachte Laufzeit vor der Vereinfachung $\mathcal{O}(n^3)$, beträgt sie bei dem reduzierten Modell $\mathcal{O}(n^{2,5})$, wobei n die Anzahl der Instruktionen in der Funktion ist, auf die die Allokation angewendet wird.

Auch eine Trennung des Registerallokationsproblems in zwei Phasen führt zu einer erheblichen Beschleunigung [AG01]. Die beobachtete Laufzeit wird mit $\mathcal{O}(n^{1,3})$ angegeben. In der ersten Phase wird mittels eines ILP ermittelt, welche virtuellen Register im Registersatz untergebracht werden können, und welche in den Speicher ausgelagert werden müssen. Eine Zuordnung der virtuellen Register zu einzelnen Registern erfolgt erst in der zweiten Phase. Durch die Aufteilung des Registerallokationsproblems in diese zwei Teilprobleme ist aber nicht mehr garantiert, dass die Lösung optimal für das Gesamtproblem ist.

2.2.3.4 Weitere Registerallokationsverfahren

Ein weiteres globales Allokationsverfahren, mit dem das Registerallokationsproblem optimal gelöst werden kann, ist das Verfahren von Scholz und Eckstein [SE02]. Sie führen das Problem der Registerallokation auf ein *partitioned boolean quadratic optimization problem* (PBQP) zurück. Da die Bestimmung einer optimalen Lösung zeitintensiv ist, das PBQP ist NP-schwer, schlagen sie vor, das PBQP mit einer Heuristik zu lösen, die auf dynamischer Programmierung basiert. Die Laufzeit dieses Lösungsverfahren entspricht ungefähr der eines Graphfärbungsallokators. Die Güte der Ergebnisse wird ebenfalls durch einen Vergleich mit einem Graphfärbungsallokator bewertet. Sie kann je nach untersuchtem Beispiel um 10% besser sein, aber auch Verschlechterungen um fast 3% sind möglich.

Koes und Goldstein [KG06] formulieren das Registerallokationsproblem mit Hilfe von einem Flussnetzwerkproblem, genauer mit einem *multi-commodity network flow problem* (MCNF). Auch dieses Problem ist NP-schwer. Die Modellierung hat den Vorteil, dass schnell eine initiale Lösung bestimmt werden kann, die dann mit Hilfe eines allgemeinen Optimierungsverfahren verbessert werden kann. Jederzeit kann die Suche nach einer besseren Lösung abgebrochen werden und aus der besten Lösung, die bis dahin gefunden wurde, eine gültige Lösung für das Registerallokationsproblem erstellt werden. Auf diese Weise kann zwischen der Laufzeit der Allokation und der Güte der Lösung abgewogen werden. Ein Nachteil der Modellierung des Registerallokationsproblem durch ein MCNF Problem ist, dass dieses kaum dazu geeignet ist, um besondere Architekturmerkmale von irregulären Architekturen zu berücksichtigen. Auch dauert die Bestimmung einer optimalen Lösung länger als bei den ILP-basierten Verfahren.

2.3 WCET (Worst-Case Execution Time)

Die maximale Ausführungszeit eines Programms (*worst-case execution time* (WCET)) ist im Bereich der Echtzeitsysteme eine wichtige Größe. In einem Echtzeitsystem muss jede Ausführung eines Programms innerhalb einer festgelegten Zeitspanne abgeschlossen sein. Um diese Forderung einzuhalten, muss garantiert werden, dass die WCET des Programms die vorgegebene Zeitspanne nicht überschreitet.

Puschner und Koza definierten die maximale Ausführungszeit eines Programms wie folgt:

Definition 2.8 *„Die maximale anwendungsspezifische Ausführungszeit eines Programms ist die Zeit, die maximal nötig ist, um dieses Programm in dem gegebenen Anwendungskontext auszuführen, vorausgesetzt, dass alle benötigten Ressourcen verfügbar sind, das Programm nicht unterbrochen wird und das Verhalten der Hardware bekannt ist.“[PK89, Definition 1]*

Die WCET hängt also zum einen von dem Programm selbst, von der Menge aller möglichen Eingaben, aber zum anderen auch von der konkreten Hardware ab, auf der das Programm ausgeführt wird. Der Einfluss anderer Programme und eines Schedulingverfahrens auf das Ausführungsverhalten des Programms wird durch die WCET nur eingeschränkt durch den Anwendungskontext berücksichtigt. Durch diesen ist festgelegt, welche anderen Programme vorher ausgeführt werden können und in welchen Zuständen sich die Hardware in Folge dessen befinden kann.

Die Voraussetzungen, die in der Definition von Puschner und Koza genannt werden, stellen sicher, dass die WCET eines Programms eindeutig ist. Der Begriff der WCET kann ebenso auf Programmfragmente wie Funktionen und Basisblöcke übertragen werden.

Wechselwirkungen, die auftreten, wenn das Programm unterbrochen und zwischenzeitlich ein anderes Programm ausgeführt wird, müssen zusätzlich zu der WCET berücksichtigt werden. Ein Beispiel für eine Wechselwirkung, die zwischen Programmen auftreten kann, ist das gegenseitige Verdrängen von Cache-Inhalten, das zum *cache related preemption delay* [NMR03] und damit zu einer Verlängerung der Ausführungszeiten führt.

Enthält die Hardware Komponenten, die ein nichtdeterministisches Verhalten aufweisen, müssen diese bei der Bestimmung der WCET durch Komponenten ersetzt werden, die sich deterministisch verhalten, und die Operationen in keinem Fall schneller ausführen als die nichtdeterministischen Komponenten. Werden z. B. Caches mit zufälliger Ersetzungsstrategie verwendet, muss immer dann, wenn ein Zugriff auf einen Cacheblock erfolgt, dessen Inhalt unbekannt ist, von einem Cache-Miss ausgegangen werden.

Unglücklischerweise kann für beliebige Programme nicht entschieden werden, ob es überhaupt eine WCET gibt, oder ob das Programm nicht terminiert. Dies käme dem Lösen des Halteproblems gleich, das nicht entscheidbar ist (siehe [Weg93]). Auch falls es die WCET gibt, ist es schwierig, die WCET genau zu bestimmen, da die Menge der möglichen Eingaben sehr groß und die Hardware sehr komplex sein kann. Aus diesem Grund wird fast immer nicht die tatsächliche maximale Ausführungszeit eines Programms ermittelt, sondern nur eine obere Schranke geschätzt. Häufig wird diese obere Schranke ebenfalls als WCET bezeichnet. Wird in dieser Arbeit der Begriff WCET verwendet, ist damit fortan ebenfalls die geschätzte WCET gemeint. Ist eine deutliche Unterscheidung zwischen der geschätzten und der tatsächlichen WCET nötig, wird die tatsächliche WCET als WCET_{real} und die geschätzte WCET als WCET_{est} bezeichnet.

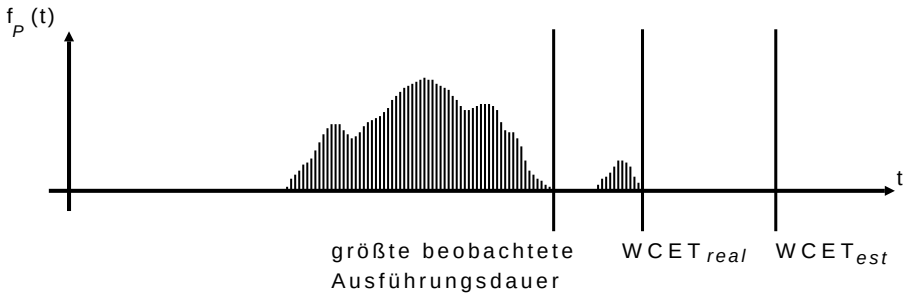

Abbildung 2.2: Beziehungen zwischen WCET_{real}, WCET_{est}, der maximalen beobachteten Ausführungsdauer und der Verteilungsdichtefunktion $f_P(t)$ der Ausführungszeiten eines Programms nach [WEE+08].

Die geschätzte WCET wird nach zwei Gesichtspunkten bewertet. Zum einen muss es sich wirklich um eine obere Schranke handeln. Gibt es auch nur eine Eingabe, die zu einer längeren Ausführungsdauer führt, als durch die WCET angegeben wird, ist sie in der Regel unbrauchbar. Eine geschätzte WCET, die eine obere Schranke für die tatsächliche WCET ist, wird als sicher bezeichnet. Zum anderen soll die WCET eine möglichst kleine obere Schranke für die tatsächliche WCET sein. Je kleiner der Abstand zwischen der geschätzten und der tatsächlichen WCET ist, desto vorteilhafter ist dies für den Entwurf von Echtzeitsystemen.

Abbildung 2.2 verdeutlicht nochmal den Zusammenhang zwischen den Ausführungszeiten eines Programms und der WCET. In diesem Beispiel ist für jede Ausführungszeit aufgetragen, wie oft die Ausführung eines Programms genau diese Zeit erfordert. Die tatsächliche WCET ist die kleinste obere Schranke für die Ausführungsdauer eines Programms. Alle geschätzten WCET mussen mindestens so groß sein die tatsächliche WCET, damit sie als sicher bezeichnet werden können. Dies trifft auf die eingezeichnete WCET_{est} zu.

Bei dem Entwurf erfolgt die Auslegung des Systems an Hand der geschätzten WCET. In dem Echtzeitsystem muss jede Ausführung des Programms innerhalb einer festgelegten Zeitspanne abgeschlossen sein. Ist das Programm und die Menge der möglichen Eingaben gegeben, muss die Hardware so entworfen werden, dass sie so viele Ressourcen bereitstellt und sie eine so hohe Leistung aufweist, dass die WCET die vorgegebene Zeitspanne nicht übersteigt. Je größer der Abstand zwischen der geschätzten und der tatsächlichen WCET ist, desto länger bleiben die Ressourcen und die Leistung des Systems ungenutzt. Kann eine bessere sichere Schranke für die tatsächliche WCET angegeben werden, sind somit Einsparungen bei der Hardware möglich.

Neben der Möglichkeit, eine bessere WCET durch eine genauere Analyse zu ermitteln, ist auch die Modifikation des Programms in Betracht zu ziehen. Da Einsparungen bei der Hardware von der geschätzten WCET abhängen, sind Optimierungen immer dann sinnvoll, wenn sie zu einer kleineren WCET führen, auch wenn ihre Auswirkungen auf die tatsächliche WCET unklar sind. Aus diesem Grund wird in dieser Diplomarbeit die erstellte Registerallokation bezüglich ihres Einflusses auf die geschätzte WCET untersucht. Die tatsächliche WCET ist schließlich im Allgemeinem unbekannt.

Für die Bestimmung der WCET gibt es zwei Ansätze: die dynamische und die statische Analyse. Bei der dynamischen Analyse wird das Programm auf der Hardware ausgeführt

und die Laufzeit gemessen. Bei diesem Vorgehen kann die maximale Ausführungszeit für jeweils eine einzelne Eingabe sehr genau ermittelt werden. Normalerweise ist aber die entscheidene Eingabe, die zu der maximalen Ausführungszeit führt und daher *worst-case* Eingabe genannt wird, nicht bekannt. Auch ist die Menge der möglichen Eingaben in der Regel zu groß, um die Ausführungszeiten für jede Eingabe zu ermitteln. Aus diesem Grund muss die Analyse auf eine repräsentative Menge von Mengen beschränkt werden. Befindet sich unter den Eingaben nicht die worst-case Eingabe, ist die größte gemessene Ausführungszeit jedoch nur eine untere Schranke für die WCET_{real} und kann keinesfalls als sicher angesehen werden, wie in Abbildung 2.2 zu erkennen ist. Die größte bei einer Messung beobachte Ausführungszeit wird daher noch um einen Anteil erhöht, in der Hoffnung, dass dieser Sicherheitspuffer für die praktische Anwendung ausreicht.

Bei der statischen Analyse wird eine WCET_{est} ermittelt, indem der Programmcode untersucht wird. Hier werden alle möglichen Eingaben dadurch berücksichtigt, dass keine Annahmen über die Eingabe getroffen werden. Dies betrifft unmittelbar zwei Teilaspekte der Analyse. Sowohl der Kontrollfluss des Programms als auch der Zustand der Hardware hängen von der Eingabe des Programms ab.

Da auf Grund der Unentscheidbarkeit des Halteproblems der Kontrollfluss für beliebige Programme nicht vorhergesagt werden kann, muss der Nutzer der Analyse zusätzliche Informationen über den Kontrollfluss bereitstellen. Solche Informationen werden als *flow facts* bezeichnet. Durch sie werden untere und obere Schranken für die Ausführungshäufigkeit von Programmabschnitten angegeben. Dies können absolute Werte sein, sie können aber auch durch Relationen zu den Ausführungshäufigkeiten anderer Programmabschnitte ausgedrückt werden. Auf diese Weise lassen sich die Anzahl von Schleifenwiederholungen und die Tiefe von Rekursionen beschränken, und auch *infeasible paths* können angegeben werden.

Um eine Ausführungszeit ermitteln zu können, wird neben dem Programmcode und den *flow facts* noch ein Modell der Hardware benötigt, das alle für die Ausführungszeit relevanten Eigenschaften möglichst genau wiedergibt. Dazu zählt für Modelle moderner Prozessoren insbesondere das Verhalten von Pipelines, Caches und anderer Speicher [NR95]. Je genauer das Modell ist, desto genauer kann auch das Ergebnis der Analyse sein[1]. Alle Ungewissheiten, die bei der statischen Analyse auftreten, müssen pessimistisch abgeschätzt werden, um eine in jedem Fall sichere WCET zu erhalten.

Weil die statische Analyse nur an Hand des Programmcodes, der *flow facts* und des Hardwaremodells durchgeführt werden kann, kann sie schon vorab, bevor die Hardware real existiert, vorgenommen werden. Aus dem gleichem Grund kann sie bereits in Compilern angewendet werden und dazu genutzt werden, um effizienten Code zu generieren.

Einen Überblick über verschiedene Analyseverfahren und Analysewerkzeuge geben Wilhelm et al. [WEE+08]. Von besonderem Interesse ist dabei das Analysewerkzeug aiT der Firma AbsInt Angewandte Informatik GmbH [HF06], das in dieser Diplomarbeit für die Bestimmung von WCETs verwendet wurde. Es wird noch in Abschnitt 3.4 vorgestellt.

Ein Teilproblem bei der statischen Analyse der WCET ist die Ermittlung des Kontrollpfades mit der längsten Ausführungsdauer. Dieser Pfad wird als *worst-case execution path* (WCEP) bezeichnet.

[1] Bei der dynamischen Analyse kann dieses Problem dadurch gelöst werden, dass ein genaues Modell in Form einer technischen Realisierung für die Laufzeitmessung verwendet wird.

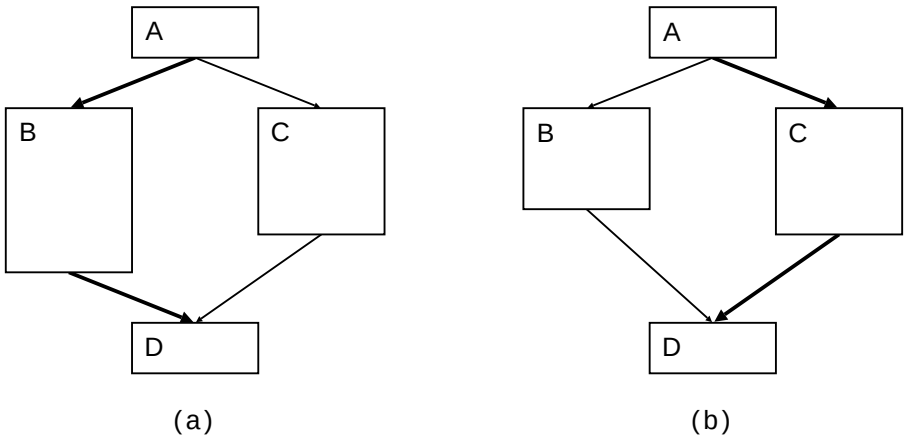

Abbildung 2.3: Beispiel für die Änderung des WCEP in Folge von Optimierungen. (a) WCEP vor der Verkürzung der Ausführungszeit von Basisblock B. (b) WCEP nach der Verkürzung der Ausführungszeit von Basisblock B.

Definition 2.9 *Der* worst-case execution path *ist der Pfad im Kontrollflussgraphen des Programms, der bei der* worst-case *Eingabe ausgeführt wird.*

Somit gibt es eine direkte Beziehung zwischen der WCET_{real} und dem WCEP eines Programms. WCET_{real} ist die Ausführungszeit des WCEPs.

Sollen Optimierungen an dem Programm vorgenommen werden, um die WCET zu senken, so sind diese nur dann wirkungsvoll, wenn sie die Ausführungszeit von Programmabschnitten auf dem WCEP verkürzen. In Folge der Veränderungen am Programm kann sich der WCEP wiederum ändern. Abbildung 2.3 zeigt hierfür ein Beispiel. Dargestellt zwei ein Kontrollflussgraphen, bei dem Instruktionsfolgen zu Basisblöcken zusammen gefasst wurden. Die Größe der Knoten in dem Kontrollflussgraphen sei proportional zur maximalen Ausführungsdauer der Basisblöcke. Die Kanten, die den Kontrollfluss des WCEP wiedergeben, sind durch eine größere Linienbreite hervorgehoben. In dem linken Graphen ist der WCEP der Pfad, auf dem die Basisblöcke A, B und D liegen. Die Verkürzung der Ausführungszeit von Basisblock B führt zu dem Graphen, der rechts dargestellt ist. Auch wenn sich an der Struktur des Graphen nichts geändert hat, liegt B nicht mehr auf dem WCEP. Eine weitere Verkürzung der Ausführungszeit von Basisblock B hat keine Auswirkungen mehr auf die WCET.

Geht eine Verkürzung der Ausführungszeit von Basisblock B zu Lasten der Ausführungszeit von Basisblock C, kann die WCET sogar steigen. Dies kann zum Beispiel bei der Registerallokation eintreten. Hier kann die Wahlmöglichkeit bestehen, entweder in Basisblock B oder C Spillcode einzufügen. Eine gleichmäßige Verteilung könnte die beste Lösung sein, eine komplette Verlagerung in einen der beiden Basisblöcke sich dagegen nachteilig auswirken.

Kapitel 3

WCC (WCET-aware C-Compiler)

In einem eingebetteten System sind die Hardware und die Software eng aufeinander abgestimmt. Um Hardwareressourcen einzusparen und damit Kosten bei Produktion und Betriebe des Systems zu senken, ist die Erzeugung effizienter Software unumgänglich. An Compiler für den Bereich der eingebetteten Systeme wird daher die Anforderung gestellt, dass sie Code erzeugen, der eine geringe durchschnittliche Ausführungsdauer, eine geringe Codegröße und einen geringen Energieverbrauch aufweist. Häufig sind eingebettete Systeme auch Echtzeitsysteme. Dann ist auch die WCET ein Optimierungsziel. In jedem Fall sind präzise Analysen notwendig, damit der Compiler während des Übersetzungsprozesses Entscheidungen so treffen kann, dass sie die Güte der erzeugten Software positiv beeinflussen.

Der WCC (WCET-aware C-Compiler) [FLT06] ist der erste Compiler, der für die Analyse ein professionelles Analysewerkzeug in den Übersetzungsprozess einbezieht, um WCET-Daten zu gewinnen. In diesem Fall handelt es sich um das WCET-Analysewerkzeug aiT der Firma AbsInt Angewandte Informatik GmbH [HF06]. Der WCC selbst wurde am Lehrstuhl 12 der Fakultät für Informatik der Technischen Universität Dortmund von der Forschungsgruppe für eingebettete Systeme entwickelt. Er übersetzt Programme, die in der Programmiersprache C beschrieben sind, in Maschinencode für den Infineon TC1796 32 Bit-Mikrocontroller [Inf08], der bislang die einzige Zielarchitektur des Compilers ist.

Im folgenden Abschnitt 3.1 wird nun zunächst der Aufbau des WCC vorgestellt. Anschließend wird die vom WCC verwendete *low-level intermediate representation* (LLIR) vorgestellt. Sie ist das Eingabe- und Ausgabeformat des Registerallokators. In Abschnitt 3.3 wird erläutert, wie der Programmierer mit Hilfe von Flow Facts zusätzliche Informationen für die WCET-Analyse bereitstellt. Diese wird vom Analyse-Tool aiT durchgeführt, das in Abschnitt 3.4 beschrieben wird. Da die Registerallokation eine hardwarenahe Optimierung ist, werden schließlich in Abschnitt 3.5 die wesentlichen Merkmale der Zielarchitektur vorgestellt.

3.1 Aufbau des WCC

Der klassische Aufbau eines Compilers gleicht dem Softwaremodell eines Fließbandes. Für die Übersetzung eines Programms aus einer Hochsprache in Maschinencode führt der Compiler der Reihe nach einzelne Übersetzungsschritte aus, wobei die Ausgabe des einen Übersetzungsschrittes die Eingabe des folgenden Schrittes ist. Die Struktur des Compilers kann in mehrere Bereiche unterteilt werden [ASU86]. In dem ersten Teil, *Front-End* genannt, wird das Programm aus der Hochsprache durch Anwendung der

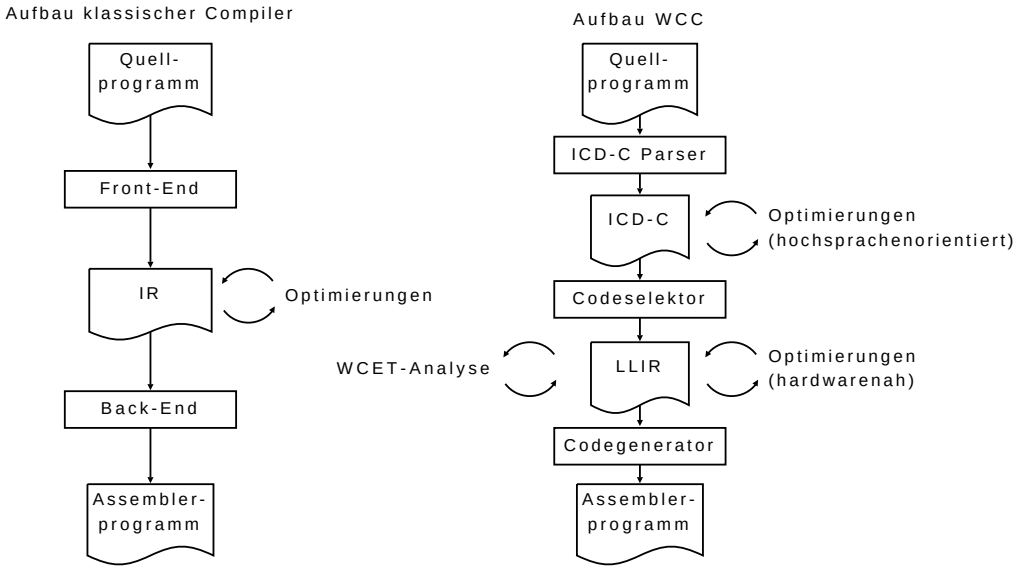

Abbildung 3.1: Vereinfachte Darstellung des Aufbau eines klassischen Compilers und des WCC.

Übersetzungsschritte in eine Zwischendarstellung (*intermediate representation* (IR)) überführt. Auf dieser werden allgemeine Optimierungen angewendet, die das Ziel verfolgen, das Programm hinsichtlich eines Gütekriteriums zu verbessern. Anschließend wird aus der IR durch eine weitere Folge von Übersetzungschritten, die das *Back-End* bilden, Assemblercode für die Zielarchitektur generiert. Ein Assembler und ein Linker erzeugen aus dem Assemblercode schließlich das ausführbare Maschinenprogramm.

Die verwendete IR ist bei diesem Aufbau nach Möglichkeit sowohl unabhängig von der Hochsprache als auch von der Zielarchitektur. Das Ziel ist, dass nur durch Austausch des *Front-Ends* der Compiler Programme aus anderen Hochsprachen in Maschinencode für dieselbe Zielarchitektur erzeugen kann. Weiterhin soll durch den Austausch des *Back-Ends* Maschinencode für andere Zielarchitekturen erzeugt werden können. Häufig verfügt ein Compiler über mehrere *Front-Ends*, in denen die Übersetzungschritte jeweils an eine Hochsprache angepasst sind. Je nach der vorliegenden Hochsprache wird bei der Übersetzung das passende Front-End ausgewählt und das Programm in die IR überführt. Ebenso stehen dem Compiler in der Regel mehrere *Back-Ends* zur Verfügung. In jedem sind die Übersetzungsschritte an die Gegebenheiten der Zielarchitektur angepasst. Damit *Front-End* und *Back-End* unabhängig voneinander gewählt werden können, müssen die Übersetzungsschritte im *Front-End* unabhängig von der Zielarchtektur erfolgen, die Schritte im *Back-End* unabhängig von der Hochsprache.

Der Aufbau des WCC folgt nicht in allen Punkten diesem klassischen Aufbau eines Compilers. Abbildung 3.1 stellt den klassischen Aufbau und den Aufbau des WCC nach [FLT06] gegenüber. Als Compiler für eingebettete Systeme steht nicht die Retargierbarkeit des Compilers im Mittelpunkt, sondern die Erzeugung hoch effizienten Codes. Aus diesem Grund verwendet der WCC zwei Zwischendarstellungen, die beide nicht sowohl programmiersprachen- als auch architekturunabhängig sind.

Zunächst wird das Quellprogramm aus der Programmiersprache C in eine *high-level intermediate representation* (HIR) überführt. Dabei handelt es sich um ICD-C [ICD06], das

vom Informatik Centrum Dortmund (ICD) entwickelt wurde. Hierauf können Optimierungen ausgeführt werden, die gezielt Eigenschaften der Hochsprache ausnutzen. Auf der zweiten IR, der *low-level intermediate representation* (LLIR) [ICD07], die ebenfalls vom ICD stammt, können dann Optimierungen angewendet werden, die auf die Zielarchitektur ausgerichtet sind. Die Verbindung zwischen ICD-C und der LLIR stellt der Codeselektor her. Er überführt ICD-C in die LLIR und ist daher sowohl von der Hochsprache als auch von der Zielarchitektur stark geprägt.

Die Registerallokation, die Thema dieser Diplomarbeit ist, erfolgt unabhängig von der Hochsprache, in der das Programm formuliert ist. Sie benötigt dagegen genaue Informationen über den Registersatz der Zielarchitektur und ist daher eine hardwarenahe Optimierung. Sie ist in dem Aufbau des WCC nach dem Codeselektor einzuordnen. Ihre Aufgabe ist, in der LLIR alle Vorkommen von virtuellen Registern durch die Bezeichnungen physikalischer Register zu ersetzen, ohne dabei die Semantik des Programms zu ändern. Alle auf die Registerallokation folgenden Optimierungen dürfen keine virtuellen Register mehr verwenden. Die Verwendung virtueller Register erleichtert aber die Ausführung von Optimierungsschritten. Daher wird die Registerallokation als eine der letzten Optimierungen vor der Erzeugung des finalen Maschinenprogramms ausgeführt.

Für eine LLIR, die nur noch Bezeichnungen von physikalischen Registern enthält, kann der WCC eine WCET-Analyse durchführen [Lok05]. Das Analysewerkzeug, mit dem die Analyse durchgeführt wird, ist aiT. Der Ablauf einer WCET-Analyse wird in Abschnitt 3.4 beschrieben. Durch die Verwendung eines professionellen WCET-Analysewerkzeuges an Stelle einer proprietären Lösung können vergleichsweise genaue Ausführungszeiten ermittelt werden. Diese Informationen können dann genutzt werden, um das Ergebnis der bisher vorgenommenen Übersetzungsschritte zu bewerten und gegebenenfalls erneut auszuführen, um ein hoffentlich besseres Ergebnis zu erzielen. Prinzipiell ist dieses Vorgehen, ein Programm zu übersetzen, eine WCET-Analyse auszuführen und mit den Informationen erneut zu übersetzen, auch mit anderen Compilern möglich.

Das Problem dabei ist, dass sich die Ergebnisse der WCET-Analyse auf das Maschinenprogramm beziehen und sich nicht auf die Bestandteile des Programms in der Hochsprache übertragen lassen. Ein Compiler, der diese Daten importieren würde, könnte daraus nicht zuverlässig Entscheidungen ableiten, die tatsächlich die WCET auch verkürzen.

Der große Vorteil der Stuktur des WCC ist, dass durch die WCET-Analyse der LLIR die WCET-Daten vor dem Abschluss des Übersetzungsprozesses vorliegen und einzelnen Objekten in der LLIR zugeordnet werden können. Auf diese Weise lassen sich bessere Rückschlüsse auf die Übersetzungsschritte ziehen, die schon ausgeführt wurden. Folgende Übersetzungsschritte können die WCET-Daten der einzelnen Objekte in der LLIR für die Optimierung nutzen. Sie können Veränderungen in der LLIR vornehmen, eine WCET-Analyse der geänderten LLIR vornehmen und so die genauen Auswirkungen auf die WCET_{est} ermitteln. Auf diese Weise unterstützt der WCC gezielte Optimierungen der WCET.

3.2 LLIR (Low-Level Intermediate Representation)

Die ICD-LLIR [ICD07] ist eine Klassenbibliothek für die objektorientierte Darstellung von Assemblercode, die vom Informatik Centrum Dortmund (ICD) entwickelt wurde. Bei der *low level intermediate representation* (LLIR), die der WCC für hardwarenahe

Optimierungen verwendet, handelt es sich um eine konkrete Instanz der Klasse LLIR dieser Bibliothek, mir der der Assemblercode eines gesamten Programms repräsentiert wird. Durch Nutzung weiterer Klassen wird der Assemblercode in logische Abschnitte gegliedert. Er ist in Funktionen und dann weiter in Basisblöcke und einzelne Instruktionen unterteilt.

Mit Hilfe der Basisblock-Objekte wird gleichzeitig der Kontrollflussgraph einer Funktion repräsentiert. Jeder Basisblock hat Verweise auf die Basisblöcke in der Funktion, die unmittelbar vor und unmittelbar nach dem Basisblock ausgeführt werden könnten. Die Kontrollflussinformation für Instruktionen über die vorhergehende und die nachfolgende Instruktion in dem Basisblock ist ebenfalls verfügbar. So sind Traversierungen des Assemblercodes auf Basisblock- und Instruktionsebene möglich.

Eine Instruktion kann potenziell aus mehreren Mikrooperationen bestehen, die wiederum durch eine eigene Klasse repräsentiert werden, ebenso wie die Parameter einer Operation. Ein Parameter kann dabei eine ganzzahlige Konstante, eine Marke, ein Operator oder ein Register sein. Die Register, bei denen es sich sowohl um virtuelle als auch um physikalische Register handeln kann, werden ebenfalls durch eine eigene Klasse modelliert. Zusätzlich zu dem Typ und dem Wert bzw. Bezeichnung des Parameters ist angegeben, ob der Parameter definiert oder genutzt wird. Basierend auf diesen Informationen können durch Methoden der ICD-LLIR Analysen wie Def-Use- und Lebendigkeitsanalysen durchgeführt werden, die z. B. für die Registerallokation benötigt werden.

Alle direkten Beziehungen zwischen den Objekten werden durch Referenzen zwischen den Objekten dargestellt, die ein einfaches Navigieren in der Objekthierarchie erlauben. So können z. B. für ein Register die Operationen ermittelt werden, von denen es definiert oder genutzt wird, ohne den gesamten Assemblercode nach den Vorkommen des Registers absuchen zu müssen.

Ein weiterer Vorteil der Zwischendarstellung ist, dass zu jedem Objekt Zusatzinformationen gespeichert werden können. Dies sind z. B. für die Registerallokation Informationen zur Vorfärbung. Für Kontrollflussanalysen werden *flow facts* verwaltet, und nach einer WCET-Analyse können WCET-Daten einzelner Programmabschnitte gespeichert werden.

Obwohl die ICD-LLIR eine Repräsentation von Assemblercode ist, ist ihre Verwendung nicht auf eine Zielarchitektur beschränkt. Die grundlegende Struktur ist unabhängig von einer kokreten Zielarchitektur gestaltet worden. Die ICD-LLIR wird erst durch eine Beschreibungsdatei an die Architektur angepasst. Diese Datei enthält unter anderem Informationen zum Register- und zum Befehlssatz der Architektur.

3.3 Flow Facts

Flow facts beschreiben Eigenschaften der Kontrollflüsse eines Programms. Teilweise sind Analyseprogramme nicht in der Lage, die für die Analyse wesentlichen Eigenschaften der Kontrollflüsse aus einer Programmbeschreibung abzuleiten. Der Nutzer der Analyse muss dann selbst *flow facts* formulieren und der Analyse zur Verfügung stellen. Erst so ist z. B. eine WCET-Analyse möglich, die anderenfalls das Halteproblem lösen müsste, das für beliebige Programme unentscheidbar ist.

Ein Nutzer des WCC kann zwei Arten von flow facts explizit in Quellcode angeben [Sch07]. Zum einen können durch *loopbounds* eine untere und eine obere Schranke für die

Iterationshäufigkeit einer Schleife angegeben werden. Zum anderen können durch *flow restrictions* sehr allgemein Beziehungen zwischen den Ausführungshäufigkeiten einzelner Programmabschnitte spezifiziert werden. Durch sie kann die Rekursionstiefe einer Funktion beschränkt werden und die Ausführungshäufigkeit mehrerer Basisblöcke in ein Verhältnis gesetzt werden. Dadurch kann ebenfalls die Iterationshäufigkeit von Schleifen begrenzt, aber auch Informationen zu *feasible paths* formuliert werden.

Die *flow facts* werden durch den Nutzer im Quellcode notiert. Dort beziehen sie sich auf Sprachelemente der Programmiersprache C. Bei der Überführung des Quellprogramms in die Zwischendarstellungen ICD-C und LLIR müssen daher auch die *flow facts* auf die Zwischendarstellungen übertragen werden. Auch können Optimierungen die Zwischendarstellungen so verändern, dass die *flow facts* angepasst werden müssen. Aus diesem Grund verwaltet und transformiert der WCC die *flow facts* während des gesamten Übersetzungsprozesses. So stehen sie dann auch für eine WCET-Analyse der LLIR zur Verfügung.

3.4 WCET-Analyse mittels aiT

Das WCET-Analysewerkzeug aiT [HF06] führt eine statische WCET-Analyse für ein vollständiges Programm durch. Das Programm kann wahlweise als Binärdatei vorliegen oder durch die Control Flow Representation Language Version 2 (CRL2) [Abs06] beschrieben sein. Liegt es als Binärdatei vor, überführt aiT in einem ersten Schritt die Darstellung des Programms in die CRL2. Dazu decodiert aiT zum einen die im Binärformat codierten Instruktionen und rekonstruiert zum anderen den Kontrollflussgraphen.

Neben dem Programm benötigt aiT eine weitere Datei mit Zusatzinformationen. In ihr sind z. B. obere Schranken für die Rekursiontiefe für Funktionen und die Iterationshäufigkeit von Schleifen angegeben. Auch flow facts werden auf diese Weise aiT mitgeteilt.

Um eine möglichst kleine obere Schranke für die WCET des Programms zu bestimmen, führt aiT eine Reihe von Analyseschritten durch, die Ausführungskontexte berücksichtigen und auf dem Prinzip der abstrakten Interpretation basieren. Ein Ausführungskontext gibt an, in welchen Zuständen sich die Hardware zu einem bestimmten Zeitpunkt bei der Programmausführung befinden kann. Wird z. B. ein Basisblock mehrmals ausgeführt, weil er Teil einer Schleife ist, kann die Menge der Zustände, in dem sich die Hardware befindet, jedesmal eine andere sein. Da der Hardwarezustand unmittelbare Auswirkungen auf die Ausführungszeit der Programmbefehle hat, kann eine bessere WCET_{est} bestimmt werden, wenn die Menge der möglichen Hardwarezustände möglich stark eingegrenzt werden kann. Dazu kann aiT verschiedene Ausführungskontexte unterscheiden. Ein Ausführungskontext repräsentiert dann nicht alle Hardwarezustände, die vor der Ausführung eines Basisblocks eintreten können, sondern nur die Zustände, die vor einer bestimmten Ausführungen des Basisblocks vorkommen können. Z. B. kann es je einen Ausführungskontext für die erste und die zweite Ausführung geben, sowie ein weiteren, der für alle restlichen Ausführungen gilt.

Abstrakte Interpretation bezeichnet ein Verfahren, bei dem Operationen ausgewertet werden, obwohl die konkreten Werte der Operanden nicht bekannt sein müssen. Beispielsweise kann einer Variable x der Wert einer Variable a zugewiesen werden. Anchließend wird der Wert der Variablen x um drei erhöht. Auch wenn die Werte von x und a nicht bekannt sind, steht das Ergebnis eines Vergleichs der beiden Variablen a und x fest.

Die Reihe der Analyseschritte beginnt mit einer Werteanalyse. Ziel der Werteanalyse ist, für jede Stelle im Programm und für jeden Ausführungskontext die Werte, die in den Registern des Prozessors vorliegen, vorherzusagen. Zwar können oft nicht die genauen Werte der Register bestimmt werden, da dies eine Simulation der Programmausführung für alle Eingaben erfordern würde, aber die Bereiche, aus denen Werte angenommen werden können, werden eingeschränkt. In Folge dessen kann genauer vorhergesagt werden, auf welche Bereiche im Speicher durch Speicherinstruktionen mit indirekter Addressierung zugegriffen wird, d. h. durch Instruktionen, bei denen ein Register ein Operand bei der Berechnung der Adresse der Speicherstelle ist. Zusätzlich kann für einfache Schleifen vorhergesagt werden, wann deren Abbruchbedingung erfüllt ist, die ebenfalls von den Werten in den Registern abhängt.

Genau diese Analyse der Iterationshäufigkeiten führt aiT im nächsten Analyseschritt aus. Es versucht für einfache Schleifen eine obere Schranke für die Iterationshäufigkeit zu bestimmen. Wann immer dies aiT nicht gelingt, muss diese Information durch den Nutzer gegeben sein. Auf diese Weise kann die Ausführungshäufigkeit der Basisblöcke beschränkt werden.

In der folgenden Cache-Analyse werden Speicherzugriffe danach klassifiziert, ob sie in jedem Fall zu einem Cache Hit führen, oder ob keine gesicherte Aussage über die Qualität des Zugriffs getroffen werden kann. Würden alle Cachezugriffe vereinfacht als Cache Misses angenommen werden, könne dies zu einer Überabschätzung der WCET um Größenordnung von mehreren Vielfachen führen.

In der Pipeline-Analyse werden dann die WCETs der Basisblöcke bestimmt. Durch einen Ausführungskontext ist die Menge der möglichen Hardwarezustände vor der Ausführung des Basisblocks gegeben. Nach und nach werden die Auswirkungen der Ausführung jeder Instruktion des Basisblocks ähnlich wie bei in einer zkylengenauen Simulation untersucht. Dabei werden die Ergebnisse aus der Werte- und Cache-Analyse sowie das Pipelineverhalten des Prozessors berücksichtigt. Die Ausführung der Instruktion kann je nach Ausgangssituation unterschiedlich lange dauern und zu unterschiedlichen Folgezuständen führen. Die Menge aller möglichen Folgezustände bildet dann die Ausgangsmenge für die Untersuchung der nächsten Instruktion. Die WCET ist dann eine obere Schranke für die Anzahl der Prozessorzyklen, die von den Instruktionen des Basisblocks benötigt werden. Die WCET des Basisblocks wird auf diese Weise für jeden Kontext einzeln bestimmt.

Abschließend wird mit der Pfad Analyse der WCEP im Programm ermittelt. Hierzu wird die *implicit path enumeration technique* (IPET) [LM95] angewendet. In einem ILP wird der Kontrollflussgraph des Programms durch Constraints modelliert. Für jeden Basisblock wird ein Constraint aufgestellt, das sicherstellt, dass ein Kontrollfluss den Basisblock über die eingehenden Kanten in der Summe genauso oft erreicht, wie er ihn über die ausgehenden Kanten verlässt. Jene Summe ist die Ausführungshäufigkeit des Basisblocks auf dem WCEP. Für den ersten Basisblock einer Funktion und die Basisblöcke, über die die Funktion verlassen werden kann, werden andere Constraints formuliert, weil sie keine Vorgänger bzw. Nachfolger in der Funktion haben. Für den ersten Basisblock ist die Ausführungshäufigkeit auf Eins festgesetzt und der Kontrollfluss verlässt diesen Basisblock genau einmal. Für die Basisblöcke, die keine Nachfolger in der Funktion haben, muss kein Constraint aufgestellt werden. Ein Kontrollfluss, der diese Basisblöcke erreicht, kann sie nicht mehr verlassen. Mit weiteren Constraints werden zusätzliche Bedingungen an den Kontrollfluss gestellt, wie sie z. B. durch *flow facts* formuliert sind. Die Zielfunktion des ILPs ist die WCET des Programms. Sie enthält für jeden Basis-

block das Produkt aus seiner Ausführungshäufigkeit und seiner WCET. Alle Produkte in der Zielfunktion sind durch die Addition verknüpft. Die Maximierung der Zielfunktion erzwingt, dass der durch die Ausführungshäufigkeiten der Basisblöcke implizit modellierte Kontrollfluss den maximalen Wert für die WCET annimmt. In der Lösung des ILPs sind die Ausführunghäufigkeiten der Basisblöcke so gewählt, dass die geschätzte WCET von keiner geschätzten Ausführungsdauer eines anderen möglichen Ausführungspfades übertroffen wird. Da die WCETs der einzelnen Basisblöcke sicher sind, ist auch die so ermittelte WCET für das gesamte Programm sicher.

Damit der WCC eine WCET-Anylse mit Hilfe von aiT durchführen kann, muss er zunächst das Programm in eines der beiden Eingabeformate überführen. Die CRL2 ist ähnlich wie die LLIR eine Zwischendarstellung für Assemblercode. Dies nutzt der WCC. Er ist in der Lage, zu einer LLIR eine CRL2 Instanz erzeugen und braucht daher für die Analyse der LLIR keine ausführbare Datei zu erstellen. Allerdings kann die LLIR nur dann analysiert werden, wenn sie bereits ausführbaren Assemblercode repräsentiert. Dies bedeutet insbesondere, dass die LLIR keine virtuellen Register enthalten darf. Es muss also schon eine Registerallokation stattgefunden haben.

Der WCC konvertiert die LLIR in eine CRL2 Instanz [Lok05] und startet aiT. Dieses gibt nicht nur die WCET des Programms aus, sondern auch wieder eine CRL2 Instanz, in der nun aber zusätzlich WCET-Daten für die einzelnen CRL2-Elemente vorhanden sind. Der WCC überträgt anschließend an die WCET-Analyse die WCET-Daten auf die Objekte der LLIR, wo sie dann den Optimierungen des WCC zur Verfügung stehen.

3.5 Zielarchitektur

Der TC1796 Mikrocontroller von Infineon [Inf08] ist für den Einsatz in eingebetteten Echtzeitsystemen entwickelt worden. Er implementiert eine 32 Bit TriCore v1.3 Architektur [Inf05a], die sowohl charakteristische Merkmale eines RISC-Prozessors, eines Mikrocontrollers als auch eines digitalen Signalprozessors (DSP) aufweist. Dies prägt vor allem den Befehlssatz, dessen Merkmale in Abschnitt 3.5.2 beschrieben werden. Zunächst aber wird in Abschnitt 3.5.1 der Registersatz beschrieben, dessen Eigenschaften zweifellos durch die Registerallokation berücksichtigt werden müssen und bestmöglichst genutzt werden sollten. Obwohl die Befehlsausführung in den Pipelines von einer Registerallokation gewöhnlich nicht berücksichtigt wird, werden die Pipelines des TriCores in Abschnitt 3.5.3 beschrieben, da einige Eigenschaften der Pipelines für die Bestimmung der WCET-Daten berücksichtigt werden. Abschließend wird in Abschnitt 3.5.4 die Speicherhierarchie des TC1796 kurz vorgestellt. Auf weitere Merkmale des TC1796 wie Interruptbehandlung und Perpherie wird hier nicht eingegangen, da diese bei der Registerallokation nicht berücksichtigt werden.

