TECHNISCHE UNIVERSITÄT DORTMUND
FAKULTÄT FÜR INFORMATIK

Diplomarbeit

Instruction Scheduling-Verfahren zur Minimierung der WCET

Andre Smolarczyk

17. Mai 2010

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik XII
(Technische Informatik und Eingebettete Systeme)

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

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

1 Einleitung

In dieser Einführung werden zunächst die Motivation des Autors (Kapitel 1.1) sowie Ziele dieser Diplomarbeit vorgestellt (Kapitel 1.2). Anschließend wird in Kapitel 1.3 auf verwandte Arbeiten eingegangen, bevor in Kapitel 1.4 der weitere Aufbau der Diplomarbeit beschrieben wird.

1.1 Motivation

Begriffe wie *Pervasive Computing*, *Ubiquitous Computing* oder *Ambient Intelligence* deuten es bereits an: Nach dem Ende der Mainframe-Epoche ist auch das Zeitalter des *Personal Computing* am Ende angelangt [HMNS01]. Wir befinden uns nun am Beginn der *Post-PC Ära*. Die Vision dieser Ära ist die Schaffung einer *intelligenten Umgebung*, in der informationsverarbeitende Systeme *allgegenwärtig* sind. Dadurch soll *jederzeit* von *Jedermann* ein Zugriff auf Informationen möglich sein. Bei der technischen Umsetzung dieser Vision spielen *Eingebettete Systeme* wie Mobiltelefone, PDAs, MP3-Player und Set-Top-Boxen für digitalen Fernsehempfang eine entscheidende Rolle.

Als Eingebettete Systeme werden informationsverarbeitende Systeme, die in ein umgebendes Produkt integriert sind und somit normalerweise vom Benutzer nicht direkt als Computer wahrgenommen werden, bezeichnet [Mar06]. In einem Automobil nutzen wir demnach implizit zahlreiche Eingebettete Systeme wie Infotainment-Systeme, Stabilitätskontrolle (ESP), Bremsassistent, Tempomat oder Einparkhilfe. Aber auch in Produktionsmaschinen oder Haushaltsgeräten spielen sie eine wichtige Rolle bei der Steuerung und Kontrolle. Eingebettete Systeme sind somit schon heute aus unserem Alltag nicht mehr wegzudenken.

Die steigende Bedeutung Eingebetteter Systeme lässt sich auch rein quantitativ belegen [Tur09] [BBI10]:

- Nur ein Bruchteil aller produzierten Mikroprozessoren wird mittlerweile in klassischen PC verwendet, der überwiegende Teil (über 98%) wird in Eingebetteten Systemen eingesetzt.

- Das Marktvolumen Eingebetteter Systeme entwickelte sich in den letzten Jahren in Deutschland stabil mit Zuwachsraten von bis zu 8%. Nach BITKOM Schätzungen wird das Volumen 2010 bei über 19Mrd. € liegen.

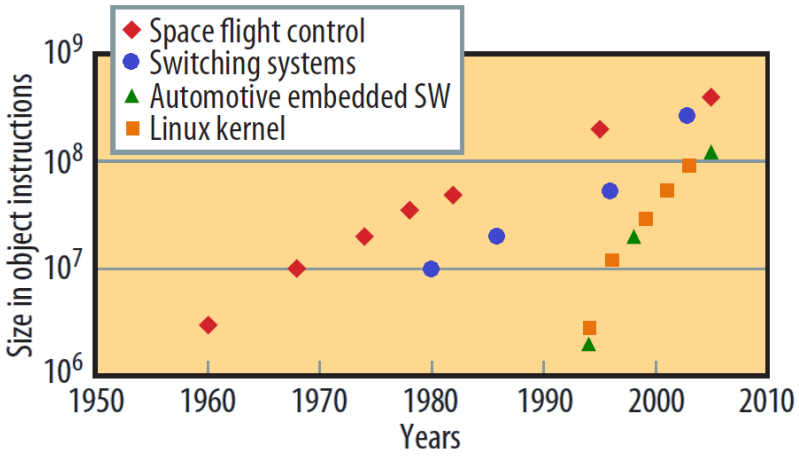


Abbildung 1.1: Entwicklung der Komplexität von ES-Software (Quelle: [EJ09])

- Bereits 2008 arbeiteten in Deutschland in Anbieterunternehmen für Eingebettete Systeme mehr als 40.000 Menschen (bei steigender Tendenz).

- In den Anwenderunternehmen der Embedded-Technologien, beispielsweise in der Automobil- oder der Maschinenbau-Industrie haben derzeit ca. 250.000 Mitarbeiter unmittelbar mit Eingebetteten Systemen zu tun.

Neben der Verbreitung Eingebetteter Systeme steigt gleichzeitig auch deren Komplexität rasant: Abbildung 1.1 zeigt die Entwicklung der durchschnittlichen Codegröße der System-Software in Raumfahrzeugen, Telekommunikations-Vermittlungssystemen, Automobilen und des Linux-Kernels zwischen 1960 und 2005. Der Linux-Kernel ist selbst zwar kein Eingebettetes System, wird aber häufig als deren Basis eingesetzt.

Das Wachstum entwickelte sich in den letzten Jahren immer schneller: Innerhalb von 10 Jahren ist die durchschnittliche Codegröße (gemessen in Instruktionen im Binärprogramm) in Eingebetteten Systemen in Automobilen und die Größe des Linux-Kernels ungefähr um den Faktor 100 gestiegen. Damit weisen Neuwagen der Premium-Klasse mit 20 bis 70 Eingebetteten Systemen und einer Codegröße von fast 1 GByte bereits eine ähnliche Komplexität wie die Onboard-Software eines Raumschiffes auf [EJ09].

Viele Eingebettete Systeme weisen ähnliche Charakteristika und Anforderungen auf, und stellen unter anderem Anforderungen an Zuverlässigkeit, Effizienz, Sicherheit und Echtzeitverhalten. Mit steigender Komplexität der Systemsoftware wird es immer schwieriger diese Anforderungen einzuhalten. Vor allem die Notwendigkeit, bestimmte Zeitschranken einzuhalten, ist im Bereich sicherheitskritischer Systeme allerdings unerlässlich. Sicherheitsrelevante System sind meist sogenannte *harte Echtzeitsysteme*: Bei diesen Systemen ist eine Reaktion innerhalb der gegebenen Zeitschranken zwingend erforderlich. Die Überschreitung einer Zeitschranke kann in solchen Systemen katastrophale Folgen haben. Beispielsweise ist eine Airbag-Steuerung, die nicht garantiert innerhalb einer festen Zeitschranke Messwerte der Sensoren verarbeiten und entscheiden kann, ob und wie stark der Airbag ausgelöst wird (Reaktionszeit ca. 1ms), praktisch nutzlos.

Um das Einhalten dieser Zeitschranken garantieren zu können, muss die maximale Ausführungszeit (*Worst-Case Execution Time, WCET*) des auf dem System ausgeführten Programms bekannt sein. Für eine exakte Bestimmung der WCET ist einerseits die Kenntnis der Eingabedaten notwendig, die zu einer Worst-Case Laufzeit führen. Aufgrund der bereits angesprochenen Komplexität der Software sind diese Eingaben für nicht-triviale Programme allerdings nicht bekannt. Andererseits hängt die WCET auch vom jeweiligen Startzustand der Hardware ab. Technologien wie Caches, virtueller Speicher und Sprungvorhersagen, die auch in Eingebetteten Systemen immer häufiger eingesetzt werden, sorgen dafür, dass hier zahlreiche unterschiedliche Zustände betrachtet werden müssen. Eine *exakte* manuelle Bestimmung der WCET ist demnach aufgrund der Komplexität praktisch nicht möglich.