3.5.1 Registersatz

Der Registersatz der Infineon TriCore Architektur besteht aus insgesamt 32 *general purpose*-Registern, die jeweils eine Breite von 32 Bit haben. Um die gleichzeitige Ausführung von Daten- und Adressoperationen zu erleichtern, ist der Registersatz in zwei Hälften unterteilt. Die 16 Register A[0] bis A[15] können nur als Adressregister verwendet werden, und die 16 Register D[0] bis D[15] nur als Datenregister. Es handelt sich daher um einen irregulären Registersatz, da nicht jedes Register die Funktion eines beliebigen anderen

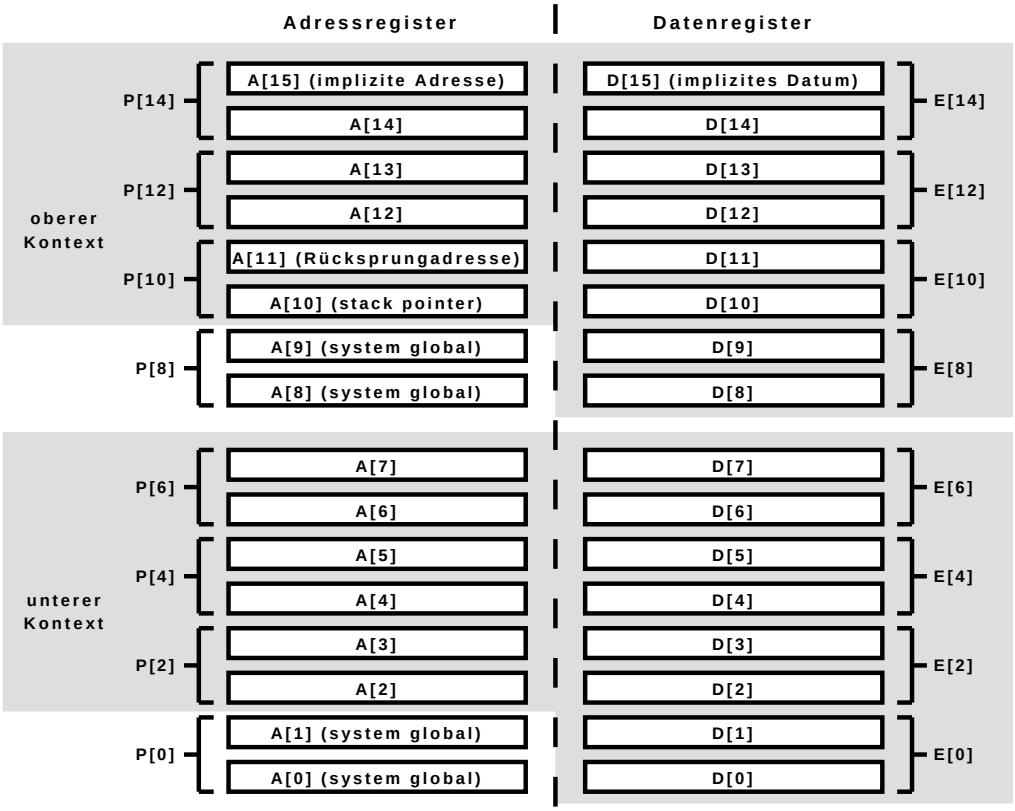

Abbildung 3.2: Struktur des Registersatzes der Infineon 32 Bit TriCore v1.3 Architektur nach [Inf05a].

Registers übernehmen kann. Spezielle Register für das Speichern von Gleitkommazahlen gibt es nicht. Hierfür müssen ebenfalls die Datenregister verwendet werden.

Zwei benachbarte Daten- oder zwei benachbarte Adressregister können auch als ein 64 Bit Register verwendet werden, wenn das eine Register einen geraden Index hat und das andere Register einen um eins größeren Index. Auf diese Weise ist ausgeschlossen, das sich erweiterte Register überlappen. Abbildung 3.2 zeigt eine Übersicht über den Registersatz und verdeutlicht, aus welchen 32 Bit Registern die erweiterten Register bestehen. Die Bezeichnungen der erweiteren Adressregister beginnen mit einem P, die der Datenregister mit einem E. Der Index des Registers ist der kleinere, gerade Index des Registerpaares. Z. B. bilden D[2] und D[3] das erweiterte Register E[2].

Einige Register übernehmen eine festgelegte Funktion. Das Register A[10] speichert den Wert des *stack pointer*. In Register A[11] ist die Rücksprungadresse vom letzten Funktionsaufruf abgelegt. Die Register A[15] und D[15] können implizit von Befehlen im verkürzten Befehlsformat verwendet werden. Da das verkürzte Befehlsformat nur 16 Bit umfasst, wurden Bits für die Codierung der verwendeten Register eingespart.

Die Register A[0], A[1], A[8] und A[9] werden auch als *system global*-Register bezeichnet und sind gewöhnlich einem Betriebssystem vorbehalten. Sie sollen helfen, den Aufwand für die Kommunikation zwischen Prozessen zu senken. Um auch den Aufwand bei Funktionsaufrufen zu senken, unterstützt der TriCore schnelle Kontextwechsel durch das gleichzeitige Speichern und Laden mehrerer Register. Dazu ist der Registersatz in zwei

Kontexte unterteilt, die jeweils durch einen einzelnen Befehl komplett gespeichert oder geladen werden können.

Die Addressregister A[10] bis A[15] und die Datenregister D[8] bis D[15] gehören zum oberen Kontext. Sie werden bei einem Funktionsaufruf, der mittels eines *call*-Befehls ausgeführt wird, automatisch gesichert. Bei einem Rücksprung, der durch einen *ret*-Befehl realisiert wird, werden die Inhalte der Register bei der Ausführung des Befehls wiederhergestellt.

Die Register A[2] bis A[7] und D[0] bis D[7] gehören dem unteren Kontext an. Sie werden bei Funktionsausrufen und Rücksprüngen nicht durch den *call*-Befehl gesichert und den *ret*-Befehl wiederhergestellt. Auf diese Weise eignen sich diese Register für die Übergabe von Parametern an Funktionen und für die Rückgabe von Funktionsergebnissen. Die *system global*-Register gehören keinem Kontext an.

3.5.2 Befehlssatz

Der TriCore-Prozessor erhebt den Anspruch, die Funktionen eines DSP, eines Mikrocontrollers und eines RISC-Prozessors zu vereinen. Aus diesem Grund verfügt er über einen sehr umfangreichen Befehlssatz [Inf05b], [Inf03]. Dieser enthält zum einem übliche Instruktionen eines RISC-Prozessors, zu denen Instruktionen für einfache arithmetische und logische Operationen, für Vergleiche, Sprünge und Speicherzugriffe zählen. Weiterhin enthält er Instruktionen für die Manipulation einzelner Bits oder Bitfelder. Zusätzlich gehören viele Multiplikations- und MAC-Instruktionen zum Befehlssatz, die charakteristisch für DSPs sind. Durch Nutzung der *loop*-Instruktion können *zero overhead loops* realisiert werden.

Der TriCore ist eine Load/Store-Architektur. Dies bedeutet, dass alle Operanden der Instruktionen in Registern vorliegen müssen oder als Konstanten im Befehlswort codiert sind. Der Zugriff auf Daten im Speicher ist nur mittels Lade- und Speicheroperationen möglich.

Für die Codierung der Instruktionen werden 32 oder 16 Bit verwendet. Instruktionen mit beiden Befehlslängen können ohne Einschränkung gemeinsam in einem Programm verwendet werden. Für jede Instruktion gibt es ein 32 Bit-Befehlswort. Für einige Instruktionen, die in der Regel häufig verwendet werden, gibt es zusätzlich auch ein nur 16 Bit langes Befehlswort. Auf diese Weise kann Speicherplatz, der für die Speicherung des Programmcodes benötigt wird, eingespart werden. Als weitere Folge kann ein Cache-Block des Instruktionscaches mehr Instruktionen enthalten, was die Anzahl der Cache-Misses und damit die Ausführungszeit positiv beeinflussen kann. Allerdings ist die Wahlfreiheit des Nutzers bezüglich der verwendbaren Parameter gegenüber dem längeren 32 Bit-Befehlsformat eingeschränkt. So ist z. B. der Wertebereich von Konstanten verkleinert, und statt eines bliebigen Adress- oder Datenregisters wird implizit das Adressregister A[15] oder das Datenregister D[15] verwendet.

3.5.3 Pipeline

Der TriCore-Prozessor ist ein superskalarer Prozessor. Er verfügt über drei vierstufige Pipelines [Inf00], in denen die Verarbeitung der Instruktionen *in-order* beginnt.

In der ersten Pipeline werden ausschließlich Instruktionen verarbeitet, die Operationen auf Datenregistern ausführen. Diese Pipeline wird als *integer processing* bzw. *IP*-Pipeline

bezeichnet. In der zweiten Pipeline, der *Load/Store*-Pipeline, werden Instruktionen zur Adressarithmetik, unbedingte Sprünge, bedingte Sprünge, die von den Inhalten in Adressregistern abhängen, sowie Lade- und Speicherinstruktionen verarbeitet. Die dritte Pipeline dient hauptsächlich der Ausführung der *loop*-Instruktionen, mit der *zero overhead loops* realisiert werden. Sie wird daher als *Loop*-Pipeline bezeichnet.

Die Pipelines ermöglichen die gleichzeitige Ausführung von Instruktionen. Bei störungsfreiem Betrieb der Pipelines kann eine Instruktion im Durchschnitt in einem Zyklus ausgeführt werden. Eine Ausnahme bilden z. B. die Multiplikationsinstruktionen, die zwei bis drei Zkylen im Durchschnitt benötigen, oder Divisionsinstruktionen, die vier Zyklen benötigen. Durch die getrennte Verarbeitung der Instruktionen in verschiedenen Pipelines kann die durchschnittliche Ausführungszeit einer Instruktion im Idealfall auf einen halben Zyklus verkürzt werden. Dazu muss auf eine IP-Instruktion eine Instruktion folgen, die von der *Load/Store*-Pipeline verarbeitet wird. Nur in diesem Fall werden die Instruktionen in den Pipelinestufen parallel ausgeführt.

Register werden von der *IP*-Pipeline und der *Load/Store*-Pipeline innerhalb eines Zyklus zu unterschiedlichen Zeiten beschrieben. Auf diese Weise können einige *write after write* Abhängigkeiten aufgelöst werden. Um *read after write* Abhängigkeiten zu begegnen, verfügt die Architektur über *Forwarding*. Für bedingte Sprünge wird eine statische Sprungvorhersage verwendet, die die Verzögerung der Instruktionsausführung durch Sprünge verringern soll [Inf03].

Dennoch können nicht alle Hazards vermieden werden. Insbesonders Lese- und Schreibzugriffe auf langsame Speicher führen dazu, dass eine Instruktion in einer Pipelinestufe länger als einen Zyklus verweilt. Wann immer ein Konflikt nicht aufgelöst werden kann, muss die betroffene Pipeline angehalten werden. Wird die *IP*-Pipeline oder die *Load/Store*-Pipeline angehalten, so werden in der Regel auch die beiden anderen Pipelines angehalten.

3.5.4 Speicherarchitekur

Wie die meisten heutigen RISC-Architekturen ist auch der Infineon TriCore eine Harvard Architektur [Inf08], [Inf04]. Der Prozessorkern selbst enthält zwar keinen Speicher, der TC1796 verfügt aber über eine Vielzahl von unterschiedlichen Speichern, die sich auf dem selben Chip wie der Prozessor befinden. Dies ermöglicht, dass Zugriffe auf Speicher, die direkt an den Prozessorkern angebunden sind, innerhalb eines Zyklus ausgeführt werden können. In Abbildung 3.3 ist die Speicherhierarchie des TC1796 skizziert.

An die Programmspeicher-Schnittstelle (*program memory interface* (PMI)) des Prozessorkerns sind unmittelbar ein 48 Kilobyte großer Scratchpad-Speicher und ein 16 Kilobyte großer Instruktionscache angeschlossen. Als nicht-flüchtiger Programmspeicher stehen ein 16 Kilobyte großer Boot-ROM und ein 2 Megabyte großer Flash-Speicher in der Programmspeicher-Einheit (*program memory unit* (PMU)) zur Verfügung. Diese Einheit ist über einen lokalen Programmspeicher-Bus (*program local memory bus* (PLMB)) an das PMI angeschlossen. Bei Zugriffen auf die Programmspeicher-Einheit fungiert das PMI als Brücke zwischen dem lokalen Programmspeicher-Bus und dem Prozessorbus für Instruktionen.

Daten werden in der ersten Hierarchiestufe in einem 64 Kilobyte großen SRAM-Speicher abgelegt, der vom dem Prozessorkern direkt über die Datenspeicher-Schnittstelle (*data memory interface* (DMI)) angesprochen werden kann. Wie auch das PMI übt das DMI

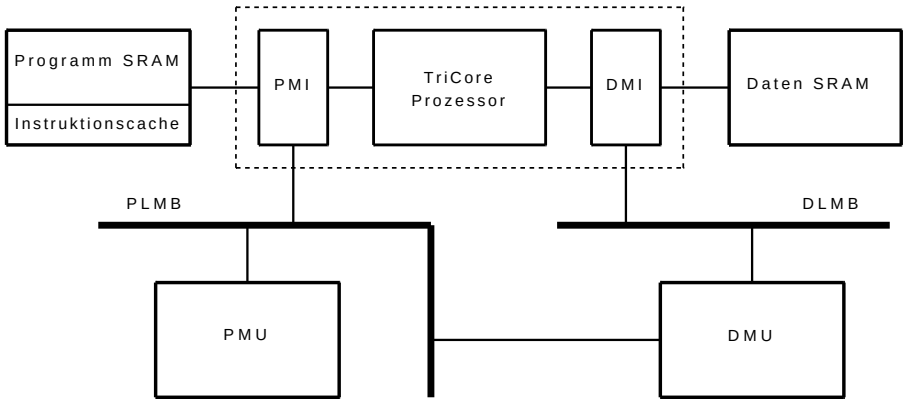

Abbildung 3.3: Vereinfachte Darstellung der Speicherhierarchie des TC1796 nach [Inf04] und [Inf08].

eine Doppelfunktion aus. Es stellt nicht nur eine Datenspeicher-Schnittstelle zwischen dem Prozessorkern und dem SRAM-Speicher dar, sondern ist gleichzeitig eine Brücke zwischem dem Prozessorbus für Daten und dem lokalen Datenspeicher-Bus (*data local memory bus* (DLMB)). Über diesen lokalen Datenspeicher-Bus (*data local memory bus* (DLMB)) kann auf die Datenspeicher-Einheit (*data memory unit* (DMU)) zugegriffen werden. Sie enthält einen weiteren 64 Kilobyte großen SRAM und einen 16 Kilobyte großen Standy-by Speicher, die die zweite Hierarchiestufe bilden. Zusätzlich erfüllt die Datenspeicher-Einheit die Funktion einer Brücke zwischen dem lokalen Datenspeicher-Bus und dem lokalen Programmspeicher-Bus. Somit ist die Programmspeicher-Einheit auch über die Datenspeicher-Schnittstelle erreichbar. Hier befindet sich neben den Programmspeichern ein Flash-Speicher, der bis zu 128 Kilobyte Daten enthalten kann. Das Fehlen eines Datencaches erleichtert die Vorhersage der Zugriffszeiten auf die Daten.

Kapitel 4

Modellierung des Registerallokationsproblems als ILP

Der in der Diplomarbeit erstellte Registerallokator beschreibt das Registerallokationsproblem durch ein ganzzahliges lineares Programm (*integer linear program* (ILP)). Da es sich um einen globalen Registerallokator handelt, wird für jede Funktion ein separates ILP formuliert. Dieses kann in zwei Teile gegliedert werden. Durch eine Menge von Constraints wird sichergestellt, dass aus der Lösung des ILP unmittelbar eine korrekte Registerallokation abgeleitet werden kann. Durch zusätzliche Constraints und die Zielfunktion wird das Gütekriterium modelliert. Dies ist die WCET der Funktion, die minimiert werden soll.

In diesem Kapitel wird der erste Teil des ILP beschrieben, der das Allokationsproblem ohne ein Gütekriterium modelliert. Hierfür wurde die Modellierung von Goodwin und Wilken [GW96] für Load/Store-Architekturen verwendet, die in den nächsten Abschnitten vorgestellt wird.

Zunächst werden in Abschnitt 4.1 wesentliche Eigenschaften der Modellierung vorgestellt und die Annahmen, die getroffen werden, genannt. In Abschnitt 4.2 wird beschrieben, wie mit Hilfe von binären Entscheidungsvariablen die Allokation der Register modelliert wird. Im anschließenden Abschnitt 4.3 werden die Constraints vorgestellt, die die Gültigkeit der Allokation gewährleisten.

Das ILP, das auf diese Weise entsteht, ist möglicherweise nicht lösbar, da es eine Stelle geben kann, an der nicht allen gleichzeitig lebendigen Registern physikalische Register zugeordnet werden können. In diesem Fall muss Spillcode eingefügt werden. Zum einem müssen die Stellen identifiziert werden, an denen das Einfügen von Spillcode sinnvoll ist. Zum anderen müssen die Entscheidungen über das Einfügen von Lade- und Speicherinstruktionen, sowie die nötigen Voraussetzungen und ihre Folgen modelliert werden. Welche Entscheidungsvariablen und Constraints dem ILP hinzugefügt werden, um diese Aspekte zu berücksichtigen, wird in Abschnitt 4.4 dargestellt. In Abschnitt 4.5 wird erläutert, wie *calling conventions* in das ILP-Modell integriert werden.

Abschließend wird in Abschnitt 4.6 beschrieben, in mit welchen Änderungen das Modell von Goodwin und Wilken im Rahmen dieser Diplomarbeit umgesetzt wurde. Mit der Modellierung des Gütekriteriums der Registerallokation befasst sich dann das folgende Kapitel 5.

4.1 Optimale Registerallokation (ORA)

Die grundlegende Vorgehensweise bei der optimalen Registerallokation (ORA) ist typisch für ein ILP-basiertes Verfahren. Sie gliedert sich grob in drei Schritte: Die Eingabe des

Ausgangsproblems wird in ein ILP überführt. Dessen Lösung wird von einem ILP-Solver bestimmt. Aus der Lösung des ILP wird dann die Lösung für das Ausgangsproblem erstellt.

Im Fall der globalen Registerallokation ist die Eingabe eine Zwischendarstellung einer Funktion. Für diese wird gemäß eines Regelwerkes, das in den nächsten Abschnitten vorgestellt wird, ein ILP erstellt. Das ILP beschreibt dann auf eine sehr strukturierte Weise das Registerallokationsproblem, das sich konkret auf die Zwischendarstellung dieser Funktion bezieht. Um die Registerallokation an Eigenschaften der zu Grunde liegenden Hardware anzupassen, sind nur Änderungen am Regelwerk nötig. Darüber hinaus kann das ILP so erweitert werden, dass es neben der eigentlichen Allokation auch noch weitere Optimierungen umfasst, die einen Bezug zur Registerallokation haben. Dies sind z. B. die Optimierungen *coalescing* und *rematerialization*. Diese Erweiterungen sind nicht in dem Allokator enthalten, der in der Diplomarbeit erstellt wurde, und werden daher in dieser Diplomarbeit nicht vorgestellt. Die Optimierung *live range splitting* wird von dem ORA-Grundmodell dagegen grundsätzlich immer mit ausgeführt.

Für die Modellierung des Allokationsproblems nach dem ORA-Ansatz werden ausschließlich binäre Entscheidungsvariablen verwendet. Sie codieren die Entscheidungen, die bei der Allokation getroffen werden müssen. Zum einen muss entschieden werden, welche Zustände eingenommen werden, und zum andern welche Aktionen ausgeführt werden. Die Bedeutung der Variablen ist immer so definiert, dass genau dann wenn die Variable den Wert Eins annimmt, der zugehörige Zustand eingenommen oder die zugehörige Aktion ausgeführt wird.

Für die Lösung des ILP wird ein ILP-Solver eingesetzt, der eine optimale Lösung bestimmt. An Hand dieser wird schließlich eine Lösung für die Registerallokation erstellt. Dazu führt die Registerallokation die Entscheidungen aus, die durch die binären Entscheidungsvariablen codiert werden, denen in der Lösung des ILP der Wert Eins zugeordnet wurde.

Eine optimale Lösung für das ILP repräsentiert nur dann gleichsam eine optimale Lösung für die Registerallokation, wenn die folgenden Annahmen zutreffen. Die Registerallokation soll Spillinstruktionen so in die Zwischendarstellung einfügen, dass eine Zielfunktion minimiert wird. Dabei wird für das ORA-Modell angenommen, dass der Einfluss, den das Einfügen einer Spillinstruktion auf den Wert der Zielfunktion hat, nur von dem Typ der Spillinstruktion und der Ausführungshäufigkeit des Basisblocks abhängt, in den die Instruktion eingefügt wird. Die Registerallokation benötigt dann keine Kenntnis über das Verhalten der Pipeline des zu Grunde liegenden Prozessors.

Zusätzlich wird angenommen, dass die Ausführungshäufigkeiten der Basisblöcke, die einen Einfluss auf die Zielfunktion haben, die eines Kontrollflusses sein können. Auf diese Weise ist sichergestellt, das ein Basisblock, der nur einen Vorgänger oder einen Nachfolger hat, nicht öfter als sein einziger Vorgänger bzw. Nachfolger ausgeführt wird. Auf Grund dieser und der vorherigen Annahmen können Spillinstruktionen in den Vor- bzw. Nachfolger verschoben werden, ohne dass sich der Wert Zielfunktion verschlechtert. Dies wird benötigt, um die Stellen einzugrenzen, an denen Entscheidungen über das Einfügen von Spillinstruktionen getroffen werden.

Darüber hinaus sollen auch all die Annahmen gelten, die üblicherweise für eine Registerallokation getroffen werden. In dieser Diplomarbeit werden von der Registerallokation keine weiteren Optimierungen wie *coalescing* und *rematerialization* durchgeführt. Daher dürfen in der Zwischendarstellung nur die virtuellen Register durch physikalische Regis-

ter ersetzt werden und Spillinstruktionen in die Zwischendarstellung eingefügt werden. Andere Änderungen sind nicht erlaubt. Weiterhin wird angenommen, dass alle Pfade in dem Kontrollflussgraphen mögliche Ausführungspfade repräsentieren.

4.2 Modellierung der Registerzuordnung

In diesem Abschnitt wird beschrieben, wie die Abbildung der virtuellen Register auf die physikalischen Register nur mit Hilfe von binären Entscheidungsvariablen modelliert wird. Da die ORA-Modellierung grundsätzlich ein *live range splitting* beinhaltet, kann ein virtuelles Register an verschiedenen Stellen in der Zwischendarstellung der betrachteten Funktion unterschiedlichen physikalischen Registern zugeordnet sein. Für die Zuordnung der Entscheidungsvariablen zu einzelnen Abschnitten im Kontrollflussgraphen der Funktion werden Lebendigkeitsgraphen (*live range graphs* (LRG)) verwendet. Für jedes virtuelle Register s gibt es einen solchen Graphen LRG_s. Seine Struktur geht unmittelbar aus dem Kontrollflussgraphen der Funktion hervor.

Definition 4.1 *Ein Lebendigkeitsgraph LRG_s ist ein Tupel (V, E). Die Knotenmenge V enthält für jede Instruktion I, vor der oder nach der das virtuelle Register s unmittelbar lebendig ist, einen Knoten v_I. Zusätzlich enthält V für jeden Basisblock B genau dann einen Knoten v_B^{start}, wenn das virtuelle Register s unmittelbar vor der ersten Instruktion des Basisblocks B lebendig ist, und genau dann einen Knoten v_B^{end}, wenn das virtuelle Register s unmittelbar nach der letzten Instruktion des Basisblocks B lebendig ist. Zwischen zwei Knoten v_I und v_J existiert genau dann eine Kante, wenn sie zwei Instruktion I und J repräsentieren, die zum selben Basisblock gehören und die Instruktion I unmittelbarer Vorgänger der Instruktion J ist. Weiterhin existiert für jeden Knoten v_B^{start} eine Kante von v_B^{start} zu dem Knoten v_I, der die erste Instruktion I in dem Basisblock B repräsentiert, und es existiert für jeden Knoten v_B^{end} eine Kante von dem Knoten v_J, der die letzte Instruktion J des Basisblocks B repräsentiert, zu dem Knoten v_B^{end}. Zwischen einem Knoten $v_{B'}^{start}$, der den Anfang des Basisblocks B' repräsentiert, und dem Knoten $v_{B''}^{end}$, der das Ende des Basisblocks B'' repräsentiert, existiert eine Kante genau dann, wenn Basisblock B' ein unmittelbarer Vorgänger von Basisblock B'' ist.*

Der Lebendigkeitsgraph LRG_s ist somit der Teilgraph des Kontrollflussgraphen, in dem das Register s lebendig ist, und der so erweitert wurde, dass jeder Knoten in dem Graphen, der eine Instruktion repräsentiert, höchstens einen Vorgänger und einen Nachfolger hat. Für eine bessere Lesbarkeit werden im Folgenden die Begriffe Knoten und Instruktion synonym verwendet. So kann auf die umständlichen Formulierungen wie: „der Knoten v_I, der die Instruktion I repräsentiert“ bzw. „die Instrutkion I, die durch den Knoten v_I repräsentiert wird“, verzichtet werden.

Für jedes physikalische Register p, dem das virtuelle Register s zugeordnet werden kann, gibt es schließlich eine Abbildung $DV_{s\to p} : E \mapsto \mathcal{P}\{x_1, \ldots, x_n\}$, die den Kanten des Lebendigkeitsgraphen eine Menge von Entscheidungsvariablen zuordnet. Die Menge der Entscheidungsvariablen beschreibt für die jeweilige Stelle, ob das virtuelle Register s auf das physikalische Register r abgebildet wird. Sobald eine Variable aus der Menge $DV_{s\to p}(e)$ mit $e = (v_I, v_J)$ den Wert Eins annimmt, bedeutet dies, dass das virtuelle Register s dem physikalischen Register p unmittelbar nach der Instruktion I und unmit-

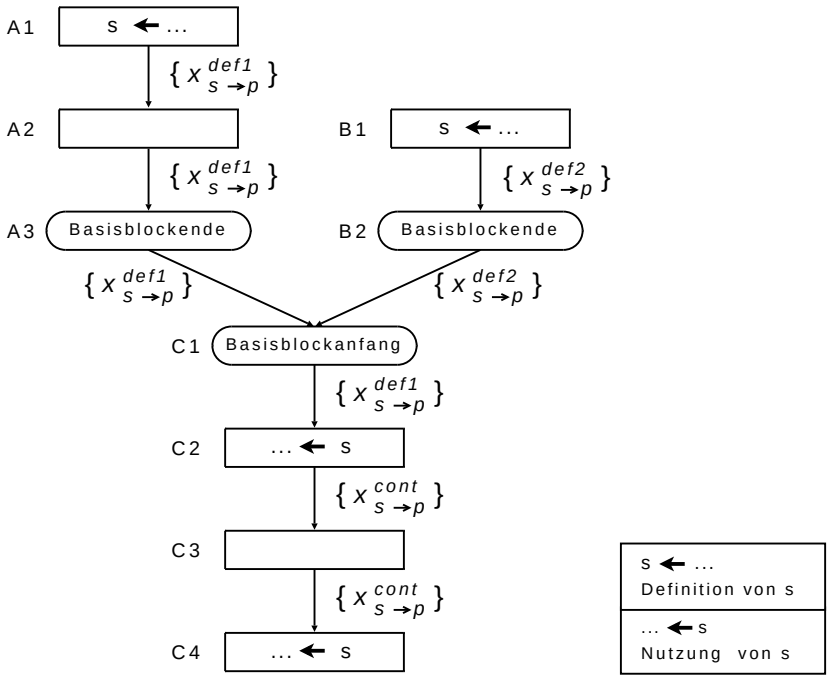

Abbildung 4.1: Lebendigkeitsgraph mit Entscheidungsvariablen

telbar vor der Instruktion J zugeordnet ist. Wenn alle Variablen in der Menge den Wert Null aufweisen, ist s p nicht zugeordnet.

Für eine einfachere Schreibweise und ein leichteres Verständnis werden nun für jede Instruktion die Mengen der Entscheidungsvariablen unmittelbar vor und nach der Instruktion eine weitere Bezeichnung definiert. Für eine Instruktion I soll $LiveOutDV_{s\to p}(I) := DV_{s\to p}(e)$ mit $e = (v_I, \bullet)$ die Menge der Entscheidungsvariablen sein, die angeben, ob unmittelbar nach der Instruktion I das virtuelle Register s dem physikalischen Register p zugeordnet ist. $LiveInDV_{s\to p}(I) := DV_{s\to p}(e)$ mit $e = (\bullet, v_I)$ sei die Menge der Entscheidungsvariablen, die angeben, ob unmittelbar vor der Instruktion I das virtuelle Register s dem physikalischen Register p zugeordnet ist. Diese Mengen $LiveOutDV_{s\to p}(I)$ und $LiveOutDV_{s\to p}(I)$ sind eindeutig, da jeder Knoten, der in dem Lebendigkeitsgraphen eine Instruktion repräsentiert, höchstens eine ausgehende und höchstens eine eingehende Kante aufweist.

Erweiterte virtuelle Register werden erst in einfache virtuelle Register zerlegt, so dass die einfachen virtuellen Register jeweils einem einzelnen physikalischen Register zugeordnet werden können. Wird das erweiterte Register s in die virtuellen Register s_1 und s_2 zerlegt, so gibt es jeweils für s_1 und s_2 einen Lebendigkeitsgraph LRG_{s_1} bzw. LRG_{s_2} und Abbildungen $DV_{s_1\to p_1}$ bzw. $DV_{s_1\to p_2}$. Die physikalischen Register p_1 und p_2, denen s_1 und s_2 zugewiesen werden kann, sind die physikalischen Register, aus denen sich ein physikalischen Registerpaar $p = (p_1, p_2)$ zusammen setzt, dem das erweiterte virtuelle Register s zugeordnet werden kann. Für das erweiterte Register s selbst gibt es keine weiteren Lebendigkeitsgraphen und Abbildungen, da die Registerzuordnung für das erweiterte Register aus der Allokation der einfachen Register s_1 und s_2 hervor geht.

Im Folgenden soll nun beschrieben werden, welche Entscheidungsvariablen in den Mengen enthalten sind, die den Kanten durch die Abbildung $DV_{s\to p}$ zugeordnet werden. Zur Veranschaulichung der Regeln, nach denen die Mengen der Entscheidungsvariablen bestimmen werden, ist in Abbildung 4.1 ein Lebendigkeitsgraph für ein virtuelles Register s dargestellt. In diesem Beispiel definieren die Instruktionen $A1$ und $B1$ das virtuelle Register s. Die Instruktionen $C2$ und $C4$ nutzen das virtuelle Register. Die Instruktionen $A2$ und $C3$, die durch leere Rechtecke repräsentiert sind, definieren und nutzen das virtuelle Register nicht. $A3$ und $B2$ sind die Enden von Basisblöcken. Der Anfang eines Basisblocks ist in diesem Beispiel der Knoten $C1$.

Die Kanten sind mit den Entscheidungsvariablen beschriftet, die den Kanten für ein physikalisches Register p durch die Abbildung $DV_{s\to p}$ zugeordnet sind. Kann das virtuelle Register s n verschiedenen physikalischen Registern zugeordnet werden, gibt es für jede Kante n Mengen. Da die Regeln, nach denen die Mengen der Entscheidungsvariablen gebildet werden, für alle physikalischen Register gleich sind, sind für eine bessere Übersicht nur für ein physikalisches Register p die Mengen in der Abbildung angegeben.

Definiert eine Instruktion I der betrachteten Funktion das virtuelle Register s, so enthält jede Menge $LiveOutDV_{s\to p}(I)$ jeweils genau eine Entscheidungsvariable $x^{def}_{s\to p}$. Nimmt eine Entscheidungsvariable $x^{def}_{s\to p}$ den Wert Eins an, so ist bei der Definition das virtuelle Register s dem physikalischem Register p zugeordnet. Hat sie den Wert Null, liegt diese Zuordnung nicht vor. In Abbildung 4.1 wird daher der Kante, die aus dem Knoten $A1$ herausführt, die Entscheidungsvariable $x^{def1}_{s\to p}$ zugeordnet. Ebenso besteht die Menge der Entscheidungsvariablen für die Kante zwischen den Knoten $B1$ und $B2$ nur aus der Entscheidungsvariablen $x^{def2}_{s\to p}$.

Nach der Definition oder einer Nutzung kann das virtuelle Register s in dem physikalischen Register p verbleiben, oder von einem anderen virtuellen Register aus dem physikalischen Register verdrängt werden. Dabei ist es unerheblich, ob das physikalische Register p unmittelbar in der nächsten oder in einer der darauf folgenden Instruktionen neu definiert wird. Entscheidend ist, ob das virtuelle Register s bis zur nächsten Nutzung dem physikalischen Register p zugeordnet bleibt, oder nicht. Für eine einzelne Instruktion, die ein virtuelles Register s weder nutzt noch definiert, wird daher nicht unterschieden, ob die Zuordnung erhalten bleibt. Die Entscheidungsvariablen sind so gewählt, dass sie nur Aussagen darüber treffen, ob eine Zuordnung während eines ganzen Bereiches vorliegt, der sich von einer Definition oder Nutzung bis zur folgenden Nutzung erstreckt.

Jeder Pfad im Lebendigkeitsgraphen, der nur Knoten enthält, die das virtuelle Register s weder nutzen noch definieren, oder das Ende eines Basisblocks repräsentieren, ist ein Teilstück eines solchen Bereichs zwischen einer Definition bzw. Nutzung und einer folgenden Nutzung. Allen Kanten in dem Pfad ist darum die gleiche Menge von Entscheidungsvariablen zugeordnet. Für jede Instruktion I, vor der das virtuelle Register s unmittelbar lebendig ist, und die das virtuelle Register s weder nutzt noch definiert, ist somit die Menge $LiveOutDV_{s\to p}(I)$ gleich der Menge $LiveInDV_{s\to p}(I)$. Für jeden Knoten im Lebendigkeitsgraphen, der das Ende eines Basisblocks repräsentiert, gilt mit der gleichen Begründung, dass den Kanten, die von ihm ausgehen, jeweils die gleiche Menge von Entscheidungsvariablen zugeordnet ist wie seiner eingehenden Kante. In dem Beispiel, das in Abbildung 4.1 dargestellt ist, ist somit die Menge, die den Kanten zwischen den Knoten $A2$, $A3$ und $C1$ zugeordnet ist, jedesmal $\{x^{def1}_{s\to p}\}$.

Ein Knoten v_B^{start}, der den Anfang eines Basisblocks B repräsentiert, kann mehrere eingehende Kanten haben. Sei v^{use} der Knoten auf einem Pfad beginnend in v_B^{start}, der die nächste Instruktion repräsentiert, die das virtuelle Register nutzt. Die Mengen von Entscheidungsvariablen, die den eingehenden Kanten zugeordnet sind, geben wie auch die Menge, die der ausgehenden Kante zugeordnet ist, an, ob die Zuordnung des virtuellen Registers s zum physikalischen Register p bis zu dem Knoten v^{use} Bestand hat. Da die Mengen die gleiche Bedeutung haben, ist an dieser Stelle die Menge, die der ausgehenden Kante zugeordnet ist, mit einer der Mengen identisch, die einer eingehenden Kante zugewiesen ist. Sie muss aber nicht mit den Mengen aller eingehenden Kanten identisch sein, da diese unterschiedliche Entscheidungsvariablen enthalten können.

In Abbildung 4.1 repräsentiert der Knoten $C1$ den Anfang eines Basisblocks. Es ist zu erkennen, dass den beiden Kanten, die auf den Knoten $C1$ gerichtet sind, unterschiedliche Mengen von Entscheidungsvariablen zugeordnet sind. Der ausgehenden Kante wurde in diesem Fall die gleiche Menge von Entscheidungsvariablen zugeordnet, wie der Kante zwischen $A3$ und $C1$.

Bleibt nun nur noch die Mengen von Entscheidungsvariablen anzugeben, die den Kanten zugeordnet sind, die von einem Knoten des Lebendigkeitsgraphen ausgehen, der eine Instruktion I repräsentiert, die das virtuelle Register s nutzt, aber nicht definiert. Nachdem das virtuelle Register s genutzt wurde, stellt sich erneut die Frage, ob das virtuelle Register bis zu einer folgenden Nutzung einem physikalischem Register p zugeordnet bleibt, oder nicht. In diesem Fall können die Mengen $LiveInDV_{s\to p}(I)$ und $LiveOutDV_{s\to p}(I)$ nicht identisch sein, da sich ihre Bedeutung unterscheidet. Während $LiveInDV_{s\to p}(I)$ die Registerzuordnung bis zu der Nutzung durch I beschreibt, gibt $LiveOutDV_{s\to p}(I)$ die Zuordnung bis zu einer folgenden Nutzung an, die sich von dieser Nutzung unterscheiden kann. Jede Menge $LiveOutDV_{s\to p}(I)$ enthält daher ausschließlich eine neue Entscheidungsvariable $x_{s\to p}^{cont}$, die eine Aussage darüber trifft, ob die Zuordnung des virtuellen Registers s zum physikalischen Register p bis zur folgenden Nutzung bestehen bleibt. Aus diesem Grund wird die Entscheidungsvariable $x_{s\to p}^{cont}$ auch Kontinuitätsvariable genannt. Sie hat den Wert Eins, wenn das virtuelle Register s von dieser bis zur folgenden Nutzung dem physikalischen Register p zugeordnet ist und anderenfalls den Wert Null.

Instruktionen, die das virtuelle Register s nutzen werden in Abbildung 4.1 durch die Knoten $C2$ und $C4$ repräsentiert. Da nach der Nutzung von $C4$ das virtuelle Register s nicht mehr lebendig ist, führt nur aus dem Knoten $C2$ eine Kante heraus. Die Menge der Entscheidungsvariablen, die dieser Kante zugeordnet ist besteht nur aus der Entscheidungsvariable $x_{s\to p}^{cont}$.

4.3 Modellierung der Allokationsbedingungen

Damit durch die Menge der Entscheidungsvariablen eine gültige Registerallokation beschrieben wird, müssen einige Bedingungen erfüllt sein. Diese sollen in diesem Abschnitt genannt und durch Constraints beschrieben werden. Allein schon auf Grund der Bedeutung der Entscheidungsvariablen bestehen Zusammenhänge, die durch die Constraints formuliert werden müssen.

So kann ein virtuelles Register s nur dann nach einer Instruktion I, die s nutzt, einem physikalischem Register p zugeordnet bleiben, wenn es ihm auch schon unmittelbar vor

der Instruktion I zugeordnet war. War s p zugeordnet, ist es aber nach der Instruktion I aber nicht mehr, so fand eine Aufhebung der Zuordnung statt. Diese Aufhebung wird durch eine neue binäre Entscheidungsvariable $x_{s\to p}^{end}$ modelliert wird. Sie hat den Wert Eins, wenn eine Aufhebung der Zuordnung erfolgt und sonst den Wert Null.

Die Menge $LiveInDV_{s\to p}(I)$ beschreibt die Abbildung des virtuellen Registers s auf das physikalische Register p unmittelbar vor der Instruktion. Für die Kontinuitätsvariable $x_{s\to p}^{cont}$ und die Entscheidungsvariable $x_{s\to p}^{end}$, die die Aufhebung der Zuordnung modelliert, muss daher gelten, dass sie nur dann den Wert Eins annehmen können, wenn eine Entscheidungsvariable aus der Menge $LiveInDV_{s\to p}(I)$ den Wert Eins hat.

Für jede Instruktion I und jedes virtuelle Register s, das von I genutzt, aber nicht definiert wird, und unmittelbar nach der Instruktion I weiterhin lebendig ist, bestehen die Mengen $LiveOutDV_{s\to p}(I)$ jeweils nur aus der Entscheidungsvariable $x_{s\to p}^{cont}$. Für jede dieser Variablen muss gelten:

$$\forall p: \sum_{x_{s\to p}\in LiveInDV_{s\to p}(I)} x_{s\to p} = x_{s\to p}^{cont} + x_{s\to p}^{end} \tag{4.1}$$

Für Instruktionen I, die das virtuelle Register s weder nutzen noch definieren, ist ein solches Constraint nicht erforderlich. Mit Hilfe der Entscheidungsvariablen wird nicht zwischen der Zuordnung vor und nach der Instruktion I unterschieden. Statt dessen wird modelliert, ob die Zuordnung für einen gesamten Bereich von einer vorhergehenden Definition bis zu einer folgenden Nutzung gilt. Ohnehin ist für solche Instruktionen $LiveInDV_{s\to p}(I) = LiveOutDV_{s\to p}(I)$.

Im vorherigen Abschnitt 4.2 wurde schon festgehalten, dass an einem Knoten v_B^{start}, der den Anfang eines Basisblocks repräsentiert, die Bedeutung der Mengen an allen eingehenden Kanten gleich ist. Obwohl die Mengen unterschiedliche Entscheidungsvariablen enthalten können, wie in Abbildung 4.1 zu erkennen ist, müssen sie die gleiche Registerzuordnung für das jeweilige virtuelle Register s und das physikalische Register p beschreiben. Zwei Mengen von Entscheidungsvariablen beschreiben genau dann die gleiche Registerzuordnung, wenn in beiden Mengen mindestens eine Entscheidungsvariable den Wert Eins hat, oder wenn in beiden Menge alle Entscheidungsvariablen den Wert Null haben. Um die Registerzuordnungen der Mengen zu vergleichen, können daher auch die Summen der Entscheidungsvariablen in den Mengen betrachtet werden. Für jeden Knoten v_B^{start}, der den Anfang eines Basisblocks repräsentiert, wird sichergestellt, dass die Menge der Entscheidungsvariablen, die je zwei verschiedene eingehenden Kanten e_1 und e_2, zugeordnet sind, die gleiche Registerzuordnung beschreiben, indem für die Summen der Entscheidungsvariablen in den Mengen Gleichheit gefordert wird.

$$\forall s\ \forall p: \sum_{x_{s\to p}\in DV_{s\to p}(e_1)} x_{s\to p} = \sum_{x_{s\to p}\in DV_{s\to p}(e_2)} x_{s\to p} \tag{4.2}$$

Eine Instruktion, die ein Register definiert, speichert den Wert des virtuellen Registers s in genau einem physikalischen Register p. Das virtuelle Register s muss daher bei der Instruktion genau einem physikalischen Register zugeordnet sein. Genau eine der Entscheidungsvariablen $x_{s\to p}^{def}$, die die Registerzuordnung des virtuellen Registers s nach der Definition modellieren, muss den Wert Eins annehmen, alle anderen den Wert Null. Für jede Instruktion I und jedes virtuelle Register s, das von I definiert wird, muss daher die *must-allocate*-Bedingung gelten:

$$\sum_{p} \sum_{x^{def}_{s\to p} \in LiveOutDV_{s\to p}(I)} x^{def}_{s\to p} = 1 \tag{4.3}$$

Bei der Nutzung eines virtuellen Registers s muss dagegen nur garantiert sein, dass das virtuelle Register in mindestens einem physikalischen Register p vorliegt. Folglich muss mindestens eine der Entscheidungsvariablen, die die Registerzuordnung unmittelbar vor der Nutzung modellieren, den Wert Eins haben, es dürfen aber auch mehrere Variablen den Wert Eins annehmen. Für jede Instruktion I und jedes virtuelle Register s, das von I genutzt wird, wird die *must-allocate*-Bedingung daher so formuliert:

$$\sum_{p} \sum_{x_{s\to p} \in LiveInDV_{s\to p}(I)} x_{s\to p} \geq 1 \tag{4.4}$$

Wenn es sich bei s um ein erweitertes Register handelt, müssen die beiden einfachen virtuellen Register s_1 und s_2, in die s zerlegt werden kann, in jeweils einem physikalischen Register vorliegen. Zusätzlich müssen sich die virtuellen Register s_1 und s_2 in zwei physikalischen Registern p_1 bzw. p_2 befinden, die ein Registerpaar bilden. Für jede Instruktion I und jedes erweiterte Register s, das von I genutzt wird und in einfache virtuelle Register s_1 und s_2 zerlegt werden kann, und für jedes Registerpaar (p_1, p_2), dem s zugeordnet werden kann, muss erzwungen werden, dass die Zuordnung von s_1 zu dem physikalischen Register p_1 genau dann vorliegt, wenn s_2 dem physikalischen Register p_2 zugeordnet ist.

$$\sum_{x_{s_1\to p_1} \in LiveInDV_{s_1\to p_1}(I)} x_{s_1\to p_1} = \sum_{x_{s_2\to p_2} \in LiveInDV_{s_2\to p_2}(I)} x_{s_2\to p_2} \tag{4.5}$$

Um ein solches Constraint nicht auch bei der Definition des erweiterten Registers s' durch eine Instruktion I formulieren zu müssen, kann für $x^{def}_{s_1\to p_1}$ und $x^{def}_{s_2\to p_2}$ ein und dieselbe Entscheidungsvariable verwendet werden. Es gilt dann für die beiden einfachen virtuellen Register $s1$ und $s2$ $LiveOutDV_{s_1\to p_1}(I) = LiveOutDV_{s_2\to p_2}(I)$.

Schließlich muss für eine gültige Registerallokation noch berücksichtigt werden, dass jedem physikalischen Register höchstens ein virtuelles Register gleichzeitig zugeordnet werden kann. Dies wird als *single-symbolic*-Bedingung bezeichnet. Da nur durch eine Definition einem physikalischen Register ein virtuelles Register zugeordnet werden kann, muss auch nur nach einer Definition eines virtuellen Registers diese Bedingung überprüft werden. Es muss also gefordert werden, dass unmittelbar nach jeder Instruktion I, die ein virtuelles Register definiert, höchstens eine der Entscheidungsvariablen den Wert Eins hat, die sich auf ein physikalisches Register p bezieht. Für jede Instruktion I, die ein virtuelles Register s definiert, und für jedes physikalische Register p, dem s zugeordnet werden kann, muss deshalb diese Bedingung erfüllt sein:

$$\forall p : \sum_{s} \sum_{x_{s\to p} \in LiveOutDV_{s\to p}(I)} x_{s\to p} \leq 1 \tag{4.6}$$

4.4 Modellierung von Spill-Entscheidungen

Soll ein virtuelles Register in den Speicher ausgelagert werden, gibt es eine Vielzahl von möglichen Stellen in der Zwischendarstellung der Funktion, an denen das Register gesichert und wiederhergestellt werden kann. Besonders weil der ORA-Ansatz ein optimales

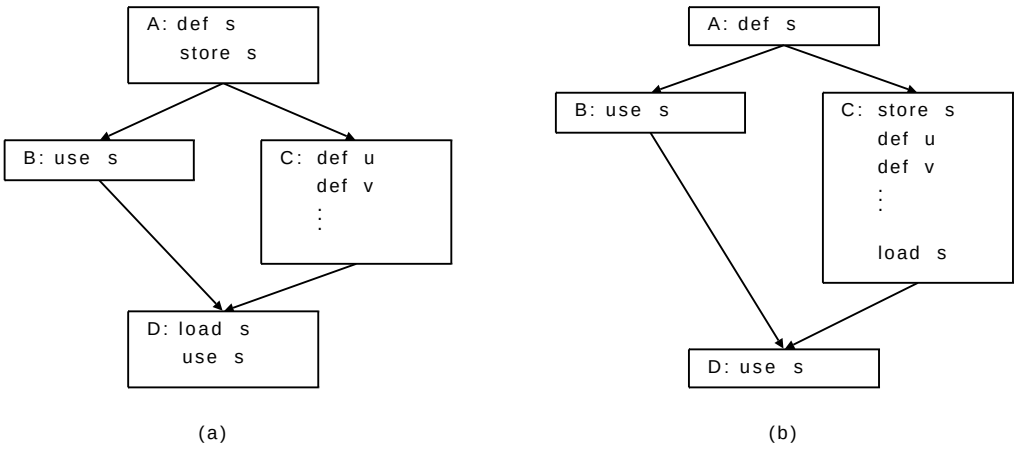

Abbildung 4.2: Kontrollflussgraph mit Spillcode. In Basisblock C bestehe der Zwang das virtuelle Register s in den Speicher auslagern zu müssen. Typische Positionierung von Spillcode (a). Vorteilhafte Verteilung von Spillcode bei *live range splitting* (b).

live range splitting modelliert, ist es nicht sinnvoll, ein virtuelles Register immer nur unmittelbar nach einer Definition zu sichern und immer nur unmittelbar vor einer Instruktion, die das Register nutzt, wiederherzustellen, wie das Beispiel in Abbildung 4.2 zeigt.

Die Abbildung 4.2 (a) zeigt einen Kontrollflussgraphen mit vier Basisblöcken. In dem Basisblock A wird ein virtuelles Register s definiert, das in den Basisblöcken B und D genutzt wird. Für Basisblock C soll angenommen werden, dass dort so viele virtuelle Register definiert und genutzt werden, dass das virtuelle Register s in den Speicher ausgelagert werden muss. Es muss daher angenommen werden, dass zu Beginn von Basisblock D sich das virtuelle Register in keinem physikalischem Register befindet. Typischerweise würde in diesem Fall Spillcode unmittelbar nach der Definition von s und vor der Nutzung in Basisblock D eingefügt werden. Oftmals würde sogar bei einem einfachen regelbasierten Vorgehen vor jeder Nutzung eine Spillinstruktion zum Wiederherstellen des Registers eingefügt werden. In diesem Beispiel wurde in Basisblock B aber kein Spillcode eingefügt. Er wäre überflüssig, da die Nutzung unmittelbar auf die Definition folgt und sich daher das virtuelle Register s noch in einem pyhsikalischem Register befinden wird. Insgesamt werden somit auf einem Pfad vom Basisblock A zum Basisblock D immer eine Spillinstruktion zum Speichern und eine Spillinstruktion zum Laden des Registers ausgeführt. In Abbildung 4.2 (b) ist für dieses Beispiel eine bessere Positionierung des Spillcodes dargestellt. In diesem Fall müssen nur auf dem Pfad vom Basisblock A zum Basisblock D, der über den Basisblock C führt, Spillinstruktionen ausgeführt werden.

In Abbildung 4.3 ist der Lebendigkeitsgraph dargestellt, der aus dem Kontrollflussgraph des Beispiels in Abbildung 4.2 mit zwei Änderungen hervorgeht. Nach der Definition des virtuellen Registers s durch Instruktion $A1$ wurde der Knoten $A2$ eingefügt, der das Register weder definiert noch nutzt. Er könnte einen Sprungbefehl repräsentieren, der in der Abbildung 4.2 für eine bessere Übersicht weggelassen wurde. Jede Instruktion in dem Basisblock C müsste eigentlich durch einen eigenen Knoten repräsentiert werden. Für eine kompaktere Darstellung wurden alle diese Knoten zu $C2$ zusammengefasst. Dieser

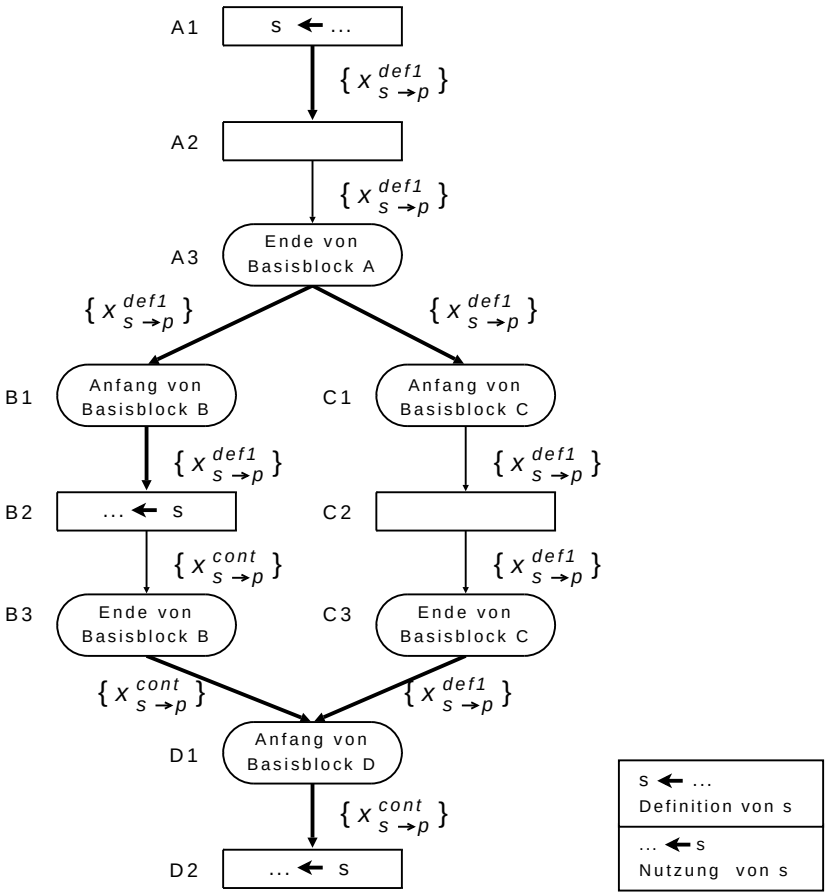

Abbildung 4.3: Lebendigkeitsgraph für ein virtuelles Register s. Kanten, die Stellen von Spill-Entscheidungen repräsentieren, sind durch eine größere Linienbreite hervorgehoben.

Lebendigkeitsgraph soll im Folgenden als Anschauungsbeispiel genutzt werden.

Bislang enthalten die Lebendigkeitsgraphen des virtuellen Registers s nur für die Instruktionen Knoten, die in der Zwischendarstellung der Eingabe vorkommen. Die Stellen, an denen Spillinstruktionen für das virtuelle Register s eingefügt werden können, werden durch die Kanten der Lebendigkeitsgraphen repräsentiert.

Goodwin und Wilken treffen zwei Annahmen, durch die sie die Anzahl der Stellen, die für ein optimales Einfügen von Spillinstruktionen betrachtet werden müssen, stark einschränken können. Zum einem sollen die Auswirkungen einer Spillinstruktion auf die Zielfunktion der Registerallokation nur vom Typ der Instruktion und dem Basisblock abhängt, in den Spillinstruktionen eingefügt werden. Zum anderen nehmen sie an, dass die Ausführungshäufigkeiten der Basisblöcke denen eines möglichen Kontrollfluss entsprechen. Es müssen also an jedem Basisblock die Kirchhoffsche Knotenregel erfüllt sein, was so viel bedeutet, dass jeder Basisblock über die eingehenden Kanten in der Summe genauso oft erreicht wird, wie er über die ausgehenden Kanten verlassen wird.

Goodwin und Wilken zeigen, dass es unter diesen Annahmen für das optimale Platzieren von Spillinstruktionen, die das virtuelle Registers s im Speicher sichern, ausreicht, nur zwei Arten von Kanten zu berücksichtigen. Sowohl die Kanten, die aus einem Knoten

herausführen, der eine Instruktion repräsentiert, die das virtuelle Register s definiert, als auch die Kanten, die aus einem Knoten herausführen, der das Ende eines Basisblocks mit mehreren Nachfolgern repräsentiert, repräsentieren die Stellen im Lebendigkeitsgraphen, an denen Entscheidungen über das Einfügen von Spillcode getroffen werden müssen. Die Kanten, die eine dieser Stellen im Lebendigkeitsgraphen repräsentieren, an der das virtuelle Register s möglicherweise im Speicher gesichert wird, werden *spill store edges* genannt. In dem Beispiel in Abbildung 4.3 sind dies die Kante, die die Knoten $A1$ und $A3$ verlassen.

Weiterhin zeigen Goodwin und Wilken, dass es unter der zuvor genannten Annahmen auch für das optimale Platzieren von Spillinstruktionen, die das virtuelle Register s in einem Register wiederherstellen, ausreicht, nur eine Teilmenge der Kanten des Lebendigkeitsgraphen zu betrachten. Zu dieser Teilmenge gehören die Kanten, die auf einen Knoten gerichtet sind, der eine Instruktion repräsentiert, die das virtuelle Register s nutzt, oder die auf einen Knoten gerichtet sind, der den Anfang eines Basisblocks repräsentiert, der mehrere Vorgänger im Kontrollflussgraphen hat. Die Kanten in dieser Teilmenge werden *spill load edges* genannt. Die *spill load edges* in Abbildung 4.3 sind die Kanten, die auf die Knoten $B1$, $D1$ und $D2$ gerichtet sind.

Da Spillinstruktionen virtuelle Register definieren oder nutzen und damit die Registerzuordnung beeinflussen, sollten sie auch in den Lebendigkeitsgraphen durch Knoten repräsentiert werden. Zudem kann durch die *spill store edges* und *spill load edges* allein nicht modelliert werden, dass sich die Registerzuordnung zwischen den Anfangs- und Endknoten der gerichteten Kanten ändert. Jeder Kante ist genau eine Menge von Entscheidungsvariablen zugeordnet, die einen Zustand, aber isoliert betrachtet, keine Zustandsänderung beschreibt. Bevor die Registerallokationsbedingungen durch Constraints formuliert werden, wird daher auf jeder *spill store edge* und jeder *spill load edge* eine Transformation angewendet. Sie ersetzt die Kante $e = (v_1, v_2)$ durch einen Knoten v_{spill} und zwei Kanten $e_1 = (v_1, v_{spill})$ und $e_2 = (v_{spill}, v_2)$ [1]. Der Knoten v_{spill} modelliert dabei die Spillinstruktion, die möglicherweise in die Zwischendarstellung eingefügt wird.

Die Menge von Entscheidungsvariablen, die der Kante e_1 zugeordnet ist, ist die gleiche Menge von Entscheidungsvariablen, die der Kante e zugeordnet war. Das Einfügen einer Instruktion hat schließlich keine Auswirkung auf eine Registerallokation vor der Instruktion. Welche Menge der Kante e_2 zugeordnet wird, hängt davon ab, ob die Kante e eine *spill store edge* oder *spill load edge* war. War die Kante e eine *spill store edge*, so repräsentiert der neu eingefügt Knoten v_{spill} eine Instruktion I_{store}, die das virtuelle Register s speichert und damit nutzt. Wie auch bei den andern Kanten, die von einem Knoten ausgehen, der eine Instruktion repräsentiert, die s nutzt, besteht daher die Menge, die der Kante e_2 im Lebendigkeitsgraphen LRG_s zugeordnet ist, nur aus einer Kontinuitätsvariable $x^{cont}_{s \to p}$. Dies ist in Abbildung 4.4 noch einmal dargestellt. In dieser Abbildung sind die Knoten, die die Kante e verbindet, durch leere Rechtecke dargestellt. Dies heißt aber nicht, dass sie diese Instruktionen repräsentieren, die das virtuelle Regis-

[1] Da auf Grund der Transformation der *spill store edges* und der *spill load edges* neue Kanten in die Lebendigkeitsgraphen eingefügt werden, gibt es Knoten, deren eingehende Kanten vor Anwendung der Transformation andere Mengen von Entscheidungsvariablen zugeordnet waren als nach der Anwendung der Transformation. In jedem Fall gelten zwischen den Mengen zweier Kanten, die einen gemeinsamen Knoten haben die Beziehungen, die in Abschnitt 4.2 beschrieben wurde. Die Mengen, die durch die Abbildung $DV_{s \to p}$ den nicht transformierten Kanten zugeordnet werden, müssen gegebenenfalls so angepasst werden, dass die Beziehungen wieder gelten.

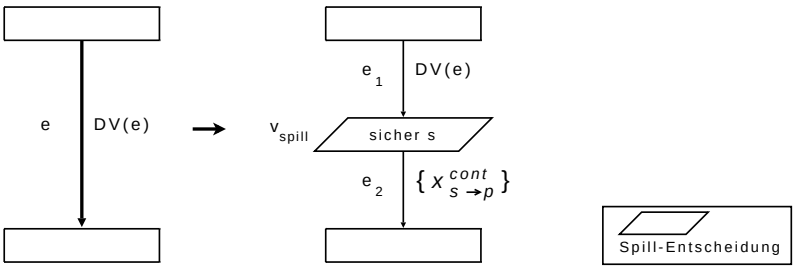

Abbildung 4.4: Transformation einer *spill store edge*.

ter s weder definieren noch nutzen dürfen. Sie sind in diesem Fall Platzhalter für beliebige Knoten. Bei dem Knoten, von dem die Kante e ausgeht, kann es sich aber nur um das Ende eines Basisblocks oder eine Instruktion handeln, die das virtuelle Register s definiert. Ansonsten wäre die Kante e keine *spill store edge*.

Die Kontinuitätsvariable $x_{s\to p}^{cont}$ nimmt den Wert Eins an, wenn die Registerzuordnung des Registers s zum physikalischen Register p nach dem Sichern des Registers s auch weiterhin vorliegt, und anderenfalls den Wert Null. Für die Kontinuitätsvariable muss wiederum gelten, dass sie nur dann den Wert Eins nehmen kann, wenn das virtuelle Register s auch unmittelbar vor der Instruktion I_{store} dem physikalischen Register p zugeordnet war, was durch diese Ungleichung formuliert wird:

$$\sum_{x_{s\to p}\in DV_{s\to p}(e_1)} x_{s\to p} \geq x_{s\to p}^{cont} \tag{4.7}$$

Die Entscheidung, ob auch wirklich eine Spillinstruktion, die das virtuelle Register s sichert, in die Zwischendarstellung eingefügt wird, wird durch eine neue Entscheidungsvariable $x_{s\to p}^{store}$ modelliert. Sie hat den Wert Eins, wenn eine Speicherinstruktion eingefügt und das virtuelle Register s gesichert wird. Wird die Instruktion nicht eingefügt, hat sie den Wert Null. Natürlich kann das virtuelle Register s nur dann gespeichert werden, wenn es in dem physikalischen Register p auch vorliegt. Es muss daher gelten, dass der Wert der Variable $x_{s\to p}^{store}$ nur dann Eins sein kann, wenn eine Variable den Wert Eins hat, die die Registerzuordnung des virtuellen Registers s zum physikalischen Register p unmittelbar vor der Spillinstruktion I_{store} modelliert. Es muss also diese Ungleichung gelten:

$$\sum_{x_{s\to p}\in DV_{s\to p}(e_1)} x_{s\to p} \geq x_{s\to p}^{store} \tag{4.8}$$

War die Kante e in dem Lebendigkeitsgraphen LRG_s eine *spill load edge*, so repräsentiert der neu eingefügte Knoten v_{spill} eine Instruktion I_{load}, die das virtuelle Register s in das physikalische Register p lädt. Die Entscheidung, ob diese Instruktion auch wirklich in die Zwischendarstellung eingefügt, und damit das virtuelle Register s in das physikalische Register p geladen wird, wird durch eine neue Entscheidungsvariable $x_{s\to p}^{load}$ modelliert. Nimmt sie den Wert Eins an, wird die Instruktion eingefügt, anderenfalls nicht. Da das Laden des virtuellen Registers s auch eine Definition von s darstellt, müssen für die Instruktion I_{load} die Nebenbedingungen aufgestellt werden, die durch die Ungleichungen 4.6 beschrieben sind. Sie stellen sicher, dass jedem physikalischen Register unmittelbar nach I_{load} höchstens ein virtuelles Register gleichzeitig zugeordnet ist.

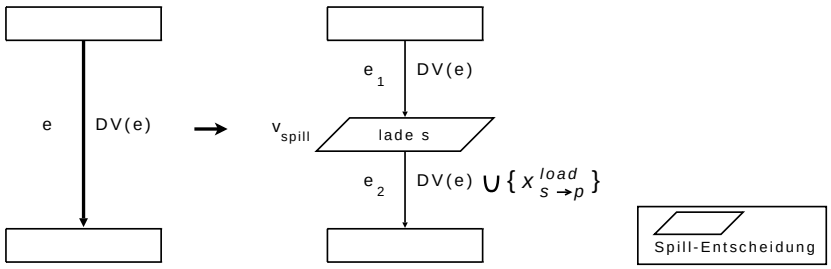

Abbildung 4.5: Transformation einer *spill load edge*.

Zusätzlich zu der Möglichkeit, dass das virtuelle Register s schon unmittelbar vor der Ladeinstruktion dem physikalischen Register p zugeordnet war, kann nun auch durch das Laden des virtuellen Registers s erreicht werden, dass sich s in p befindet. Die Menge von Entscheidungsvariablen, die der Kante e_2 durch die Abbildung $DV_{s\to p}$ zugeordnet ist, ist somit gleich der Menge $DV_{s\to p}(e_1)$, die um die Entscheidungsvariable $x^{load}_{s\to p}$ erweitert ist. Diese Transformation der *spill load edges* ist in Abbildung 4.5 dargestellt. Das virtuelle Register s, kann aber nur von einer Instruktion aus dem Speicher geladen werden, wenn es auf jedem Pfad zu der Instruktion mindestens einmal in den Speicher geschrieben wurde. Daraus ergibt sich die Notwendigkeit, dass auch der Zustand des Speichers modelliert werden muss, in den s ausgelagert werden kann.

Zunächst soll aber wieder das Beispiel aus Abbildung 4.3 berachtet werden. Werden alle Kanten in dem Beispielgraphen transformiert, die *spill store edges* und *spill load edges* sind, entsteht der Lebendigkeitsgraph, der in Abbildung 4.6 dargestellt ist. Die Kanten des Lebendigkeitsgraphen sind mit den Mengen von Entscheidungsvariablen beschriften, die ihnen durch die Abbildung $DV_{s\to p}$ zugeordnet ist. Erstmals wird nun deutlich, dass die Mengen mehr als eine Entscheidungsvariablen enthalten können.

Die Speicherstelle, in die ein virtuelles Register ausgelagert werden kann, kann bei der Modellierung wie ein physikalisches Register gehandhabt werden, das permanent und ausschließlich für dieses eine virtuelle Register zur Verfügung steht. Erneut wird ein Graph verwendet, der für die Zuordnung von Entscheidungsvariablen zu Stellen im Kontrollflussgraph genutzt wird. Da der Graph die Speicherstelle eines virtuellen Registers modelliert, wird er als Speichergraph (*memory graph* (MG)) bezeichnet. Weil ein virtuelles Register an genau eine Stelle im Speicher ausgelagert werden kann, gibt es für jedes virtuelle Register s auch genau einen Speichergraph MG_s. Die Definition des Speichergraphen unterscheidet sich nicht von der eines Lebendigkeitsgraphen, d. h. er weist die gleiche Struktur auf wie der Lebendigkeitsgraph von s nach der Anwendung der Transformationen auf die *spill store edges* und *spill load edges*.

Abermals weist eine Abbildung $DV_s : E \mapsto \mathcal{P}\{x_1, \ldots, x_n\}$ den Kanten des Speichergraphen Mengen von Entscheidungsvariablen zu. Sie geben an, ob das virtuelle Register s in die Speicherstelle geschrieben wurde. Hat eine Variable aus der Menge $DV_s(e)$ den Wert Eins, bedeutet dies, dass das virtuelle Register s an der Stelle, die durch die Kante e beschrieben wird, in der Speicherstelle vorliegt. Sei v^{load} ein Knoten, der die nächste Spillinstruktion repräsentiert, die das virtuelle Register s lädt. Haben alle Variablen den Wert Null, wurde das virtuelle Register s nicht auf allen Pfaden gespeichert, die zu einem Knoten v^{load} führen und die Kante e enthalten.

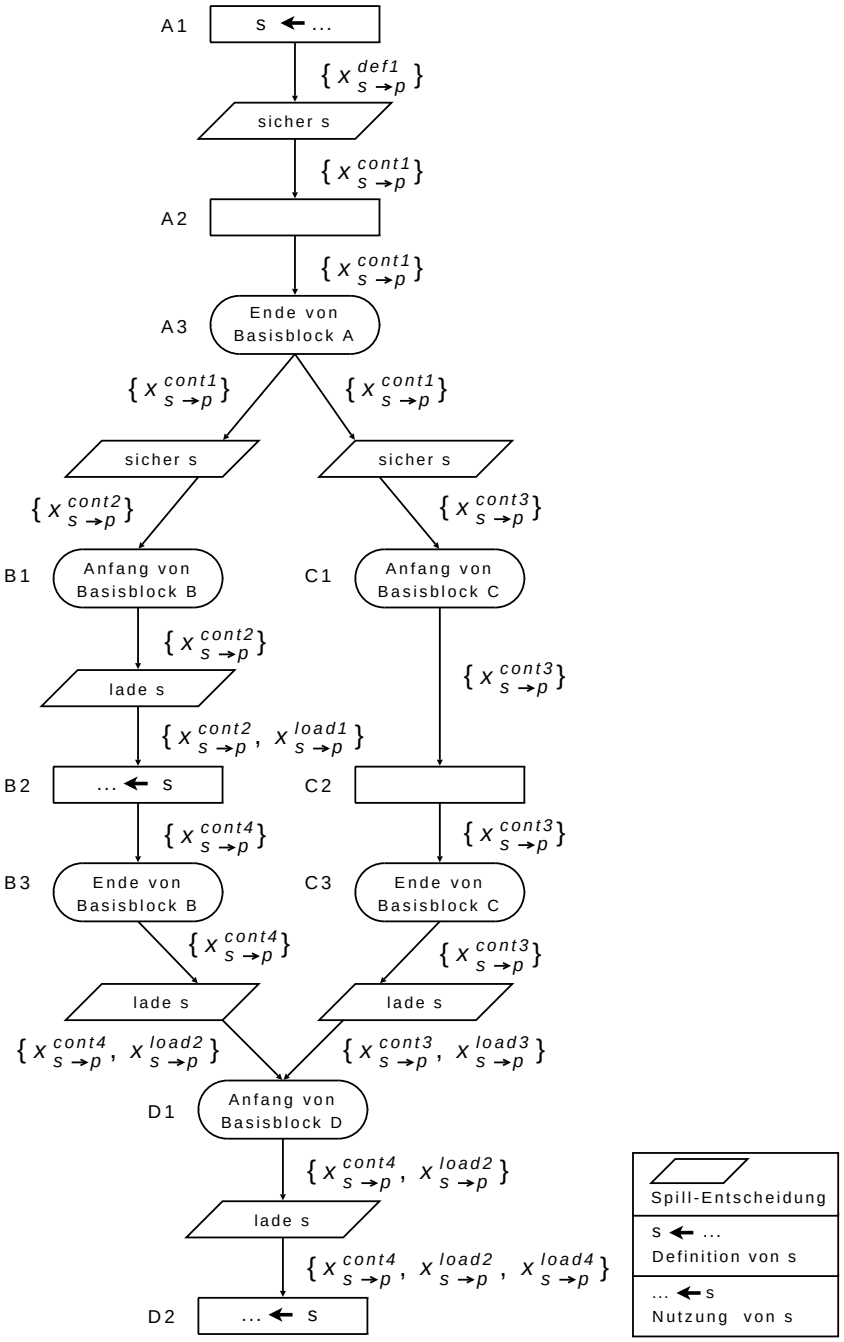

Abbildung 4.6: Lebendigkeitsgraph nach der Modellierung der Spill-Entscheidungen.

Für die Mengen $DV_s(e_1)$ und $DV_s(e_2)$ zweier Kanten e_1 und e_2, die einen gemeinsamen Knoten haben, der keine Spillinstruktion repräsentiert, gelten auch weiterhin die Beziehungen, die in Abschnitt 4.2 beschrieben wurden. Zu beachten ist, dass es nun aber keine Instruktionen außer den Spillinstruktionen gibt, die die Speicherstelle des virtuellen Registers s definiert oder nutzt. Den Kanten, die aus Knoten herausführen, die Instruktionen repräsentieren, die das virtuelle Register s definieren und die keine Spillinstruktionen sind, ist die leere Menge zugeordnet. An diesen Stellen kann der neu definierte Wert des virtuellen Registers s nicht bereits auch in der Speicherstelle von s vorliegen .

Ein Knoten v_{store} in dem Speichergraphen MG_s, der eine Spillinstruktion I_{store} repräsentiert, die das virtuelle Register s im Speicher sichert, wird wie ein Knoten gehandhabt, der für eine Ladeinstruktion in einem Lebendigkeitsgraphen steht. Für jedes physikalische Register p, dem das virtuelle Register s zugeordnet werden kann, gibt es bereits eine Entscheidungsvariable $x_{s\to p}^{store}$. Sie gibt an, ob das virtuelle Register s gesichert wird, indem der Inhalt des physikalischen Registers p durch die Spillinstruktion I_{store} gespeichert wird. Dabei handelt es sich um die selben Entscheidungsvariablen, die die Entscheidung modellieren, ob die Instruktion I_{store} in die Zwischendarstellung eingefügt wird. Da es für das Vorliegen des virtuellen Registers s in der Speicherstelle unerheblich ist, aus welchem physikalischen Register heraus es in die Speicherstelle geladen wurde, kann das virtuelle Register s aus zwei Gründen in der Speicherstelle vorhanden sein. Es kann in der Speicherstelle vorliegen, weil es schon vor der Spillinstruktion I_{store} im Speicher vorlag, was durch die Menge $LiveInDV_s(I_{store})$ angegeben wird, oder weil die Spillinstruktion I_{store} eingefügt wurde, was durch die Entscheidungsvariablen $x_{s\to p}^{store}$ angegeben wird. Die Menge der Entscheidungsvariablen $LiveOutDV_s(I_{store})$ ist daher die Menge $LiveInDV_s(I_{store})$, die um all die Variablen $x_{s\to p}^{store}$ erweitert wurde.

In Abbildung 4.7 ist der zu dem Lebendigkeitsgraphen aus Abbildung 4.6 zugehörige Memorygraph dargestellt. Es wurden diesmal die Mengen von Entscheidungsvariablen für den Fall angegeben, dass das virtuelle Register s zwei physikalischen Registern $p1$ und $p2$ zugeordnet werden kann. Dadurch wird noch einmal deutlich, dass alle Entscheidungsvariablen bei der Modellierung der Speicherstelle von s zusammengeführt werden, die das Einfügen einer Spillinstruktion modellieren, die das virtuelle Register s sichern.

Analog dazu wird in dem Speichergraphen MG_s ein Knoten v_{load}, der eine Spillinstruktion I_{load} repräsentiert, die das virtuelle Register s aus dem Speicher lädt, wie ein Knoten gehandhabt, der eine Speicherinstruktion in einem Lebendigkeitsgraphen darstellt. Ob eine Ladeoperation I_{load}, die das virtuelle Register s in das physikalische Register p lädt, in die Zwischendarstellung eingefügt wird, wird bereits durch Entscheidungsvariablen $x_{s\to p}^{load}$ angegeben. Nun wird die Bedingung durch ein Constraint beschrieben, die sicherstellt, dass das virtuelle Register s nur dann geladen werden kann, wenn es im Speicher vorliegt. Sie kann für alle diese Entscheidungsvariable $x_{s\to p}^{load}$ durch ein einzelnes Constraint so formuliert werden:

$$\sum_{x_s \in LiveInDV_s(I_{load})} x_s \geq \sum_p x_{s\to p}^{load} \tag{4.9}$$

Da das virtuelle Register s in seiner Speicherstelle nicht überschrieben wird, liegt es dort auch weiterhin vor, nachdem es geladen wurde. Die Menge $LiveOutDV_s(I_{load})$ besteht daher aus genau einer Kontinuitätsvariable $x_s^{memory\_cont}$. Die Kontinuitätsvariable gibt an, ob das virtuelle Register s in jedem Fall bei einer folgenden Spillinstruktion, die

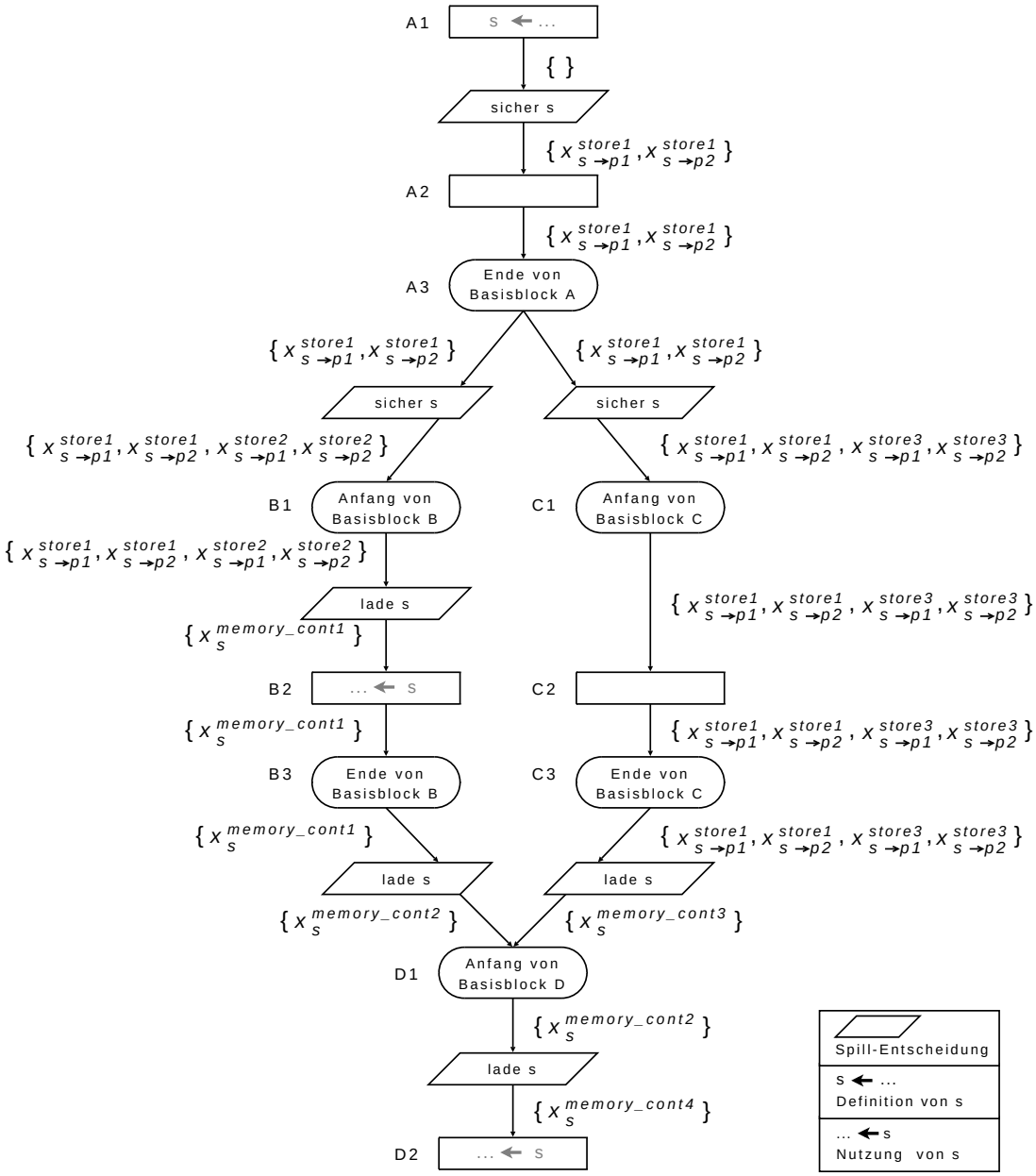

Abbildung 4.7: Memorygraph für ein virtuelles Register s, das zwei physkalischen Registern $p1$ und $p2$ zugeordnet werden kann.

das virtuelle Register s lädt, im Speicher vorliegt und nicht nur, ob das virtuelle Register unmittelbar nach der betrachteten Spillinstruktion I_{load} weiterhin im Speicher vorliegt. Sie hat den Wert Eins, wenn ersteres erfüllt ist, anderenfalls hat sie den Wert Null. Zusätzlich muss sie die Bedingung erfüllen, die durch die Ungleichung 4.1 beschrieben wird. Die Kontinuitätsvariable muss dabei auf Grund ihrer Bedeutung nicht immer die Gleichheit erfüllen. Sie kann an einem Knoten v_B^{start}, der den Anfang eines Basisblocks mit mehreren Vorgängern repräsentiert, mit der Summe einer anderen Menge von Entscheidungsvariablen gleichgesetzt werden. Diese Menge gibt aber vielleicht an, dass das virtuelle Register s auf einem anderen Pfad nicht mit Sicherheit gesichert wird. Die Summe der Entscheidungsvariablen aus dieser Menge ist dann Null. Die Kontinuitätsvariable müsste aber ohne das Constraint an dieser Stelle, die durch den Knoten v_{load} repräsentiert wird, nicht den Wert Null annehmen, wenn das virtuelle Register s in der Speicherstelle vorlag. Für alle vom Knoten v_B^{start} ausgehenden Pfade, die zu einem Knoten führen, der eine Spillinstruktion repräsentiert, die das virtuelle Register s lädt, darf dann nicht angegeben werden, dass s im Speicher vorliegt, wenn nicht auf allen Pfaden s in den Speicher geschrieben wird.

4.5 Modellierung von Calling-Conventions

Calling conventions ordnen physikalische Register der Menge der *callee-saved register* und der Menge der *caller-saved register* zu. Einem *caller-saved register* darf an der Stelle, an der der Funktionsaufruf erfolgt, kein virtuelles Register zugeordnet sein. Sei die Instruktion, die zu einem Funktionsaufruf führt, die Instruktion I_{call}, die in dem Lebendigkeitsgraphen LRG_s durch Knoten $v_{s\to p}^{call}$ repräsentiert wird. Dann muss für jedes virtuelle Register s, das unmittelbar vor der Instruktion I_{call} lebendig ist, und nicht durch diese Instruktion definiert wird, erzwungen werden, dass die Entscheidungsvariablen, die eine Zuordnung des virtuellen Registers s auf ein *caller-saved register* p beschreiben, den Wert Null annehmen. Die Entscheidungsvariable wird dafür einfach gleich Null gesetzt.

Für ein *callee-saved register* muss gelten, dass vor der ersten Verwendung des Registers in der Funktion, der Wert, den es enthält, in den Speicher gesichert wird, und wiederhergestellt wird, bevor die Funktion wieder verlassen wird. Aus diesem Grund wird jedes *callee-saved register* p durch ein neues virtuelles Register s modelliert, das zu Beginn der Funktion dem physikalischen *callee-saved register* p zugeordnet ist. Dies kann erreicht werden, indem am Anfang der Funktion eine Pseudo-Instruktion eingefügt wird, die das virtuelle Register s definiert. Natürlich kann das virtuelle Register s nur dem physikalischen Register p zugeordnet werden. Zusätzlich wird unmittelbar vor jedem Funktionsende eine Pseudo-Instruktion in die Zwischendarstellung der Funktion eingefügt, die das virtuelle Register s nutzt. Dadurch wird erzwungen, dass das *callee-saved register* am Ende der Funktion wiederhergestellt ist.

Da das *callee-saved register* während der gesamten Funktion lebendig ist, werden auch für das virtuelle Register s wie für alle anderen virtuellen Register ein Lebendigkeitsgraph LRG_s, ein Speichergraph MG_s und die dazugehörigen Abbildungen erstellt. Das virtuelle Register s wird wie die anderen Register gehandhabt und daher, bevor es aus p verdrängt wird, in den Speicher gesichert und spätestens vor dem Funktionsende wieder geladen. Auf diese Weise wird auch für *callee-saved register* optimal Spillcode eingefügt.

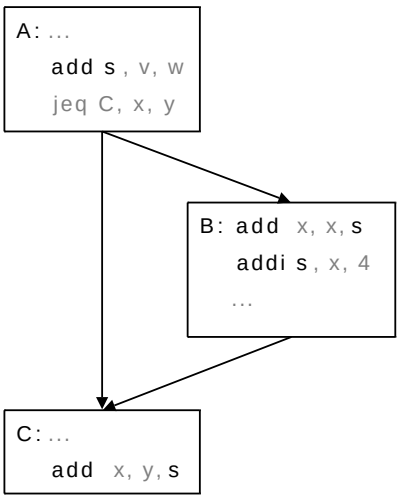

Abbildung 4.8: Beispiel für einen Kontrollflussgraphen.

4.6 Anmerkungen zur praktischen Umsetzung

Bei der Beschreibung ihres ORA-Ansatzes lassen Goodwin und Wilken [GW96] wichtige Aspekte aus, die bei einer praktischen Anwendung ihren Modells unbedingt beachtet werden müssen. Das größte Hindernis bei der Anwendung des ORA-Ansatzes tritt durch die Modellierung der Spillentscheidungen auf. Sei s ein virtuelles Register mit dem Lebendigkeitsgraphen, der in Abbildung 4.9 (a) dargestellt ist. Alle Kanten, die von dem Knoten $A3$ ausgehen, der das Ende eines Basisblocks A mit mehreren Nachfolgern repräsentiert, sind *spill store edges*. Folglich wird eine Transformation auf die Kanten angewendet, die die Kanten teilt und neue Knoten einfügt. Diese Transformation wurde bereits in Abbildung 4.4 veranschaulicht und führte zu dem Graphen in Abbildung 4.6. Die neuen Knoten repräsentieren Stellen, an denen Entscheidungen über das Einfügen von Spillinstruktionen getroffen werden, die das virtuelle Register s auf dem Laufzeitstack sichern. Unglücklicherweise kann an solch einer Stelle keine Spillinstruktion eingefügt werden, da sie sich zwischen zwei Basisblöcken befindet.

Der Lebendigkeitsgraph 4.9 ging aus dem Kontrollflussgraphen hervor, der in Abbildung 4.8 dargestellt ist. A ist der Basisblock vor der Stelle und B ist Basisblock, der auf die Stelle folgt, an der die Entscheidung über das Einfügen der Spillinstruktionen getroffen wird. Ein neuer Basisblock D, der nur die Spillinstruktion enthält, die zwischen den beiden Basisblöcken A und B liegt, kann nicht einfach zwischen A und B in die Zwischendarstellung eingefügt werden. Wenn B am Ende keinen Sprungbefehl enthält, müsste in D ein Sprungbefehl eingefügt werden, der einen Sprung zu Basisblock C ausführt. Einen solchen Sprungbefehl in die Zwischendarstellung einzufügen ist der Registerallokation aber nicht erlaubt. Soll eine Spillinstruktion eingefügt werden, muss sie einem der beiden Basisblöcke zugeordnet werden. Dadurch kann die Optimalität des Modells verloren geht.

Das virtuelle Register s wird in dem Lebendigkeitsgraphen aus Abbildung 4.9 (b) von den Instruktionen $A1$ und $B3$ definiert und von den Instruktionen $B2$ und $C3$ genutzt. Die Instruktionen $A2$ und $B4$ sind hier Platzhalter für eine Folge von Instruktionen, die das virtuelle Register s weder nutzen noch definieren. Bei ihnen stehe für das virtuelle Register s kein physikalisches Register zur Verfügung. Das virtuelle Register s müsste

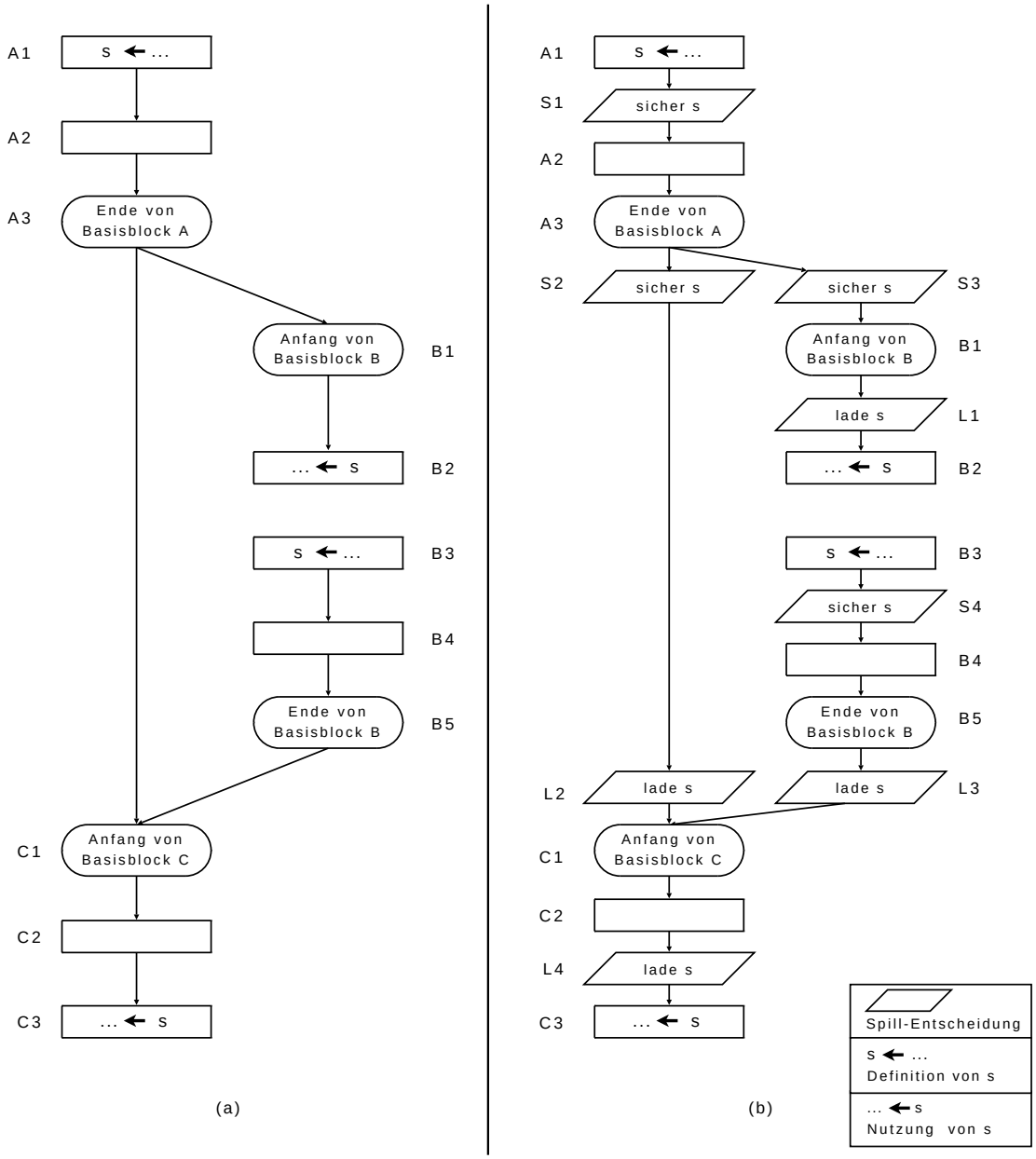

Abbildung 4.9: Lebendigkeitsgraphen eines virtuellen Register s. (a) Lebendigkeitsgraph vor der Berücksichtigung von Spill-Entscheidungen. (b) Lebendigkeitsgraph nach der Berücksichtigung von Spill-Entscheidungen.