Bei der Entwicklung Eingebetteter Systeme kommen daher meist sogenannte *dynamische Analysen* zum Einsatz. Dabei wird versucht die WCET durch Messung der Ausführungszeit eines Programms mit unterschiedlichen Probeläufen zu bestimmen. Diese Probeläufe können entweder auf echter Hardware oder mittels eines Simulators durchgeführt werden. Der große Vorteil dieses Verfahrens ist, dass es keine internen Kenntnisse der Hardware erfordert. Sobald die Zielhardware oder ein Simulator dafür zur Verfügung steht, kann dieses Verfahren eingesetzt werden.

Auch bei diesem Verfahren besteht allerdings das Problem, dass nicht klar ist, welche Eingaben zu der Worst-Case Laufzeit führen. Die Betrachtung aller möglichen Eingaben ist bei komplexeren Programmen ausgeschlossen. Um eine Unterschätzung der WCET zu verhindern, werden die gemessenen Laufzeiten daher mit einem Sicherheitsfaktor multipliziert. Dies garantiert allerdings nicht, dass nicht doch Fälle auftreten können, in denen diese Werte in der Praxis überschritten werden.

Kann die mittels dynamischer Analyse ermittelte WCET von der eingesetzten Zielhardware nicht eingehalten werden, bleiben dem Entwickler zwei Möglichkeiten:

- Es wird eine leistungsstärkere Hardware eingesetzt. Diese Systeme sind in der Regel teurer. Da Eingebettete Systeme oft in sehr großen Stückzahlen produziert werden, ist dieser Ansatz also unwirtschaftlich.

- Es wird versucht die WCET manuell durch Optimierungen am Programmcode oder Compiler-Parametern zu reduzieren. Anschließend wird das Programm erneut übersetzt, und die neue WCET bestimmt. Dieses Vorgehen ist allerdings äußerst zeitintensiv und fehleranfällig.

In Zeiten, in denen das Schlagwort *"Green IT"* eine immer wichtigere Rolle spielt, darf auch nicht vernachlässigt werden, dass ein möglichst gut auf die Anforderungen angepasstes System (also ein System, das die Zeitschranke sicher einhält, aber keine zu großen Reserven bietet) häufig energieeffizienter arbeitet, als leistungsstärkere Hardware. Ein zu groß gewählter Sicherheitsfaktor führt allerdings häufig zu einer Überdimensionierung der Hardware. Eine möglichst genaue Kenntnis der WCET ist daher vorteilhaft.

Da dynamische Methoden unsicher und zeitaufwändig sind, wäre es wünschenswert die WCET sicher automatisch durch ein Programm bestimmen zu können. Ein Programm,

das dies leistet, könnte allerdings auch das als unentscheidbar bekannte Halteproblem lösen. Im Allgemeinen ist dies daher für nicht-triviale Programme nicht möglich.

Eine Alternative stellt die Berechnung oberer Schranken für die WCET mittels einer statischen Analyse dar. Die von einem solchen Verfahren ermittelten Schranken sind sicher, daher muss im Gegensatz zu dynamischen Methoden nicht befürchtet werden, dass die ermittelten Werte in der Praxis überschritten werden können. Ein Programm, das eine (semi-)automatische Bestimmung solche Schranken für die WCET von in C geschriebenen Programmen ermöglicht, ist aiT der Firma AbsInt [Abs10].

Im Compiler *WCC* (WCET-aware C Compiler) des Lehrstuhl 12 der TU Dortmund wird aiT eingesetzt, um aufbauend auf den ermittelten WCET-Werten automatische Optimierungen durchzuführen.

1.2 Ziele

Mit dem WCET-aware C Compiler (WCC) [Inf10b] wurde in den letzten Jahren am Lehrstuhl 12 der TU Dortmund ein Compiler Framework für Eingebettete Systeme konzipiert und entwickelt. Der WCC übersetzt in ANSI C geschriebene Programme in ausführbaren Maschinencode. Dabei werden typische Compiler-Optimierungen, wie *Constant Code Folding*, *Dead Code Elimination* oder *Function Inlining* durchgeführt.

Die Besonderheit am WCC stellt aber die Möglichkeit dar, während des Kompilierungsprozesses erstmals automatisch, gezielte WCET-gesteuerte Optimierungen durchzuführen. Dadurch kann häufig auf die beschriebene aufwändige, manuelle Optimierung der WCET oder die Überdimensionierung der Hardware verzichtet werden. Ermöglicht wird dies durch die Integration von aiT in den Kompilierungsprozess. Standardoptimierungen können dadurch adaptiert werden, um WCET Information zu nutzen und so eine gezielte Reduktion der WCET zu erreichen.

Im WCC sind bereits zahlreiche WCET-fähige Optimierungen wie eine ILP-gesteuerte Registerallokation [Sch08], Loop Nest Splitting [FS06], Procedure Cloning / Function Specialization [LFMT08a] sowie eine Superblock-basierte DCE (Dead Code Elimination) und SCE (Common Subexpression Elimination) [LKM10] implementiert.

In dieser Arbeit wird eine weitere Standardoptimierung angepasst, um eine gezielte Reduktion der WCET zu erreichen. Bei dieser Optimierung handelt es sich um das Instruction Scheduling [Puc09]. Ziel des Instruction Scheduling ist im Allgemeinen die Reduktion der durchschnittlichen Programmlaufzeit (*Average-Case Execution Time, ACET*). Dazu werden zwei Maßnahmen eingesetzt:

- In vielen Prozessorarchitekturen kann eine Instruktion j, deren Ausführung von einer Instruktion i abhängig ist, erst nach einer gewissen Zeit (*Latenz*) ausgeführt werden. Eine Unordnung der Instruktionsreihenfolge kann dazu führen, dass zwischen den abhängigen Instruktionen i und j eine weitere unabhängige Instruktion j ausgeführt werden kann, so dass ein unnötiger Wartezyklus (*Stall*) verhindert werden kann.

- Die gleichzeitige Ausführung mehrerer unabhängiger Instruktionen (*multiple instruction issue*) ist eine Fähigkeit zahlreicher Prozessorarchitekturen. Durch eine geschickte Anordnung der Instruktionen kann eine möglichst hohe Auslastung der parallelen Recheneinheiten gewährleistet werden (*Instruction Level Parallelism*, ILP).

In einigen Fällen wird durch ein Instruction Scheduling nicht nur eine Reduktion der ACET sondern gleichzeitig auch eine Reduktion der WCET erreicht. Da dies allerdings nicht das explizite Ziel des Instruction Scheduling ist, sind auch Fälle möglich, in denen die WCET verschlechtert wird.

Das primäre Ziel von Optimierungen für Eingebettete Systeme ist allerdings oft die Minimierung der WCET. Daher sollen in dieser Arbeit zwei bekannte Instruction Scheduling-Verfahren adaptiert werden, um nicht wie üblich eine Reduktion der ACET, sondern der WCET zu erreichen. Bei diesen Verfahren handelt es sich um das *Trace Scheduling* sowie das *Superblock Scheduling*. Da nicht direkt offensichtlich ist, welches dieser Instruction Scheduling-Verfahren ein höheres Potential für die Minimierung der WCET aufweist, wurde nicht nur ein einzelnes Instruction Scheduling-Verfahren, sondern beide umgesetzt.