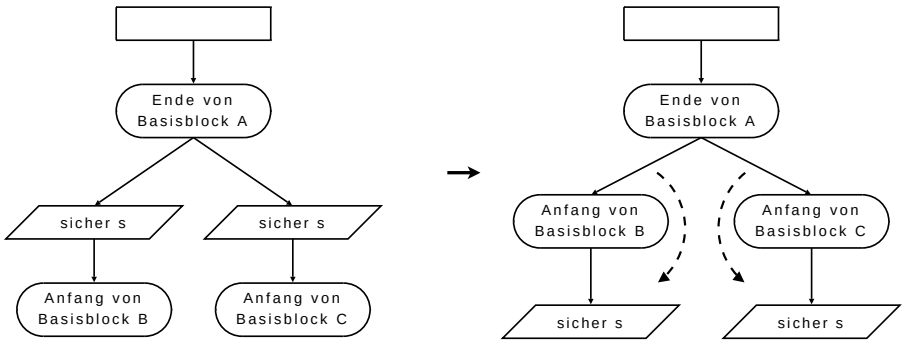

Abbildung 4.10: Verschiebung der Entscheidungsstellen.

sich daher bei Ausführung der Instruktionen $A2$ und $B4$ im Speicher befinden. Würde das virtuelle Register an den Stellen $S2$ und $S4$ in den Speicher geschrieben werden, würde im schlimmsten Fall auf allen Ausführungspfaden zwischen den Instruktionen $A1$ und $C3$, nur jeweils eine Speicherinstruktion ausgeführt werden. Würde dagegen die Speicherinstruktion von der Stelle $S2$ vor $A3$ oder zwischen $C1$ und $C2$ verschoben werden, müssten auf dem Kontrollpfad, der über Basisblock B führt, zwei Speicherinstruktionen ausgeführt werden. Zum einen würde sich entweder die von der Stelle $S2$ verschobene Spillinstruktion auf dem Ausführungspfad befinden oder die Spillinstruktion an der Stelle $S1$ wird eingefügt, damit das virtuelle Register vor der Instruktion $C2$ gesichert ist. Zum anderen müsste auch die Spillinstruktion unmittelbar nach $B3$ an der Stelle $S4$ eingefügt werden, da die Instruktion $B3$ das virtuelle Register s neu definiert, es aber bei Instruktion $B4$ aus dem physikalischen Register verdrängt wird.

Für das Verschieben der Spillinstruktion in den Vorgängerbasisblock A oder den nachfolgenden Basisblock B gibt es jeweils Gründe, die dagegen sprechen. Die Spillinstruktion in den vorhergehenden Basisblock A einzufügen hätte den Nachteil, dass dann die Instruktion immer am Ende des Basisblocks ausgeführt werden würde, auch wenn für allen andern von A ausgehenden *spill store edges* entschieden wurde, keine Spillinstruktion einzufügen. Ein ähnliches Argument kann angeführt werden, wenn die Spillinstruktion in den Nachfolgebasisblock B eingefügt wird, und der Basisblock mehrere eingehende Kanten hat.

Unproblematisch ist es dagegen, wenn der Basisblock B nur den Vorgänger A hat. Dann kann ohne einen Nachteil die Spillinstruktion in den nachfolgenden Basisblock eingefügt werden. Dies kann der Abbildung 4.10 entnommen werden. Wie für die optimale Platzierung der Spillinstruktion zwischen den Basisblöcken gilt weiterhin, dass die Spillinstruktion nach der letzten Instruktion des vorhergehenden Basisblocks ausgeführt wird und nur vor der ersten Instruktion des nachfolgenden Basisblocks.

Für das Einfügen der Spillinstruktion am Ende eines Basisblocks gibt es dagegen keinen solchen unproblematischen Fall. *Spill store edges* verlaufen nur dann zwischen Basisblöcken, wenn der vorhergehende Basisblock A mehrere Nachfolger hat. Natürlich gibt es auch Spezialfälle, bei denen das Einfügen der Spillinstruktion am Ende eines Basisblocks Vorteile hat. Um aber eine einheitliche Regelung zu schaffen, die in der Mehrheit der Fälle vorteilhaft ist, werden in dieser Diplomarbeit die Spillinstruktionen, die ein virtuelles Register zwischen zwei Basisblöcken sichern sollen, immer an den Anfang des nachfolgenden Basisblocks eingefügt.

Damit die Bedingungen der Registerallokation eingehalten werden, muss auch die Entscheidungsstelle schon bei der Modellierung des Registerallokationsproblems in den Basisblock verschoben werden. Anderenfalls ist z. B. nicht gewährleistet, dass dort, wo die Spillinstruktion eingefügt wird, sich das virtuelle Register s auch in einem physikalischen Register befindet und gesichert werden kann.

Für die Spillinstruktionen, die ein virtuelles Register in ein physikalisches Register laden, kann das gleiche Problem auftreten. Repräsentiert ein Knoten v_B^{start} in einem Lebendigkeitsgraphen den Anfang eines Basisblocks B mit mehreren Vorgängern, sind alle Kanten, die auf v_B^{start} gerichtet sind, *spill load edges*. Auch auf sie wird eine Transformation angewendet, die zur Folge hat, dass sich eine Entscheidungsstelle über das Einfügen einer Spillinstruktion, die das virtuelle Register lädt, zwischen zwei Basisblöcken befindet. Da die Rollen von dem vorhergehenden und dem nachfolgenden Basisblock vertauscht sind, werden Entscheidungsstellen über das Einfügen der Ladeinstruktionen in den vorhergehenden Basisblock verschoben.

Hierbei muss darauf geachtet werden, dass die letzte Instruktion des Basisblocks ein Sprungbefehl oder ein Funktionsaufruf sein kann. Nach einem Funktionsaufruf wird mit der Instruktion fortgefahren, die auf den Aufruf folgt. Ein Basisblock B, der einen Funktionsaufruf am Ende enthält, hat daher nur einen Nachfolger. Zwischen dem Basisblock B und seinem Nachfolger kann in diesem Fall ein neuer Basisblock eingefügt werden, in dem nur die Spillinstruktion ausgeführt wird. Sein Nachfolger ist der einzige Nachfolger des Basisblocks B. Folglich ist kein Sprungbefehl am Ende des neuen Basisblocks nötig, da auch B keinen Sprungbefehl enthielt.

Enthält dagegen ein Basisblock B am Ende einen Sprungbefehl, muss die Stelle, an der die Entscheidung über das Einfügen einer Spillinstruktion betroffen wird, sogar noch vor den Sprungbefehl am Ende des Basisblocks verschoben werden. Da auch die Bedingungen, die für eine gültige Registerallokation nötig sind, für diese Stelle aufgestellt werden, ist garantiert, dass der Wert keines Registers, das gegebenenfalls für die Auswertung einer Sprungbedingung benötigt wird, von der Spillinstruktion überschrieben wird.

Ein ähnlicher Aspekt, den Goodwin und Wilken nicht behandeln, ist die Definition eines virtuellen Registers s durch bedingte Sprünge. Der Infineon TriCore-Prozessor hat zwei bedingte Sprungbefehle `jnei` (*jump if not equal and increment*) und `jned` (*jump if not equal and decrement*), die unabhängig von der Sprungbedingung den Wert eines Registers um Eins erhöhen bzw. erniedrigen. Da sie den Programmfluss verzweigen, sind sie immer die letzten Instruktionen in einem Basisblock. Nach ihnen kann keine Spillinstruktion zum Sichern des Registers eingefügt werden. Die ist aber auch nicht notwendig, wie aus Abbildung 4.11 hervorgeht. Der Basisblock B mit einem bedingten Sprung am Ende, der ein virtuelles Register s definiert, hat zwei Nachfolger. Seine ausgehenden Kanten sind daher *spill store edges*. Nach den vorherigen Überlegungen werden die Spill-Entscheidungsstellen in die nachfolgenden Basisblöcke C und D verschoben. Es wird also in beiden Basisblöcken zu Beginn eine Entscheidung über das Einfügen von Spillinstruktionen getroffen, die das virtuelle Register s sichern. Direkt nach der Definition des virtuellen Registers s besteht daher schon die Möglichkeit, den Wert des Registers zu sichern. Für bedingte Sprünge kann also die normalerweise unmittelbar auf die Definition folgende Spillentscheidung einfach weggelassen werden.

In ihrer Beschreibung erwähnen Goodwin und Wilken ebenfalls nicht, wie sie mit virtuellen Registern umgehen, deren Lebendigkeit nicht an einer Definition beginnt und mit einer Nutzung endet. Zu solchen Registern zählen zum z. B. Funktionsparameter und

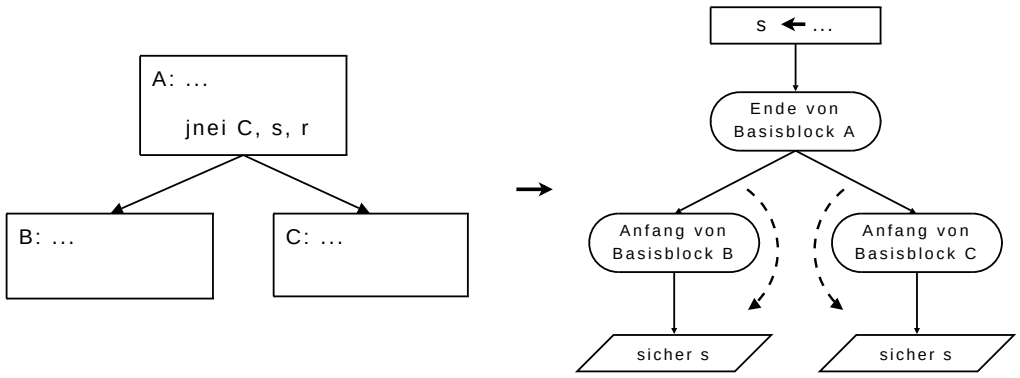

Abbildung 4.11: Kontrollflussgraph mit einem bedingten Sprung und resultierender Lebendigkeitsgraph. Die Pfeile mit geschrichelten Linien geben die Verschiebung der Entscheidungsstellen an.

Rückgabewerte einer Funktion. Da Funktionsparameter bereits bei der ersten Instruktion lebendig sind, wird für sie einfach der Knoten in dem Lebendigkeitsgraphen von s, der den Start des ersten Basisblocks repräsentiert, als die Stelle aufgefasst, an der die Definition des virtuellen Registers erfolgt. Genauso werden für die Rückgabewerte an Funktionsenden Bedingungen wie bei der Nutzung eines Registers aufgestellt (siehe Ungleichung 4.4).

Für virtuelle Register, die bei einer Nutzung möglicherweise undefiniert sind, gibt es in dem Kontrollflussgraphen einen Pfad von dem Funktionsanfang bis zu der Nutzung, auf der keine Definition erfolgt. Auch diese Register sind nach den Definitionen 2.6 und 2.7 bereits vor der ersten Instruktion lebendig. Für sie werden am Funktionsanfang die Mengen der Entscheidungsvariablen festgelegt und nur die *must-allocate*-Bedingung aufgestellt (siehe Gleichung 4.3). Da es am Funktionsanfang mehr gleichzeitig lebendige Register als physikalische Register geben kann, könnte das Aufstellen der *single-symbolic*-Bedingung (siehe Ungleichung 4.6) für den Funktionsanfang zu einem unlösbaren ILP führen. Es wird daher erst vor der ersten Instruktion die *single-symbolic*-Bedingung formuliert. Auf diese Weise könnten zum Funktionsanfang mehrere virtuelle Register einem physikalischen Register zugeordnet sein. Aber höchstens eins von den virtuellen Registern ist ein Funktionsparameter, alle anderen Register sind Register, die bei einer Nutzung möglicherweise undefiniert sind. Da die *single-symbolic*-Bedingung aber vor der ersten Instruktion formuliert wird, ist sichergestellt, dass überzählige Register in den Speicher ausgelagert werden und unmittelbar vor der ersten Instruktion nur noch ein virtuelles Register pro physikalischem Register vorliegt.

Bei einem virtuellen Register s, das eine Definition hat, auf die keine Nutzung folgt, ist das virtuelle Register nach der Definition nicht mehr lebendig, und in dem Lebendigkeitsgraphen gibt es keine Kante, die aus dem Knoten herausführt, der die Definition repräsentiert. Dennoch muss darauf geachtet werden, dass es für die Definition eine Entscheidungsvariable x^{def} gibt und diese bei der Aufstellung der *single-symbolic*-Bedingung berücksichtigt wird. Anderenfalls könnte durch die Definition von s der Wert eines anderen virtuellen Registers überschrieben werden.

Kapitel 5

WCET-Modellierung

In dem vorherigen Kapitel wurde vorgestellt, wie das Registerallokationsproblem für eine Funktion als ein ILP formuliert werden kann. Dabei wurde die Zielfunktion der Registerallokation erst einmal außer Acht gelassen. In diesem Kapitel wird nun erläutert, wie in dieser Diplomarbeit das ILP um die Zielfunktion erweitert wird, damit es das Registerallokationsproblem einer Funktion vollständig beschreibt.

Goodwin und Wilken [GW96] beschreiben, wie die Zielfunktion des Registerallokationsproblems durch die Zielfunktion des ILP modelliert werden kann, wenn jede Spillinstruktion eine Auswirkung auf die Zielfunktion hat. Dies ist dann der Fall, wenn die Zielfunktion der Registerallokation z. B. die Codegröße oder die durchschnittliche Ausführungszeit der Funktion ist. Bei der WCET wirken sich aber nur die Spillinstruktionen auf die Zielfunktion aus, die auf dem WCEP liegen. Da in dieser Diplomarbeit ein globales ILP-basiertes Registerallokationsverfahren entwickelt werden soll, das die WCET der betrachteten Funktion minimiert, muss mit Hilfe weiterer Entscheidungsvariablen und Constraints auf irgend eine Weise der WCEP der Funktion modelliert werden. Schließlich muss die Zielfunktion des ILP die WCET der Funktion wiedergeben. Durch das Minimieren der Zielfunktion des ILP wird dann erreicht, dass Spillcode so in die Funktion eingefügt wird, dass die WCET der Funktion minimiert wird.

Für den ORA-Ansatz wurde bezüglich der Zielfunktion die Annahme getroffen, dass die Ausführungshäufigkeiten der Basisblöcke, die einen Einfluss auf die Zielfunktion haben, die eines Kontrollflusses sein können. Durch die Modellierung der WCET haben genau die Basisblöcke einen Einfluss auf den Zielfunktionswert, die auf dem WCEP liegen. Somit trifft diese Annahme bei einer genauen Modellierung der Pfade in einem Kontrollflussgraphen zu.

Ein Verfahren, mit dem ein ILP formuliert werden kann, bei dem der Kontrollfluss einer Funktion durch Constraints modelliert wird und die Zielfunktion die WCET der Funktion angibt, wurde bereits in Abschnitt 3.4 genannt. Mittels der *implicit path enumeration technique* (IPET) [LM95] können implizit alle Pfade der Funktion modelliert werden. Durch das Maximieren der Zielfunktion wird schließlich erreicht, dass der Pfad mit der längsten Ausführungsdauer gebildet wird. Da diese Technik den längsten Pfad bestimmt, indem sie ihn durch Wahl der Ausführungshäufigkeiten der Basisblöcke bildet, ist sie für die Erweiterung des ILP ungeeignet, das das Registerallokationsproblem beschreiben soll. Denn hängt die Ausführungsdauer der Pfade von dem Spillcode ab, der auf dem Pfad eingefügt wird, wird ein Pfad mit maximaler Ausführungszeit gebildet, indem möglichst viel Spillcode auf den Pfad eingefügt wird. Dies erzeugt einen Pfad mit maximaler Ausführungszeit, minimiert aber nicht die Ausführungszeit des WCEP.

Ein geeignete Modellierung der WCET schlagen Suhendra et al. [SMRC05] vor. Bei dieser Modellierung der WCET müssen die Ausführungshäufigkeiten der Basisblöcke vor-

ab bekannt sein. Da innerhalb einer Funktion nur die Basisblöcke mehrfach ausgeführt werden können, die zu einer Schleife gehören, müssen Schleifen identifiziert werden. In Abschnitt 5.1 wird dazu erst einmal definiert, was eine Schleife in einem Kontrollgraphen ist. Zusätzlich werden noch Begriffe eingeführt, die für die Erklärung der WCET-Modellierung benötigt werden.

In Abschnitt 5.2 wird vorgestellt, wie Suhendra et al. die WCET einer Funktion in einem ILP modellieren. Im darauf folgenden Abschnitt 5.3 wird die WCET-Modellierung beschrieben, die im Rahmen dieser Diplomarbeit genutzt wird und eine Erweiterung des Ansatzes von Suhendra et al. darstellt.

5.1 Schleifen in Kontrollflussgraphen

Eine Schleife L in einem Kontrollflussgraphen zu einer Funktion F ist eine Menge von Knoten, die Anweisungen repräsentieren, die wiederholt ausgeführt werden können. Wenn die Funktion F in einer Hochsprache spezifiziert ist, soll es nach Möglichkeit zu jeder Schleife, die in der Hochsprache notiert ist, eine Entsprechung im Kontrollflussgraphen geben. Eine Schleife im Kontrollflussgraphen soll also die gleichen Anweisungen umfassen, wie eine Schleife in der Hochsprache. Insbesondere sollen auch Beziehungen zwischen den Schleifen berücksichtigt werden, die angeben, ob eine Schleife in einer anderen enthalten ist.

Definition 5.1 *Eine Schleife L_{in} ist in eine Schleife L_{out} verschachtelt, und die Schleife L_{out} umschließt die Schleife L_{in}, wenn L_{in} eine echte Teilmenge von L_{out} ist.*

Definition 5.2 *Eine Schleife L_{in} ist direkt in einer Schleife L_{out} verschachtelt, und die Schleife L_{out} umschließt die Schleife L_{in} direkt, wenn es keine Schleife L mit $L_{in} \subset L \subset L_{out}$ gibt.*

Definition 5.3 *Eine Schleife L ist eine äußerste Schleife, wenn es keine Schleife gibt, die L umschließt.*

Definition 5.4 *Eine Schleife L ist eine innerste Schleife, wenn es keine in L verschachtelte Schleife gibt.*

Eine Schleife enthält Anweisungen, die wiederholt ausgeführt werden können. Damit sie wiederholt ausgeführt werden können, müssen die Anweisungen in einem Kontrollflussgraphen durch Knoten repräsentiert sein, die sich auf einem Kreis befinden. Ein Kreis beschreibt aber nur einen Teil einer Schleife, die auf Grund von Alternativen mehrere Ausführungspfade erlaubt. Wie in Abbildung 5.1 zu erkennen ist, entstehen aus der Schleife, die in einer Hochsprache spezifiziert ist, ein Teil eines Kontrollflussgraphen, der mehrere Kreise enthält. Der Kreis K_1 besteht aus den Anweisungen, die ausgeführt werden, wenn die Bedingung der `if`-Anweisung erfüllt ist. Ist die Bedingung nicht erfüllt, werden in der Iteration die Anweisungen ausgeführt, die sich auf dem Kreis K_2 befinden. Es ist daher sinnvoll eine Schleife als die Vereinigung von Kreisen mit gemeinsamen Knoten aufzufassen. Dies führt zu dem Begriff der starken Zusammenhangskomponente.

Definition 5.5 *Eine starke Zusammenhangskomponente $SZHK$ in einem Graphen $G = (V, E)$ ist eine Teilmenge $SZHK \subseteq V$, so dass es zwischen je zwei Knoten $q, r \in SZHK$ einen gerichteten Pfad von q nach r und einen Pfad von r nach q gibt.*

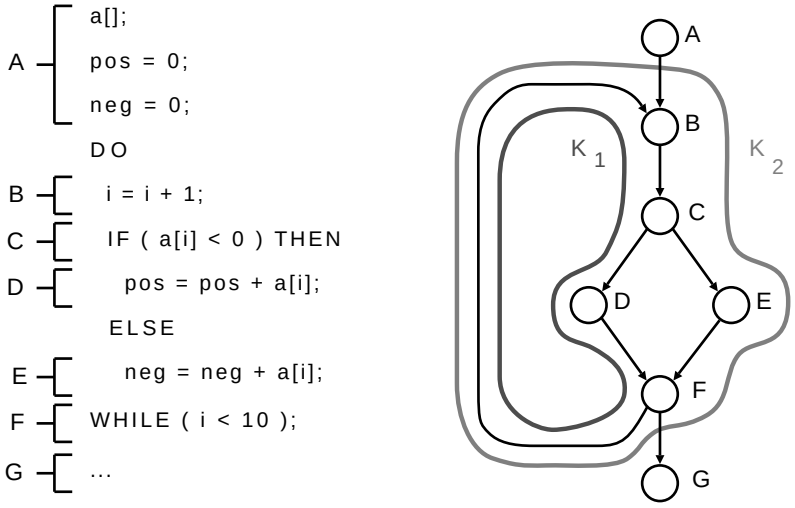

Abbildung 5.1: Kontrollflussgraph zu einem Programm mit einer Schleife.

Definition 5.6 *Eine starke Zusammenhangskomponente ist eine maximale starke Zusammenhangskomponente, wenn sie keine echte Teilmenge einer anderen starken Zusammenhangskomponente ist.*

Einer maximalen starken Zusammenhangskomponente können also keine Knoten hinzugefügt werden, ohne die Eigenschaft des starken Zusammenhangs zu verlieren. Wenn sie nicht nur aus einem einzigen Knoten v besteht, für den es keine Kante (v, v) gibt, entspricht die maximale starke Zusammenhangskomponente einer Schleife, die in keiner anderen Schleife enthalten ist.

Definition 5.7 *„Eine äußerste Schleife ist eine maximale starke Zusammenhangskomponente mit mindestens einer inneren Kante.“[Hav97, Definition 1]*

Der erste Knoten auf einem Pfad vom initialen Knoten des Flussgraphen zu einem Knoten in einer Schleife L, der zu der Schleife gehört, wird als Kopf der Schleife bezeichnet. Er stellt somit einen Eintrittspunkt in die Schleife dar. Es wird in der Regel angenommen, dass die Anweisung, die durch den Schleifenkopf repräsentiert wird, in keiner verschachtelten Schleife enthalten ist.

Definition 5.8 *„Die Schleifen, die in einer Schleife L mit Kopf h verschachtelt sind, sind die äußersten Schleifen bezüglich des Teilgraphen mit der Knotenmenge $(L - h)$.“ [Hav97, Definition 2]*

In allgemeinen Kontrollflussgraphen muss der Kopf einer Schleife aber nicht eindeutig sein. Es ist möglich, dass die Schleife über mehr als einen Knoten betreten werden kann. Welcher der Knoten der reguläre Eintrittspunkt im Sinne einer Schleife aus der Hochsprache ist, und welcher Knoten das Ziel einen Sprunges in die Schleife darstellt, kann ohne weitere Informationen nicht unterschieden werden. Je nach Wahl des Kopfes h kann die Menge der verschachtelten Schleifen jedesmal eine andere sein.

Dieses Problem kann vermieden werden, wenn ausschließlich reduzierbare Flussgraphen betrachtet werden. Für die Definition von reduzierbaren Flussgraphen nach Hecht und Ullman [HU72] werden zwei Transformation T_1 und T_1 benötigt:

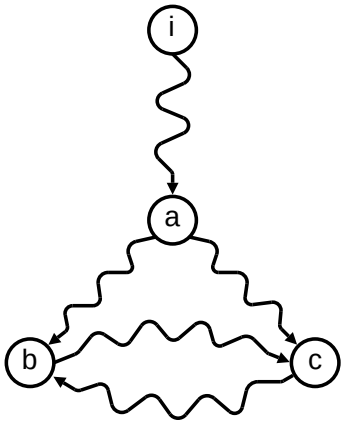

Abbildung 5.2: Nicht reduzierbarer Teilgraph nach [HU72].

Definition 5.9 *Sei $G = (V, E, i)$ ein Flussgraph mit einem Knoten n und einer Kante $e = (n, n)$. Die Transformation T_1 des Knotens n entfernt die Kante e aus dem Graphen.*

Definition 5.10 *Sei $G = (V, E, i)$ ein Flussgraph mit zwei Knoten n_1 und n_2 und einer Kante $e = (n_1, n_2)$. Zusätzlich sei e die einzige eingehende Kante von n_2 und n_2 nicht der initale Knoten i des Flussgraphen. Die Transformation T_2 des Knotenpaares (n_1, n_2) verschmilzt die Knoten n_1 und n_2 zu einem Knoten n. Dazu werden die Knoten n_1 und n_2, sowie die Kante e aus dem Graphen entfernt und durch einen neuen Knoten n ersetzt. Alle Kanten, die auf den Knoten n_1 gerichtet waren, sind nun auf den neuen Knoten n gerichtet. Für alle Kanten, die von einem der Knoten n_1 oder n_2 ausgingen, ist nun n der Startknoten.*

Ein reduzierbarer Kontrollflussgraph hat die Eigenschaft, dass er durch die wiederholte Anwendung der zwei Transformationen T_1 und T_2 auf einen Knoten reduziert wird. Dabei ist es unerheblich, in welcher Reihenfolge die beiden Transformationen angewendet werden.

Definition 5.11 *Ein Flussgraph ist genau dann reduzierbar, wenn er durch erschöpfende Anwendung der Transformationen T_1 und T_2 in einen Graphen überführt wird, der aus nur einem einzigen Knoten besteht.*

Hecht und Ullman [HU72] zeigen, dass diese Eigenschaft gleichbedeutend mit der Eigenschaft ist, dass der Flussgraph keinen Teilgraphen von der Form hat, wie sie in Abbildung 5.2 dargestellt ist. Ist der Flussgraph $G = (V, E, i)$ reduzierbar, existieren keine drei verschiedene Knoten $a, b, c \in V$, wobei b und c nicht der initiale Knoten i sind, so dass es knotendisjunkte Pfade von i nach a, von a nach b, von a nach c und zwischen b und c gibt.

Da reduzierbare Graphen keinen Teilgraphen enthalten, wie er in der Abbildung 5.2 angegeben ist, folgt, dass es für jede maximale starke Zusammenhangskomponente $SZHK$ nur genau einen Schleifenkopf gibt. Die Schleifen in einem reduzierbaren Flussgraphen sind daher eindeutig bestimmbar. Durch die Definition der Schleifen über maximale starke Zusammenhangskomponenten ist zudem sicher gestellt, dass zwei Schleifen entweder

zwei disjunkte Knotenmengen beschreiben, oder eine Schleife komplett in der anderen enthalten ist.

Schließlich sollen nun noch Namen für signifikante Knoten einer Schleife eingeführt werden.

Definition 5.12 *Ein Eingang einer Schleife L ist ein Knoten aus L, der einen Vorgänger hat, der nicht in L liegt.*

Definition 5.13 *Ein Ausgang einer Schleife L ist ein Knoten aus L, der einen Nachfolger hat, der nicht zu L gehört.*

Der Eingang einer Schleife ist also das, was bisher als Eintrittspunkt bezeichnet wurde. In reduzierbaren Graphen hat jede Schleife genau einen Eingang, der auch Schleifenkopf genannt wird. Ein Knoten einer verschachtelten Schleife kann nur dann erreicht werden, wenn zuvor ein Knoten der direkt umschließenden Schleife passiert wurde. Dies ist in jedem Fall der Schleifeneingang der direkt umschließenden Schleife. Die Schleife kann in diesem Sinne nur auf regulärem Weg betreten werden, indem alle umschließenden Schleifen nacheinander betreten wurden. Beim Verlassen der Schleife ist es aber möglich, dass mehrere Schleifen auf einmal verlassen werden.

Definition 5.14 *Ein Ausgang a einer verschachtelten Schleife L wird als regulär bezeichnet, wenn der Nachfolger des Knotens a, der nicht zu L gehört, zu der L direkt umschließenden Schleife gehört.*

Ein gebräuchlicher Algorithmus zur Identifikation von Schleifen in reduzierbaren Graphen ist der Algorithmus von Tarjan [Tar73]. Eigentlich entwickelt, um Graphen in fast linearer Zeit auf Reduzierbarkeit zu testen, kann er so geändert werden, dass er zusätzlich Schleifen identifiziert und basierend auf der Verschachtelungsrelation eine Schleifenhierarchie erstellt [Ram99]. Wie sich in dem Abschnitt 5.3 zeigen wird, ist ein Verfahren zur Schleifenidentifikation, das auf reduzierbare Graphen beschränkt ist, für diese Diplomarbeit ausreichend.

5.2 WCET-Modellierung von Suhendra et al.

Die WCET-Modellierung von Suhendra et al. [SMRC05] basiert auf dem Kontrollflussgraphen einer Funktion, in dem jeder Knoten einen Basisblock repräsentiert. Ziel der Modellierung ist ein ILP zu erstellen, dessen Zielfunktion die WCET der Funktion beschreibt, die durch den Kontrollflussgraphen repräsentiert wird. Diese soll in dem ILP minimiert werden.

5.2.1 Voraussetzungen

Es wird vorausgesetzt, dass für jeden Basisblock B die maximale Ausführungsdauer g_B bekannt ist. Für Entscheidungen, die dazu führen, dass Änderungen am Basisblock vorgenommen werden, ist ebenfalls die Auswirkung auf die Ausführungszeit des Basisblocks bekannt. Sie ist eine feste Größe, die nur von der Entscheidung selbst abhängt und nicht durch andere Entscheidungen beeinflusst wird. Folglich können unter dieser Annahme die

Auswirkungen der Entscheidungen isoliert betrachtet werden. Ist x_B^k die Entscheidungsvariable, die eine Entscheidung modelliert, die sich auf den Basisblock B bezieht, wird mit $c(x_B^k)$ die Auswirkung auf die Ausführungszeit des Basisblocks bezeichnet. Die Entscheidungsvariablen x_B^k sind binäre Entscheidungsvariablen, die so definiert sind, dass sie den Wert Eins haben, wenn die Entscheidung zu einer Änderung am Basisblock führt, und dass sie anderenfalls den Wert Null annehmen. Im Fall der Registerallokation sind dies genau die Entscheidungsvariablen, die die Entscheidungen über das Einfügen von Spillcode modellieren. Mit $c(x_B^k)$ wird dann die Verlängerung der Ausführungszeit des Basisblocks B angegeben, die entsteht, wenn eine Spillinstruktion in den Basisblock eingefügt wird und dies auf Grund der Entscheidung erfolgte, die durch die Entscheidungsvariable x_B^k modelliert wird. Die maximale Ausführungsdauer eines Basisblocks B lässt sich mit Berücksichtigung der Entscheidungen, die den Basisblock betreffen, durch eine Entscheidungsvariable d_B modellieren, die Werte aus dem Bereich der reelen Zahlen annehmen darf. Ihr Wert ist die Summe aus der maximalen Ausführungsdauer des unveränderten Basisblocks und den Auswirkungen von Entscheidungen, die zu Änderungen am Basisblock führen:

$$d_B = g_B + \sum_k \left(c(x_B^k) \cdot x_B^k \right) \tag{5.1}$$

Zusätzlich wird angenommen, dass alle Pfade in dem Kontrollflussgraphen mögliche Kontrollflüsse repräsentieren. Es gibt demnach keine *infeasible paths*. Eine weitere Annahme ist, dass die Schleifen in dem Kontrollflussgraphen identifiziert sind und zu jeder Schleife die maximale Iterationshäufigkeit bekannt ist.

5.2.2 Vorgehensweise

Bei dem Ansatz von Suhendra et al. wird zunächst die maximale Ausführungsdauer innerster Schleifen modelliert. Zu jeder innersten Schleife gibt es dann eine Entscheidungsvariable, die die Ausführungsdauer der Schleife repräsentiert. Diese wird genutzt, um die Ausführungszeit der direkt umschließenden Schleife zu repräsentieren. Dazu wird die gesamte innerste Schleife in dem Kontrollflussgraphen durch einen Knoten ersetzt, dessen Ausführungsdauer die maximale Ausführungsdauer der Schleife ist. Kanten, die in die Schleife hineinführen, werden auf den Ersatzknoten gerichtet. Die Kanten, die aus der Schleife herausführen, werden zu Kanten, die vom Ersatzknoten ausgehen.

Nachdem alle verschachtelten Schleifen auf diese Weise ersetzt wurden, sind die direkt umschließenden Schleifen selbst zu innersten Schleifen geworden, die auf die gleiche Art behandelt werden. Nach und nach werden so die Schleifen aus dem Kontrollflussgraphen entfernt, bis er keine Schleife mehr enthält.

Für die Modellierung der Ausführungsdauer einer innersten Schleife wird nur der Schleifenrumpf betrachtet, d. h. die zum Schleifenkopf zurückführenden Kanten werden außer Acht gelassen. Da der Schleifenrumpf innerster Schleifen und der resultierende Graph zyklenfrei ist, muss nur eine Modellierung für die Ausführungsdauer von azyklischen Graphen angegeben werden.

Der Kopf der Schleife wird im Folgenden auch als Quelle bezeichnet. Die Knoten, von denen aus der Rücksprung zum Kopf der Schleife erfolgte, haben im azyklischen Graphen keinen Nachfolger mehr und werden Senken genannt. Existiert mehr als eine Senke, wird ein neuer Knoten in den azyklischen Graphen eingefügt. Für jede Senke wird eine Kante

eingefügt, die von der Senke ausgeht und auf von neuen Knoten gerichtet ist. Auf diese Weise erhält der Graph eine eindeutige Senke. Diese ist der neu eingefügte Knoten. Weil er keine Anweisung der Funktion repräsentiert, sondern einen leeren Basisblock darstellt, wird seine Ausführungszeit auf Null festgesetzt.

Um die WCET des Pfades in einer Schleife zu modellieren, der von der Quelle zu der Senke der Schleife führt, wird nun induktiv die WCET der Pfade modelliert, die bei einem Knoten aus der Schleife beginnen und an der Senke enden. Es müssen somit nicht alle Pfade einzeln betrachtet werden, die in der Schleife möglich wären. Statt dessen wird für jeden Knoten nur die maximale Ausführungsdauer durch eine Entscheidungsvariable w_B modelliert, die ein Pfad haben kann, der in dem Knoten B startet.

Die Ausführungsdauer eines Pfades, der nur die Senke enthält, ist unmittelbar durch die Ausführungsdauer des Basisblocks gegeben, der durch die Senke repräsentiert wird.

$$w_{Senke} = d_{Senke} \tag{5.2}$$

Die maximale Ausführungsdauer eines Pfades, der an einem anderen Knoten B startet und an der Senke endet, setzt sich aus der Ausführungsdauer des zugehörigen Basisblocks und der größten maximalen Ausführungsdauer eines Pfades zusammen, der bei einem Nachfolger des Knoten startet und an der Senke endet. Sie ist also mindestens so groß wie die Ausführungsdauer eines Pfades, der bei einem Nachfolger beginnt, zu der die Ausführungsdauer des Basisblocks hinzu addiert wird. Für jeden Nachfolger N, den B im Schleifenrumpf hat, wird daher ein Constraint dieser Form formuliert:

$$\forall N, (B, N) \in E : w_B \geq d_B + w_N \tag{5.3}$$

Die maximale Ausführungsdauer der Schleife ist dann die maximale Ausführungsdauer des Pfades von der Quelle zu der Senke, multipliziert mit der maximalen Iterationshäufigkeit lb der Schleife. Die Ausführungsdauer des Ersatzknotens Z der Schleife wird daher durch dieses Constraint beschrieben:

$$g_Z = lb \cdot w_{Quelle} \tag{5.4}$$

Für den Kontrollflussgraphen, der entsteht, wenn alle Schleifen durch stellvertretende Knoten ersetzt werden, wird die maximale Ausführungszeit wie für einen Schleifenrumpf modelliert, der genau einmal ausgeführt wird.

Durch die Constraints des ILP wird so eine untere Schranke für die geschätzte WCET der Funktion beschrieben, die oberhalb der tatsächlichen WCET der Funktion liegt. Die Zielfunktion des ILP ist die geschätzte WCET der Funktion. Diese ist die maximale Ausführungsdauer eines Pfades, der am initialen Basisblock I der Funktion beginnt, und zu dem Funktionsende führt. Folglich ist die Zielfunktion die Entscheidungsvariable w_I. Durch das Minimieren der Zielfunktion wird schließlich erreicht, dass diese den Wert der unteren Schranke annimmt, die durch die Constraints gebildet wird.

5.2.3 Fazit

Die WCET-Modellierung von Suhendra et al. hat den Vorteil, dass sie auf beliebige Kontrollflussgraphen angewendet werden kann. Allerdings wird bei der Ersetzung der Schleifen durch Ersatzknoten der Kontrollflussgraph ohne Rücksicht auf die Ein- und Ausgänge der Schleifen verändert. Auf diese Weise werden Pfade modelliert, die es vorher

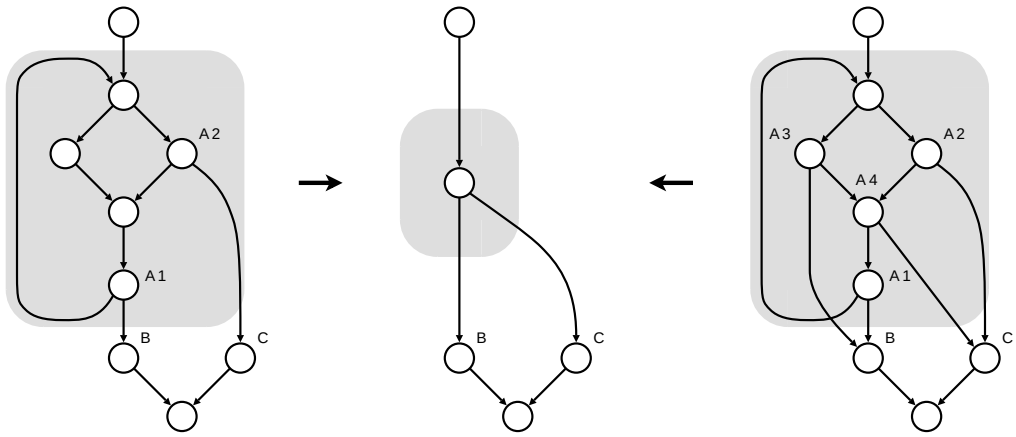

Abbildung 5.3: Zwei verschiedene Graphen, die nach der Ersetzung der Schleife nicht mehr unterschieden werden können.

nicht gab. In dem ursprünglichen Graphen, der in Abbildung 5.3 links zu sehen ist, kann der Knoten B nur über den Ausgang A_1 erreicht werden. Der Graph, der nach der Ersetzung entsteht, könnte aber auch aus dem Graphen entstanden sein, der in Abbildung 5.3 rechts dargestellt ist. In ihm ist der Knoten B von zwei Ausgängen erreichbar.

Die Constraints, die bei der Modellierung aufgestellt werden, beschreiben aber weder die Kontrollflüsse in dem linken noch in dem rechten Graphen. Da die Ausführungsdauer einer Schleife durch einem Pfad von ihrer Quelle zu ihrer Senke modelliert wird, wird eigentlich der Graph modelliert, bei dem die Senke der einzige Ausgang der Schleife ist. Alle Knoten, die unmittelbar nach Verlassen der Schleife ausgeführt werden können, sind bei dieser Modellierung Nachfolger der Senke. Andere Ausgänge der Schleife verlieren durch die Ersetzung der Schleife ihre Nachfolger. Gibt es neben der Quelle des Schleifenrumpfes andere Eingänge, würden diese im dem Modell ebenfalls keine Vorgänger mehr haben. Es muss also angenommen werden, dass die Quelle der einzige Eingang und die Senke der einzige Ausgang einer Schleife sind. Anderenfalls gibt das ILP die Struktur des Kontrollflussgraphen nicht richtig wieder.

5.3 WCET-Modellierung für reduzierbare Graphen

In diesem Abschnitt wird die Modellierung der WCET einer Funktion durch ein ILP beschrieben, wie sie in dieser Diplomarbeit vorgenommen wird. Ziel der Modellierung ist, die Struktur des Kontrollflussgraphen so genau wie möglich zu beschreiben. Genau die Pfade, die in dem Kontrollflussgraphen vorkommen, sollen in dem ILP modelliert werden.

5.3.1 Voraussetzungen

Das Modellierungsprinzip ist auf reduzierbare Graphen beschränkt. In einem reduzierbaren Graphen hat jede Schleife genau einen Eingang, kann aber mehrere Ausgänge haben. Dennoch stellt das Prinzip eine Erweiterung gegenüber der WCET-Modellierung von Suhendra et al. dar. Ihre Modellierung kann nur dann exakt sein, wenn jede Schleife genau einen Eingang und einen Ausgang hat. Anderenfalls modellieren sie auch Pfade, die es

in dem Kontrollflussgraphen nicht gibt, oder Pfade, die es im Kontrollflussgraphen gibt, werden nicht modelliert.

Die Beschränkung auf reduzierbare Graphen ist vertretbar. Im Gegensatz zu allgemeinen Flussgraphen haben reduzierbare Graphen den Vorteil, dass die Schleifen in dem Graphen eindeutig sind und jede Schleife genau einen Eingang hat. Viele Funktionen weisen einen reduzierbaren Kontrollflussgraphen auf [ASU86]. Werden bei der Spezifikation einer Funktion in einer Programmiersprache wie C nur die Sprachelemente `if-then-else`, `while-do`, `continue` und `break` verwendet, hat die Funktion in jedem Fall einen reduzierbaren Graphen. Ein C-Programm aus dem ein nicht reduzierbarer Kontrollflussgraph hervorgeht, ist z. B. ein Programm, bei dem mit Hilfe einer `goto`-Anweisung in eine Schleife gesprungen wird. Das Verbieten von `goto`-Anweisungen reicht aber nicht aus, um reduzierbare Kontrollflussgraphen zu garantieren, wie *duff's device* zeigt [Duf88]. Sollte eine Funktion durch einen nicht reduzierbaren Kontrollflussgraphen repräsentiert werden, kann dieser mittels der Technik *node splitting* in einen reduzierbaren Kontrollflussgraphen überführt werden. Welches die Schleifen in dem resultierenden Graphen sind, hängt dann aber davon ab, wie das *node splitting* angewendet wird.