Bei beiden Verfahren kann es durch das Instruction Scheduling zu einer Vergrößerung des Binärcodes kommen (*Codewachstum*). Die Codegröße stellt bei Eingebetteten Systemen mit ihrem meist stark begrenzten Speicherplatz eine wichtiges Kriterium dar. Im Allgemeinen geht man davon aus, dass das Verfahren des Trace Scheduling im Vergleich zum Superblock Scheduling zu weniger Codewachstum führt. Allerdings zeigt [Gre01] für einige Benchmark, dass Superblock Scheduling dort zu weniger Codewachstum als Trace Scheduling führt. Dem Autor sind keine Arbeiten bekannt, die die Auswirkung beider Verfahren auf die WCET vergleichen. Da des Weiteren die Implementierung eines Superblock Scheduling, aufbauend auf einem Trace Scheduling, mit relativ geringem Aufwand möglich ist, wurden in dieser Diplomarbeit beide Verfahren umgesetzt und miteinander verglichen.

Zusammenfassend sollen folgende Ziele umgesetzt werden

- Trace-Selektion: Durch die von aiT bereitgestellten Information kann ermittelt werden, welche Programmpfade zur WCET beitragen. Das Instruction Scheduling wird anschließend auf diesen Pfaden ausgeführt.

- WCET-fähiges Trace Scheduling: Aufbauend auf den ermittelten Programmpfaden wird ein WCET-fähiges Trace Scheduling implementiert, das auf dem bereits im WCC vorhanden Instruction Scheduling aufbaut.

- WCET-fähiges Superblock Scheduling: Mit Hilfe der ermittelten Programmpfade werden sogenannte *Superblöcke* gebildet, und das Superblock Scheduling darauf durchgeführt.

- Benutzung weiterer Metriken: Durch die Benutzung der Simulationsplattform CoMET sind *Profiling-Informationen* innerhalb des WCC verfügbar. Um eine Aussage der Effizienz des WCET-basierten Scheduling zu ermöglichen, sollen diese Information genutzt werden, um alternativ eine Reduktion der durchschnittlichen Laufzeit zu erreichen.

1.3 Verwandte Arbeiten

Dieses Kapitel stellt zunächst verwandte Arbeiten zum Thema Instruction Scheduling vor. Anschließend werden einige Ansätze, die sich mit der Reduktion der WCET beschäftigen, vorgestellt.

1.3.1 Instruction Scheduling

Bei Instruction Scheduling-Verfahren kann zwischen *lokalen* und *globalen* Verfahren unterschieden werden. Lokale Verfahren betrachten jeweils einzelne Basisblöcke eines Programms isoliert. Um die vorhandene Parallelität der Hardware auszunutzen oder Stalls zu vermeiden, müssen Instruction Scheduling-Verfahren in der Lage sein, unabhängige Instruktionen zu finden. Innerhalb eines einzelnen Basisblocks ist die Anzahl unabhängiger Instruktionen allerdings meist gering. Dies schränkt die Möglichkeiten lokaler Verfahren ein.

Globale Verfahren bauen auf lokalen Verfahren auf, berücksichtigen aber gleichzeitig mehrere Basisblöcke des Programms und haben so mehr Möglichkeiten, unabhängige Instruktionen zu finden. Nachfolgend werden daher ausschließlich diese globalen Instruction Scheduling-Verfahren betrachtet.

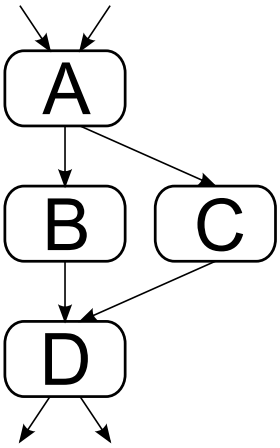
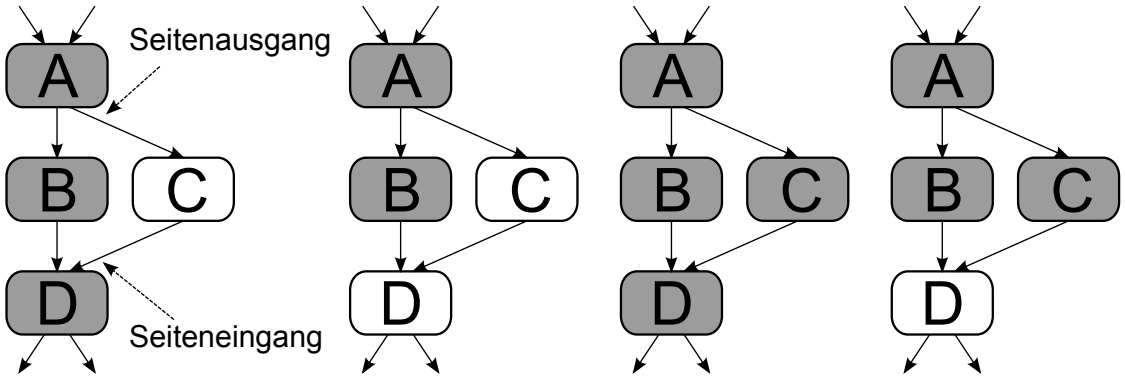


Abbildung 1.2: Ausschnitt Kontrollflussgraph

Die Unterschiede der vorgestellten Instruction Scheduling-Verfahren sollen anhand eines einfachen Kontrollflusses wie Abbildung 1.2 ihn darstellt erläutert werden. Die Blöcke repräsentieren dabei Basisblöcke eines Progamms. Zwei Blöcke sind mit einer gerichteten Kante verbunden, wenn zwischen ihnen ein Übergang stattfinden kann. Block A stellt eine Verzweigung dar. Hier kann der Kontrollfluss entweder zu Block B oder zu Block C wechseln. An einem Block mit zwei eingehenden Kanten, werden alternative Kontrollflüsse zusammengeführt. Dies ist in diesem Beispiel in Block D der Fall.



(a) Trace (b) Superblock (c) Hyperblock (d) Treegion

Abbildung 1.3: Übersicht globale Instruction Scheduling-Verfahren

In [Fis81] beschreibt Fisher eines der ersten *Single Path Global Instruction Scheduling-Verfahren*. Bei diesem Trace Scheduling genannten Verfahren wird zunächst mittels Heuristiken oder über ein Profiling eine möglichst häufig ausgeführte, schleifenfreie Folge von Basisblöcken (*Trace*) selektiert. In Abbildung 1.3a deuten die grau hinterlegten Blöcke A, B und C beispielhaft einen Trace an. Ein Trace darf dabei sowohl Basisblöcke enthalten, an denen unterschiedliche Kontrollflüsse zusammengeführt werden (*Seiteneingänge*), als auch Basisblöcke an denen der Kontrollfluss verzweigt (*Seitenausgänge*). Bei einer Verzweigung darf allerdings nur einer der nachfolgenden Basisblöcke Teil des Trace sein.

Nach der Selektion eines Trace wird ein Instruction Scheduling wie auf einem einzelnen Basisblock durchgeführt. Das Verfahren arbeitet in dieser Phase also wie ein lokales Instruction Scheduling. Seiteneingänge und -ausgänge werden nicht besonders beachtet. Dies hat zur Folge, dass die Semantik des Programms beim Durchlaufen sogenannter *Off-Trace Pfade* sehr wahrscheinlich verändert wird. Um die ursprüngliche Semantik nach dem Instruction Scheduling wieder herzustellen, findet anschließend eine *Bookkeeping-Phase* statt. In dieser Phase wird sogenannter *Kompensationscode* erzeugt. Beispielsweise muss eine Instruktion, die unterhalb einer Verzweigung verschoben wurde, auch in alle anderen Pfade, die diese Verzweigung verlassen, kopiert werden. Auf dieses Verfahren wird auf Seite Seite 41 in Kapitel 5 noch ausführlich eingegangen.

Die Komplexität des Bookkeeping beim Trace Scheduling basiert weitestgehend auf Verschiebungen, die Seiteneingänge betreffen. Da Seiteneingänge außerdem andere Optimierungen wie *Copy Propagation* erschweren, führten Hwu et al. im IMPACT-Projekt das Superblock Scheduling ein [HMC+93].

Informell ist ein Superblock ein Trace ohne Seiteneingänge. Der Kontrollfluss kann in einem Superblock also über mehrere Seitenausgänge verlassen werden. Ein Eingang ist allerdings nur am Beginn des Superblocks möglich. In Abbildung 1.3b bilden die beiden Blöcke A und B beispielsweise einen Superblock. Superblöcke ähneln erweiterten Basisblöcken. Der wesentliche Unterschied liegt in deren Erzeugung: Superblöcke werden anhand von Profiling-Information gebildet, und Seiteneingänge können durch eine *Tail Duplication* eliminiert werden, um die Superblöcke zu vergrößern (hier nicht gezeigt).

Innerhalb Kontrollfluss-intensiver Programme ist es oft nicht möglich, einen einzelnen Pfad durch das Programm zu bestimmen, der bevorzugt werden sollte. Das von Mahlke et al. entwickelte *Hyperblock-Scheduling* [MLC+92] berücksichtigt dies: Wie in Abbildung 1.3c gezeigt, kann ein *Hyperblock* mehrere parallele Kontrollflüsse enthalten. Mittels bestimmter Transformationen werden innerhalb des Hyperblocks Sprung-Instruktionen zu *bedingten Vergleichs-Instruktionen* umgewandelt, und anschließend das Instruction Scheduling wie auf einem einzelnen Basisblock durchgeführt. Da die TriCore-Architektur bedingte Instruktionen nicht beziehungsweise nur sehr eingeschränkt unterstützt, ist das Hyperblock-Scheduling im Rahmen dieser Diplomarbeit allerdings nicht anwendbar.

Das von Hsu und Davidson [HD86] sowie Havanaki et al. [HBC98] beschriebene Treegion-Scheduling baut auf dem Hyperblock-Scheduling auf. Anders als die bisher vorgestellten Profiling-basierten Ansätze, ist dieses Verfahren allerdings ein strukturbasierter Ansatz: Die Bildung von Treegions erfolgt ohne Profiling-Informationen ausschließlich aufgrund der Programmstruktur. Eine Treegion ist dabei ein Baum innerhalb des Kontrollflussgraphen, in dem alle Knoten (Basisblöcke) außer der Wurzel genau eine eingehende Kante aufweisen (Abbildung 1.3d). Eine Treegion kann also wie ein Hyperblock parallele Pfade enthalten.

Da das Treegion-Scheduling auf dem Hyperblock-Scheduling basiert, benötigt es prinzipiell eine Unterstützung bedingter Instruktionen. Im WCC ist ein Variante eines Treegion-Schedulings implementiert, die ohne die Verwendung dieser bedingter Instruktionen auskommt [Puc09].

1.3.2 WCET

In diesem Kapitel werden zunächst einige Arbeiten vorgestellt, die eine Minimierung der WCET durch Low-Level (Assembler-Ebene) Optimierungen anstreben, anschließend folgen einige der zahlreichen am LS12 entwickelten hochsprachlichen Ansätze.

Zhao et al. ermitteln in [ZKW+06] Worst-Case Pfade mittels eines Timing-Analyzers und wenden nach Bildung von Superblöcken einige weitere Low-Level wie Optimierungen *Code Sinking* und *Dead Assignment Elimination* an, um die WCET zu minimieren. Die verwendeten Benchmarks sind allerdings im Vergleich zur umfangreichen Benchsuite des WCC sehr klein (150 Zeilen) und daher wenig aussagekräftig.

Die schlechte Vorhersagbarkeit des Verhaltens von Caches ist für eine exakte WCET-Bestimmung problematisch. In [PLM09] stellen Plazar et al. einen ILP-Algorithmus vor, der den Instruction-Cache in Abhängigkeit von WCET-Informationen partitioniert. Durch die Zuweisung einzelner Cache-Partitionen für jeden Task kann keine Verdrängung durch andere Tasks stattfinden, so dass die Vorhersagbarkeit und damit die WCET verbessert wird.

Mit ihrer hervorragenden Vorhersagbarkeit stellen *Scratchpad memories* (SPMs) für Eingebettete Systeme eine Alternative dar. In [FK09] wird ein ILP-basiertes Verfahren zur statischen Abbildung von Programmcode auf SPMs vorgestellt, das zu einer minimalen WCET führt.

In[LFM08] nutzen Lokuciejewski et al. die bekannte Optimierung *Procedure Positioning* in einer neuartigen Weise, um mittels WCET-Informationen eine Anordnung von Funktionen im Speicher zu erreichen, die zu weniger Cache-Fehlzugriffen und damit zu einer Verringerung der WCET führt.

Im Gegensatz zum Procedure Positioning handelt es sich bei *Procedure Cloning* und den nachfolgend vorgestellten Arbeiten um High-Level Optimierung. In [LFMT08b] wird Procedure Cloning aufbauend auf WCET-Daten genutzt, um eine präzisere Annotation von Schleifengrenzen und damit eine Verbesserung der WCET-Schätzung zu ermöglichen.

Vor allem Multimedia-Anwendungen bestehen häufig aus tief verschachtelten Schleifen (*Loop Nests*), in denen der Kontrollfluss durch *if-Ausdrücke* bestimmt wird. Dies führt unter anderem durch auftretende Pipeline-Stalls und Unsicherheiten der Sprungvorhersage zu einer Ungenauigkeit und damit großen Überabschätzung der WCET-Analyse. Mittels des in [FS06] vorgestellten *Loop Nest Splitting* können Loop Nests aufgeteilt, und damit die Anzahl ausgeführter if-Ausdrücke minimiert werden, was zu einer deutlich präziseren WCET-Analyse führen kann.

In [LKM10] wird erstmals das Low-Level Konzept der Superblöcke auf die hochsprachliche Ebene übertragen und darauf aufbauend die beiden bekannten Optimierung Dead Code Elimination (DCE) und Common Subexpression Elimination (CSE) als WCET-Optimierungen implementiert. Durch den größeren Umfang der Superblöcke ermöglicht die Superblock-CSE Ersetzungen von Ausdrücken, die bei einer lokalen Betrachtung von Basisblöcken nicht möglich wären. Die Superblock-DCE dient anders als die klassische DCE nicht dazu *toten* Code zu eliminieren, sondern Anweisungen, die innerhalb des betrachteten Superblocks keinen Effekt haben, aus diesem Superblock zu verschieben.

1.4 Aufbau der Arbeit

Die Begriffe WCET (Worst-Case Execution Time) und darauf aufbauend *WCEP (Worst-Case Execution Path)* stellen zentrale Begriffe dieser Arbeit dar. Daher erfolgt in Kapitel 2 zunächst eine Definition, bevor in Kapitel 2.3 eine kurze Einführung in statische Verfahren zur Bestimmung der WCET gegeben wird. Kapitel 2.4 liefert einen Überblick über die Funktionsweise des im WCC verwendeten WCET-Analysewerkzeugs aiT. Zuletzt wird in Kapitel 2.5 auf die Schwierigkeiten bei der Optimierung der WCET eingegangen.