Wie bei der Modellierung von Suhendra et al. wird auch bei dieser Modellierung ein Kontrollflussgraph betrachtet, bei dem die einzelnen Instruktionen zu Basisblöcken zusammengefasst wurden, so dass jeder Knoten des Kontrollflussgraphen einen Basisblock repräsentiert. Es wird ebenfalls angenommen, dass es keine *infeasible paths* gibt. Für jeden Pfad in dem Kontrollflussgraphen wird angenommen, dass er eine mögliche Ausführungsreihenfolge der Basisblöcke repräsentiert. Alle Schleifen in dem Graphen seien identifiziert und ihre maximale Iterationshäufigkeit angegeben. Eine weitere Annahme, die auch schon Suhendra et al. trafen, ist, dass die Ausführungsdauer eines einzelnen Basisblocks B durch eine Entscheidungsvariable d_B beschrieben werden kann. Durch sie wird gleichzeitig der Einfluss der Entscheidungen über das Einfügen von Spillinstruktionen auf die Ausführungsdauer des Basisblocks berücksichtigt. Die Entscheidungsvariable ist auf die gleiche Weise definiert, wie es im vorherigen Abschnitt angegeben wurde. Für jeden Pfad im Kontrollflussgraphen wird angenommen, dass sich seine maximale Ausführungszeit durch die Summe der maximalen Ausführungszeiten der Basisblöcke auf dem Pfad berechnet.

5.3.2 Entscheidungsvariablen für Schleifen

Für die Modellierung der WCET einer Schleife L wird angenommen, dass die Schleife immer so oft ausgeführt wird, wie es durch die maximale Iterationshäufigkeit der Schleife $lb(L)$ angegeben ist. Da der Aussprung aus der Schleife in der letzten Iteration erfolgt, wird unter dieser Annahme der Pfad mit der größten Ausführungsdauer innerhalb der Schleife $(lb(L) - 1)$-mal ausgeführt und dann ein Pfad von dem Eingang zu einem der Ausgänge. Für den Graphen, der in der Abbildung 5.3 links dargestellt ist, würde demnach ein Pfad vom Eingang zum Knoten A_1 $(lb(L) - 1)$-mal ausgeführt werden, und dann entweder ein weiteres Mal der Pfad vom Eingang zum Knoten A_1 oder der Pfad vom Eingang zum Knoten A_2. Zwar ist der Pfad zum Knoten A_1 länger als der zum Knoten A_2, ob der WCEP der Funktion diesen Abschnitt aber $lb(L)$-mal hintereinander enthält, hängt auch von den Ausführungszeiten der Basisblöcke B und C ab. Wird nur die Schleife alleine betrachtet, kann nicht entschieden werden, über welchen Ausgang die Schleife bei der *worst-case*-Eingabe verlassen wird.

Aus diesem Grund wird im Gegensatz zu der Modellierung von Suhendra et al. die Ausführungsdauer für jede Schleife nicht nur durch eine einzelne Entscheidungsvariable modelliert. Statt dessen gibt es so viele Entscheidungsvariablen g_A^L, wie die Schleife L Ausgänge hat. Für jeden Ausgang A der Schleife L gibt die Entscheidungsvariable g_A^L an, wie groß die maximale Ausführungsdauer der Schleife ist, wenn sie über diesen Ausgang verlassen wird. Wie auch die Entscheidungsvariablen g_B, die die Ausführungsdauer eines einzelnen Basisblocks beschreiben, dürfen die Entscheidungsvariablen g_A^L reelle Werte annehmen.

5.3.3 Transformation des Kontrollflussgraphen

In diesem Abschnitt werden Transformationen des Kontrollflussgraphen beschrieben. Diese dienen aber nur der Anschaulichkeit, d. h. der Kontrollflussgraph bleibt während der Modellierung unverändert. Die beschriebenen Transformationen sollen dem Leser helfen zu verstehen, was die verwendeten Entscheidungsvariablen bedeuten, und in welcher Weise Constraints formuliert werden.

Die besondere Behandlung einer Schleifeniteration, üblicherweise sind es die ersten Iterationen, ist als Schälen einer Schleife bzw. *loop peeling* bekannt [BGOS94]. Hier kommt nun der letzten Iteration der Schleife eine besondere Bedeutung zu. Um verschiedene Pfade für die ersten und die letzte Iteration modellieren zu können, werden dem Kontrollflussgraphen neue Knoten hinzugefügt. In Abbildung 5.4 sind der ursprüngliche Kontrollflussgraph und das Ergebnis dieses Prozesses dargestellt. Die Knoten der Schleife, die auf einem Pfad vom Eingang zu einem Ausgang liegen, der keinen Rücksprung zum Eingang enthält, werden dupliziert. Der Eingang selbst und der Ausgang gehören diesem Pfad ebenfalls an, und werden daher ebenfalls dupliziert. Jeder duplizierte Knoten repräsentiert weiterhin den Basisblock, der von dem jeweiligen Originalknoten repräsentiert wird.

Der Teilgraph, der aus den neuen Knoten gebildet wird, wird als die Schale der Schleife bezeichnet. Sie werden genau so wie ihre Originale in der Schleife mit Kanten verbunden. Nur Rücksprünge zum duplizierten Eingang werden nicht eingefügt. Die Nachfolger eines Schleifenausgangs, die nicht zur Schleife gehören, sind auch die Nachfolger seiner Kopie. Da angenommen wird, dass in den ersten Iterationen kein Aussprung erfolgt, werden die Kanten, die von den Ausgängen der Schleife ausgehen, weggelassen. Für jeden Rücksprung zum Eingang der Schleife werden Kanten zwischen dem Ausgangsknoten des Rücksprungs und dem Duplikat des Einganges eingefügt. Die Iterationshäufigkeit der Schleife, die auf diese Weise geschält wurde, wird um eins verringert.

Innerste Schleifen in dem Kontrollflussgraphen werden zuerst betrachtet und auf die beschriebene Weise abgeschält. Auf den so entstehenden Graphen könnte nun die Modellierung von Suhendra et al. angewendet werden. Dabei würde anschaulich aber nur die geschälte Schleife selbst durch einen Knoten ersetzt werden, nicht aber die Schale der Schleife. Befindet sich eine verschachtelte Schleife auf einem Pfad vom Eingang zu einem Ausgang in der umschließenden Schleife, würde beim Schälen der umschließenden Schleife die Schale der verschachtelten Schleife dupliziert. Mit jeder weiteren umschließenden Schleife würde die Schale immer weiter vervielfacht werden und die Anzahl der Knoten im Graph würde steigen. Gleichzeitig müssten die Pfade in jedem Duplikat der Schale erneut modelliert werden. Um dem entgegen zu wirken, werden eine geschälte Schleife und ihre Schale weiterhin als eine Einheit betrachtet und gemeinsam durch Knoten ersetzt.

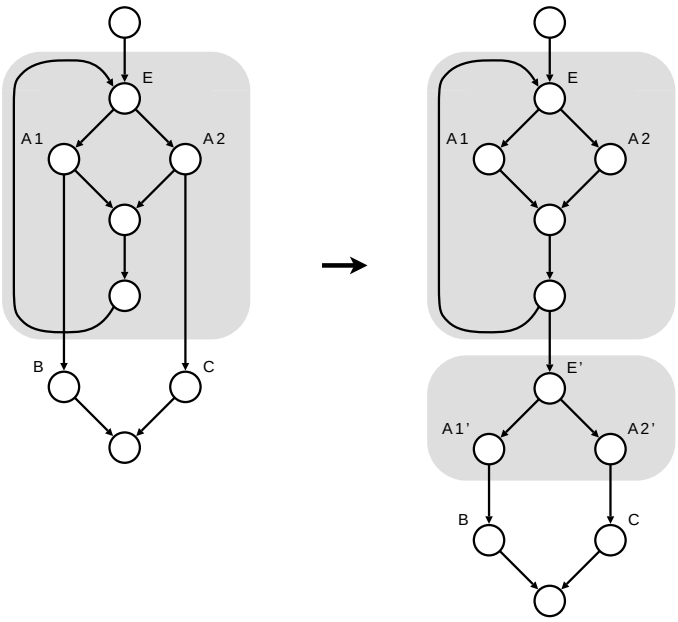

Abbildung 5.4: Schälen einer Schleife

Um die verschiedenen Übergänge zwischen der Schale und der übrigen Knoten im Graphen richtig berücksichtigen können, reicht dazu nicht immer ein Knoten aus. Die Schleife und ihre Schale werden durch so viele Knoten ersetzt, dass es für jedes Duplikat eines Ausgangs einen Ersatzknoten gibt. In dem ursprünglichen Graphen wird also eine Schleife letztendlich durch so viele Knoten ersetzt, wie sie Ausgänge hat. Dies ist in Abbildung 5.5 für eine Schleife mit zwei Ausgängen $A1$ und $A2$ dargestellt. Für die beiden Ausgänge $A1$ und $A2$ wurden Ersatzknoten $A1_{Ersatz}$ und $A2_{Ersatz}$ eingefügt. $A1_{Ersatz}$ repräsentiert die Schleife, wenn sie über Ausgang $A1$ verlassen wird. $A2_{Ersatz}$ repräsentiert die Schleife dementsprechend, wenn die Schleife über Ausgang $A1$ verlassen wird. Die Vorgänger der Ersatzknoten sind die Vorgänger des einzigen Schleifeneingangs E. Der Nachfolger eines Ersatzknotens ist der Nachfolger der Kopie des Ausgangs auf den sich der Ersatzknoten bezieht. Die Ausführungsdauer jedes Ersatzknotens für die Schleife L, der sich auf einen Ausgang A bezieht, ist durch den Wert der Entscheidungsvariable g_A^L gegeben.

Nach und nach werden die maximalen Ausführungszeiten innerster Schleifen modelliert und die Schleifen durch einzelne Knoten ersetzt. Wurden die Ausführungszeiten aller verschachtelten Schleifen modelliert, wird mit der umschließenden Schleife fortgefahren. Schließlich sind alle Schleifen sukzessiv durch Knoten ersetzt worden.

Damit die Ausführungszeiten innerster Schleifen modelliert werden können, muss jedes mal ein azyklischer Teilgraph vorliegen, der die möglichen Pfade in der Schleife repräsentiert. In dieser Modellierung besteht der Teilgraph genau aus der geschälten Schleife und ihrer Schale. Zyklen können nur in der Schleife vorkommen, nicht aber in der Schale. Wie bei der Modellierung von Suhendra et al. könnte die Schleife durch einen Knoten ersetzt werden und der resultierende Graph wäre zyklenfrei. Alternativ wird aber das im Folgenden beschriebe Verfahren angewendet.

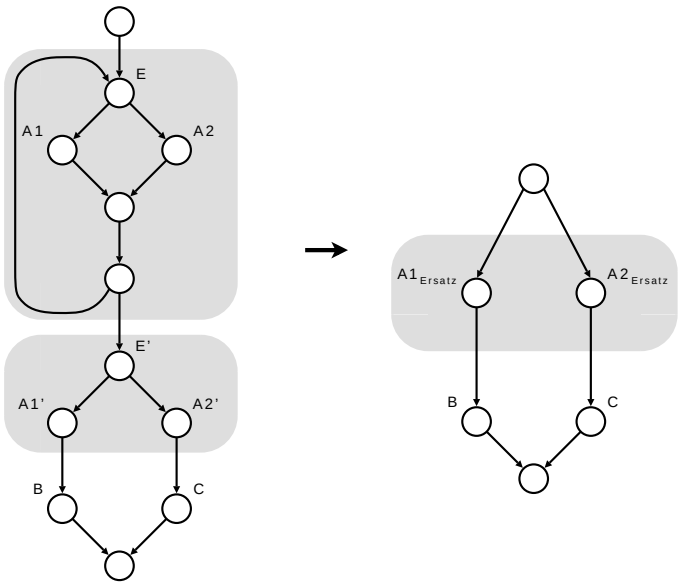

Abbildung 5.5: Ersetzung einer geschälten Schleife und ihrer Schale durch einzelne Knoten.

Eine geschälte Schleife hat nur einen einzigen Eingang und einen einzigen Ausgang. Wird in jeder Iteration der geschälten Schleife der WCEP der Schleife ausgeführt, beginnt dieser am Eingang und endet am einzigen Ausgang. Werden die Ausführungskontexte einer Iteration außer Acht gelassen und angenommen, dass jede Ausführung eines Pfades immer maximal gleich lange dauert, kann für jede Iteration derselbe Pfad betrachtet werden. Werden in zwei Iteration zwei verschiedene Pfade ausgeführt, müssten beide Pfade die gleiche maximale Ausführungsdauer haben. Ansonsten könnte eine größere maximale Ausführungsdauer der Schleife bestimmt werden, indem in jeder Iteration der Pfad mit der größeren maximalen Ausführungsdauer ausgeführt wird. Wenn aber die Pfade die gleiche WCET haben, kann in jeder Iteration der gleiche Pfad gewählt werden und die WCET der Schleife ändert sich nicht. Für alle Basisblöcke, die sich auf dem WCEP der Schleife befinden, gilt also, dass sie in jeder Iteration ausgeführt werden.

Auf Grund der Annahme, dass sich die maximale Ausführungsdauer eines Pfades additiv aus den Ausführungszeiten der Basisblöcke auf dem Pfad zusammen setzt, können die Basisblöcke auf dem Pfad genauso gut in einer anderen Reihenfolge ausgeführt werden, ohne dass sich an der maximalen Ausführungszeit des Pfades etwas ändert. Wird eine Schleife also $(lb-1)$-mal hintereinander ausgeführt, können für die Berechnung der maximalen Ausführungsdauer die Ausführungszeiten der Basisblöcke so gruppiert werden, dass jeder Basisblock $(lb-1)$-mal hintereinander ausgeführt wird, und dafür nur ein Durchlauf der Schleife erfolgt. Dieser Sachverhalt wird genutzt, um Zyklen zu entfernen, wie dies in Abbildung 5.6 demonstriert wird. Um aus einer geschälten Schleife einen azyklischen Teilgraphen zu bilden, werden die Rückkanten in der Schleife entfernen. In dem entstehenden azyklischen Schleifenrumpf wird die Ausführungsdauer der Basisblöcke mit der Iterationshäufigkeit der geschälten Schleife multipliziert.

An dieser Stelle könnte vielleicht der Eindruck entstehen, dass es auch möglich sei, bei der Modellierung der WCET der ursprünglichen Schleife ohne das Schälen der Schleife

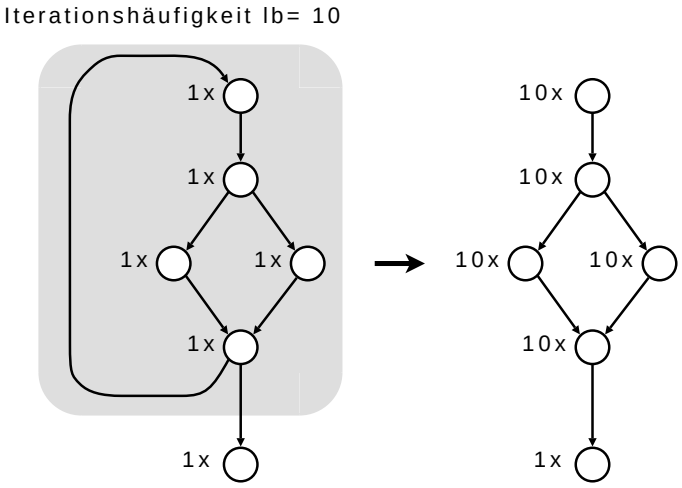

Abbildung 5.6: Entfernung einer Rückkante. Die Faktoren an den Knoten geben an, mit welchem Wert die Ausführungsdauer des Knotens multipliziert wird.

auszukommen. Die Ausführungsdauer eines Basisblocks, der auf einem zyklenfreien Pfad vom Eingang der Schleife zu einem Ausgang liegt, würde mit der Iterationshäufigkeit der Schleife multipliziert werden, und die Ausführungsdauer der übrigen Basisblöcke mit der um eins verringerten Iterationshäufigkeit. Die Rücksprünge zum Eingang der Schleife würden dann einfach weggelassen. Damit würde aber die implizite Annahme getroffen, dass in der letzten Iteration nur solche Basisblöcke ausgeführt werden, die auch in allen vorherigen Iterationen ausgeführt wurden. Es könnte dann nicht mehr zwischen einem Pfad unterschieden werden, der in den Iterationen ausgeführt wird, die mit einem Rücksprung zum Eingang enden, und einem Pfad, der in der letzten Iteration ausgeführt wird. Der zuerst genannte Pfad startet bei dem Eingang der Schleife und endet bei einem Knoten, von dem ein Rücksprung erfolgt. Der als zweites genannte Pfad startet ebenfalls beim Eingang der Schleife und endet an einem Ausgang. Dieser Pfad muss kein Teilstück des ersten Pfades sein. Bis auf den Startknoten könnten beide Pfade knotendisjunkt sein. Die Ausführungshäufigkeit eines Basisblocks wird daher dadurch bestimmt, ob er entweder auf einem der beiden Pfade, auf beiden Pfaden oder keinem der Pfade liegt. Würde die Ausführungsdauer des Basisblocks mit der Iterationshäufigkeit der Schleife multipliziert werden, könnte der Basisblock entweder nur auf beiden oder auf keinem der Pfade liegen. Folglich könnte die WCET der Funktion nicht korrekt modelliert werden.

5.3.4 Modellierung der WCET von Schleifen

Für jeden Ausgang A einer Schleife L wurde eine Entscheidungsvariable g_A^L eingeführt. Sie modelliert die maximale Ausführungsdauer der Schleife für den Fall, dass die Schleife über diesen Ausgang verlassen wird. Nachdem die Schleife geschält wurde, repräsentiert die Entscheidungsvariable g_A^L die maximale Ausführungsdauer eines Pfades, der an dem Eingang der Schleife beginnt und an der Kopie des Ausgangs endet. Sie kann auf eine Weise ermittelt werden, die der Modellierung der WCET von Schleifen nach dem Ansatz von Suhendra et al. sehr ähnelt. Bei ihrem Ansatz wird angenommen, dass alle Pfade mit maximaler Ausführungsdauer in einem Schleifenrumpf von der Quelle zur Senke führen.

Zuerst wird der Pfad betrachtet, der nur aus der Senke besteht. Schrittweise wird dann die Betrachtung auf Pfade ausgeweitet, die den Pfad in Richtung der Quelle verlängern. Für jeden Knoten B in der Schleife wird dabei die maximale Ausführungszeit der Pfade von dem Knoten zur Senke durch eine Entscheidungsvariable w_B modelliert.

Würde diese Modellierung auf die geschälte Schleife übernommen werden, müsste dieses Vorgehen für jeden Ausgang der Schleife wiederholt werden. Jedes mal müssten neue Entscheidungsvariablen w_B für die Knoten verwendet werden, da ihre Werte von den Nachfolgern im Graphen abhängen. Je nachdem welches Duplikat eines Ausgangs als Senke angesehen wird, ergeben sich für die Entscheidungsvariablen andere Werte. Hat eine Schleife n Ausgänge, gäbe es auch für jeden Knoten n Entscheidungsvariablen. Die n verschiedenen Entscheidungsvariablen an dem Schleifeneingang würden die gesuchten maximalen Ausführungszeiten der Schleife in Abhängigkeit davon repräsentieren, über welchen Ausgang sie verlassen wird. Da bei diesem Vorgehen sehr viele Entscheidungsvariablen und Constraints aufgestellt werden müssten, wird eine etwas andere Modellierung gewählt.

Alle Pfade mit maximaler Ausführungszeit, die von dem Eingang der geschälten Schleife zu einem der Duplikate der Ausgänge führen, haben den gleichen Startpunkt und führen frühestens auseinander, nachdem sie den Knoten passiert haben, der das Duplikat des Schleifeneingangs ist. Es wäre daher vorteilhaft, wenn Zwischenergebnisse wiederverwendet werden könnten. Aus diesem Grund wird bei der gewählten Modellierung die Betrachtungsweise der Pfade geändert. Zuerst wird der Pfad betrachtet, der nur aus dem Eingang der Schleife besteht. Dann wird die Betrachtung auf Pfade ausgeweitet, die die betrachteten Pfade um einen Knoten verlängern und ein Stück weiter in Richtung eines der duplizierten Ausgänge führen. Für jeden Knoten in der Schleife und ihrer Schale gibt es wieder eine Entscheidungsvariable. Diesmal gibt sie aber die maximale Ausführungszeit der Pfade an, die am Eingang der Schleife starten und in dem Knoten enden. Somit ist der Wert der Entscheidungsvariablen unabhängig davon, zu welchem Duplikat eines Ausgangs der Pfad fortgesetzt wird, immer gleich. Für jeden Knoten wird deshalb nur genau eine Entscheidungsvariable benötigt, die bei jeder Modellierung der maximalen Ausführungszeit der Schleife verwendet wird. Um die umgekehrte Wachstumsrichtung der betrachteten Pfade zu verdeutlichen, werden diese Entscheidungsvariablen mit m_B bezeichnet, das an ein umgedrehtes w erinnern soll. w_B ist die Bezeichnung, die in dem Modell von Suhendra et al. verwendet wird.

Nun sollen die Constraints angegeben werden, mit denen die maximalen Ausführungszeiten der Pfade beschrieben werden, die am Eingang der Schleife starten und zu der Kopie eines Ausgangs führen. Es wird mit dem Pfad begonnen, der nur aus dem Eingang H der Schleife besteht. Die maximale Ausführungsdauer dieses Pfades ist die Ausführungsdauer d_H des Eingangs, die mit der Iterationshäufigkeit der geschälten Schleife multipliziert wird. War $lb(L)$ die Iterationshäufigkeit der ursprünglichen nicht geschälten Schleife, ist $lb-1$ die Iterationshäufigkeit der geschälten Schleife. Der Wert der Entscheidungsvariable m_H, der die maximale Ausführungsdauer des Pfades angibt, der nur aus dem Eingang besteht, muss daher diese Bedingung erfüllen:

$$m_H = (lb - 1) \cdot d_H \tag{5.5}$$

Anschließend werden die Pfade betrachtet, die an den übrigen Knoten enden. Die WCET der Pfade, die am Eingang der Schleife starten und in dem Knoten C enden,

wird durch die Entscheidungsvariable m_C modelliert. Hat der Knoten C nur genau einen Vorgänger B, ist die WCET der Pfade, die am Eingang der Schleife starten und in dem Knoten C enden, die Summe aus der WCET der Pfade, die am Eingang der Schleife starten und im Vorgängerknoten enden, und der WCET des Basisblocks C selbst. Wenn der Basisblock der geschälten Schleife angehört, muss seine WCET noch mit der Iterationshäufigkeit der geschälten Schleife multipliziert werden. Sei

$$It(C) = \begin{cases} lb - 1 & \text{wenn C ein Knoten in der geschälten Schleife ist} \\ 1 & \text{sonst} \end{cases} \tag{5.6}$$

Dann lässt sich die Bedingung für die Entscheidungsvariable m_C mit dieser Gleichung beschreiben:

$$m_C = It(C) \cdot d_C + m_B \tag{5.7}$$

Hat der Knoten C dagegen mehrere Vorgänger in dem Kontrollflussgraphen $G = (V, E, i)$, muss unter den maximalen Ausführungszeiten der Pfade, die am Eingang der Schleife starten und in den Vorgängerknoten enden, der größte Wert ausgewählt werden. Zusammen mit der Ausführungsdauer des Basisblocks ergibt sich dann die maximale Ausführungsdauer m_C. Für die WCET der Pfade, die am Eingang der Schleife beginnen und in dem Knoten C enden, gilt also, dass sie mindestens so groß ist, wie jede Summe aus der WCET des Basisblocks C und der WCET eines Pfades, der am Eingang der Schleife beginnt und am Vorgängerknoten endet.

$$\forall B \ (B, C) \in E : m_C \geq It(C) \cdot d_C + m_B \tag{5.8}$$

Die Entscheidungsvariable $m_{A'}$ des Duplikates A' eines Ausgangs A ist die Entscheidungsvariable, die die maximale Ausführungsdauer eines Pfades vom Eingang der Schleife zu dem Knoten A' angibt. Sie ist gleich der WCET der nicht geschälten Schleife, die über den Ausgang A verlassen wird. Für jeden Ausgang A der Schleife L ist daher der Wert der Entscheidungsvariable g_A^L, die die WCET der Schleife in Abhängigkeit des Ausgangs A modelliert, gleich dem Wert der Entscheidungsvariable $m_{A'}$, wobei A' das Duplikat des Ausgangs A ist.

$$g_A^L = m_{A'} \tag{5.9}$$

5.3.5 Modellierung der WCET einer Funktion

Die WCET einer Funktion ist die maximale Ausführungsdauer eines Pfades, der am initialen Knoten i des Kontrollflussgraphen $G = (V, E, i)$ beginnt und in einem Knoten endet, der ein Funktionsende repräsentiert. Nachdem für jede Schleife in dem Kontrollflussgraphen der Funktion die WCET modelliert und jede Schleife durch Knoten ersetzt wurde, liegt ein azyklischer Graph vor. Die WCET der Funktion kann also wie für eine geschälte Schleife modelliert werden. Da die Schleifen aus dem Graphen entfernt wurden, gibt es keinen Knoten in dem Graphen, der mehrfach ausgeführt werden könnte. Die Funktion $It(C)$ ist daher für jeden verbliebenen Basisblock C Eins.

Wurde die WCET wie bei einer Schleife modelliert, gibt es für jedes Funktionsende D eine Entscheidungsvariable m_D. Sie repräsentiert die maximale Ausführungsdauer ei-

nes Pfades, der von dem initialen Knoten zu dem Funktionsende führt. Die WCET der Funktion ist dann das Maximum, das über den Werten dieser Variablen gebildet wird.

Sei m_F die Entscheidungsvariable, die die WCET der Funktion F in dem ILP repräsentiert. Ein Funktionsende der Funktion F ist ein Knoten D, der keinen Nachfolger in dem Kontrollflussgraphen der Funktion hat. Für jedes Funktionsende wird die Bedingung formuliert, dass die WCET der Funktion F mindestens so groß ist, wie die WCET des Pfades, der von dem initialen Knoten zu dem Funktionsende führt.

$$\forall D \nexists B \ (D, B) \in E : m_F \geq m_D \tag{5.10}$$

Da die Zielfunktion des ILP die WCET der Funktion beschreiben soll, ist m_F die Zielfunktion des ILP. Weil der Wert der Zielfunktion in dem ILP minimiert wird, wird erreicht, dass der Wert von m_F möglichst klein gewählt wird. Er nimmt daher den Wert der größten unteren Schranke an, die durch die Ungleichung 5.10 angegeben ist. Folglich entspricht der Wert m_F genau dem Maximum über den Werten m_D.

5.3.6 Handhabung von Funktionsaufrufen

Die WCET einer Funktion umfasst auch die Ausführungsdauer von Funktionen, die von der Funktion aufgerufen werden. Dies setzt voraus, dass die WCET der Funktionen, die in der Funktion aufgerufen werden, bereits bestimmt wurde. Es muss also zuerst eine Registerallokation für die Funktionen erfolgen, die keine andere Funktion aufrufen. Erst dann kann ein ILP für die Allokation der Funktionen formuliert werden, die diese Funktionen aufrufen.

Um eine Reihenfolge angeben zu können, in der die Registerallokation für die Funktionen durchgeführt werden kann, wird eine Relation **A** definiert. Eine Funktion F_1 steht in Relation **A** zu einer Funktion F_2, wenn die Funktion F_1 von der Funktion F_2 aufgerufen wird. Mit der Relation kann nur dann eine Ordnung auf den Funktionen definiert werden, wenn die Relation anti-symmetrisch ist. Dass bedeutet, dass es keine zwei Funktionen geben darf, die sich gegenseitig aufrufen. Aber auch eine Funktion, die sich selbst rekursiv aufruft, stellt ein Problem dar. Um das ILP für diese Funktion modellieren zu können, wird bereits die WCET der Funktion benötigt. Ansonsten kann die Ausführungszeit für den Funktionsaufruf nicht berücksichtigt werden. Aus diesem Grund sind rekursive Aufrufe auszuschließen. Anderenfalls wird in der Modellierung der rekursive Funktionsaufruf einfach ignoriert.

Ist die Relation **A** anti-symmetrisch, kann mit ihr eine Reihenfolge festgelegt werden, in der die Registerallokation für die Funktionen ausgeführt werden soll. Die Registerallokation für eine Funktion F_1 wird dann ausgeführt, wenn die Registerallokation für alle Funktionen F_2, die von F_1 aufgerufen werden, durchgeführt wurde.

Die WCET einer aufgerufenen Funktion ist durch den Wert der Zielfunktion des ILP gegeben, das für die Registerallokation der Funktion formuliert und gelöst wurde. Um die maximale Ausführungsdauer einer aufgerufenen Funktion in der aufrufenden Funktion zu berücksichtigt, wird die Ausführungszeit der Basisblöcke, in denen der Aufruf der Funktion erfolgt, um die WCET der Funktion erhöht.

5.4 Größe des ILP

Der Registerallokator, der in dieser Diplomarbeit erstellt wurde, beschreibt das Registerallokationsproblem durch ein ILP, das sich aus zwei Teilen zusammen setzt. Der erste Teil des ILP modelliert das Registerallokationsproblem ohne eine Zielfunktion und wurde in dem Kapitel 4 beschrieben. Der zweite Teil des ILP modelliert außschließlich die Zielfunktion des Registerallokationsproblems. Für diese Diplomarbeit handelt es sich bei der Zielfunktion um die WCET einer Funktion. In dem vorherigen Abschnitt 5.3 wurde beschrieben, auf welche Weise sie modelliert wird.

Während in dem ersten Teil des ILP nur binäre Entscheidungsvariablen verwendet werden, werden für die Modellierung der WCET Variablen genutzt, die Werte aus dem Berecih der reellen Zahlen annehmen dürfen. Insgesamt wird daher ein *mixed integer linear program* (mILP) gebildet.

In diesem Abschnitt soll nun für beide Teile des ILP die jeweilige Größe abgeschätzt werden. Der erste Teil des ILP beruht auf dem ORA-Ansatz von Goodwin und Wilken [GW96]. Die Anzahl der Nebenbedingungen, die für die Beschreibung des Allokationsproblems formuliert werden, hängt von der Struktur des Kontrollflussgraphen und der Lebendigkeiten der Register ab. Dies erschwert eine realistische Größenordnung für ihre Anzahl nur aus dem Regelwerk abzuleiten. So bestimmen Goodwin und Wilken die Anzahl der Nebenbedingungen experimentell. Sie untersuchen dazu wie viele Nebenbedingungen für Beispielprogramme aufgestellt werden. Aus ihren Beobachtungen schließen sie auf eine Größenordnung von $\mathcal{O}(i^{1,3})$. Hierbei ist i die Anzahl der Instruktionen in der Zwischendarstellung.

Für den zweiten Teil des ILP wird nun die Anzahl der Entscheidungsvariablen und Nebenbedingungen abgeschätzt, die für die Modellierung der WCET einer Funktion nach dem Ansatz benötigt werden, der für diese Diplomarbeit gewählt wurde. Um die Abschätzung zu vereinfachen, wird angenommen, dass alle Ausgänge von Schleifen reguläre Ausgänge sind. Dies hat zur Folge, dass jeder Ausgang nur Ausgang einer Schleife ist (siehe Definition 5.14). Sei n die Anzahl der Knoten in dem Kontrollflussgraphen der Funktion, in dem jeder Knoten einen Basisblock repräsentiert, und m die Anzahl der Kanten.

Für jeden Basisblock B gibt es eine Entscheidungsvariable d_B, die die maximale Ausführungsdauer des Basisblocks angibt. Jeder Basisblock kann einer Schleife angehören, und wird daher möglicherweise beim Schälen der Schleife dupliziert. Da danach die geschälte Schleife zusammen mit ihrer Schale in dem Graphen ersetzt wird, gibt es für jeden Basisblock maximal zwei Entscheidungsvariablen m_B und $m_{B'}$. Bei Ersetzen einer Schleife wird für jeden Ausgang der Schleife ein Ersatzknoten erzeugt. Für jeden dieser Knoten gibt es eine Entscheidungsvariable g_A^L. Sind alle Ausgänge in dem Graphen reguläre Ausgänge, gibt es für jeden Ausgang nur einen Ersatzknoten. Schließlich ist ein regulärer Ausgang kein Ausgang für mehrere Schleifen. Insgesamt enthält das ILP dann für jeden Basisblock maximal vier Entscheidungsvariablen.

Die Anzahl der Constraints in dem ILP wird primär durch die Kanten in dem Kontrollflussgraphen bestimmt. Alle Kanten in einer Schleife, die keine Rücksprungkante sind, können beim Schälen der Schleife dupliziert werden. Bei der Modellierung der WCET der Schleifen wird für jede Kante ein Constraint formuliert, wie es durch die Gleichung 5.7 bzw. die Ungleichung 5.8 beschrieben ist. Für jede Kante im Ursprungsgraphen kann es daher zwei solcher Constraints geben. Die Gleichung 5.5 wird für jeden Schleifenein-

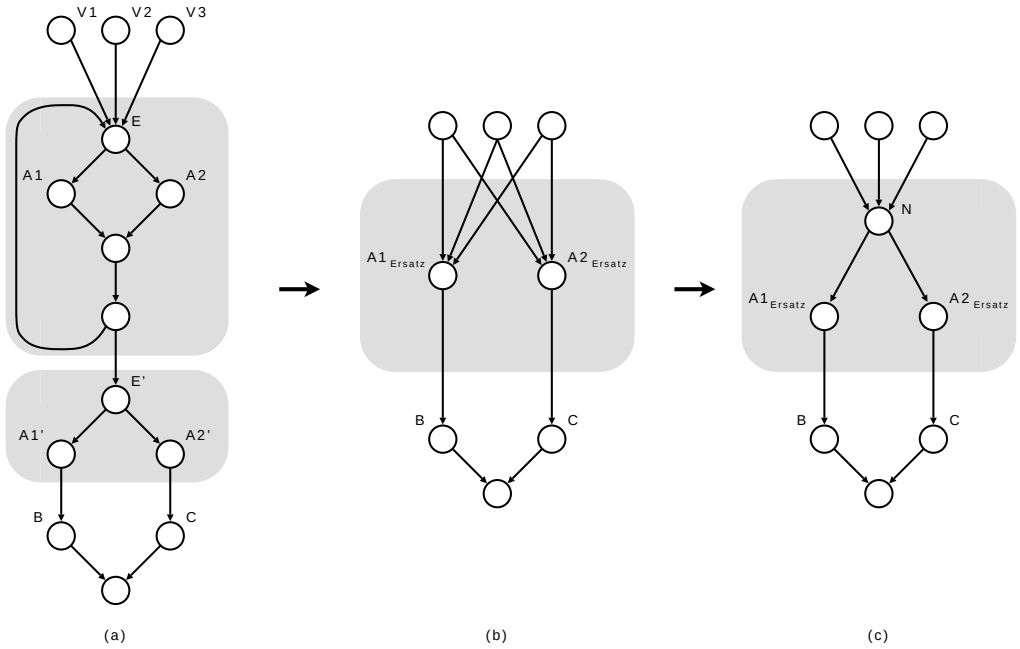

Abbildung 5.7: Transformation einer Schleife. (a) Schleife nach dem Schälen. (b) Graph nach der Ersetzung der Schleife. (c) Einfügen eines neuen Knoten vor den Ersatzknoten.

gang aufgestellt. Da den Rücksprüngen der Schleife bislang nur ein Constraint für den Übergang zwischen der geschälten Schleife und der Schale zugeordnet wurde, wird dieses Constraint ebenfalls diesen Kanten zugerechnet. Die Gleichung 5.9 wird für jeden Ausgang aufgestellt. Es kann einer der Kanten zugeordnet werden, die von dem Ausgang ausgehen. Bislang werden also für jede Kante maximal drei Constraints erstellt.

Bei dem Ersetzen einer Schleife werden die Kanten vervielfacht, die auf den Eingang der Schleife gerichtet sind. Von jeder Kante gibt es dann so viele Abbilder, wie die Schleife Ausgänge hat. Es können also pro Kante nicht mehr als $n - 1$ neue Kanten entstehen. Für jede Kante in dem ursprünglichen Kontrollflussgraphen werden daher insgesamt nicht mehr als $3n$ Constraints erstellt. Für den ungünstigen Fall, dass Schleifeneingänge viele eingehende Kanten aufweisen, muss die Anzahl der Constraint somit durch $\mathcal{O}(n \cdot m)$ abgeschätzt werden.

Die Anzahl der Variablen und Constraints kann noch verringert werden, indem die Variablen, die nur auf linken Seiten von Gleichungen vorkommen, durch den Ausdruck auf der rechten Seite substituiert wird. Auf diese Weise ersetzte Variablen können zusammen mit der Gleichung aus dem ILP entfernt werden. Für typische Funktionen mit Verzweigungen und Verschmelzungen vor und nach Basisblöcken wird sich die asymptotische Größenordnung dadurch aber nicht ändern.

Eine lineare Anzahl an Entscheidungsvariablen und Constraints kann dagegen erreicht werden, indem bei der Ersetzung einer Schleife ein weiterer Knoten erstellt wird, wie in Abbildung 5.7 zu sehen ist. Er wird zwischen den Vorgängern der Schleife und den Ersatzknoten eingefügt, die die Schleife repräsentieren, die über einen bestimmten Ausgang verlassen wird. Da er keinen Basisblock repräsentiert, wird für ihn eine Entscheidungsva-

riable erzeugt, die seine Ausführungsdauer als Null ausweist. Diese Entscheidungsvariable kann dem Eingang einer Schleife zugerechnet werden. Jeder Knoten in einem reduzierbaren Graphen kann nur Eingang einer Schleife sein. Der neue Knoten kann beim Schälen einer umschließenden Schleife dupliziert werden kann. Er ist aber kein Ausgang einer Schleife, da er immer Nachfolger in der Schleife hat. Somit werden für jeden Basisblock maximal zwei weitere Entscheidungsvariablen benötigt.

Alle Kanten, die auf den Eingang der Schleife gerichtet sind, werden auf den neuen Knoten gerichtet und nicht mehr vervielfacht. Zwischen dem neuen Knoten und den Ersatzknoten wird jeweils eine Kante eingefügt. Es entsteht also pro Ausgang der Schleife eine neue Kante. Die Constraints, die für die neue Kante erzeugt werden, können den Kanten zugerechnet werden, die aus einer Schleife herausführen. Einer Kante sind dann nicht mehr als fünf Constraints zugeordnet. Insgesamt kann die Anzahl der Entscheidungsvariablen weiterhin mit $\mathcal{O}(n)$ abgeschätzt werden, die Anzahl der Constraints kann jetzt aber mit $\mathcal{O}(m)$ angegeben werden.

Typischerweise hat ein Schleifeneingang aber nur einen Vorgänger außerhalb der Schleife. In ihm werden z. B. Variablen initialisiert. Für jede Schleife einen weiteren neuen Knoten einzufügen lohnt daher im Anwendungsfall nicht.

Für typische Programme, bei denen Schleifen nur einen oder wenige Vorgänger haben, werden für die Modellierung der WCET eine Anzahl von $\mathcal{O}(m)$ Nebenbedingungen benötigt. Da jeder Basisblock in der Regel höchstens zwei ausgehende Kanten hat, ist die Anzahl der Kanten in dem Kontrollflussgraphen linear zu der Anzahl der Knoten in dem Graphen. Die Knoten repräsentieren Basisblöcke, daher ist die Anzahl der Knoten nicht größer als die Anzahl i der Instruktionen in der Zwischendarstellung, wenn angenommen wird, dass es keine leeren Basisblöcke gibt. Die Anzahl der Nebenbedingungen in dem zweiten Teil des ILP kann daher auch mit $\mathcal{O}(i)$ angegeben werden. Die Größe des ILP insgesamt kann also weiterhin durch $\mathcal{O}(i^{1,3})$ abgeschätzt werden, wobei i die Anzahl der Instruktionen in der Zwischendarstellung ist.

Kapitel 6

Ermittlung der WCET-Daten

In dem vorherigen Kapitel wurde beschrieben, wie in dieser Diplomarbeit die WCET einer Funktion in der ILP-basierten Registerallokation modelliert wird, damit sie bei der Allokation minimiert wird. In der Modellierung werden Konstanten für die Ausführungszeit der Spillinstruktionen und von Basisblöcken sowie für die maximalen Ausführungswiederholungen von Schleifen verwendet. In diesem Kapitel soll die Frage beantwortet werden, wie die Werte für diese Konstanten ermittelt werden.

Obere Schranken für die Ausführungswiederholungen einer Schleife können im Allgemeinen nicht aus dem Programm selbst abgeleitet werden. Dennoch existiert in dem WCC eine Schleifenanalyse [Cor08]. Basierend auf abstrakter Interpretation und der Auswertung von Polytopen, versucht sie vorherzusagen, nach wie vielen Iterationen spätestens die Abbruchbedingung der Schleife erfüllt ist. In sehr vielen Fällen, wenn auch nicht allen, kann sie so die obere Schranken für die Ausführungswiederholungen der Schleifen automatisch nur an Hand des Programmcodes bestimmen. Für alle Schleifen, bei denen die Analyse nicht erfolgreich ist, müssen die Schranken vom Programmierer in Form von *flow facts*, in diesem Fall in Form von *loop bounds*, angegeben werden (siehe Abschnitt 3.3). Die Verwendung von *flow restrictions* zur Begrenzung der Iterationshäufigkeit von Schleifen wird von der in dieser Diplomarbeit beschriebenen Registerallokation nicht unterstützt. Der WCC verwaltet die Angaben zu den Schleifenwiederholungen während des Übersetzungsprozesses und speichert sie in der LLIR. In der Registerallokation kann dann für eine Schleife die Schranke für die Iterationshäufigkeit direkt aus dem Datenobjekt ausgelesen werden, das den Basisblock repräsentiert, der der Eingang der Schleife ist.

Im nächsten Abschnitt 6.1 wird angegeben, welche Werte für die Ausführungszeiten der Spillinstruktionen verwendet werden. Im folgenden Abschnitt 6.2 werden die Verfahren vorgestellt, mit denen eine Ausführungszeit für Basisblöcke bestimmt wird.

6.1 Ausführungszeiten von Spillinstruktionen

Eine Entscheidung darüber, ob eine Spillinstruktion in die Zwischendarstellung der Funktion eingefügt werden soll, wird in dem ILP durch eine Entscheidungsvariablen x_B^k modelliert. Die Auswirkung auf die Ausführungszeit des Basisblocks, in den die Instruktion eingefügt wird, wird durch eine Konstante $c(x_B^k)$ angegeben. Ihr Wert soll nur von dem Typ der Spillinstruktion abhängen, damit in dem ILP nicht auch noch das Verhalten der Pipeline des Prozessors der Zielarchitektur modelliert werden muss.

Für den Infineon TriCore-Prozessor (siehe Abschnitt 3.5), der für diese Diplomarbeit der Prozessor der Zielarchitektur ist, gibt es zahlreiche verschiedene Lade- und Speicherbefehle. Als Spillinstruktionen wurden davon nur vier Befehle verwendet. Je nachdem, ob

ein Adressregister oder ein Datenregister im Laufzeitstack gespeichert oder aus ihm geladen werden muss, wird eine andere Instruktion benötigt. Da erweiterte virtuelle Register in der Modellierung in einfache Register zerlegt werden, werden keine Spillinstruktionen für erweiterte Register verwendet.

Als Speicherbereich für den Laufzeitstack wird von dem WCC der SRAM-Datenspeicher genutzt, der Teil der Datenspeicher-Schnittstelle (DMI) des Prozessorkerns ist (siehe Abschnitt 3.5.4). Dadurch kann jede Speicherinstruktion, die auf den Laufzeitstack zugreift, im Normalfall in durchschnittlich einem Zyklus ausgeführt werden. Aus diesem Grund weisen die vier verschiedenen als Spillinstruktion verwendeten Speicher- und Ladebefehle das gleiche Zeitverhalten auf. Da der TriCore-Prozessor die Speicherinstruktionen in einer separaten Pipeline abarbeitet, können die Speicherinstruktionen z. T. parallel zu anderen Instruktionen ausgeführt werden (siehe Abschnitt 3.5.3). Dann führt das Einfügen einer Spillinstruktion zu keiner Verlängerung der Ausführungsdauer eines Basisblocks in der Pipeline. Weil das Einfügen von Instruktionen aber zu einer größeren Codegröße führt, kann in jedem Fall die Anzahl von Cache-Misses im Instruktionscache steigen. Selbst wenn die Spillinstruktion parallel zu anderen Instruktionen ausgeführt werden kann, würde sich dann die Ausführungsdauer des Basisblocks oder auch anderer Basisblöcke verändern. Dieser Effekt ist ebenfalls für alle Spillinstruktionsarten identisch, da nur Instruktionen mit einer Befehlswortlänge von 32 Bit verwendet werden. Für eine einzelne Spillinstruktion kann aber nicht konkret vorhergesagt werden, welche Auswirkungen sie auf das Cacheverhalten hat, da dazu eine Cache-Analyse nötig wäre.

In Anbetracht, dass die Verlängerung der Ausführungszeit eines Basisblocks durch das Einfügen einer Spillinstruktion sehr unterschiedlich ausfallen kann, ist die Wahl eines Wertes für die Konstanten $c(x_B^k)$ schwierig. Da er nur von dem Typ der Spillinstruktion abhängen soll, der Kontext, in den die Instruktion eingefügt wird, aber unberücksichtigt bleibt, ist es nicht möglich, einen Wert festzulegen, der an jeder Stelle die Verlängerung der Ausführungszeit richtig wiedergibt.

Ein Cache-Miss führt dazu, dass eine Cachezeile mit acht Instruktionen neu geladen wird. Dies dauert im Durchschnitt vier Prozessorzyklen. Eine Spillinstruktion verursacht also bei der ersten Ausführung durchschnittlich eine Verzögerung von einen halben Zyklus. Für alle weiteren Ausführungen ist unklar, ob die Instruktion im Cache vorliegt, oder erneut in den Cache geladen werden muss. Wird die Instruktion parallel zu einer anderen Instruktion ausgeführt, muss keine zusätzliche Ausführungsdauer für die Instruktion berechnet werden. Anderenfalls ist ein Zyklus zu veranschlagen, wenn vom günstigsten Fall ausgegangen wird, dass die Instruktion in der Pipeline störungsfrei verarbeitet wird. Alles in allem erscheint, dass der Wert Eins für die Konstanten $c(x_B^k)$ einen vertretbaren Kompromiss darstellt.

6.2 Bestimmung der Ausführungszeiten von Basisblöcken

In dem ILP, das das Registerallokationsproblem beschreibt, bei dem die WCET minimiert werden soll, wird die WCET einzelner Basisblöcke durch Entscheidungsvariablen d_B modelliert. Ihr Wert ist die Summe aus der maximalen Ausführungszeit der Instruktionen der Zwischendarstellung und der Verlängerung der Ausführungsdauer durch die Spillinstruktionen, die vom dem Registerallokator in den Basisblock eingefügt werden. Die Verlängerung der Ausführungsdauer setzt sich aus Konstanten $c(x_B^k)$ zusammen, de-

ren Wahl im vorherigen Abschnitt erläutert wurde. Die maximale Ausführungsdauer der Instruktionen der Zwischendarstellung, die dem Basisblock B angehören, wird durch die Konstante g_B repräsentiert. Wie die Werte dieser Konstanten ermittelt werden, wird in diesem Abschnitt vorgestellt.

Die Konstanten g_B sind so zu wählen, dass sie die maximale Ausführungsdauer des Basisblocks B nach der Registerallokation angeben, wenn in ihn kein Spillcode eingefügt werden würde. Dies ist aber keinesfalls die Ausführungsdauer des Basisblocks vor der Registerallokation. Vor der Registerallokation ist die Ausführungsdauer nicht ermittelbar, da die Zwischendarstellung virtuelle Register enthält. Solange die Zuordnung zu physikalischen Registern fehlt, kann eine WCET-Analyse keine Aussage über das Verhalten der Hardware treffen und daraus Ausführungszeiten ableiten. Eine WCET-Analyse setzt daher stets voraus, dass eine ausführbare Beschreibung des zu analysierenden Programmabschnitts vorliegt. Dies bedeutet insbesondere, dass in dem Programmabschnitt ausschließlich physikalische Register verwendet werden.

Die in dieser Diplomarbeit erstellte Registerallokation ist ein Übersetzungsschritt im WCC, der für die WCET-Analyse das Analysewerkzeug aiT verwendet (siehe Abschnitt 3.4). Es führt eine Analyse für ein gesamtes Programm durch, um die Ausführungskontexte von Funktionen und Basisblöcken berücksichtigen zu können. Damit sollen im Vergleich zur isolierten Betrachtung der Basisblöcke genauere Analyseergebnisse erzielt werden. Vor der WCET-Analyse müssen also alle virtuellen Register in allen Funktionen des Programms durch physikalische Register ersetzt werden. Weiterhin nimmt das Analysewerkzeug aiT eine Werteanalyse für die Inhalte der physikalischen Register vor. Folglich reicht es nicht aus, irgendeine Zuordnung von virtuellen zu physikalischen Registern vorzunehmen. Für die WCET-Analyse muss also für jede Funktion eine korrekte globale Registerallokation vorgenommen werden.

Auf den ersten Blick erscheint das Problem der Registerallokation, die die WCET einer Funktion minimieren soll, vielleicht paradox. Für die Registerallokation muss der WCEP modelliert werden. Dafür werden Ausführungszeiten von Basisblöcken benötigt, die erst nach einer Registerallokation ermittelt werden können. Dieses Problem kann aber gelöst werden, indem vor der WCET-Analyse eine andere Registerallokation durchgeführt wird, die diese Information nicht benötigt. Diese vorab durchgeführte Registerallokation wird im Folgenden Pre-Allokation genannt. Nach der WCET-Analyse und der Berechnung der Konstanten g_B wird das Ergebnis der Pre-Allokation verworfen und die ILP-basierte Allokation durchgeführt, die die WCET der Funktion minimieren soll.

Sehr wahrscheinlich wird die Pre-Allokation den virtuellen Registern andere physikalische Register zuordnen als die Registerallokation, die die WCET einer Funktion minimieren soll. Glücklicherweise sind bei dem Prozessor der Zielarchitektur, dem Infineon TriCore, alle Register eines Typs gleichwertig, wenn keine Instruktionen mit verkürzter Befehlswortlänge verwendet werden (siehe Abschnitt 3.5.1). Unterschiede in der Ausführungsdauer eines Basisblocks ohne Spillcode können dann nur durch Lese- und Schreibabhängigkeiten zwischen den Instruktionen entstehen. Der TriCore-Prozessor vermeidet für viele Abhängigkeiten z. B. durch *Forwarding*, dass aus ihnen Verlängerungen der Ausführungsdauer resultieren. Daher wird im Folgenden angenommen, dass die Ausführungsdauer eines Basisblocks ohne Spillcode nicht von der Registerzuordnung abhängt.

Ein Registerallokationsverfahren, das nie Spillcode einfügen muss, gibt es nicht. Dieses würde nebenbei ja auch schon das Registerallokationsproblem lösen, bei dem die WCET einer Funktion minimiert werden soll. Auf Grund der Anzahl der verfügbaren

physikalischen Register und der Lebendigkeiten der virtuellen Register besteht manchmal die Notwendigkeit, Spillcode einzufügen. Dadurch tritt ein zusätzliches Problem auf. Die Ausführungszeit der Basisblöcke soll nicht die Ausführungsdauerverlängerung der Spillinstruktionen umfassen. Die Basisblöcke können bei der WCET-Analyse aber Spillcode enthalten. Die Ergebnisse der WCET-Analyse sind somit nicht unmittelbar die gesuchten Werte für die Konstanten g_B. Dies erfordert, dass mit Kenntnis der eingefügten Spillinstruktionen von den Ergebnissen der WCET-Analyse Rückschlüsse auf die Ausführungszeit der Basisblöcke gezogen werden, die sie hätten, wenn sie kein Spillcode enthielten.

An dieser Stelle erscheint es vielleicht zweckmäßiger, wenn die Ausführungsdauer der Basisblöcke d_B in dem ILP auf eine andere Weise modelliert worden wäre. Statt auf eine Grundausführungsdauer die zusätzliche Ausführungsdauer der Spillinstruktionen zu addieren, hätte auch von der WCET des Basisblocks, der alle denkbaren Spillinstruktionen enthält, die Einsparung bei der Ausführungsdauer abgezogen werden können, die eintritt, wenn eine Spillinstruktion nicht eingefügt wird. Durch diesen Ansatz sind aber zwei Nachteile zu erwarten.

Zum einem können bei der ILP-basierten Registerallokation an sehr vielen Stellen Spillinstruktionen eingefügt werden. Besonders nach Verzweigungen und vor Verschmelzungen im Kontrollflussgraphen können an der Basisblockgrenze potenziell alle dort lebendigen Register auf den Laufzeitstack gesichert bzw. aus ihm geladen werden. All diese Spillinstruktionen müssten in den zu analysierenden Basisblöcken vorhanden sein, damit es für jede Spillinstruktion, die der ILP-basierte Registerallokator einfügen kann, eine Entsprechung gibt. Die zu analysierenden Basisblöcke würden dann sehr viel Spillcode enthalten, was die WCET-Analyse verlangsamen würde.

Zum anderen sind die Werte $c(x_B^k)$, die für eine nicht eingefügte Spillinstruktion von der WCET des Basisblocks subtrahiert werden würden, nach den Überlegungen gewählt, die im vorherigen Abschnitt dargelegt wurden. Sie stellen einen Kompromiss dar, der an jeder Stelle der falsche Wert sein kann. Die maximale Ausführungszeit des Basisblocks in dem Modell kann immer stärker von der tatsächlichen WCET abweichen, je öfter diese Konstanten von der WCET des Basisblocks mit Spillcode subtrahiert wird, d. h. je mehr Spillinstruktionen nicht eingefügt werden. Dass der ILP-basierte Registerallokator viele mögliche Spillinstruktionen nicht einfügt, ist aber sehr wahrscheinlich. Als Konsequenz wird die vorgeschlagene alternative Modellierung der Ausführungszeit der Basisblöcke in dieser Diplomarbeit nicht verfolgt. Es wird erwartet, dass die gewählte Modellierung die maximale Ausführungszeit der Basisblöcke genauer wiedergibt, wenngleich auch jeder Summand, sowohl die Ausführungsdauer des Basisblocks ohne Spillcode, als auch die Ausführungsverlängerung der Spillinstruktionen, geschätzt ist.

In dieser Diplomarbeit wird darauf Wert gelegt, dass das Pre-Allokationsverfahren einfach zu implementieren ist und eine geringe Laufzeit aufweist. Sie soll gegenüber dem ILP-basierten Verfahren, das die WCET der Funktion verringern soll, nicht ins Gewicht fallen. Aus diesem Grund wird in dieser Diplomarbeit für die Pre-Allokation ein naiver Allokator eingesetzt. Dieser naive Allokator verwaltet alle virtuellen Register auf dem Laufzeitstack. Er sichert jedes virtuelle Register unmittelbar nach seiner Definition in dem Speicher und lädt es unmittelbar vor seiner Nutzung wieder in ein physikalisches Register. Zwar fügt er auf diese Weise vergleichsweise viel Spillcode ein, aber dennoch bedeutend weniger als ein Allokator, der wie zuvor beschrieben an jeder Stelle Spillcode einfügt, an der auch der ILP-basierte Allokator Spillcode einfügen könnte.

Alternativ könnte für die Pre-Allokation auch ein Linear Scan Allokator eingesetzt werden. Z. B. könnte der Linear Scan Allokator von Poletto und Sarkar [PS99] verwendet werden, der in Abschnitt 2.2.3.2 vorgestellt wurde. Er hat den Vorteil, dass er ebenfalls eine lineare Laufzeit aufweist, aber im Vergleich zum naiven Registerallokator deutlich weniger Spillcode einfügt. Die auf die Pre-Allokation folgende WCET-Analyse könnte dann schneller ausgeführt werden und vielleicht könnten die Ausführungszeiten der Basisblöcke stellenweise besser abgeschätzt werden. Da die Implementierung aber zeitaufwändiger als die Implementierung des naiven Allokators ist, wurde in dieser Diplomarbeit ein naiver Allokator für die Pre-Allokation gewählt.

Nachdem für alle Funktionen eine Pre-Allokation vorgenommen wurde, wird mittels aiT eine WCET-Analyse für das gesamte Programm durchgeführt (siehe Abschnitt 3.4). Für jeden Basisblock liegt dann eine Zahl vor, die die von aiT berechnete WCET des Basisblocks inklusive Spillcode in Vielfachen von Prozessorzyklen angibt. Anschließend wird die maximale Ausführungszeit des Basisblocks ohne Spillcode geschätzt. In dieser Diplomarbeit werden dafür zwei Ansätze verfolgt. Es wird entweder eine einfache oder eine aufwändigere Schätzung vorgenommen.

Bei der einfachen Schätzung wird für jeden Basisblock der Anteil der Instruktionen, die keine Spillinstruktionen sind, an der Gesamtzahl der Instruktionen des Basisblocks bestimmt. Mit diesem Faktor wird die von aiT ermittelte WCET des Basisblocks gewichtet. Auf diese Weise wird der Anteil der maximalen Ausführungsdauer des Basisblocks geschätzt, der nicht auf Spillinstruktionen zurückgeht.

Bezeichne s_B die Anzahl der Spillinstruktionen in dem Basisblock B und n_B die Anzahl der übrigen Instruktionen. Ist $wcet_B$ die von aiT ermittelte WCET des Basisblocks, so wird g_B, das die Ausführungszeit des Basisblocks ohne Spillcode angeben soll, durch folgende Gleichung berechnet:

$$g_B = wcet_B \cdot \frac{n_B}{n_B + s_B} \tag{6.1}$$

Für leere Basisblöcke, d. h. Basisblöcke, die keine Instruktion enthalten, wird g_B auf Null gesetzt.

Bei der aufwändigeren Schätzung wird ebenfalls für jeden Basisblock die Anzahl s_B der Spillinstruktionen und die Anzahl n_B der übrigen Instruktionen bestimmt. Die Idee ist nun aber, mit dem Verhältnis aus der Anzahl der Instruktionen, die keine Spillinstruktionen sind, und der Gesamtzahl der Instruktionen nur den Anteil der Ausführungsdauer zu gewichten, der weder auf die Ausführungsdauer der Spillinstruktionen noch auf die Ausführungsdauer der übrigen Instruktionen eindeutig zurückzuführen ist. Dieser Anteil wird im Folgenden BCET-Abweichung genannt, da er die Abweichung der WCET von der minimalen Ausführungszeit (*best-case execution time* (BCET)) ist. Er umfasst die Dauer aller Verzögerungen, die laut der WCET-Analyse bei der Ausführung des Basisblocks auftreten können. Dazu zählen z. B. Verzögerungen durch Ressourcenbeschränkungen und Cache-Misses.

Um die BCET-Abweichung zu berechnen, wird die minimale Ausführungsdauer des Basisblocks geschätzt. Im Gegensatz zur WCET ist eine sichere BCET-Schätzung guter Qualität leicht möglich. Aber selbst die Auswirkungen einer schlechten Schätzung sind für die Schätzung der WCET des Basisblocks ohne Spillcode unschädlich. Warum dies so ist, wird später erläutert. Für die BCET-Schätzung ist für jede Instruktion eine minimale Anzahl von Zyklen festgelegt, die für die Ausführung der Instruktion mindestens

veranschlagt werden muss. Diese Daten wurden den Beschreibungen des Zielprozessors, dem Infineon TriCore, entnommen [Inf00], [Inf03]. Für jede Instruktion, die in der *Load-Store*-Pipeline parallel zu anderen Instruktionen in der *IP*-Pipeline ausgeführt werden kann, wird keine Ausführungszeit berechnet. Die BCET des Basisblocks wird schließlich ermittelt, indem die minimalen Ausführungszeiten der Instruktionen kumuliert werden. Die aufwändigere Schätzung kann daher nicht wie die einfache Schätzung in konstaner Zeit durchgeführt werden. Ihre Dauer ist linear zu der Anzahl der Instruktionen in dem Basisblock.

Die so geschätzte BCET des Basisblocks wird von der WCET subtrahiert. Das Ergebnis ist die BCET-Abweichung. Nachdem diese mit dem Verhältnis aus der Anzahl der Instruktionen, die keine Spillinstruktionen sind, und der Gesamtzahl der Instruktionen gewichtet wurde, wird schließlich noch die BCET des Basisblocks ohne Spillcode hinzu addiert. Diese wird wie die BCET des Basisblocks mit Spillcode geschätzt.

Die geschätzte maximale Ausführungszeit des Basisblocks g_B, wenn er keinen Spillcode enthielte, setzt sich also aus zwei Summanden zusammen. Der eine Summand ist die minimale Ausführungszeit, die für die Instruktionen geschätzt wurde, die keine Spillinstruktionen sind. Diese Ausführungszeit wird in jedem Fall benötigt. Der zweite Summand ist ein Anteil an der BCET-Abweichung, dessen Größe durch das Verhältnis zwischen den Spillinstruktionen und der übrigen Instruktionen bestimmt wird.

Sei s_B und n_B abermals die Anzahl der Spillinstruktionen bzw. der übrigen Instruktionen im Basisblock B. $wcet_B$ bezeichne weiterhin die von aiT ermittelte WCET des Basisblocks. $bcet_B^{spill}$ ist die geschätzte minimale Ausführungszeit des Basisblocks mit Spillcode, $bcet_B^{ohne}$ die geschätzte BCET des Basisblocks ohne Spillcode. Die WCET des Basisblocks ohne Spillcode g_B berechnet sich bei diesem Ansatz durch:

$$g_B = (wcet_B - bcet_B^{spill}) \cdot \frac{n_B}{n_B + s_B} + bcet_B^{ohne} \tag{6.2}$$

An der Gleichung lässt sich leicht die Auswirkung einer schlechten BCET-Schätzung erkennen. Eine schlechte Abschätzung der BCET führt nur dazu, dass der Wert von g_B sich dem Wert annähert, der sich durch Gleichung 6.1 nach dem einfachen Ansatz berechnet. Sowieso sind beide Gleichungen identisch, wenn beide BCET-Werte Null sind, oder wenn für die Schätzung der BCET stets ein Zyklus pro Instruktion angenommen wird. In diesem Fall ist der Betrag von $bcet_B^{spill}$ gleich der Summe $n_B + s_B$ und der Betrag von $bcet_B^{ohne}$ ist n_B. Die aufwändigere Abschätzung führt dann zu einem besseren Ergebnis, wenn Spillinstruktionen parallel zu anderen Instruktionen ausgeführt werden und der Anteil der Instruktionen an der Ausführungsdauer des Basisblocks nicht einen Zyklus beträgt.

In keinem Fall darf erwartet werden, dass sich mit einem der beiden Ansätze die WCET des Basisblocks ohne Spillcode genau berechnen lässt. Durch das Einfügen von Spillcode kann sich das Ausführungsverhalten stark ändern und Effekte wie z. B. Cache-Misses auslösen, die es ohne Spillcode nicht gäbe. Solche Auswirkungen werden in den Ansätzen nicht vollständig kompensiert. Zudem wird bei beiden Ansätzen angenommen, dass die BCET-Abweichung zu gleichen Anteilen von den Spillinstruktionen und den übrigen Instruktionen verursacht wird.

Beide Ansätze können noch nicht einmal garantieren, dass die berechnete WCET weiterhin eine sichere obere Schranke für die WCET des Basisblocks ohne Spillcode darstellt. Die Konstanten g_B wie auch die Konstanten $c(x_B^k)$ werden nur dazu verwendet, in

dem ILP möglichst genau die WCET der Funktion zu modellieren. Deshalb sind sichere maximale Ausführungszeiten auch nicht zwingend erforderlich. Die WCET der Funktion, die von dem ILP berechnet wird, wird daher nur innerhalb der Registerallokation weiterverwendet. Für eine verlässliche Aussage über die WCET der Funktion nach der Registerallokation muss eine erneute WCET-Analyse durchgeführt werden.

Kapitel 7

Ergebnisse

Viele Optimierungen in einem Compiler sind optional, d. h. sie können wahlweise angewendet oder unterlassen werden. Ihre Güte kann einfach bewertet werden, indem mit dem Compiler einmal Maschinencode mit und einmal ohne Anwendung der Optimierung erstellt wird. Durch den Vergleich der beiden resultierenden Maschinenprogramme kann eine Aussage über die Güte der Optimierung getroffen werden. Dieses einfache Verfahren ist für die Bewertung eines Registerallokators leider nicht anwendbar. Werden in dem Compiler virtuelle Register verwendet, ist die Registerallokation eine Optimierung, die in jedem Fall ausgeführt werden muss. Aus diesem Grund können Aussagen über die Güte der Registerallokation nur getroffen werden, indem sie mit anderen Registerallokationsverfahren verglichen wird.

Neue Registerallokationsverfahren werden fast immmer durch Vergleiche mit Graphfärbungsverfahren (siehe Abschnitt 2.2.3.1) bewertet. Auch für die Evaluation des in dieser Diplomarbeit erstellten Registerallokators wird ein Registerallokator verwendet, der auf dem Graphfärbungsansatz beruht. Der Registerallokator, der in dieser Diplomarbeit mit dem Ziel erstellt wurde, die WCET eines Programms zu verringern, wird im Folgenden WCET-optimierender Registerallokator genannt. Der Registerallokator, der auf dem Graphfärbungsansatz beruht, wird als Graphfärbungsallokator bezeichnet. Basierend auf dem ORA-Modell, das für den WCET-optimierenden Registerallokator genutzt wird, konnte noch ein weiterer ILP-basierter Registerallokator erstellt werden. Er unterscheidet sich von dem WCET-optimierenden Registerallokator lediglich in der Zielfunktion. Statt die WCET eines Programms zu optimieren, minimiert er die Codegröße des Programms. Er wird daher Codegröße-optimierender Registerallokator genannt. Während bei der Gegenüberstellung des Graphfärbungsallokators und des WCET-optimierenden Registerallokators zwei grundlegend verschiedene Allokationsverfahren verglichen werden, soll der Codegröße-optimierende Registerallokator bessere Aussagen über die WCET-Modellierung zulassen, die in den Kapiteln 5 und 6 vorgestellt wurde.

Zunächst sollen diese drei Allokationsverfahren genauer vorgestellt werden:

Graphfärbungsallokator

Bei dem Graphfärbungsallokator handelt es sich um den Registerallokator, der zur Zeit in dem WCC verwendet wird. Er realisiert das Allokationsverfahren eines Registerallokators nach dem Ansatz von Briggs [Bri92], wendet also die *optimistic spill heuristic* an. Er erstellt basierend auf den Überschneidungen von Lebendigkeiten der Register einen Interferenzgraphen und versucht eine k-Färbung des Graphen zu bestimmen, indem er ihn durch die Entnahme von Knoten vereinfacht. Enthält der Interferenzgraph in der Phase, in dem der Graph vereinfacht wird, nur noch Knoten mit mindestens k Nachbarn, wird derjenige Knoten mit den meisten Nachbarn

ausgewählt. Optimierungen wie *coalescing*, *rematerialization* oder *live rang splitting* führt der Allokator nicht aus. Somit werden lediglich den virtuellen Registern physikalische Register zugeordnet und Spillcode eingefügt. Der Allokator eignet sich daher besonders für Vergleiche mit anderen Allokatoren, bei denen die Allokatoren nur bezüglich des eingefügten Spillcodes beurteilt werden sollen.

Die Anzahl k der für die Färbung zur Verfügung stehenden Farben entspricht genau der Anzahl der verfügbaren physikalischen Register. Der Graphfärbungsallokator verwendet von dem TriCore-Prozessor nur die Register in dem oberen Kontext (siehe Abschnitt 3.5.1), da diese bei Funktionsaufrufen automatisch gesichert und wiederhergestellt werden. Weil der *stack pointer* und das Register, in dem die Rücksprungadresse für Funktionsaufrufe gespeichert ist, ebenfalls dem oberen Kontext angehören, stehen dem Allokator somit vier *general purpose*-Adressregister und acht *general purpose*-Datenregister frei zur Verfügung.

WCET-optimierender Registerallokator

Der WCET-optimierende Registerallokator ist der ILP-basierte Registerallokator, der das in den Kapiteln 4 bis 6 beschriebene Allokationsverfahren implementiert. Da für die Abschätzung der WCET der Basisblöcke zwei Ansätze zur Verfügung stehen, wird in diesem Kapitel zwischen zwei WCET-optimierenden Registerallokatoren unterschieden. Der Allokator, der die einfache Abschätzung für die WCET der Basisblöcke verwendet (siehe Abschnitt 6.2), wird im Folgenden einfacher WCET-Registerallokator genannt, der andere, der von der aufwändigeren Abschätzung Gebrauch macht, wird aufwändigerer WCET-Registerallokator genannt.

Für einen fairen Vergleich mit dem Graphfärbungsallokator nutzen beide WCET-optimierenden Registerallokatoren wie dieser nur Register aus dem oberen Kontext des Registersatzes, wenn nicht eine Vorfärbung die Verwendung eines Registers aus dem unteren Kontext erzwingt. Im Gegensatz zu dem Graphfärbungsallokator führt ein ILP-basierter Registerallokator aber inhärent ein *live rang splitting* durch. Weitere Optimierungen nehmen die WCET-optimierenden Registerallokatoren aber nicht vor.

Codegröße-optimierender Registerallokator

Der Codegröße-optimierende Registerallokator basiert wie der WCET-optimierende Registerallokator auf dem ORA-Ansatz, der in Kapitel 4 vorgestellt wurde. Das ILP, das von dem Codegröße-optimierenden Registerallokator zur Lösung des Registerallokationsproblems formuliert wird, stimmt mit dem des WCET-optimierenden Registerallokators in dem Teil überein, der nur das Allokationsproblem ohne eine Zielfunktion beschreibt. Die Zielfunktion des Registerallokators kann direkt nur durch die Zielfunktion des ILP modelliert werden. Jede Entscheidungsvariable, die eine Entscheidung über das Einfügen von Spillcode modelliert, hat den Wert Eins, wenn eine Spillinstruktion eingefügt werden soll, und sonst den Wert Null. Die Werte dieser Entscheidungsvariablen werden mit der Anzahl der Bytes gewichtet, die die Codegröße der einzelnen Spillinstruktion wiedergeben, und anschließend aufsummiert. Das Ergebnis ist die Codegröße des Spillcodes insgesamt und definiert den Wert der Zielfunktion des ILP.

Die Übersetzungsschritte, die der WCC vor der Registerallokation durchführt, sind deterministisch und für jedes der Registerallokationsverfahren identisch. Daher wird jeder der Allokatoren für ein und die selbe Eingabe des WCC auf die gleiche Zwischendarstellung angewendet. Nach der Registerallokation werden keine weiteren Optimierungen durchgeführt. Auch redundante Kopierinstruktionen wurden nicht entfernt, so dass sich die Ausgaben der Registerallokatoren neben der Verwendung der physikalischen Register nur durch den eingefügten Spillcode unterscheiden. Nach der Registerallokation erfolgt unmittelbar die Erzeugung des Maschinencodes. Die Unterschiede in den Ausgaben der verschiedenen Registerallokationsverfahren bleiben somit erhalten.

Die im Folgenden vorgestellten Ergebnisse resultieren aus Testläufen des WCC auf einem PC-System mit einem Intel Xeon Prozessor mit 2,4 GHz Taktfrequenz und 4 MB Cache und 8 GB RAM Arbeitsspeicher. Die WCET-Analyse in dem WCC erfolgt mit dem Analysewerkzeug aiT Version 2.0, Build #86374 der Firma AbsInt Angewandte Informatik GmbH [HF06] für den Infineon TC1796 Version 1.3.

Die Formulierung der ILP erfolgte mit der Klassenbibliothek LIBILP des WCC. Diese stellt auch eine Methode zum Lösen der ILP bereit, bei der ein externer mILP-Solver genutzt wird. Für die Lösung der ILP wurde in dieser Diplomarbeit der mILP-Solver CPLEX Version 11.0.0 der Firma ILOG eingesetzt [ILO08]. Da das Bestimmen einer optimalen Lösung sehr lange dauern kann, wurde für jedes ILP eine Zeitschranke von 600 Sekunden festgelegt. Möglicherweise reicht diese Zeitspanne für CPLEX nicht aus, eine nachweislich optimale Lösung zu finden. Dann können zwei Fälle eintreten. Entweder hat CPLEX immerhin eine Lösung gefunden, die allen Nebenbedingungen genügt. Oder aber, CPLEX gibt keine Lösung für das ILP aus. Im ersten Fall ist die gefundene Lösung möglicherweise nicht die optimale Lösung, oder es ist die optimale Lösung, nur konnte noch nicht ausgeschlossen werden, dass es keine bessere Lösung gibt. Da aber alle Nebenbedingungen erfüllt sind, kann aus der Lösung, die CPLEX mit dem Vermerk suboptimal ausgibt, eine Lösung für die Registerallokation erstellt werden. Im zweiten Fall gibt CPLEX keine Lösung für das ILP aus und es kann keine Registerallokation durchgeführt werden. Der WCC bricht in diesem Fall den Übersetzungsvorgang ab.

Für die Bewertung der Registerallokatoren wurde der WCC auf eine Sammlung von Testprogrammen angewendet, die von der Forschungsgruppe für eingebettete Systeme am Lehrstuhl 12 der Fakultät für Informatik der Technischen Universität Dortmund unter dem Namen WCETBENCH zusammengestellt wurde. Ein Großteil der Programme in der WCETBENCH stellt typische Anwendungen für einen digitalen Signalprozessor dar. So enthält die WCETBENCH Funktionen zur schnellen Fourier-Transformation, digitalen Filterung und Matrizenmultiplikation. Sie umfasst Programme, die von der WCET-Forschungsgruppe am Mälardalen Real-Time Research Centre (MRTC) [Mäl08] zusammengetragen wurden, sowie einen Teil der Mediabench [LPMS97]. Insgesamt umfasst die WCETBENCH etwa 100 C-Programme mit über 300 verschiedenen Funktionen. Der WCC überführt die Quellprogramme bei Optimierungsstufe O0 in Zwischendarstellungen, die bis zu 3000 Instruktionen enthalten. Das dritte Quartil liegt etwa bei 100 Instruktionen, d. h. nur 25% der Zwischendarstellungen haben mehr als 100 Instruktionen. Ein Großteil der Zwischendarstellungen weist eher eine geringe Größe auf.

Im Folgenden soll nun die Güte der Registerallokatoren verglichen werden. Da in dieser Diplomarbeit ein Registerallokator entwickelt werden sollte, der die WCET eines Programms minimiert, wird als ein Vergleichskriterium die WCET der übersetzten Testprogramme verwendet. Um diese zu ermitteln, führt der WCC erneut eine WCET-Analyse

mit aiT durch, nachdem er das Testprogramm in ein Maschinenprogramm übersetzt hat. Die Ergebnisse der Vergleiche bezüglich der WCET der Programme werden in Abschnitt 7.1 vorgestellt.

Als zweites Kriterium wird die Laufzeit der Registerallokatoren untersucht. Die Resultate der Laufzeitvergleiche werden in Abschnitt 7.2 vorgestellt. Schließlich werden in Abschnitt 7.3 die ILP betrachtet. Dabei wird die Anzahl der Nebenbedingungen in Bezug zur Größe der Zwischendarstellung gesetzt.

7.1 Vergleiche bezüglich der WCET

In den ILP-Modellen der WCET-optimierenden Registerallokatoren werden Konstanten $c(x_B^k)$ verwendet, die die Ausführungsdauerverlängerung angeben soll, die beim Einfügen einer Spillinstruktion entsteht. In Abschnitt 6.1 wurde nach einigen Überlegungen, aber doch ohne stichhaltigen Beweis, für die Konstanten der Wert Eins festgelegt. In Abschnitt 7.1.1 wird der Einfluss dieser Konstanten untersucht. Basierend auf den Ergebnissen dieser Untersuchung wird gegebenfalls ein neuer Wert gewählt, der bessere Ergebnisse verspricht. Für die folgenden Untersuchungen wird dann dieser neue Wert verwendet.

In Abschnitt 7.1.2 werden die beiden Ansätze für die Schätzung der Ausführungszeiten der Basisblöcke ohne Spillcode verglichen. Dazu werden der einfache WCET-Registerallokator und der aufwändigere WCET-Registerallokator gegenübergestellt. Der bessere der beiden Allokatoren wird schließlich in Abschnitt 7.1.3 mit dem Graphfärbungsallokator verglichen. In Abschnitt 7.1.4 erfolgt dann der Vergleich zwischen dem WCET-optimierenden und dem Codegröße-optimierenden Allokator. Eine Gegenüberstellung des Graphfärbungsallokators und des Codegröße-optimierenden Allokators in Abschnitt 7.1.5 schließen die Untersuchungen bezüglich der WCET ab.

7.1.1 Untersuchung der $c(x_B^k)$-Werte

Zunächst wird der Einfluss der Konstanten $c(x_B^k)$, die die Ausführungsdauerverlängerung durch das Einfügen einer Spillinstruktion beschreiben, im einfachen WCET-Registerallokator untersucht. Jedes Testprogramm wurde dazu nacheinander mit den vier Optimierungsstufen O0, O1, O2 und O3 des WCC übersetzt und dies mit verschiedenen Werten für die Kostanten $c(x_B^k)$ wiederholt. Als Werte für die Konstanten wurde hier 1,0 , 1,5 , 2,0 und 2,5 ausprobiert.

In Tabelle 7.1 sind beispielhaft für vier Testprogramme die Ergebnisse dargestellt. Jede Tabelle zeigt die Ergebnisse für ein Testprogramm. Hier wurden die Programme *expint*, *mult_10_10*, *petrinet* und *select* ausgewählt. In den Tabellen sind die WCET der Maschinenprogramme als Vielfache von Prozessorzyklen angegeben, die aus der Übersetzung mit dem WCC resultieren. In einer Spalte sind die Ergebnisse aufgeführt, die jeweils mit dem gleichen Wert für die Konstanten $c(x_B^k)$ erzielt wurden. Die Zeilen enthalten die Ergebnisse, die bei gleicher verwendeter Optimierungsstufe, aber unterschiedlichen Werten für die Konstanten ermittelt wurden. Der beste Wert in einer Zeile, d. h. die geringste WCET, ist durch Fettdruck hervorgehoben.

Bei der Betrachtung der Ergebnisse kann festgehalten werden, dass die Auswirkungen der verschiedenen $c(x_B^k)$-Werte auf die WCET des resultierenden Maschinenprogramms sehr unterschiedlich sind. Während für das Testprogramm *petrinet* nur bei Verwendung der Optimierungsstufe O0 geringe Unterscheide festzustellen sind, ist bei den übrigen

expint	1,0	1,5	2,0	2,5
O0	8886	**8299**	9138	9198
O1	8247	9223	**8146**	9023
O2	9167	11406	10252	**9060**
O3	**9027**	9501	9918	9502

mult_10_10	1,0	1,5	2,0	2,5
O0	90979	92425	108210	**88504**
O1	**85318**	86922	88964	94529
O2	74478	74450	74415	**71466**
O3	**68358**	69207	76648	72299

petrinet	1,0	1,5	2,0	2,5
O0	6645	**6476**	6784	**6476**
O1	5914	5914	5914	5914
O2	5869	5869	5869	5869
O3	5869	5869	5869	5869

select	1,0	1,5	2,0	2,5
O0	12279	12279	**8582**	8719
O1	11046	**10872**	12093	11046
O2	11069	**10758**	12455	10760
O3	10824	**10758**	12331	11069

Tabelle 7.1: WCET bei Anwendung des einfachen WCET-Allokators in Abhängigkeit der Konstanten $c(x_B^k)$ und der Optimierungsstufen. In jeder Zeile sind die besten Werte durch Fettdruck hervorgehoben.

	1,0	1,5	2,0	2,5
O0	18	20	25	20
O1	8	13	15	18
O2	12	20	11	18
O3	13	18	15	19

Tabelle 7.2: Anzahl der besten Ergebnisse bei Anwendung des einfachen WCET-Allokators in Abhängigkeit der Konstanten und Optimierungsstufen.

	1,0	1,5	2,0	2,5
O0	20	19	13	19
O1	21	9	13	10
O2	19	10	13	4
O3	19	8	18	10

Tabelle 7.3: Anzahl der schlechtesten Ergebnisse bei Anwendung des einfachen WCET-Allokators in Abhängigkeit der Konstanten und Optimierungsstufen.

Programmen ein deutlicher Einfluss zu erkennen. Bei dem Testprogramm *mult_10_10*, das mit Optimierungsstufe O0, übersetzt wurde, sind Abweichungen von fast 20% aufgetreten, bei dem Programm *select* sogar von 30%.

Die Beispiele zeigen weiterhin, dass es keinen Wert für die Konstanten gibt, der immer zu dem besten Ergebnis führt. Zwar können bei dem Programm *select* mit der Konstante 1,5 die besseren WCET-Werte erzielt werden, bei den Programmen *expint* und *mult_10_10* hängt der Einfluss der Konstanten aber auch stark mit der verwendeten Optimierungsstufe zusammen.

Es ist daher schwierig, einen Wert für die $c(x_B^k)$-Konstanten festzulegen, der für die weiteren Untersuchungen des einfachen WCET-Allokators genutzt wird. Um eine Hilfe bei dieser Wahl zu erhalten, wurden die WCET-Daten für 86 Programme ausgewertet, bei denen die Übersetzung erfolgreich war. D. h. der Kontrollflussgraph für Programme ist reduzierbar und sowohl die WCET-Analyse der pre-allokierten Zwischendarstellung als auch die Bestimmung einer Lösung für das ILP innerhalb von 600 Sekunden führten zu einem Ergebnis. In Abhängigkeit der verwendeten Optimierungsstufe wurde für jeden der vier untersuchten Werte für die Kostanten $c(x_B^k)$ ermittelt, wie oft er im Vergleich zu den anderen untersuchten Werten zu einem besten WCET-Wert führt. Die Fälle, bei denen alle untersuchten Werte für die Konstanten zu dem gleichen Ergebnis führen, wurden ausgelassen. Dies sind für die Optimierungsstufe O0 26 Testprogramme und für die Optimierungsstufen O1 bis O3 38 bzw. 39 Programme. Sollten aber z. B. nur zwei Werte zu den besseren Ergebnissen führen, wird dies für beide Werte angerechnet. Das Ergebnis dieser Untersuchung ist in der Tabelle 7.2 dargestellt. Eine Spalte enthält wieder die WCET für einen festen Wert der Konstanten $c(x_B^k)$. In den Zeilen wird zwischen den Optimierungsstufen unterschieden.

Zusätzlich wurde gezählt, wie oft ein untersuchter Wert zu den schlechtesten Ergebnissen führt. Auch hier werden nur die Fälle betrachtet, bei denen es mindestens einen besten und einen schlechtesten Wert gab. Das Ergebnis dieser Zählung zeigt Tabelle 7.3.

Nochmal ist die große Schwankung der Güte der Allokation in Abhängigkeit der verwendeten Werte für die Konstanten $c(x_B^k)$ und der Optimierungsstufe zu erkennen. Auch auf diese Weise lässt sich kein bester Wert für die Konstanten ermitteln. Je nach Optimierungsstufe fallen die Häufigkeiten unterschiedlich aus, und auch ein deutlicher Trend ist nicht festzustellen. Eine Vielzahl an besseren Ergebnissen und eine geringe Anzahl an schlechteren Ergebnissen verspricht der Wert 1,5. Dieser Wert wird für die Konstanten $c(x_B^k)$ bei den folgenden Untersuchungen verwendet.

Die selben Untersuchungen und Auswertungen wurden auch für den aufwändigeren WCET-Allokator vorgenommen. Zunächst wird wieder der Einfluss der Wahl der Werte für die Konstanten $c(x_B^k)$ auf die WCET für einzelne Programme betrachtet. Die Ergebnisse sind in der Tabelle 7.4 dargestellt.

Auch hier können die gleichen Beobachtungen wie für den einfacheren WCET-Allokator gemacht werden. Bei dem Testprogramm *pertinet* ist eine Abweichung in den WCET des übersetzten Programms nur für die Optimimierungsstufe O0 festzustellen. Bei den anderen Programmen sind Abweichung für alle Optimierungsstufen vorhanden. Für das Programm *select*, das mit Optimierungsstufe O1 übersetzt wurde, beträgt die Abweichung zwischen dem besten und schlechtesten beobachteten Wert rund 38%. Abermals kann kein Wert ermittelt werden, der immer oder zumindest in einer deutlichen Mehrheit der Fälle zu der geringeren WCET führt.

Die Häufigkeit, mit denen beste und schlechteste Ergebnisse mit den unterschiedlichen

expint	1,0	1,5	2,0	2,5
O0	13057	10677	10820	**9485**
O1	9336	8879	8826	**8433**
O2	**7679**	8286	8756	9230
O3	**9514**	9907	9703	10414

mult_10_10	1,0	1,5	2,0	2,5
O0	90734	93919	109058	**89918**
O1	91357	**85444**	88891	93557
O2	73357	80442	**72967**	76761
O3	71313	68692	**67030**	69388

petrinet	1,0	1,5	2,0	2,5
O0	6735	**6617**	6643	6476
O1	5914	5914	5914	5914
O2	5869	5869	5869	5869
O3	5869	5869	5869	5869

select	1,0	1,5	2,0	2,5
O0	12519	12058	**8719**	11757
O1	11996	10810	**7464**	11046
O2	11001	10772	11205	**10758**
O3	11001	10772	10898	**10621**

Tabelle 7.4: WCET bei Anwendung des aufwändigeren WCET-Allokators in Abhängigkeit der Konstanten $c(x_B^k)$ und der Optimierungsstufen. In jeder Zeile sind die besten Werte durch Fettdruck hervorgehoben.

	1,0	1,5	2,0	2,5
O0	14	16	22	18
O1	7	13	20	13
O2	13	10	15	17
O3	10	15	14	14

Tabelle 7.5: Anzahl der besten Ergebnisse bei Anwendung des aufwändigeren WCET--Allokators in Abhängigkeit der Konstanten und Optimierungsstufen.

	1,0	1,5	2,0	2,5
O0	23	16	13	18
O1	23	12	11	9
O2	19	16	9	11
O3	14	12	11	14

Tabelle 7.6: Anzahl der schlechtesten Ergebnisse bei Anwendung des aufwändigeren WCET-Allokators in Abhängigkeit der Konstanten und Optimierungsstufen.

Werten für die Kostanten $c(x_B^k)$ erzielt wurden, sind unterschieden nach der genutzten Optimierungsstufe in den Tabellen 7.5 und 7.6 aufgeführt. Dieses Mal können von dem Wert 2,0 die meisten besten und die wenigsten schlechtesten Ergebnisse erwartet werden. Für den aufwändigeren WCET-Allokator wird fortan dieser Wert für die Konstanten $c(x_B^k)$ verwendet.

Insgesamt lässt sich somit feststellen, dass die Wahl, die für die Konstanten $c(x_B^k)$ in Abschnitt 6.1 getroffen wurde, mit dem Wert 1,0 nicht gerechtfertigt ist. Je nach untersuchtem Testprogramm sind durch eine andere Wahl bessere Ergebnisse möglich. Scheinbar gibt ein Zyklus nicht die Ausführungsverlängerung durch das Einfügen einer Spillinstruktion richtig wieder. Möglicherweise sind die großen beobachteten Abweichungen darauf zurückzuführen, dass ein fester Wert die Verlängerung der Ausführungszeit für alle Instruktionen nicht richtig beschreiben kann. Ebenso könnte dies erklären, warum der Wert für die Konstanten $c(x_B^k)$, der zu dem besseren Ergebnis führt, für jedes Programm und jede angewendete Optimierungsstufe unterschiedlich sein kann.

Die nachträgliche Änderung des Wertes für die Konstanten $c(x_B^k)$ ist normalerweise natürlich problematisch. Werden die Testfälle, die für die Evaluation eines Verfahrens genutzt werden, auch für die Parameterwahl verwendet, ist das angepasste Verfahren genau auf die Testfälle optimiert. Ein objektiver Vergleich mit einem Verfahren, das nicht auf die Testprogramme zugeschnitten ist, ist dann nicht mehr möglich. Dies muss also bei den weiteren Beurteilungen der Allokationsverfahren berücksichtigt werden. Allerdings zeigen die Schwankungen in der Güte des Allokationsverfahrens, egal mit welchem Wert für die Konstanten $c(x_B^k)$ es ausgeführt wurde, dass dadurch zwar für den Einzelfall Verbesserungen erzielt werden können, bei der Betrachtungen aller Programme aber kein wesentlicher Vorteil durch die nachträgliche Änderung der $c(x_B^k)$-Werte zu erwarten ist. Auch wurden bei der Wahl der Konstanten nur vier Werte berücksichtigt, und die Auswahl wurde an Hand von Häufigkeiten getroffen. Die Qualität der Verbesserungen wurde dagegen kaum beachtet.