Danach wird in Kapitel 3.1 der am Lehrstuhl 12 entwickelte Compiler WCC, die verwendete Low-Level Darstellung *ICD-LLIR* (Kapitel 3.1.1) sowie die Zielplattform TriCore (Kapitel 3.2) vorgestellt. Besonderen Wert wird dabei auf eine Schilderung der Besonderheiten der Pipelines gelegt, die ein Instruction Scheduling für diesen Prozessor berücksichtigen muss.

Kapitel 4 geht ausführlicher auf Instruction Scheduling im Allgemeinen und auf die Umsetzung innerhalb des WCC ein. Die später vorgestellten WCET-fähigen Instruction Scheduling Verfahren bauen auf dem lokalen Scheduler des WCC auf.

Daher erfolgen in diesem Kapitel Erläuterung zu folgenden Verfahren:

- Abhängigkeiten zwischen Instruktionen
- Aufbau eines Abhängigkeitsgraphen
- List Scheduling
- Heuristiken zur Prioritätsbestimmung einzelner Anweisungen während des Instruction Scheduling

Das erste WCET-fähige Instruction Scheduling-Verfahren, das innerhalb dieser Diplomarbeit implementiert wurde, wird in Kapitel 5 genauer beschrieben. Zunächst wird in Kapitel 5.1 das Verfahren von Fisher, auf dem diese Optimierung basiert, allgemein vorgestellt. Anschließend wird genauer auf die einzelnen Schritte des Trace Scheduling eingegangen. Nachdem in Kapitel 5.5 beschrieben wurde, wie Schleifen innerhalb von Traces repräsentiert werden können, geht Kapitel 5.6 zum Schluss auf einige Details der Implementierung des Trace Scheduling innerhalb des WCC ein.

In Kapitel 6 wird das zweite in dieser Arbeit implementierte Instruction Scheduling-Verfahren beschrieben. Das WCET-fähige Superblock Scheduling baut auf dem Trace Scheduling auf. Beide Verfahren weisen große Ähnlichkeiten auf. Der größte Unterschied liegt in der im Superblock Scheduling verwendeten Tail Duplication. Dieses Verfahren wird in Kapitel 6.2 beschrieben.

Schlussendlich werden in Kapitel 7 die experimentellen Ergebnisse der implementierten Instruction Scheduling-Verfahren dargestellt, bevor in Kapitel 8 eine Zusammenfassung des Erreichten sowie ein kurzer Ausblick möglicher Erweiterungen gegeben wird.

2 WCET

Die Begriffe WCET und WCEP spielen in dieser Arbeit eine zentrale Rolle. Daher erfolgt in diesem Kapitel zunächst eine Definition der Begriffe. Anschließend wird ein Überblick über existierende Verfahren zur statischen Ermittlung der WCET (Kapitel 2.3) gegeben. Kapitel 2.4 gibt eine ausführlichere Erläuterung der Funktionsweise des WCET-Analysewerkzeugs aiT, das im WCC zum Einsatz kommt. Zuletzt wird kurz auf die Schwierigkeiten, die eine Optimierung der WCET mit sich bringt, eingegangen.

2.1 WCET Begriff

Mit dem Begriff WCET (Worst-Case Execution Time) wird die maximale Ausführungszeit eines Programms bezeichnet. Dabei müssen alle denkbaren Eingabedaten und alle möglichen Anfangszustände des zur Ausführung verwendeten Prozessors berücksichtigt werden. Wie in Kapitel 1.1 bereits angedeutet, ist die WCET im Allgemeinen nicht exakt berechenbar. Auch Messungen liefern keine sicheren Werte für die WCET. In der Praxis arbeitet man daher stattdessen oft mit *sicheren* oberen Schranken für die WCET, die von statischen Analyseverfahren bestimmt werden können. Dabei spielen zwei Begriffe eine Rolle [Mar06]

- $WCET_{real}$ - Die tatsächliche, größtmögliche Ausführungszeit eines Programms. Diese ist im Allgemeinen nicht bekannt.

- $WCET_{est}$ - Geschätzte WCET-Werte, die von Analyseverfahren bestimmt werden können. Trotz des Begriffs „geschätzt“ handelt es sich hierbei um sichere obere Schranken für die tatsächliche $WCET_{real}$. $WCET_{real}$ ist also immer kleiner oder gleich $WCET_{est}$.

Die Unterschiede zwischen $WCET_{real}$, $WCET_{est}$ und tatsächlich beobachteten Ausführungszeiten sind in Abbildung 2.1 dargestellt.

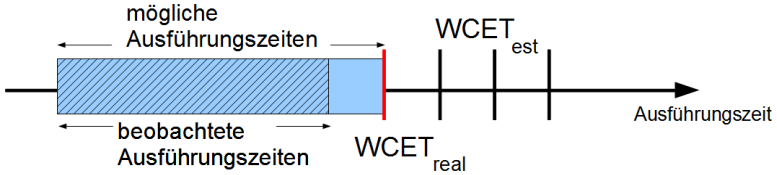


Abbildung 2.1: WCET Begriff

Da eine stark überabgeschätzte $WCET_{est}$ in der Praxis nutzlos ist, sollte ein Verfahren zur Bestimmung der $WCET_{est}$ nicht nur sichere, sondern auch möglichst *präzise* Werte liefern. Die Differenz aus $WCET_{est}$ und $WCET_{real}$ sollte also gegen 0 gehen.

Im WCC stehen die von aiT gelieferten $WCET_{est}$ Informationen zur Verfügung. Wenn im Folgenden von WCET gesprochen wird, ist daher immer die $WCET_{est}$ gemeint.

Neben dem Begriff der WCET spielt auch die ACET (Average-Case Execution Time) eine Rolle. Dies ist die durchschnittliche Programmlaufzeit, die mittels eines Profiling über verschiedene Eingabedaten ermittelt werden kann. Im WCC steht dazu die Simulationsplattform CoMET zur Verfügung. Obwohl das primäre Ziel dieser Diplomarbeit eine Optimierung der WCET ist, sollen zur Evaluation der entwickelten Instruction Scheduling Verfahren auch ACET-Werte genutzt werden und eine Optimierung anhand dieser Werte vorgenommen werden.Dadurch lässt sich der Einfluss der Optimierungen auf die unterschiedlichen Laufzeiten untersuchen.

2.2 WCEP Begriff

Die WCET stellt die maximale Ausführungszeit eines Programms auf der Zielhardware rein nummerisch dar. Von Interesse ist allerdings oft auch der Programmpfad, dessen Länge der WCET entspricht. Diesen Pfad bezeichnet man als Worst-Case Execution Path (WCEP). Zunächst erfolgt eine Definition verwendeter Begriffe.

Definition 2.1 (Basisblock).
*Ein **Basisblock** ist eine maximale Sequenz von Instruktionen, die nur an der ersten Instruktion betreten werden kann, und die nur an der letzten Instruktion verlassen werden kann.*

Verzweigungen des Kontrollflusses (durch bedingte Sprünge) sind demnach immer nur am Ende eines Basisblocks möglich. Alternative Kontrollflüsse können nur am Beginn eines Basisblocks zusammengeführt werden. Der mögliche Kontrollfluss eines Programms lässt sich mittels eines Kontrollflussgraphen darstellen.

Definition 2.2 (Kontrollflussgraph).
*Ein **Kontrollflussgraph*** (Control Flow Graph, CFG) *eines Programms ist ein gerichteter, zyklischer Graph $G = (V, E)$, wobei V die Menge der* Basisblöcke *und E die Menge der gerichteten **Kontrollflusskanten** darstellt. Eine Kontrollflusskante $e \in E$ verbindet zwei Knoten v_i und v_j, wenn v_j unmittelbar nach v_i ausgeführt werden kann.*

Der Basisblock, an dem die Startfunktion des Programms betreten wird, wird als **Quelle** bezeichnet. Alle Basisblöcke, an denen die Startfunktion wieder verlassen werden kann, werden als **Senken** bezeichnet. Damit ist es möglich, den Begriff des Programms zu definieren.

Definition 2.3 (Programm).
Ein Programm $P = \{P_0, ..., P_n\}$ ist die Menge aller möglichen Pfade durch den Kontrollflussgraph $G = (V, E)$, die in der Quelle $v_S \in V$ starten, und in einer Senke $v_E \in V$ enden: $\forall P_i \in PROG : P_i = (v_S, ..., v_E)$

Unter allen möglichen Programmpfaden aus PROG ist der WCEP derjenige, auf dem die maximale Laufzeit zustande kommt. Dabei schließt die Definition von Programmpfaden auch Pfade ein, die zwar im CFG möglich sind, aufgrund sich ausschließender Bedingungen, in einer tatsächlich Ausführung des Programms allerdings nicht vorkommen können. Solche Pfade werden als *infeasible* bezeichnet. Eine statische WCET-Analyse, die solche Pfade erkennen und bei der Berechnung der WCET ausschließen kann, ist somit in der Lage die Analysegenauigkeit zu erhöhen.

2.3 WCET-Analyse

Die von einer dynamischen WCET-Analyse gelieferten Werte sind, wie bereits in Kapitel 1.1 (Seite 1) erläutert, nicht sicher. Bei diesen Verfahren ist also nicht garantiert, dass die ermittelte $WCET_{est}$ immer größer als die $WCET_{real}$ ist. Daher werden im Folgenden ausschließlich statische Verfahren betrachtet, die sichere Werte für die WCET liefern. Diese Verfahren führen ihre Berechnungen anhand einer Analyse des Programmcodes durch. Dabei kann zwischen Syntaxbaum-, Pfad- und ILP- Verfahren unterschieden werden [SS08].

Syntaxbaum-Verfahren: Diese Verfahren gehören zu den ältesten Methoden zur WCET-Bestimmung. Sie führen eine *bottom-up* Berechnung der WCET auf dem Syntaxbaum des Programms durch. Dabei wird die WCET eines Knoten nach bestimmten Regeln anhand der WCET seiner Kinder ermittelt. Beispielsweise liefert

$$\text{WCET(S1; S2)} := \text{WCET(S1)} + \text{WCET(S2)}$$

die WCET einer sequentiellen Ausführung der Ausdrücke S1 und S2. S1 und S2 können dabei wiederum komplexere Ausdrücke sein, deren WCET sich ebenfalls über Regeln bestimmen lässt. Die WCET eines if-else-Ausdrucks lässt sich in dieser Methode mittels der Regel

$$\text{WCET(if (B) \{S1\} else \{S2\})} := \text{WCET(B)} + \max(\text{WCET(S1)}, \text{WCET(S2)})$$

bestimmen. Dabei stellt B die Bedingung des Ausdrucks dar. S1 repräsentiert den then-Teil und S2 den else-Teil des Ausdrucks.

Die WCET elementarer Anweisungen erhält man dabei durch eine Schätzung der Laufzeit des erzeugten Assemblercodes. Aufgrund ihrer Simplizität weisen diese Ansätze eine hohe Laufzeiteffizienz auf. Allerdings gestaltet es sich schwierig, Informationen über unausführbare Pfade mit in die Analyse einfließen zu lassen. Dadurch kann es bei diesen Verfahren zu einer großen Überabschätzung der WCET kommen.

Pfad-Verfahren: Diese Methoden arbeiten auf dem CFG des zu analysierenden Programms, und führen darauf eine Suche nach dem längsten Pfad durch. Um die

WCET von Schleifen zu berechnen wird der Schleifenrumpf während der Analyse abgerollt. Um dabei ein unbegrenztes Abrollen zu verhindern, benötigt die Analyse Informationen über die maximale Ausführungshäufigkeit von Schleifen (*Loop Bounds*).

Falls das eingesetzte Verfahren tatsächlich alle möglichen Pfade einzeln analysiert, können dadurch sehr exakte Ergebnisse geliefert werden. Der größte Nachteil dieser Verfahren ist das schlechte Laufzeitverhalten, da unter Umständen exponentiell viele Pfade betrachtet werden müssen.

ILP-Verfahren: Wie auch die Pfad-Verfahren, arbeiten *Integer Linear Programming* (ILP) Verfahren auf dem Kontrollflussgraphen des Programms. Für jeden Basisblock b des CFG wird zunächst eine maximale Laufzeit C_b angenommen. Die WCET des Programms kann durch einen ILP-Solver ermittelt werden, indem dieser die Zielfunktion

$$maximize \sum_{b \in B} C_b * N_b$$

berechnet, wobei B die Menge der Basisblöcke darstellt. N_b ist eine Integer Variable, die die gesamte Ausführungshäufigkeit eines Basisblock $b \in B$ angibt. Da die Funktion so natürlich unbegrenzt ist, werden abhängig vom Kontrollflussgraphen Nebenbedingungen für N_b benötigt:

- Der Startknoten des Programms P wird exakt einmal ausgeführt, also $N_b = 1$ für den Startknoten von P.
- Die Anzahl von Schleifeniterationen bzw. Rekursionsstufen muss durch eine entsprechende Bedingung beschränkt werden.
- Für alle Knoten, außer Start- und Endknoten muss gelten:
  Die Anzahl eingehender Kanten und die Anzahl ausgehender Kanten eines Knoten entsprechen der Worst-Case Ausführungshäufigkeit (*Worst-Case Execution Count, WCEC*) des Knoten.

Das Finden einer Optimallösung für ILP-Programme ist ein NP-schweres Problem. In der Praxis ist das Laufzeitverhalten allerdings meist gut, so dass diese Verfahren einen guten Kompromiss aus Präzision und Laufzeit darstellen. Daher nutzt auch das im WCC verwendete Werkzeug aiT ein ILP-Verfahren zur Analyse der WCET. In aiT kommen allerdings einige Erweiterungen zum Einsatz, um die Analysegenauigkeit zu erhöhen. Eine dieser Erweiterungen stellt die Unterstützung von *Kontexten* dar. Die Arbeitsweise von aiT und das Konzept der Kontexte wird im Folgenden beschriebenen.

2.4 WCET-Analyse mittels aiT

Das innerhalb des WCC verwendete WCET-Analysewerkzeug aiT führt eine statische Analyse des Programms durch, um die WCET zu bestimmen. Die grundlegende Arbeitsweise

soll anhand von Abbildung 2.2 erläutert werden. Als Eingabe wird ein Binärprogramm

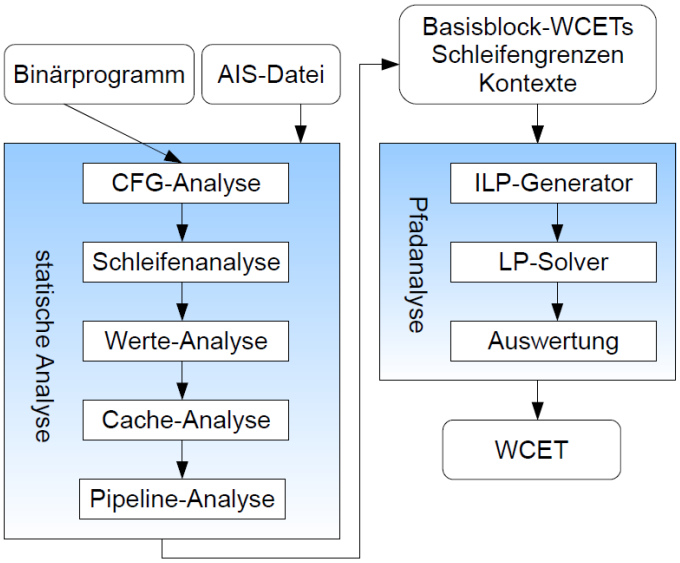


Abbildung 2.2: Funktionsweise aiT (Quelle: [Kel09])

sowie vom Benutzer zur Verfügung gestellte Informationen über die Anzahl an Schleifeniterationen, Rekursionstiefen und Analyseoptionen in Form einer *AIS-Datei* erwartet. Die anschließend durchgeführte Analyse besteht aus mehreren Phasen, die im Folgenden kurz erläutert werden:

CFG-Analyse: Im ersten Schritt der Analyse wird der Kontrollfluss des zu analysierenden Programms durch aiT rekonstruiert und in die interne Darstellung *CRL2* (Control Flow Representation Language) transformiert, die als Eingabe der weiteren Analyseschritte dient.