7.1.2 Vergleich der Schätzungsansätze

In diesem Abschnitt werden die Schätzungsansätze miteinander verglichen, mit denen für jeden Basisblock B die Ausführungsdauer g_B bestimmt wird, die dann vorläge, wenn der Basisblock keinen Spillcode enthielte. Da die Güte der Allokationsverfahren schon stark von den Werten der Konstanten $c(x_B^k)$ abhängen, werden diese bei der Untersuchung der Schätzungsansätze mit einbezogen. Es wird daher der einfache Schätzungsansatz zweimal mit dem aufwändigerem Ansatz verglichen. Einmal wird als Wert für die Konstanten $c(x_B^k)$ 1,5 und einmal 2,0 gewählt.

Für die Untersuchung der Schätzungsansätze wird für jedes Testprogramm der WCC einmal unter Nutzung des einfachen WCET-Allokators und einmal unter Nutzung des aufwändigeren WCET-Allokators ausgeführt. Wie auch schon bei der Untersuchung des Einflusses der $c(x_B^k)$-Werte, wird dies für jede Optimierungsstufe des WCC wiederholt. Für jedes resultierende Maschinenprogramm wird von aiT eine WCET-Analyse durchgeführt. An Hand dieser Ausgaben wird die relative Verbesserung des aufwändigeren Schätzungsverfahren gegenüber dem einfachen Schätzungsverfahren nach dieser Gleichung berechnet:

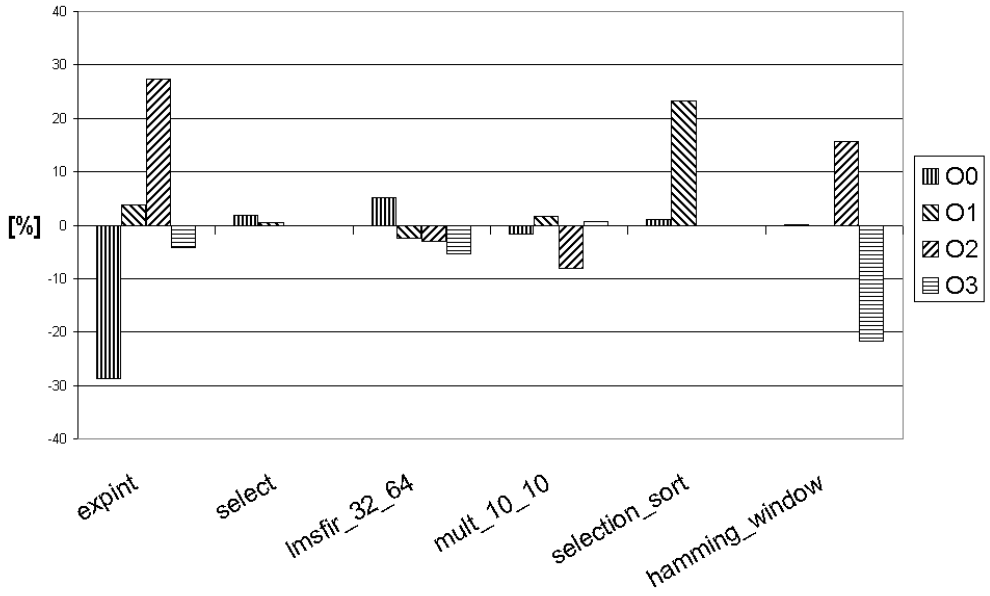

Abbildung 7.1: Verbesserungen des aufwändigeren WCET-Allokators gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$ in Prozent.

$$\text{Verbesserung} = \frac{wcet_{\text{einfach}} - wcet_{\text{aufwändig}}}{wcet_{\text{einfach}}} * 100\% \tag{7.1}$$

$wcet_{\text{einfach}}$ ist die WCET des Programms, das unter Nutzung des einfachen Schätzungsansatzes übersetzt wurde, $wcet_{\text{aufwändig}}$ ist die WCET des Programms, wenn der aufwändigere Ansatz genutzt wurde. Ist die WCET eines Maschinenprogramms größer, wenn der aufwändigere Schätzungsansatz statt des einfachen Ansatzes verwendet wird, tritt eine Verschlechterung ein. Die berechnete Verbesserung hat in diesem Fall ein negatives Vorzeichen.

Als erstes werden die beiden Schätzungsansätze für $c(x_B^k) = 1,5$ untersucht. Einige exemplarische Ergebnisse des Vergleichs der beiden Schätzungsansätze sind in Abbildung 7.1 dargestellt. Das Säulendiagramm zeigt die Verbesserung, die das aufwändigere Schätzverfahren gegenüber dem einfachen Schätzverfahren erzielt. Die Ergebnisse sind nach den Programmen gruppiert. Innerhalb einer Gruppe wird zwischen Optimierungsstufen unterschieden. Die Höhe der Säulen ist proportional zu der prozentualen Verbesserung.

Für die zwei aufgeführten Programme *select* und *lmsfir_32_64* sind zum Teil keine Änderungen oder nur geringe Änderungen in der WCET zu beobachten. Für die Programme *expint*, *selection_sort* und *hamming_window* sind dagegen deutliche Verbesserungen aber auch Verschlechterungen zu erkennen. Erneut ist festzustellen, dass für ein Testprogramm in Abhängigkeit davon, welche Optimierungsschritte vor der Registerallokation ausgeführt wurden, sowohl bessere als auch schlechtere Ergebnisse auftreten.

Mit einer Verbesserung als auch Verschlechterung von fast 30% ist das Testprogramm *expint* aber ein Einzelfall. Für fast die Hälfte der 86 Programme führen beide Ansätze zu der selben WCET, und für einen weiteren Großteil sind die relativen Abweichungen

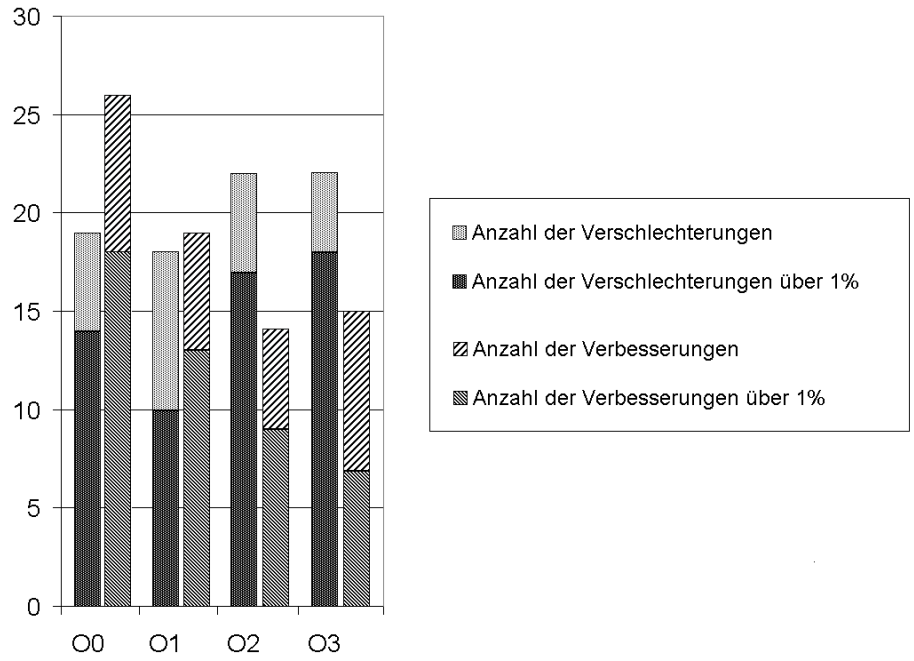

Abbildung 7.2: Anzahl der Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$.

durchschnittlicher Wert	O0	O1	O2	O3
- einer Verschlechterung in %	-4,36	-2,46	-3,15	-5,50
- einer Verbesserung in %	3,27	4,05	4,63	2,06

Tabelle 7.7: Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$.

kleiner als ein Prozent. In Abbildung 7.2 sind die Häufigkeiten, mit denen Verbesserungen und Verschlechterungen auftraten, in einem Säulendiagramm dargestellt. Der Anteil der Programme, bei denen eine Abweichung von mehr als ein Prozent auftrat, ist nochmal besonders hervorgehoben. Auf Grund der Berücksichtigung von Optimierungsstufen kann festgestellt werden, dass der aufwändigere Ansatz quantitativ bei den Optimierungsstufen O0 und O1 zu mehr Verbesserungen als Verschlechterungen führt und sich dieser Trend bei den Optimierungsstufen O2 und O3 ins Gegenteil verkehrt.

Das Ergebnis einer qualitativen Auswertung der Abweichungen zeigt Tabelle 7.7. Sie stellt die Durchschnittswerte der Abweichungen getrennt nach Optimierungsstufen dar und unterscheidet nach Verbesserungen und Verschlechterungen. Die Werte wurden auf zwei Dezimalstellen nach dem Komma gerundet. Sie ermöglichen die Aussage, dass wenn bei Optimierungsstufe O3 eine Verbesserung auftritt, sie im Durchschnitt geringer ist, als bei den anderen Optimierungsstufen, und dass wenn eine Verschlechterung bei Optimierungsstufe O3 auftritt, ihr Wert im Durchschnitt größer ist als der der Verbesserung. Setzt man dies in Verbindung mit den Aussagen des Säulendiagramms in Abbildung 7.2,

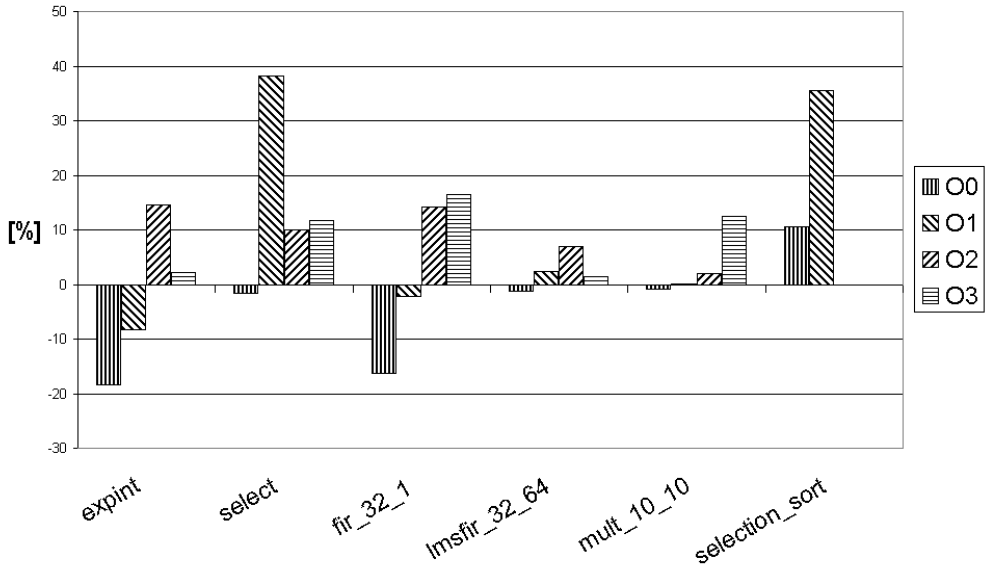

Abbildung 7.3: Verbesserungen des aufwändigeren WCET-Allokators gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 2,0$ in Prozent.

kann der Schluss gezogen werden, dass für Optimierungsstufe O3 der einfachere Ansatz zu besseren Ergebnissen führt als der aufwändigere Ansatz.

Insgesamt lässt sich aber keine eindeutige Aussage treffen, welcher der beiden Ansätze für alle Optimierungsstufen der bessere ist. Für die Optimierungsstufe O2 sind zwar die Verbesserungen im Durchschnitt größer als die Verschlechterungen, allerdings treten die Verbesserungen auch seltener auf. Bei der Optimierungsstufe O1 scheint der aufwändigere Ansatz eher zu Verbesserungen als zu Verschlechterungen zu neigen, für Optimierungsstufe O0 kann wiederum kein Ansatz als besser bezeichnet werden.

Auf Grund dieser Beobachtungen kann zunächst einmal festgehalten werden, dass für $c(x_B^k) = 1,5$ der aufwändigere Schätzungsansatz für die maximale Ausführungsdauer der Basisblöcke ohne Spillcode nicht prinzipiell besser als der einfache Ansatz ist. Da sowohl Verbesserungen als auch Verschlechterungen auftreten, hat das Schätzungsverfahren einen Einfluss auf die Güte der Registerallokation, der im Einzelfall sehr hoch sein kann, wie das Testprogramm *expint* zeigt. Dabei muss aber auch immer bedacht werden, dass eine kleine Änderungen der Allokation durch Schleifen verstärkt wird. In dem Testprogramm *expint* wird eine Schleife hundertmal ausgeführt, was die große Schwankung erklären könnte.

Für den Großteil der Programme, bei denen kein oder nur ein geringer Unterschied in der WCET festgestellt wurde, kann vermutet werden, dass sich die geschätzten Werte für die Basisblöcke kaum unterscheiden. Für diese Vermutung spricht auch, dass die Schätzungsansätze ähnlich sind.

Mit $c(x_B^k) = 2,0$ wurden die Untersuchungen der Abschätzungsverfahren wiederholt. In Abbildung 7.3 sind wieder exemplarisch die Verbesserungen für sechs Programme unter Berücksichtigung der Optimierungsstufe durch Säulen dargestellt. Im Vergleich zu der vorherigen Untersuchung sind nun mehr Abweichungen zwischen den zwei Schätzungsansätzen zu erkennen. Auch scheint die Tendenz zu Verbesserungen größer zu sein als zu

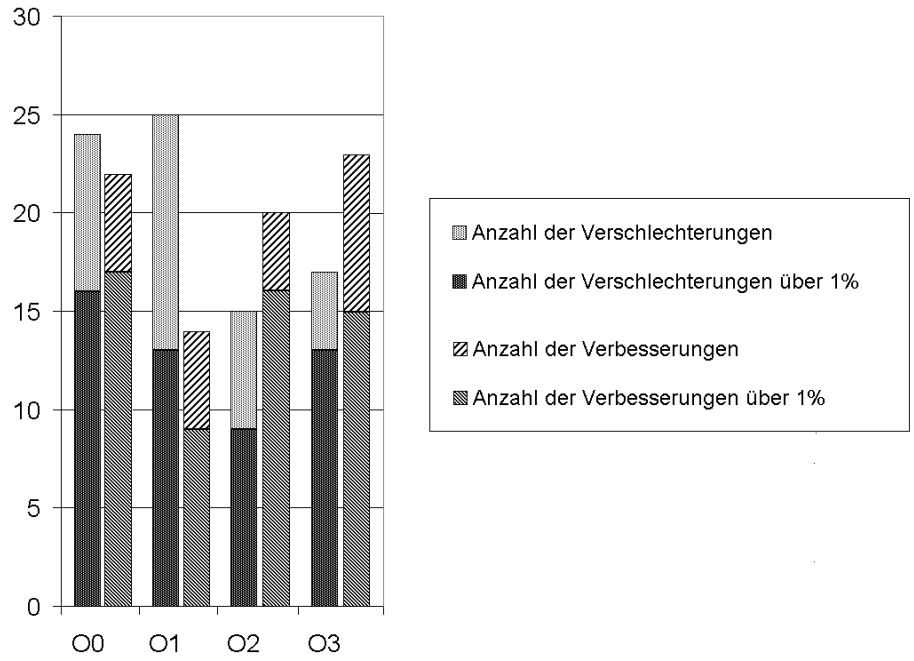

Abbildung 7.4: Anzahl der Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 2,0$.

durchschnittlicher Wert	O0	O1	O2	O3
- einer Verschlechterung in %	-3,69	-2,40	-1,58	-4,73
- einer Verbesserung in %	3,69	7,48	6,85	6,09

Tabelle 7.8: Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 2,0$.

Verschlechterungen. Dies wird sowohl von der Untersuchung der Häufigkeiten unterstützt, mit denen Verbesserungen und Verschlechterungen auftreten, als auch von der Untersuchung des durchschnittlichen Werts einer Veränderung gestützt.

Abbildung 7.4 zeigt ein Säulendiagramm, das getrennt nach Optimierungsstufen wiedergibt, wie häufig bei Verwendung des aufwändigeren Schätzungsansatzes Verbesserungen und Verschlechterungen gegenüber dem einfachen Ansatz bei den 86 untersuchten Programmen auftreten. Außer wenn die Optimierungsstufe O1 verwendet wurde, ist eine größere Anzahl an Verbesserungen über einem Prozent als Verschlechterungen über einem Prozent zu erkennen. Die durchschnittliche Qualität der Verbesserungen ist in Tabelle 7.8 aufgeführt. Es ist klar zu erkennen, dass ein Verbesserung im Durchschnitt nicht kleiner ausfällt als eine Verschlechterung. Besonders deutlich ist der Unterschied der Durchschnittswerte für die Optimierungsstufe O1. Obwohl also weniger Verbesserungen auftreten, führt der aufwändigere Schätzungsansatz für einen Einzelfall im Durchschnitt zu einer wesentlich besseren WCET als der einfachere Ansatz.

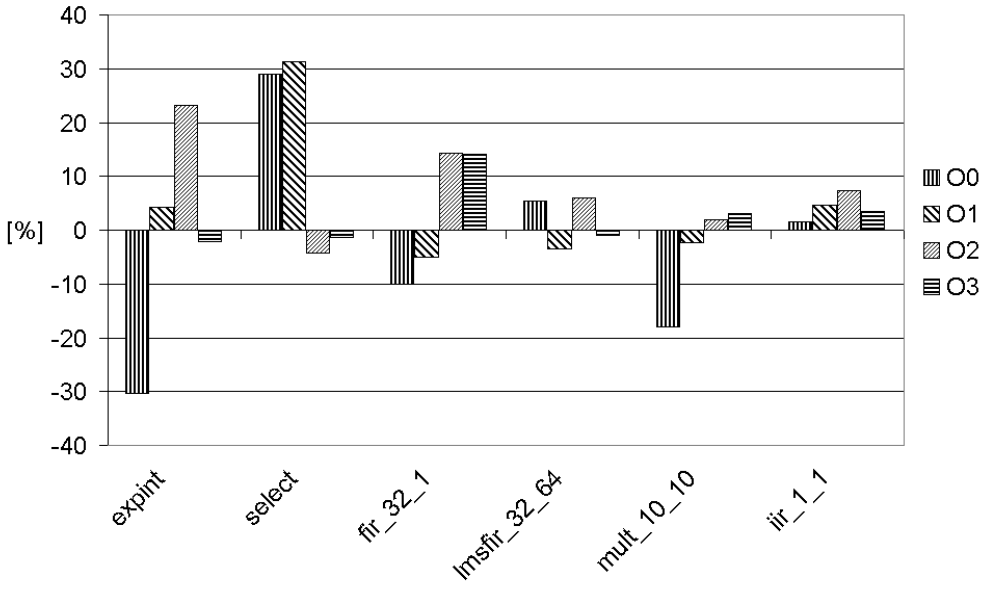

Abbildung 7.5: Verbesserungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$ in Prozent.

Aus diesen Beobachtungen kann der Schluss gezogen werden, dass für $c(x_B^k) = 2,0$ die Güte der Allokation stärker von dem Schätzungsverfahren für die maximale Ausführungsdauer der Basisblöcke ohne Spillcode abhängt als zuvor. Es spricht viel dafür, für $c(x_B^k) = 2,0$ das aufwändigere Schätzungsverfahren dem einfachen Verfahren vorzuziehen. Durch die Untersuchungen wurde zudem deutlich, dass die Werte für die Konstanten $c(x_B^k)$ und das Schätzungsverfahren aufeinander abgestimmt sein müssen. Aus ihnen errechnet sich schließlich die maximale Ausführungsdauer der Basisblöcke mit Spillcode, die die Zielfunktion der Registerallokation beeinflussen.

Für die folgenden Vergleiche mit dem Graphfärbungs- und dem Codegröße-optimierenden Allokator soll nun eine Wahl für das Schätzungsverfahren und den Wert für die Konstanten $c(x_B^k)$ getroffen werden. Um das Potenzial des in dieser Diplomarbeit erstellten Registerallokators beurteilen zu können, sollte die Wahl so getroffen werden, dass der Allokator möglichst gut ist. Da der Wert 1,5 für die Konstanten $c(x_B^k)$ für das einfache Schätzungsverfahren scheinbar die besten Ergebnisse erwarten ließ und der aufwändiger Schätzungsansatz mit $c(x_B^k) = 2,0$ insgesamt zu den meisten Verbesserungen und den wenigsten Verschlechterungen führt, werden nun diese beiden Allokatoren miteinander verglichen.

Abbildung 7.5 zeigt zunächst einmal exemplarisch die Ergebnisse für sechs Testprogramme. Dargestellt werden in diesem Säulendiagramm die relativen Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$. Auch hier wurden wieder die Optimierungsstufen des WCC berücksichtigt. Der Abbildung 7.5 ist schon zu entnehmen, dass es nicht eindeutig sein wird, welcher der beiden Allokatoren als besser anzusehen ist. Erneut treten sowohl Verbesserungen als auch Verschlechterungen auf und diese hängen auch noch mit der verwendeten Optimierungsstufe zusammen.

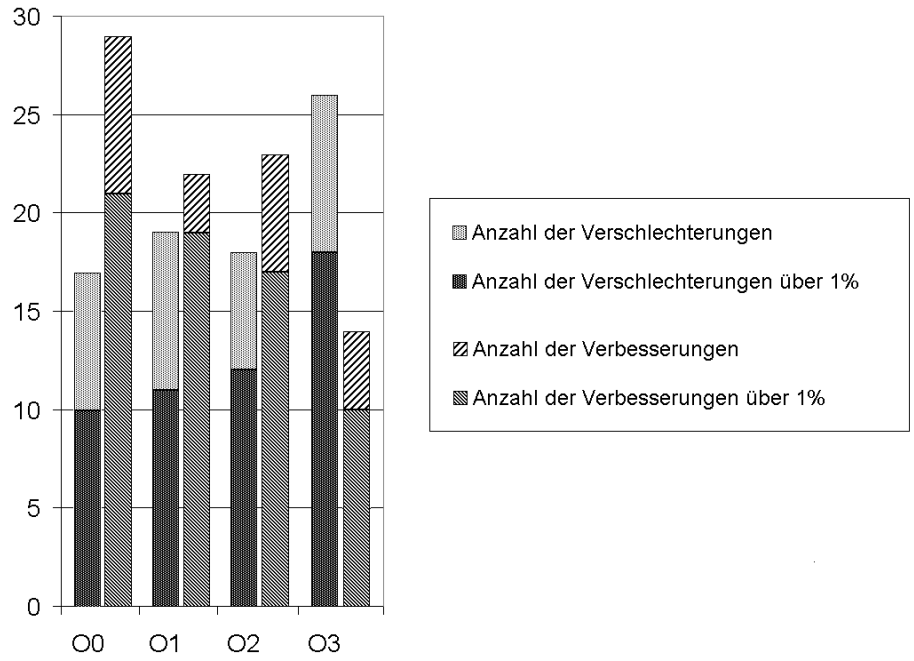

Abbildung 7.6: Anzahl der Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$.

Das Säulendiagramm in Abbildung 7.6 gibt die Anzahl der Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$ wieder. Damit kleine Abweichungen in der Güte vernachlässigt werden können, ist in Abbildung 7.6 zusätzlich die Anzahl der Verbesserungen und Verschlechterungen über 1% hervorgehoben. Aus der Darstellung geht hervor, dass die Anzahl der Verbesserungen größer ist als die Anzahl der Verschlechterungen, wenn nicht die Optimierungsstufe O3 verwendet wird. Werden nur Verbesserungen und Verschlechterungen über 1% betrachtet, verringert sich zwar dieser Abstand, dennoch existiert er weiterhin.

Für eine qualitative Beurteilung der Allokatoren sind in Tabelle 7.9 der Durchschnittswert einer Verschlechterung und einer Verbesserung für alle vier Optimierungsstufen aufgeführt. Es ist zu erkennen, dass wenn eine Verbesserung eintritt, ihr relativer Wert unabhängig von der Optimierungsstufe über dem durchschnittlichen Wert einer Verschlechterung liegt. Auch wenn die Anzahl der Verschlechterungen auf der Optimierungsstufe O3 fast doppelt so hoch ist wie die Anzahl der Verbesserungen, wird auf Grund der höheren Qualität der Verbesserungen der WCET-optimierende Allokator mit der aufwändigeren Abschätzung und dem Wert 2,0 für die Konstanten $c(x_B^k)$ in den nächsten Vergleichen verwendet.

Die durchgeführten Vergleiche der verschiedenen Kombinationen aus Schätzungsverfahren und Werten für die Kostanten $c(x_B^k)$ haben gezeigt, dass keine der untersuchten Kombinationen aus den Schätzungsverfahren und den Werten für die Konstanten als die beste bezeichnet werden kann. Daraus lässt sich folgern, dass das Ergebnis der Schätzung

durchschnittlicher Wert	O0	O1	O2	O3
- einer Verschlechterung in %	-5,52	-2,63	-2,58	-3,60
- einer Verbesserung in %	5,75	5,04	5,93	5,14

Tabelle 7.9: Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem einfachen WCET-Allokator mit $c(x_B^k) = 1,5$.

durchschnittlicher Wert	O0	O1	O2	O3
- einer Verschlechterung in %	-6,08	-7,18	-9,20	-18,73
- einer Verbesserung in %	17,34	17,88	21,97	20,76

Tabelle 7.10: Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem Graphfärbungsallokator.

und die Werte für die Ausführungsverlängerung durch eingefügte Spillintruktionen zumindest in einigen Fällen so ungenau sind, dass die WCET einer Funktion nicht exakt beschrieben und daher nicht minimiert wird.

7.1.3 Vergleich zwischen Graphfärbungs- und WCET-optimierenden Allokator

Für den Vergleich zwischen dem Graphfärbungsallokator und dem WCET-optimierenden Allokator wurden die Testprogramme aus der WCETBENCH einmal unter Nutzung des Graphfärbungsallokators und einmal unter Nutzung des WCET-optimierenden Allokators mit aufwändigerem Schätzungsansatz und $c(x_B^k) = 2,0$ übersetzt. Dies wurde für alle Optimierungsstufen wiederholt. Anschließend wurden für jedes Programm und jede Optimierungsstufe die Verbesserung des WCET-optimierenden Allokators gegenüber dem Graphfärbungsallokator nach Gleichung 7.1 berechnet. Natürlich wurde an Stelle der Ergebnisse des WCET-optimierenden Allokators mit einfachem Schätzungsansatz die WCET der Programme eingesetzt, für die der Graphfärbungsallokator die Registerallokation durchgeführt hat. Für den Vergleich der beiden Allokatoren wurden nur die Testprogramme berücksichtigt, für die der ILP-basierte WCET-optimierende Allokator eine Registerallokation ermitteln konnte.

Das Ergebnis des Vergleichs zwischen Graphfärbungs- und WCET-optimierenden Allokators fällt im Gegensatz zu den Vergleichen der WCET-optimierenden Allokatoren untereinander deutlicher aus. Abbildung 7.7, in der die Häufigkeiten der Verbesserungen und Verschlechterungen aufgeführt sind, zeigt, dass die Anzahl der Verbesserungen deutlich höher ist als die Anzahl der Verschlechterungen. Aus Tabelle 7.10 geht hervor, dass eine Verbesserung im Durchschnitt qualitativ höher ist als eine Verschlechterung. Der hohe Durschnittswert der Verschlechterungen bei Optimierungsstufe O3 geht zu einem großen Teil auf das Testprogramm *cjpeg_jpeg6b_wrbmp* zurück. In dem Säulendiagramm in Abbildung 7.8 sind die relativen Verbesserungen und Verschlechterungen für zwölf Programme dargestellt, die bei Anwendung der Optimierungsstufe O3 zu beobachten sind. Aus der Abbildung geht hervor, dass für das Programm *cjpeg_jpeg6b_wrbmp* eine sehr hohe Verschlechterung von über 70% entstanden ist. Würde dieses Programm bei der Berechnung des Durchschnittswerts einer Verschlechterung ausgelassen werden, würde sich

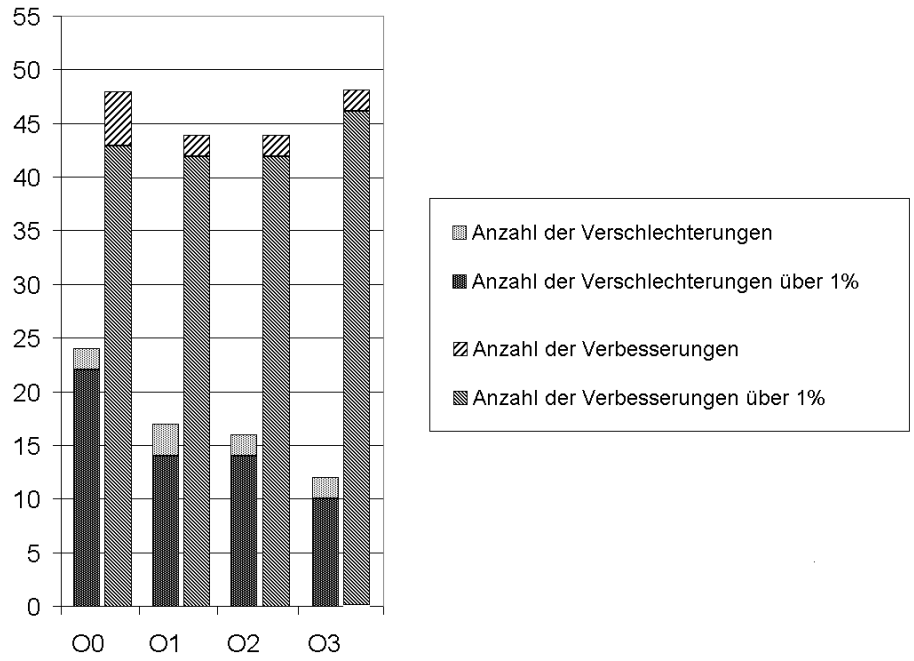

Abbildung 7.7: Anzahl der Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem Graphfärbungsallokator.

ein Durchschnittswert von immerhin noch -13,92 ergeben. Es treten also bei Optimierungsstufe O3 wenige aber hohe Verschlechterungen auf.

Eine außergewöhnlich hohe Verbesserung von über 70% konnte für das Programm *cjpeg_jpeg6b_transupp* für alle Optimierungsstufen erreicht werden. Würde dieser größte Wert bei der Berechnung des Durchschnittswertes einer Verbesserung bei Optimierungsstufe O3 weggelassen werden, würde der Durchschnittswert kaum absinken. Mit 19,70 wäre er immer noch hoch. Wird für alle Optimierungsstufen ein gemeinsamer Mittelwert aus allen Verbesserungen und Verschlechterungen gebildet, liegt dieser zwischen 9% bis 13%. Auch dies unterstreicht, dass die Verbesserungen durch den WCET-optimierenden Allokator die Verschlechterungen überwiegen.

Es kann also festgehalten werden, dass mit einer Registerallokation, die die WCET eines Programms berücksichtigt, erhebliche Verbesserungen gegenüber einem Graphfärbungsallokator erreicht werden können. Dies ist nicht verwunderlich, da der Graphfärbungsallokator keine gezielten Optimierungen bezüglich der WCET einer Funktion vornimmt. Zudem hat der ILP-basierte WCET-optimierende Allokator den Vorteil, dass er ein *live range splitting* vornimmt und damit Spillcode vermeiden kann. Die Graphfärbung führt dies nicht durch. Auch ist die Beschreibung des Registerallokationsproblem durch ein ILP wesentlich genauer als durch den Interferenzgraphen, den die Graphfärbung nutzt, und auch die Auswirkungen, die das Einfügen von Spillinstruktionen hat, werden stärker berücksichtigt. Während der ILP-basierte Allokator eine optimale Lösung für sein Modell des Registerallokationsproblems berechnet, werden für die Graphfärbung nur Heuristiken verwendet. Der WCET-optimierende Allokator hat also schon auf Grund des

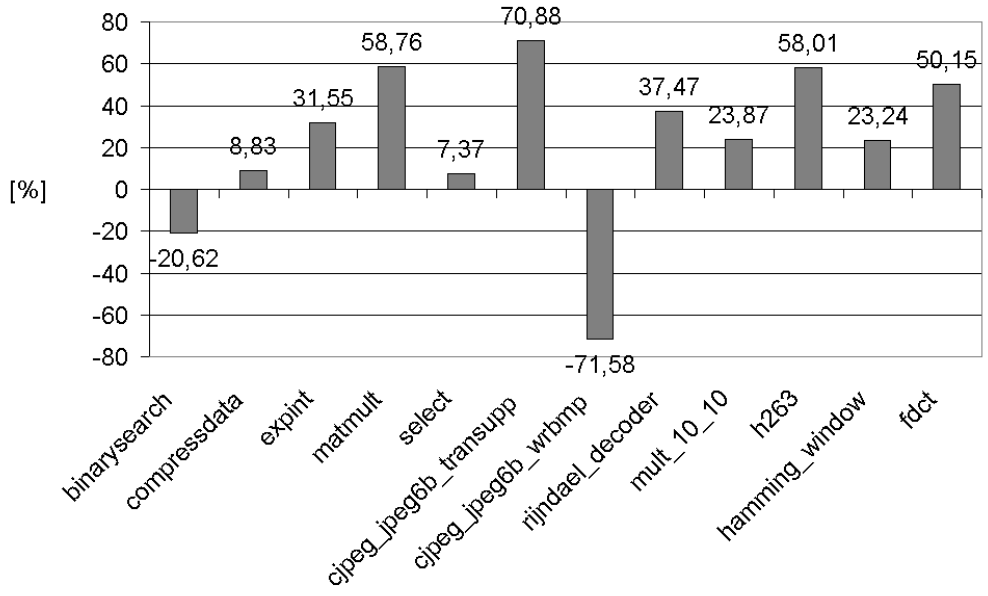

Abbildung 7.8: Relative Verbesserungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem Graphfärbungsallokator bei Optimierungsstufe O3.

Lösungsansatzes einen Vorteil gegenüber dem Graphfärbungsallokator. Deshalb wird in dem nächsten Abschnitt 7.1.4 der WCET-optimierende Allokator mit einem ebenfalls ILP-basierten Allokator verglichen.

Bei dem Vergleich zwischen dem Graphfärbungsallokator und dem WCET-optimierenden Allokator, muss festgestellt werden, dass nicht nur Verbesserungen sondern auch Verschlechterungen auftreten. Dies mag auf dem ersten Blick erstaunlich sein, da der WCET-optimierende Allokator doch die WCET gezielt zu minimieren versucht. Aus den Ergebnissen der Vergleiche bezüglich des Wertes der Konstanten $c(x_B^k)$ in Abschnitt 7.1.1 und des Schätzungsverfahrens im vorherigen Abschnitt 7.1.2 ging aber bereits hervor, dass die Zielfunktion des WCET-optimierenden Allokators die WCET einer Funktion nicht genau beschreibt. Es kann daher vorkommen, dass der Allokator von Basisblöcken, die auf dem WCEP liegen, annimmt, dass sie nicht auf dem WCEP liegen. Da es keinen Einfluss auf den Wert der Zielfunktion hat, Spillcode in Basisblöcke einzufügen, die sich in dem Modell nicht auf dem WCEP befinden, gibt es für den Allokator keinen Zwang in diese Basisblöcke möglichst wenig Spillcode einzufügen.

So konnte beobachtet werden, dass der Allokator auch redundanten Code wie in dem Beispiel in Abbildung 7.9 einfügt. Die Abbildung zeigt einen einfachen Kontrollflussgraphen mit vier Basisblöcken. In diesem Beispiel sei der WCEP in dem Modell der Pfad, dessen Kanten durch Linien mit größerer Linienbreite dargestellt sind. Der Basisblock B ist daher in diesem Beispiel nicht Teil des WCEP. In ihm werden die Inhalte der Register D[4] und D[6] gesichert und wiederhergestellt, obwohl sie in der Zwischenzeit nicht verwendet werden. Solange die Ausführungsdauer des Basisblocks dadurch nicht so stark steigt, dass der WCEP in dem Modell wechselt, hat dieser redundante Spillcode keinen Einfluss auf die Zielfunktion des ILP und wird daher von dem Allokator nicht vermieden.

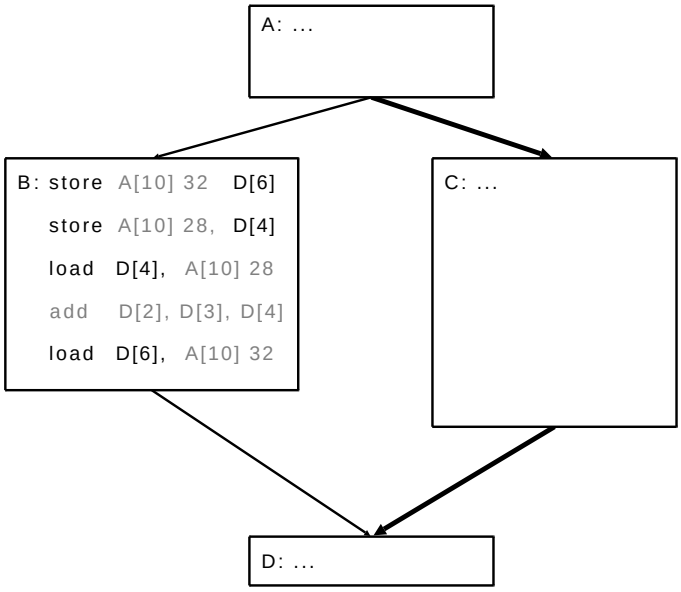

Abbildung 7.9: Kontrollflussgaph mit redundantem Spillcode.

durchschnittlicher Wert	O0	O1	O2	O3
- einer Verschlechterung in %	-6,09	-8,14	-7,35	-13,73
- einer Verbesserung in %	4,89	2,43	4,60	3,54

Tabelle 7.11: Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem Codegröße-optimierenden Allokator.

Befindet sich ein Basisblock, in den viel Spillcode eingefügt wurde, in Wirklichkeit aber auf dem WCEP, führt dies zu Verschlechterungen.

7.1.4 Vergleich zwischen Codegröße- und WCET-optimierenden Allokator

Der Vergleich des Codegröße- und des WCET-optimierenden Allokators erfolgte nach dem gleichen Schema, nach dem auch die Verbesserungen und Verschlechterungen in dem vorherigen Abschnitt ermittelt wurden. In Abbildung 7.10 sind in einem Säulendiagramm für zwölf Programme die Ergebnisse dieses Vergleichs dargestellt. Es zeigt die relativen Verbesserungen und Verschlechterungen des WCET-optimierenden Allokators gegenüber dem Codegröße-optimierenden Allokators bei Verwendung der Optimierungsstufe O3.

Aus der Abbildung 7.10 geht ein klarer Trend hervor. Verglichen mit dem Codegröße-optimierenden Allokator führt der WCET-optimierende Allokator nur zu kleinen Verbesserungen aber zu großen Verschlechterungen. Dies belegt auch Tabelle 7.11, in der die Durchschnittswerte von Verbesserungen und Verschlechterungen aufgeführt sind. Auch wenn die Anzahl der Verbesserungen und Verschlechterungen, die in Abbildung 7.11 nach Optimierungsstufen unterschieden dargestellt sind, ungefähr gleich sind, resultiert aus der Verwendung des WCET-optimierenden Allokator an Stelle des Codegröße-optimierenden Allokator insgesamt eine Verschlechterung.

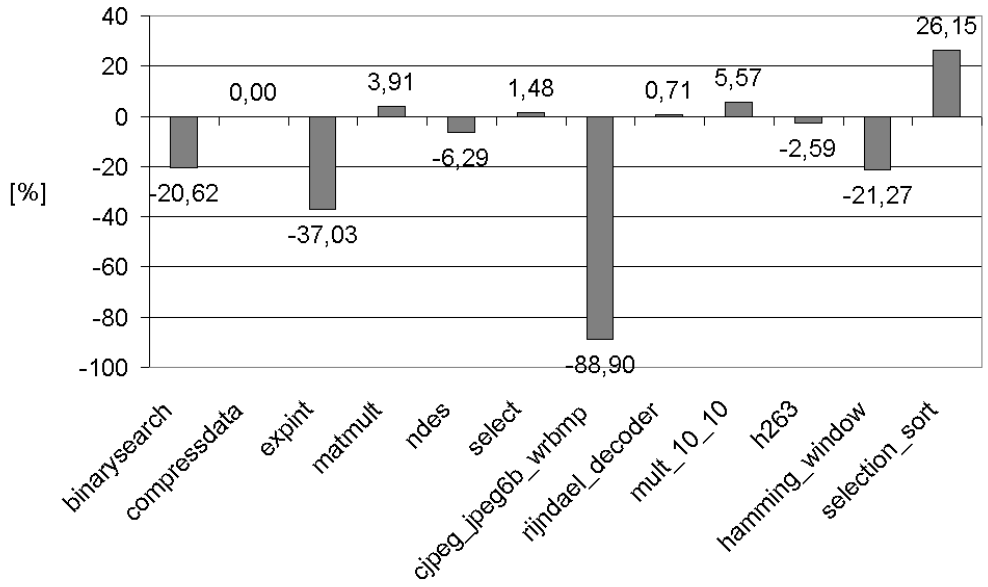

Abbildung 7.10: Relative Verbesserungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem Codegröße-optimierenden Allokator bei Optimierungsstufe O3.

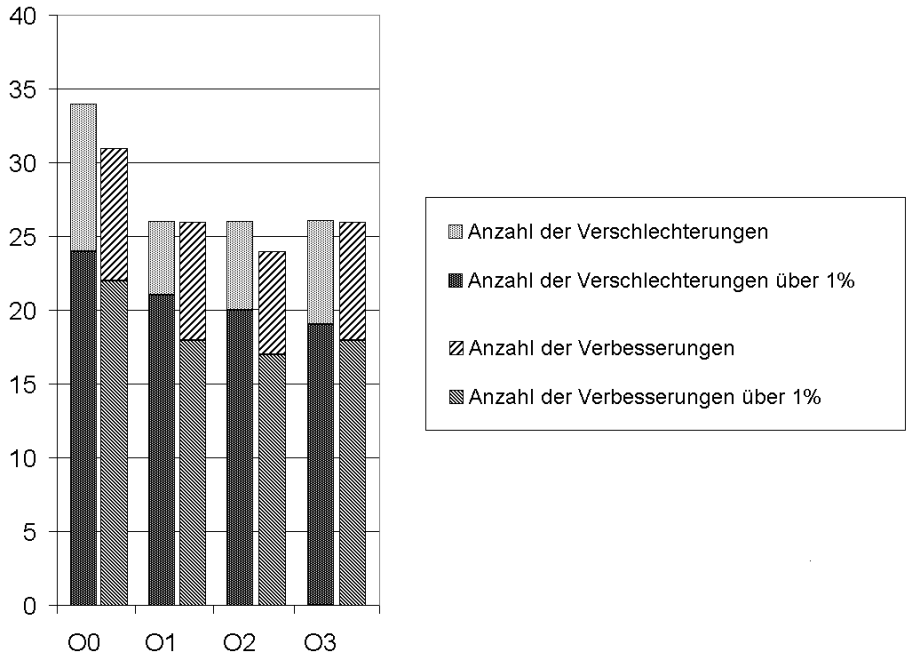

Abbildung 7.11: Anzahl der Verbesserungen und Verschlechterungen des aufwändigeren WCET-Allokators mit $c(x_B^k) = 2,0$ gegenüber dem Codegröße-optimierenden Allokator.

durchschnittlicher Wert	O0	O1	O2	O3
- einer Verschlechterung in %	-5,70	-1,84	-10,15	-9,52
- einer Verbesserung in %	18,53	20,25	22,59	20,28

Tabelle 7.12: Verbesserungen und Verschlechterungen des Codegröße-optimierenden Allokators gegenüber dem Graphfärbungsallokator.