Schleifen-Analyse: Die WCET Analyse benötigt Informationen über die Iterationshäufigkeit von Schleifen. Die Schleifenanalyse von aiT versucht, für nicht vom Benutzer annotierte Schleifen die Iterationshäufigkeit selbständig zu ermitteln. Da die Möglichkeiten dieser Analyse allerdings eingeschränkt sind, wurde am LS12 eine Schleifenanalyse in den WCC integriert, die auch komplexere Schleifen erkennen und deren Grenzen an aiT übergeben kann (Kapitel 3.1.2).

Werte-Analyse In diesem Schritt wird mittels Abstrakter Interpretation versucht zu ermitteln, welche Werte die in Prozessorregistern gehaltenen Variablen während einer Programmausführung annehmen können. Wie auch die nachfolgend vorgestellten Analyseschritte arbeitet die Werte-Analyse dabei mit dem Konzept der Kontexte.

Ein Kontext repräsentiert die Aufrufhistorie des aktuell untersuchten Basisblocks. Damit können für jeden Basisblock mehrere Analyseergebnisse gespeichert werden. Wird eine Funktion beispielsweise mit unterschiedlichen Parametern aufgerufen, so kann dies in unterschiedlichen Kontexten festgehalten werden. Auf diese Weise ist es nicht notwendig auf das schlechteste Ergebnis (im Sinne des Worst-Case Verhaltens) zurück zu fallen, und es kann eine höhere Analysegenauigkeit erreicht werden.

Diese Fähigkeit spielt auch bei Schleifen eine große Rolle, da die erste Iteration einer Schleife oft den Cache füllt. Eine kontextsensitive Analyse kann dies berücksichtigen, und nur in der ersten Iteration/Kontext von einer hohen WCET aufgrund zahlreicher *Cache-Misses* ausgehen. Eine Analyse, die ohne Kontexte auskommt, muss immer den Worst-Case annehmen, in dem sich alle Schleifeniteration bezüglich des Cache-Verhaltens genau wie die erste Iteration verhalten.

Die Analysegenauigkeit ist also prinzipiell höher, wenn die Anzahl betrachteter Kontexte hoch ist. Allerdings steigt aufgrund der damit verbundenen erhöhten Komplexität auch die Analysedauer. Daher lässt sich der maximale Wert der betrachteten Kontexte vom Benutzer festlegen. Wird die festgelegte Anzahl an Kontexten während der Analyse überschritten, so werden alle weiteren Kontexte im letzten Kontext aggregiert. Hier muss dementsprechend seitens des Nutzers eine Abschätzung zwischen Analysedauer und -genauigkeit erfolgen.

Cache-Analyse: Mittels der Cache Analyse wird für jeden Hauptspeicherzugriff das Cache-Verhalten bestimmt. Dabei wird jeder Zugriff in eine Kategorie wie *always hit* oder *always miss* eingestuft.

Pipeline-Analyse: Im anschließenden Schritt der Pipeline Analyse wird das Verhalten der Zielpipeline für einzelne Basisblöcke vorhergesagt. Dabei wird unter anderem der aktuelle Zustand der Pipeline, der Inhalt der *Prefetch Queue* und die Klassifikation der Hauptspeicherzugriffe beachtet. Hier ist also eine sehr genaue Kenntnis der Zielarchitektur notwendig. Als Ergebnis wird die Ausführungszeit jeder einzelnen Instruktion innerhalb jedes betrachteten Kontext geliefert.

Pfad-Analyse: Als letzter Schritt der WCET-Bestimmung erfolgt die Pfad-Analyse. Anders als die bisherigen Schritte baut diese nicht auf Abstrakter Interpretation auf, sondern nutzt wie in Kapitel 2.3 beschrieben, ein ILP-Modell, um unter allen möglichen Ausführungspfaden einen längsten Pfad, und damit die WCET zu bestimmen. Dabei kommen wieder die eben erwähnten Kontexte zum Einsatz.

Als Ergebnis liefert diese Analyse die gesamte WCET sowie einen CFG, der unter anderem mit Informationen über die Ausführbarkeit, WCET sowie WCEC einzelner Basisblöcke annotiert ist.

Nachfolgend werden Schwierigkeiten dargestellt, die sich im Zusammenhang mit WCET-Optimierungen ergeben.

2.5 Herausforderungen bei der WCET Optimierung

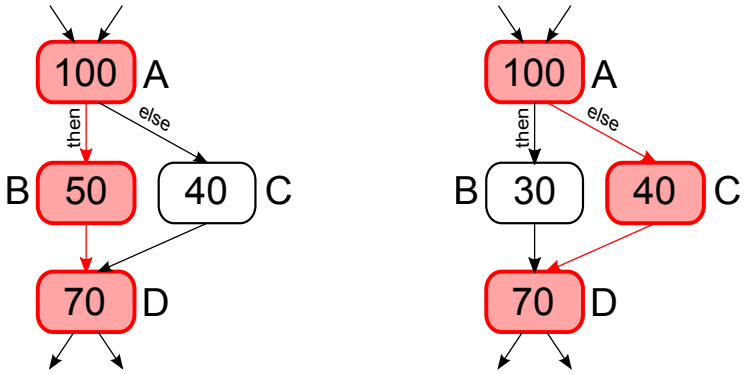
Bereits bei der Optimierung eines Programms mit dem Ziel der ACET-Minimierung ergibt sich das Problem, dass im Allgemeinen nicht entscheidbar ist, ob eine Optimierung einen positiven Effekt zur Verringerung der Programmlaufzeit beitragen wird. Es ist auch möglich, dass eine Optimierung die Laufzeit nicht verringert, sondern vergrößert. In anderen

Fällen ist es möglich, dass während eines Optimierungsprozesses zeitweise Verschlechterungen akzeptiert werden müssen, um so lokale Minima zu überwinden.

Bei einer Optimierung mit dem Ziel der Verringerung der WCET ergeben sich darüber hinaus weitere Schwierigkeiten. Ein Thema, dass in diesem Zusammenhang immer wieder diskutiert wird, ist die Problematik der *Pfadwechsel*, die in dieser Form bei einer ACET-Optimierung nicht vorkommen.

2.5.1 Pfadwechsel

Pfad-basierte Optimierungen, die eine Reduktion der ACET zum Ziel haben, arbeiten auf dem am häufigsten ausgeführten Pfad (ACEP). Ein große Vorteil gegenüber WCET basierten Optimierungen ist, dass dieser Pfad sich durch die Optimierung nicht ändern kann. Eine Optimierung die den ACEP verkürzt, leistet demnach immer einen positiven Beitrag zur Minimierung der ACET.