Als maßgebliche Gründe für diese Verschlechterungen können wieder dieselben zwei Gründe wie bei dem Vergleich mit dem Graphfärbungsallokator angeführt werden. Zum einen sind der Wert der Konsanten $c(x_B^k)$ und die Schätzung der Ausführungsdauer von Basisblöcken zu ungenau. Daher stimmt der WCEP in dem Modell des WCET-optimierenden Allokators wahrscheinlich nicht mit dem tatsächlichen WCEP überein. Zum anderen wird von dem WCET-optimierenden Allokator unnötiger Spillcode in Basisblöcke eingefügt, die sich in seinem Modell nicht auf dem WCEP befinden. Der Codegröße-optimierende Allokator vermeidet solchen redundanten Spillcode, da dieser zu einer größeren Codegröße führen würde.

Es ist zu vermuten, dass der WCET-optimierende Allokator gegenüber dem Graphfärbungsallokator nur deshalb überwiegend zu Verbesserungen führte, weil sein Lösungsverfahren dem des auf Heuritiken beruhenden Graphfärbungsallokators bezüglich der Güte der Allokation überlegen ist. Dies soll in einem Vergleich zwischen dem Graphfärbungs- und dem Codegröße-optimierenden Allokator nochmal untersucht werden.

7.1.5 Vergleich zwischen Graphfärbungs- und Codegröße-optimierenden Allokator

Für den Vergleich zwischen Graphfärbungs- und Codegröße-optimierenden Allokator wurden die Daten, die aus den Testläufen für die vorherigen Vergleiche ermittelt wurden, neu ausgewertet, und diesmal die relativen Verbesserungen und Verschlechterungen des Codegröße-optimierenden Allokators gegenüber dem Graphfärbungsallokator berechnet.

In Abbildung 7.12 sind exemplarisch für zwölf Programme die Ergebnisse der Berechnung für Optimierungsstufe O3 in einem Säulendiagramm dargestellt. Es ist zu erkennen, dass der Codegröße-optimierende Allokator fast ausschließlich zu Verbesserungen führt. Das Testprogramm *selection_sort* stellt in diesem Fall eine Ausnahme dar. In Abbildung 7.13 sind die Häufigkeiten dargestellt, mit denen Verbesserungen und Verschlechterungen auftreten. Auch dieser Abbildung ist zu entnehmen, dass überwiegend Verbesserungen auftreten. Auch der Betrag der relativen Verbesserungen ist mit durchschnittlich 20% hoch, wie aus der Tabelle 7.12 hervorgeht. In ihr sind die Durchschnittswerte der relativen Verbesserungen und Verschlechterungen in Prozent aufgeführt, die auf zwei Hundertstel gerundet sind. Würde das Programm *selection_sort* bei der Berechnung der Durchschnittswerte außer Acht gelassen, würde der durchschnittliche Wert einer Verschlechterung für Optimierungsstufe O3 auf -0,96 fallen und für Optimierungsstufe O2 auf -5,47 zurückgehen. Für die Optimierungsstufen O0 und O1 ergab sich für dieses Programm zwischen den beiden Allokationen kein Unterschied. Wird der Spitzenwert bei den Verbesserungen ausgelassen, der 20 Prozentpunkte über der zweitbesten Verbesserung liegt und bei dem Testprogramm *cjpeg_jpeg6b_transupp* beobachtet werden kann, sinken die Durchschnittswerte der Verbesserungen nur um 1-2%.

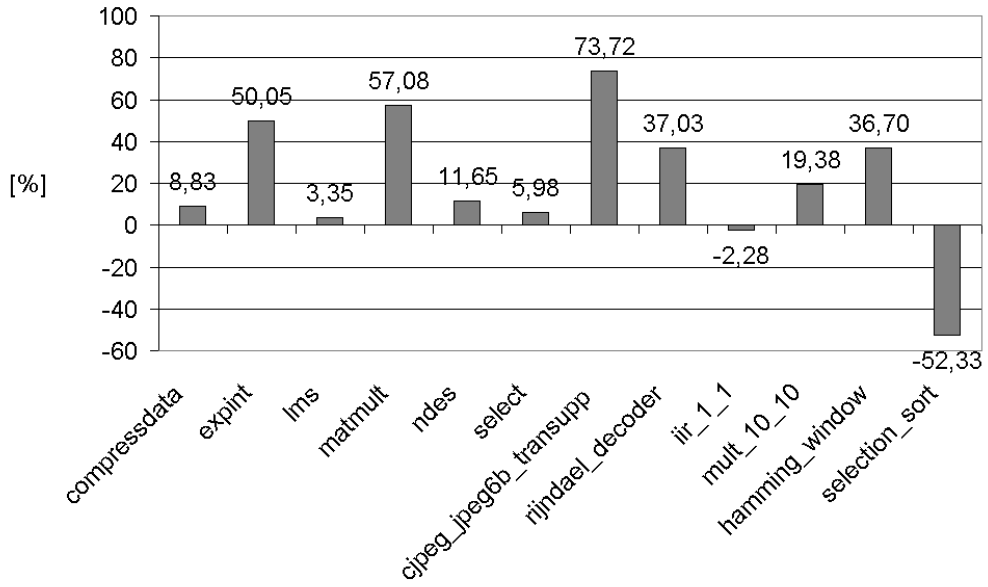

Abbildung 7.12: Relative Verbesserungen des Codegröße-optimierenden Allokators gegenüber dem Graphfärbungsallokator bei Optimierungsstufe O3.

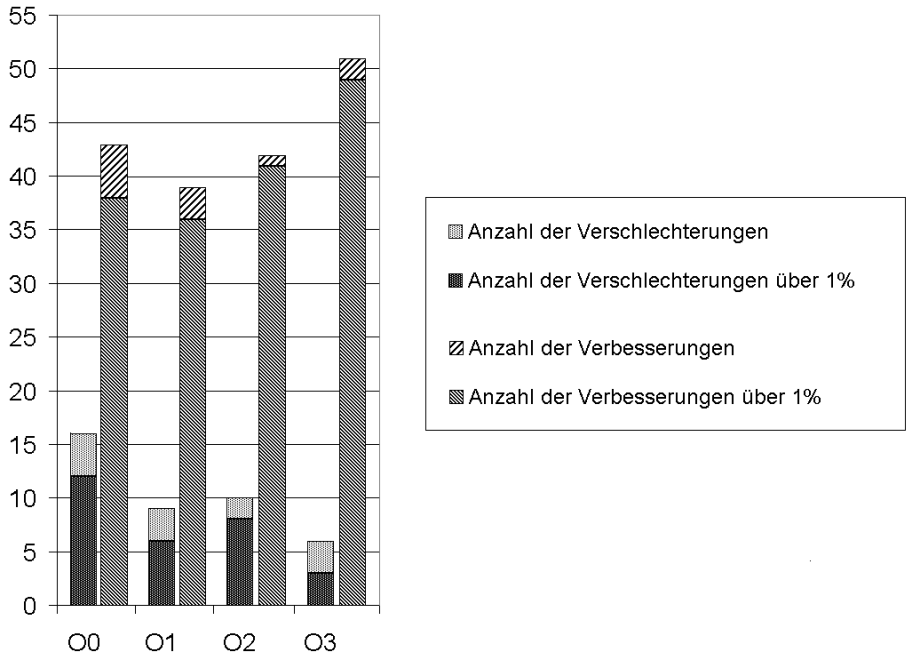

Abbildung 7.13: Anzahl der Verbesserungen und Verschlechterungen des Codegröße--optimierenden Allokators gegenüber dem Graphfärbungsallokator.

Aus diesem Vergleich wird also deutlich, dass wie zu erwarten die ILP-basierte Verfahrensweise zu Programmen mit geringerer WCET führt als das Graphfärbungsverfahren. Die Verringerung der Anzahl der Spillinstruktionen reicht in diesem Fall schon aus, um die WCET eines Programms deutlich zu verkleinern.

7.2 Laufzeitvergleiche

Für die Laufzeitvergleiche werden die Laufzeiten des Codegröße- und des WCET-optimierenden Allokators mit der Laufzeit des Graphfärbungsallokators verglichen. Die Laufzeit des Graphfärbungsallokators wird also als Bezugspunkt für die Bewertung der Laufzeiten gewählt. Für jedes Testprogramm werden die Laufzeiten der ILP-basierten Allokatoren durch die Laufzeit des Graphfärbungsallokator dividiert. In Abbildung 7.14 sind für sechs Testprogramme die Laufzeiten als Vielfache der Laufzeit des Graphfärbungsallokators in ein Säulendiagramm eingetragen. Da der Testfall *rijndael_decoder* eine Ausnahme darstellt, sollen zunächst nur die Verhältnisse für die übrigen fünf Programme betrachtet werden. Obwohl diese Auswahl im Vergleich zu den 86 Testprogrammen, die für diese Untersuchungen genutzt wurden, klein ist und nur die Allokationszeiten unter Verwendung der Optimierungsstufe O3 vergleicht, reicht sie aus, um das grundlegende Verhältnis zwischen den Laufzeiten der Allokatoren wiederzugeben.

Es kann festgestellt werden, dass die ILP-basierten Verfahren in der Regel eine höhere Laufzeit aufweisen als das Graphfärbungsverfahren. Der Codegröße-optimierende Allokator ist zum Teil kaum langsamer als der Graphfärbungsallokator, in anderen Fällen benötigt er aber mehr als doppelt so lange. Der WCET-optimierende Allokator benötigt sogar mehr als sechsmal so lange wie der Graphfärbungsallokator. Ein einzelner Extremfall, der nicht in der Abbildung 7.14 aufgeführt ist, da sonst die Skala zu sehr erweitert werden müsste, kann für das Programm *cjpeg_jpeg6b_wrbmp* beobachtet werden. Während der Graphfärbungsallokator eine Registerallokation in weniger als einer halben Minute bestimmt, benötigt der WCET-optimierende Allokator fast eine Stunde. Das Programm *rijndael_decoder* zeigt aber auch, dass in Ausnahmefällen die ILP-basierten Allokatoren sogar schneller als der Graphfärbungsallokator sein können.

Das schnellere Laufzeitverhalten der ILP-basierten Allokatoren lässt sich damit erklären, dass der Graphfärbungsallokator jedesmal den Interferenzgraphen erstellt und färbt, wenn in der *select*-Phase entschieden wurde, dass ein virtuelles Register in den Speicher ausgelagert und Spillcode eingefügt werden muss (siehe Abschnitt 2.2.3.1). Das Programm *rijndael_decoder* besteht aus sechs Funktionen, wovon eine Funktion ungefähr 1900 Instruktionen enthält. Das wiederholte Erstellen und Färben des Graphen scheint länger zu dauern, als eine Registerallokation mittels eines ILP zu bestimmen.

In fast allen Fällen benötigen die ILP-basierten Allokatoren aber länger als das Graphfärbungsverfahren. Dies kann durch die Komplexität der ILP erklärt werden. Während der Graphfärbungsallokator Heuristiken einsetzt, die in der Regel eine polynomielle Laufzeit aufweisen, lösen die ILP-basierten Allokatoren die ILP optimal, wenn das Zeitlimit von 600 Sekunden ausreicht. Dass der WCET-optimierende Allokator nochmals sehr viel mehr Laufzeit als der Codegröße-optimierende Allokator benötigt, liegt zum einem daran, dass die ILP für die Modellierung der WCET einer Funktion mehr Entscheidungsvariablen und Nebenbedingungen enthalten als die ILP, die zum Minimieren der Codegröße formuliert werden. Zum anderen führt der WCET-optimierende Allokator als

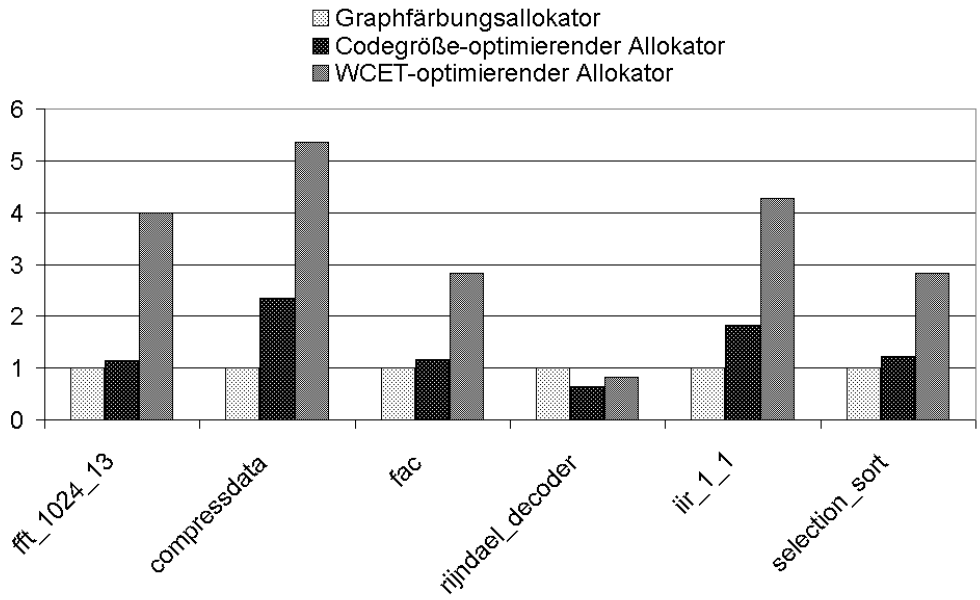

Abbildung 7.14: Laufzeitvergleich der Registerallokationen. Die Laufzeiten sind als Vielfache der Laufzeit des Graphfärbungsallokators angegeben.

einziger eine WCET-Analyse durch. Sie ist der Grund, dass die Allokation für das Programm *cjpeg_jpeg6b_wrbmp* so lange dauert. Diese Analyse benötigt 55 Minuten, während das Aufstellen und Lösen des ILP nur wenige Sekunden dauert. Da die WCET-Analyse nach der Registerallokation, mit der abschließend die WCET des Maschinenprogramms bestimmt wird, nur wenige Sekunden dauert, muss die lange Dauer der WCET-Analyse in der Registerallokation ihren Grund in der Pre-Allokation haben. Durch das Einsetzen eines naiven Allokators für die Pre-Allokation wird hier viel Spillcode eingefügt, der eine längere WCET-Analyse verursacht.

Bei diesem Laufzeitvergleich wurden die Fälle ganz außer Acht gelassen, bei denen innerhalb von 600 Sekunden keine optimale Lösung für ein ILP gefunden wird. Würden diese Fälle mit in die Betrachtung einbezogen werden, würde der Vergleich zwischen den ILP-basierten Allokatoren und dem Graphfärbungsallokator zu einem noch deutlicheren Ergebnis führen. Für einen Großteil der Testprogramme in der WCETBENCH konnte aber eine Registerallokation mit den ILP-basierten Verfahren innerhalb weniger Sekunden durchgeführt werden. Dies wird jedoch durch die geringe Größe dieser Testprogramme begünstigt.

7.3 Größe der ILP

Für die Größe der ILP wurde in Abschnitt 5.4 eine Abschätzung für die Anzahl der Nebenbedingungen angegeben. In diesem Abschnitt soll untersucht werden, wie genau diese Abschätzung für die ILP zutrifft, die für die Registerallokation bei den Testprogrammen der WCETBENCH erstellt wurden. In Abbildung 7.15 sind dazu in einem Koordinatensystem die Anzahl der Nebenbedingungen der ILP des Codegröße-optimierenden Allo-

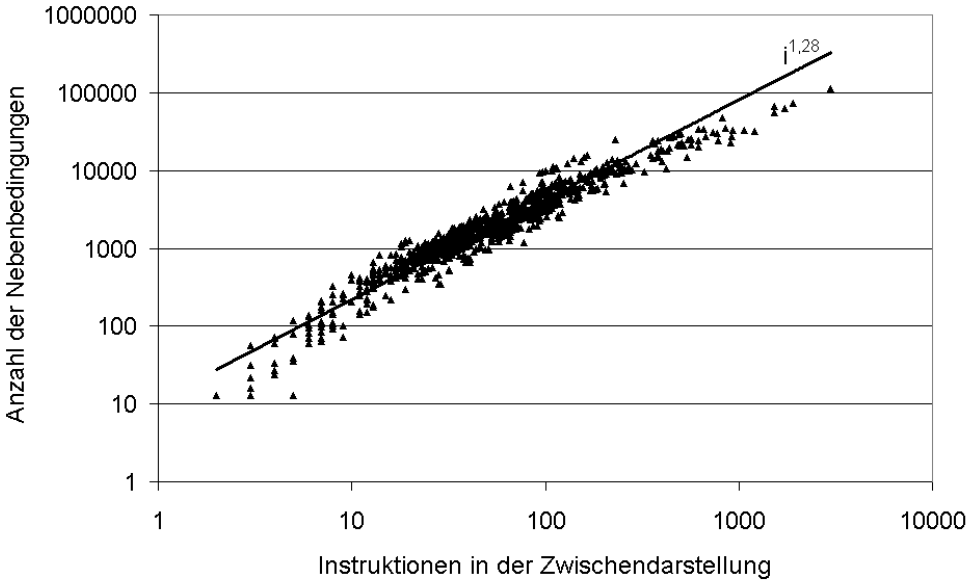

Abbildung 7.15: Anzahl der Nebenbedingung der ILP des Codegröße-optimierenden Allokators bezogen auf die Anzahl der Instruktionen in der Zwischendarstellung.

kators gegen die Anzahl der Instruktionen in der Zwischendarstellung aufgetragen. Für beide Koordinaten-Achsen wurde eine logarithmische Skalierung gewählt. In das Koordintatensystem ist zusätzlich eine polynomielle Funktion aus $\Omega(i^{1,28})$ eingetragen.

Es ist zu erkennen, dass sich die Messpunkte um die eingetragene Funktion gruppieren. Die Abschätzung von $\mathcal{O}(n^{1,3})$, die Goodwin und Wilken [GW96] für ihren ORA-Ansatz angeben, kann also bestätigt wrden.

In Abbildung 7.16 wurde dies für die ILP des WCET-optimierenden Allokators wiederholt. Da im Vergleich zu der Anzahl Nebenbedingungen des ORA-Ansatzes, die Anzahl der Nebenbedingungen gering ist, die für die WCET-Modellierung verwendet werden, sind die Punkte in dem Koordinatensystem im Vergleich zum Koordinatensystem in Abbildung 7.15 nur geringfügig verschoben. Auch hier zeigt sich, dass die Schätzung aus Abschnitt 5.4 zutrifft.

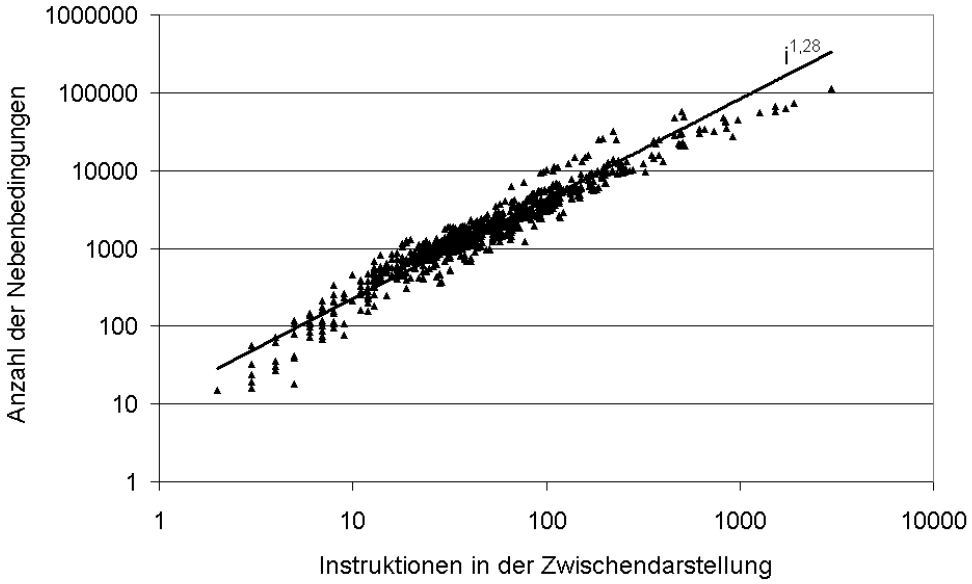

Abbildung 7.16: Anzahl der Nebenbedingung der ILP des WCET-optimierenden Allokators bezogen auf die Anzahl der Instruktionen in der Zwischendarstellung.

Kapitel 8

Zusammenfassung und Ausblick

Diese Diplomarbeit ist eine der ersten Arbeiten, die das Problem der Registerallokation unter dem Gesichtspunkt der WCET eines Programms betrachtet. Ihr Ziel war die Erstellung eines ILP-basierten Registerallokators, der WCET-Daten nutzt, um die WCET eines Programms zu minimieren. In Abschnitt 8.1 sollen nun die Ergebnisse dieser Diplomarbeit noch einmal zusammengefasst werden. In Abschnitt 8.2 wird ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben.

8.1 Zusammenfassung

Zu Beginn dieser Diplomarbeit wurden in Kapitel 2 das Problem der Registerallokation beschrieben und darauf hingewiesen, dass es sich dabei um ein NP-schweres Problem handelt. Es wurden verschiedene Lösungsverfahren vorgestellt, doch keines der bislang bekannten Verfahren optimiert gezielt die WCET eines Programms. In diesem Kapitel wurden ebenfalls die Schwierigkeiten beschrieben, die bei Optimierungen der WCET auftreten. Für diese Diplomarbeit musste besonders berücksichtigt werden, dass nur die Optimierung von Basisblöcken auf dem WCEP zu einer Verringerung der WCET führt, der WCEP während der Optimierung aber ständig wechseln kann. Daraus ergab sich die Notwendigkeit, dass während der Bestimmung einer Registerallokation der Einfluss berücksichtigt werden muss, den Entscheidungen über das Einfügen von Spillinstruktionen auf den WCEP haben.

Die in dieser Diplomarbeit erstellte Registerallokation wird innerhalb des WCC ausgeführt. Dieser wurde in Kapitel 3 vorgestellt. Er übersetzt C-Programme in Maschinenprogramme für den Infineon TC1796. Eine Registerallokation in dem WCC muss vor allem die Eigenschaften des Registersatzes dieser Zielarchitektur berücksichtigen. So müssen für den TC1796 zwischen Daten- und Adressregister unterschieden, und die Verwendung erweiteter Register unterstützt werden.

Für die Bestimmung einer Registerallokation sollte in dieser Diplomarbeit ein ILP-basierter Ansatz gewählt werden. Mit einem ILP kann das Registerallokationsproblem sehr genau beschrieben und auf einfache Weise die irreguläre Struktur des Registersatzes des TC1796 berücksichtigt werden. Die Modellierung des Registerallokationsproblems erfolgte in dieser Diplomarbeit nach dem ORA-Ansatz von Goodwin und Wilken [GW96]. Ihr Modellierungsprinzip wurde in Kapitel 4 vorgestellt.

In Kapitel 5 wurde aufgezeigt, wie das ILP ergänzt werden kann, damit die WCET eines Programms durch die Registerallokation minimiert wird. Dazu wurde die WCET des Programms durch weitere Entscheidungsvariablen und Nebenbedingungen modelliert. In Abschnitt 5.2 wurde zunächst die WCET-Modellierung von Suhendra et al. [SMRC05]

vorgestellt. Diese nimmt in einigen Fällen Vereinfachungen vor, die zu einem Informationsverlust führen können. Aus diesem Grund wurde eine genauere Modellierungstechnik entwickelt, die in Abschnitt 5.3 beschrieben wurde. Durch die Modellierung der WCET einer Funktion in dem ILP wird erreicht, dass bei der Lösung des ILP automatisch der Einfluss der Spillinstruktionen auf den WCEP berücksichtigt wird.

Für diese Modellierung werden die maximalen Ausführungszeiten von Basisblöcken ohne Spillcode benötigt. In Kapitel 6 wird begründet, dass diese aber vor einer Registerallokation nicht durch eine WCET-Analyse ermittelt werden können. Es muss daher von dem Allokator erst eine Pre-Allokation durchgeführt werden. Diese erzeugt eine Zwischendarstellung der Funktion, für die eine WCET-Analyse mit aiT ausgeführt werden kann (siehe Abschnitt 3.4). Allerdings enthält die Zwischendarstellung Spillcode. Die Ergebnisse der WCET-Analyse können deshalb nicht direkt genutzt werden. In Kapitel 6 werden daher zwei Ansätze vorgestellt, mit denen in dieser Diplomarbeit Ausführungszeiten für Basisblöcke ohne Spillcode basierend auf den Ergebnissen der WCET-Analyse geschätzt werden.

In Kapitel 7 wurde die Ergebnisse aus Vergleichen zwischen dem in dieser Diplomarbeit erstellten WCET-optimierenden Allokator, einem Graphfärbungsallokator sowie einem Codegröße-optimierenden Allokator vorgestellt. Der Vergleich zwischen dem Graphfärbungsallokator und den ILP-basierten Allokatoren zeigte, dass mit der Registerallokation die WCET eines Programms stark beeinflusst werden kann. Durch eine geschickte Registerallokation können im Durchschnitt Verbesserungen von über 20% erreicht werden. Durch die Vergleiche wurde zudem deutlich, dass für den gewählten ILP-basierten Ansatz die größte Schwierigkeit in der Bestimmung der WCET-Daten besteht. Für die gezielte Minimierung der WCET einer Funktion ist eine genaue Modellierung erforderlich. Die Werte, die für die Modellierung verwendet werden, können aber nicht genau genug betimmt werden.

Das Ziel dieser Diplomarbeit, einen ILP-basierten Registerallokator zu erstellen, wurde durch die Implementierung des ORA-Ansatzes von Goodwin und Wilken [GW96] erreicht. Das Modell wurde so angepasst, dass mit ihm eine Registerallokation für die Zielarchitektur des WCC durchgeführt werden kann. Somit wurde auch das Ziel erreicht, eine Registerallokation zu erstellen, die innerhalb des WCC ausgeführt werden kann. Für die Nutzung von WCET-Daten wurde in dieser Diplomarbeit eine Erweiterung des ILP vorgeschlagen, die die WCET einer Funktion modelliert. Im Vergleich zu dem bisher in dem WCC verwendeten Graphfärbungsallokator konnte so in vielen Fällen deutliche Verbesserungen erreicht werden. Somit ist auch dass letzte der drei Ziele erreicht worden. Der Vergleich mit dem Codegröße-optimierenden Allokator hat aber auch gezeigt, dass noch Möglichkeiten zur Verbesserung bestehen. Diese sollen in dem folgenden Abschnitt genannt werden.

8.2 Ausblick

In dieser Diplomarbeit wurde ein ILP-basierter Registerallokator entwickelt und implementiert, der das Ziel verfolgt, die WCET einer Funktion zu minimieren. Die Untersuchungen des erstellten Registerallokators in Kapitel 7 haben gezeigt, dass er die WCET eines Programms aber nicht so stark verringert, wie es von einem WCET-optimierenden Allokator hätte erwartet werden können. Hierfür wurden mehrere Gründe angeführt.

Um den Einfluss von Spillinstruktionen auf die WCET berücksichtigen zu können, modelliert der Registerallokator die WCET der Funktion in dem ILP, das das Registerallokationsproblem beschreibt. Dazu benötigt er Konstanten, die die Ausführungsdauer von Basisblöcken ohne Spillcode angegeben, und Konstanten $c(x_B^k)$, die die Verlängerung der Ausführungsdauer durch eingefügte Spillinstruktionen wiedergeben. Während letztere unabhängig von der Eingabe festgelegt sind, werden die Konstanten für die Ausführungsdauer der Basisblöcke ohne Spillcode durch ein Schätzungsverfahren bestimmt. Grundlage der Schätzung ist die Ausführungsdauer der Basisblöcke mit Spillcode. Bei der Untersuchung der Auswirkungen verschiedener Werte für die Konstanten $c(x_B^k)$ in Abschnitt 7.1.1 und der Schätzungsansätze in Abschnitt 7.1.2 wurde deutlich, dass die Konstanten zu ungenau sind, als dass die WCET einer Funktion genau genug modelliert werden könnte und der WCEP in dem Modell des Registerallokators mit dem tatsächlichen WCEP übereinstimmt.

Damit bessere Ergebnisse erzielt werden können, besteht daher der Bedarf, bessere Werte für die Konstanten zu ermitteln und das Schätzungsverfahren zu verbessern. Für alle Konstanten $c(x_B^k)$ wurden in dieser Diplomarbeit der gleiche Wert gewählt. Für eine genauere Beschreibung der Verlängerung der Ausführungsdauer durch eingefügte Spillinstruktionen, könnte der Wert aber auch abhängig von der Art der Spillinstruktion gewählt werden. Spillinstruktionen, die unmittelbar auf Instruktionen folgen, die Operationen auf Datenregister beschreiben, und in dem TriCore in der *IP*-Pipeline verarbeitet werden, können von der *Load/Store*-Pipeline parallel zu der vorherigen Instruktion ausgeführt werden (siehe Abschnitt 3.5.3). Durch die parallele Ausführung wird die Ausführungsdauer des Programms weniger verlängert als bei einer sequentiellen Ausführung.

Die Wahrscheinlichkeit, dass eine Spillinstruktion die auf eine Instruktion folgt, die in der *IP*-Pipeline verarbeitet wird, eine Spillinstruktion ist, die ein Register sichert, dürfte größer sein, als die Wahrscheinlichkeit, dass es sich dabei um eine Spillinstruktion handelt, die ein virtuelles Register wiederherstellt. Diese werden sehr häufig unmittelbar vor einer Instruktion eingefügt, um das virtuelle Register zu laden, das von der Instruktion genutzt wird. Spillinstruktionen, die ein Register sichern, werden oft unmittelbar nach einer Instruktion eingefügt, die ein virtuelles Register definiert. Sie befindet sich somit genau an der Stelle, an der sie parallel ausgeführt werden kann. Spillinstruktionen, die ein virtuelles Register wiederherstellen und vor Instruktionen eingefügt werden, können nur dann parallel ausgeführt werden, wenn sich vor ihnen nicht bereits eine andere Instruktion befindet, die von der *Load/Store*-Pipeline verarbeitet wird. Auf Grund dieser Annahme könnten für die Konstanten $c(x_B^k)$, die die Verlängerung der Ausführungszeit von Spillinstruktionen beschreiben, die ein virtuelles Register sichern, ein kleinerer Wert gewählt werden als für die Konstanten, die sich auf Spillinstruktionen beziehen, die ein Register wiederherstellen. Denkbar wäre auch, für jeden Basisblock den Unterschied zwischen der WCET eines Basisblocks mit Spillcode und der geschätzten WCET eines Basisblocks ohne Spillcode bei der Wahl der Werte mit einzubeziehen, so dass die Werte der Konstanten auch von dem Basisblock abhängen, auf den sie sich beziehen.

Aus dem Vergleich mit dem Codegröße-optimierenden Allokator in Abschnitt 7.1.4 geht hervor, dass bei der Verwendung des WCET-optimierenden Allokators es fast genauso viele Verschlechterungen wie Verbesserungen gibt wie bei der Verwendung des Codegröße-optimierenden Allokators. Die Verschlechterungen sind jedoch im Durchschnitt qualitativ höher als die Verbesserungen. Der Grund hierfür ist, dass der WCET-optimierende Al-

lokator nur in den Basisblöcken Spillcode vermeidet, die in seinem Modell der Funktion auf dem WCEP liegen. In Basisblöcken, die nicht auf dem WCEP liegen, kann er so viel Spillcode einfügen, bis sich der WCEP auf Grund der größeren Ausführungsdauer des Basisblocks ändern würde. Weicht der WCEP in dem Modell von dem tatsächlichen WCEP ab, so dass sich die Basisblöcke, in die viel Spillcode eingefügt wurde, auf dem tatsächlichen WCEP befinden, kann dies zu erheblichen Verschlechterungen führen. Die Zielfunktion des WCET-optimierenden Allokators könnte daher so erweitert werden, dass sie nicht ausschließlich die WCET einer Funktion modelliert, sondern auch ein weiteres Kriterium mit einbezieht, um redundanten Spillcode zu vermeiden. Als ein solches Kriterium könnte z. B. die Codegröße der Funktion verwendet werden. Es dürfte aber nur mit einem sehr geringen Gewicht in die Zielfunktion einfließen, damit die Minimierung der WCET weiterhin das primäre Optimierungsziel bleibt. Der Einbezug der Codegröße würde vermeiden, dass bei der Abweichung des WCEP in dem Modell des Registerallokators von dem tatsächlichen WCEP allzu große Verschlechterungen entstehen.

Die Modellierung der WCET könnte auch noch dadurch verbessert werden, dass nicht nur die Iterationshäufigkeiten von Schleifen berücksichtigt werden, sondern auch weitere *flow facts* (siehe Abschnitt 3.3). So könnte z. B. für Pfade in dem Kontrollflussgraphen, die *infeasible paths* sind, berücksichtigt werden, dass sie nicht der WCEP sein können. Zur Zeit ist nicht ausgeschlossen, dass der WCET-optimierende Registerallokator die maximale Ausführungsdauer eines *infeasible paths* minimiert.

Als weiterer Nachteil des WCET-optimierenden Allokators hat sich seine Laufzeit erwiesen, die zum Teil fünfmal so hoch ist wie die Laufzeit eines Graphfärbungsallokators. Maßgeblich bestimmt wird die Laufzeit durch die Lösungsdauer des ILP und durch die Laufzeit der WCET-Analyse, die der Registerallokator vor der Bildung der ILP durchführt. Für beide Komponenten sind Verbesserungen möglich.

In Abschnitt 2.2.3.3 wurde bereits darauf hingewiesen, dass ein ILP-Modell des Registerallokationsproblems, das nach dem ORA-Ansatz von Goodwin und Wilken [GW96] formuliert wurde, Entscheidungsvariablen enthält, die keinen Einfluss auf die Güte der Allokation haben und daher eliminiert werden können. Fu und Wilken [FW02] geben Regeln an, mit denen solche Entscheidungsvariablen identifiziert werden können. Da das ILP des WCET-optimierenden Allokators in zwei Teile gegliedert ist und der erste Teil nach dem ORA gebildet wird, könnte der Allokator so erweitert werden, dass er diese Vereinfachungen des ILP-Modells vornimmt. Fu und Wilken berichten, dass sie auf diese Weise die beobachtete Laufzeit des Allokators, der auf dem ORA-Ansatz basiert, von $\mathcal{O}(n^3)$ auf $\mathcal{O}(n^{2,5})$ senken konnten.

Auch die Laufzeit der WCET-Anaylse könnte verringert werden. In dieser Diplomarbeit wird für die Pre-Allokation ein naiver Allokator eingesetzt, der jedes virtuelle Register unmittelbar nach seiner Definition in dem Speicher sichert und unmittelbar vor der Nutzung wieder in ein physikalisches Register lädt. Auf diese Weise wird sehr viel Spillcode in das Programm eingefügt, was die Analyse verlangsamt. In Abschnitt 6.2 wurde bereits angemerkt, dass es keinen Zwang gibt, den naiven Allokators für die Pre-Allokation einzusetzen. Es können daher auch andere Registerallokatoren verwendet werden, die weniger Spillcode einfügen. Dies könnte auch bei der Schätzung der Ausführungsdauer von Basisblöcken ohne Spillcode bessere Ergebnisse zur Folge haben.

Da eine Allokation, die weniger Spillcode einfügt, zeitaufwändiger ist, muss eine Abwägung zwischen dem Laufzeitgewinn bei der WCET-Analyse und der zusätzlich für die Pre-Allokation benötigten Zeit vorgenommen werden. In Abschnitt 6.2 wurde ein Linear

Scan Allokator als Alternative vorgeschlagen. Ein Linear Scan Allokator weist eine geringe Laufzeit auf und fügt deutlich weniger Spillcode als der naive Allokator ein.

Die genannten möglichen Erweiterungen des in dieser Diplomarbeit erstellten Registerallokators lassen erwarten, dass sowohl die Güte des Allokators verbessert als auch seine Laufzeit verringert werden kann, und er dann die WCET einer Funktion stärker minimiert als andere Allokationsverfahren.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Abs06] AbsInt Angewandte Informatik GmbH (Hrsg.): *CRL Version 2*. AbsInt Angewandte Informatik GmbH, 2006. – http://www.absint.com/artist2/doc/crl2

[AG01] Appel, Andrew W. ; George, Lal: Optimal spilling for CISC machines with few registers. In: *ACM SIGPLAN Notices* 36 (2001), Nr. 5

[App98] Appel, Andrew W.: *Modern Compiler Implementation in C*. Cambridge, UK : Cambridge University Press, 1998

[ASU86] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffery D.: *Compilers - Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986

[Bel66] Belady, Laszlo A.: A study of replacement algorithms for a virtual-storage computer. In: *IBM Systems Journal* 5 (1966), Nr. 2

[BEN08] Berkelaar, Michel ; Eikland, Kjell ; Notebaert, Peter: *lp_solve 5.5*. http://lpsolve.sourceforge.net/5.5/, 2008

[BGOS94] Bacon, David F. ; Graham, Susan L. ; Oliver ; Sharp, J.: Compiler transformations for high-performance computing. In: *ACM Computing Surveys* 26 (1994), Nr. 4

[Bri92] Briggs, Preston: *Register allocation via graph coloring*. Houston, TX, USA, Rice University, Diss., 1992

[Cha82] Chaitin, Gregory J.: Register allocation & spilling via graph coloring. In: *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. New York, NY, USA : ACM, 1982

[Cor08] Cordes, Daniel: *Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib*, Universität Dortmund, Diplomarbeit, April 2008

[Duf88] Duff, Tom: *Tom Duff on Duff's Device*. http://www.lysator.liu.se/c/duffs-device.html, 1988. – Abruf am 15.09.2008

[FL98] Farach, Martin ; Liberatore, Vincenzo: On local register allocation. In: *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1998

[FLT06] Falk, Heiko ; Lokuciejewski, Paul. ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: *ESTIMedia '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. Washington, DC, USA : IEEE Computer Society, 2006

[FW02] Fu, Changqing ; Wilken, Kent: A faster optimal register allocator. In: *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA : IEEE Computer Society Press, 2002

[GW96] Goodwin, David W. ; Wilken, Kent D.: Optimal and near-optimal global register allocation using 0–1 integer programming. In: *Software - Practice and Experience* 26 (1996), Nr. 8

[Hav97] Havlak, Paul: Nesting of reducible and irreducible loops. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19 (1997), Nr. 4

[HF06] Heckmann, Reinhold ; Ferdinand, Christian: Worst-Case Execution Time Prediction by Static Program Analysis / AbsInt Angewandte Informatik GmbH. 2006. – Forschungsbericht

[HP03] Hennessy, John L. ; Patterson, David A.: *Computer architecture - a quantitative approach*. 3. Ausgabe. Amsterdam : Morgan Kaufmann, 2003

[HU72] Hecht, Matthew S. ; Ullman, Jeffrey D.: Flow graph reducibility. In: *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1972

[ICD06] Informatik Centrum Dortmund (Hrsg.): *ICD-C Compiler framework*. Informatik Centrum Dortmund, 2006. – http://www.icd.de/es/icd-c

[ICD07] Informatik Centrum Dortmund (Hrsg.): *ICD Low Level Intermediate Representation backend infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund, 2007

[ILO08] ILOG Inc.: *CPLEX*. www.ilog.com/products/cplex/, 2008

[Inf00] Infineon Technologies AG (Hrsg.): *TriCore 1 Pipeline Behaviour & Instruction Execution Timing - Application Note*. 1.1. Infineon Technologies AG, Januar 2000

[Inf03] Infineon Technologies AG (Hrsg.): *TriCore 1 32-Bit Unified Processor DSP Optimization Guide - Part 1: Instruction Set - User's Manual*. 1.3.6. Infineon Technologies AG, Januar 2003

[Inf04] Infineon Technologies AG (Hrsg.): *Memory Access Time in TriCore 1 TC1M Based Systems - Application Note*. 1.1. Infineon Technologies AG, Juli 2004

[Inf05a] Infineon Technologies AG (Hrsg.): *TriCore 1 32-Bit Unified Processor Core v1.3 Core Architecture - User's Manual*. 1.3.6. Infineon Technologies AG, Oktober 2005

[Inf05b] Infineon Technologies AG (Hrsg.): *TriCore 1 32-Bit Unified Processor Core v1.3 Instruction Set - User's Manual*. 1.3.6. Infineon Technologies AG, Oktober 2005

[Inf08] Infineon Technologies AG (Hrsg.): *TC1796 32-Bit Single-Chip Microcontroller TriCore - Data Sheet*. 1.0. Infineon Technologies AG, April 2008

[KG06] Koes, David R. ; Goldstein, Seth C.: A global progressive register allocator. In: *ACM SIGPLAN Notices* 41 (2006), Nr. 6

[KW98] Kong, Timothy ; Wilken, Kent D.: Precise Register Allocation for Irregular Architectures. In: *International Symposium on Microarchitecture*, 1998

[LM95] Li, Yau-Tsun S. ; Malik, Sharad: Performance analysis of embedded software using implicit path enumeration. In: *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*. New York, NY, USA : ACM, 1995

[Lok05] Lokuciejewski, Paul: *Design and Realization of Concepts for WCET Compiler Optimization*, Universität Dortmund, Diplomarbeit, Dezember 2005

[LPMS97] Lee, Chunho ; Potkonjak, Miodrag ; Mangione-Smith, William H.: MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In: *International Symposium on Microarchitecture*, 1997

[Mar07] Marwedel, Peter: *Eingebettete Systeme*. Berlin : Springer Verlag, 2007

[Mäl08] Mälardalen Real-Time Research Centre: *WCET project / Benchmarks*. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2008. – Abruf am 24.09.2008

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997

[NM93] Neumann, Klaus ; Morlock, Martin: *Operation Research*. München : Carl Hanser Verlag, 1993

[NMR03] Negi, Hemendra S. ; Mitra, Tulika ; Roychoudhury, Abhik: Accurate estimation of cache-related preemption delay. In: *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA : ACM, 2003

[NR95] Nilsen, Kelvin D. ; Rygg, Bernt: Worst-case execution time analysis on modern processors. In: *ACM SIGPLAN Notices* 30 (1995), Nr. 11

[PK89] Puschner, Peter ; Koza, Christian: Calculating the maximum execution time of real-time programs. In: *Real-Time Syst.* 1 (1989), Nr. 2

[PS99] Poletto, Massimiliano ; Sarkar, Vivek: Linear scan register allocation. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (1999), Nr. 5

[Ram99] Ramalingam, Ganesan: Identifying loops in almost linear time. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (1999), Nr. 2

[Sch07] Schulte, Daniel: *Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler*, Universität Dortmund, Diplomarbeit, Mai 2007

[SE02] Scholz, Bernhard ; Eckstein, Erik: Register allocation for irregular architectures. In: *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*. New York, NY, USA : ACM, 2002

[Set73] Sethi, Ravi: Complete register allocation problems. In: *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1973

[SMRC05] Suhendra, Vivy ; Mitra, Tulika ; Roychoudhury, Abhik ; Chen, Ting: WCET Centric Data Allocation to Scratchpad Memory. In: *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2005

[SS03] Sagonas, Konstantinos F. ; Stenman, Erik: Experimental evaluation and improvements to linear scan register allocation. In: *Software - Practice and Experience* 33 (2003), Nr. 11

[Tar73] Tarjan, Robert: Testing flow graph reducibility. In: *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1973

[THS98] Traub, Omri ; Holloway, Glenn ; Smith, Michael D.: Quality and speed in linear-scan register allocation. In: *ACM SIGPLAN Notices* 33 (1998), Nr. 5

[WEE+08] Wilhelm, Reinhard ; Engblom, Jakob ; Ermedahl, Andreas ; Holsti, Niklas ; Thesing, Stephan ; Whalley, David ; Bernat, Guillem ; Ferdinand, Christian ; Heckmann, Reinhold ; Mitra, Tulika ; Mueller, Frank ; Puaut, Isabelle ; Puschner, Peter ; Staschulat, Jan ; Stenström, Per: The worst-case execution-time problem—overview of methods and survey of tools. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7 (2008), Nr. 3

[Weg93] Wegener, Ingo: *Theoretische Informatik - eine algorithmenorientierte Einführung*. Stuttgart : Teubner, 1993

[Weg03] Wegener, Ingo: *Komplexitätstheorie - Grenzen der Effizienz von Algorithmen*. Berlin : Springer, 2003

[WM05] Wimmer, Christian ; Mössenböck, Hanspeter: Optimized interval splitting in a linear scan register allocator. In: *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. New York, NY, USA : ACM, 2005