(a) Vor der Optimierung (b) Nach der Optimierung

Abbildung 2.3: Pfadwechsel

Bei WCET-basierten Optimierung ist dies nicht sicher der Fall. Der längste Ausführungspfad eines Programms (WCEP) kann im Laufe der Optimierungen wechseln, so dass Optimierungen zwar eine lokale Verbesserung erreichen, insgesamt aber keine, oder nur eine geringe Verringerung der WCET eintritt. Abbildung 2.3 soll dies veranschaulichen.

In Abbildung 2.3a ist der Kontrollflussgraph einer *if-then-else*-Verzweigung dargestellt. Der Einfachheit halber wird im Folgenden davon ausgegangen, dass diese Verzweigung im betrachteten Programm genau einmal ausgeführt wird. Innerhalb der Basisblöcke sind die von aiT ermittelten WCET Werte dieser Verzweigung notiert. Die Werteanalyse von aiT war in diesem Beispiel nicht in der Lage, das Ergebnis der Bedingung statisch auszuwerten, so dass sowohl der *then*-Zweig, als auch der *else*-Zweig als ausführbar (Basisblöcke B und C) betrachtet werden müssen. Da der Basisblock B mit 50 Zyklen stärker zu der WCET beitragen würde, ermittelt aiT hier, dass der WCEP durch diesen Block läuft. Insgesamt ergibt sich also der WCEP als Basisblockfolge (A, B, D) mit einer WCET von 220 Zyklen $(100 + 50 + 70)$.

Durch Optimierungen auf diesem Pfad wird in Basisblock B eine Reduktion um 20 Zyklen

erreicht. Da nun der zum *else*-Pfad gehörende Basisblock C eine höhere WCET aufweist, findet ein Pfadwechsel statt: Der WCEP besteht wie in Abbildung 2.3b gezeigt, nun aus der Basisblockfolge (A, C, D). Insgesamt wurde dadurch, bedingt durch diesen Pfadwechsel, keine Reduktion der Gesamt-WCET um 20 Zyklen, sondern nur um 10 Zyklen erreicht.

Für eine erfolgreiche WCET-Minimierung sind also aktuelle WCET-Informationen notwendig, da weitere Optimierungen, die den Pfadwechsel nicht bemerkt haben, anschließend eventuell auf Pfaden arbeiten, die nicht (mehr) auf dem WCEP liegen, und somit nicht zu einer WCET-Reduktion beitragen können.

3 WCC

Die in dieser Arbeit implementierten Instruction Scheduling-Verfahren wurden in das am Lehrstuhl 12 der TU Dortmund entwickelte Compiler-Framework WCET-aware C Compiler (WCC) integriert. Der WCC ist ein Compiler für Eingebettete Realzeitsysteme, der in ANSI-C geschriebene Programme in Maschinencode für den Infineon TriCore TC1796 bzw. TC1797 übersetzt. Unterstützung für weitere Hardwareplattformen wie ARM7 befindet sich momentan in der (weit fortgeschrittenen) Entwicklung.

Wie bereits in der Einleitung erwähnt, spielt im Bereich der Eingebetteten Systeme, anders als bei klassischen PCs, häufig die Optimierung der WCET eine entscheidende Rolle. In den WCC wurde daher das WCET-Analysewerkzeug aiT der Firma AbsInt [Abs10] integriert. Dadurch ist es während des Kompilierungsprozesses möglich auf WCET-Informationen zuzugreifen. Dies ermöglicht erstmals automatische WCET-Optimierungen in einem Compiler.

In Kapitel 3.1 wird zunächst der Aufbau des WCC vorgestellt. Die in dieser Diplomarbeit entwickelten Instruction Scheduling-Verfahren stellen Low-Level Optimierungen dar. Daher steht die Beschreibung der Low-Level Repräsentation ICD-LLIR [Inf10d] im Vordergrund. In Kapitel 3.2 wird anschließend detaillierter auf die unterstütze TriCore Plattform eingegangen. Für das Instruction Scheduling ist dabei vor allem die Pipeline-Architektur des TriCore von Interesse (Kapitel 3.2.1).

3.1 Aufbau des WCC

Abbildung 3.1 stellt den grundlegenden Aufbau des WCC schematisch dar. Dabei deuten schwarze Pfeile den Kontrollfluss innerhalb eines normalen Compilers an. Gestrichelte Pfeile zeigen Bearbeitungsschritte innerhalb des WCC, die WCET-spezifisch sind und somit in herkömmlichen Compilern nicht vorkommen. Im Folgenden wird der WCC im Detail diskutiert.

Zunächst werden durch den *ICD-C Parser* alle Quelldateien in die High-Level Zwischendarstellung *ICD-C IR* [Inf10c] überführt. Die ICD-C IR (im Folgenden kurz IR) bietet zahlreiche integrierte, plattformunabhängige High-Level Optimierungen.

Aus der High-Level Darstellung erzeugt der am Lehrstuhl entwickelte *LLIR Code Selector* des WCC die maschinennahe Low-Level Darstellung ICD-LLIR (Informatik Centrum Dortmund - Low-Level Intermediate Representation, kurz: LLIR). Auch für die LLIR sind zahlreiche Optimierungen implementiert.

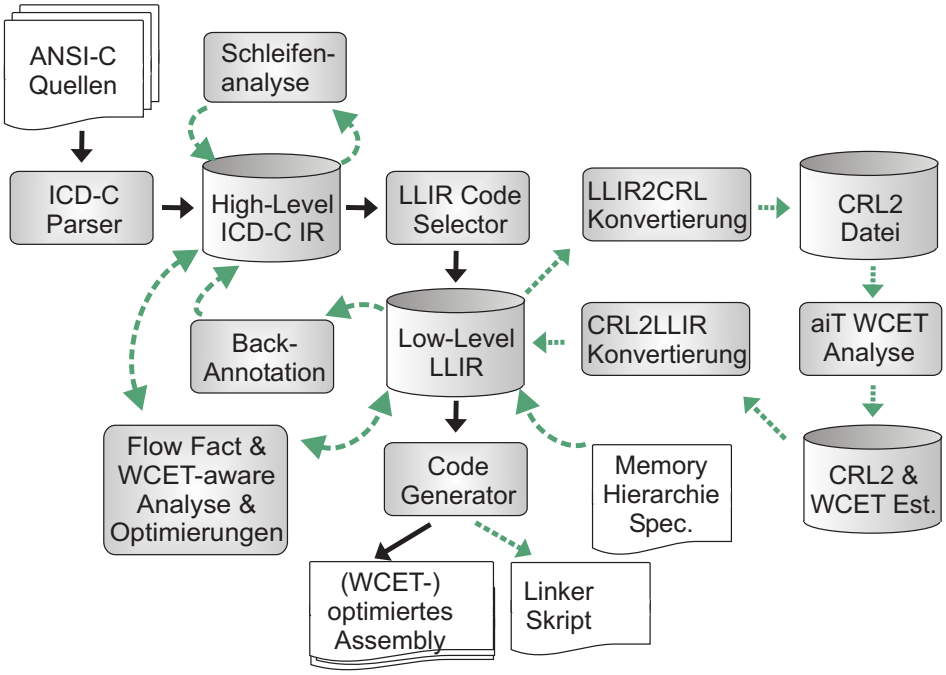


Abbildung 3.1: Struktur WCC

Das Binärprogramm für den TriCore wird durch einen Standard-Linker aus dem vom *Code Generator* erzeugten Assembler-Code und einem Linkerscript erzeugt.

Das in dieser Arbeit implementierte Instruction Scheduling stellt eine Low-Level Optimierung dar und arbeitet daher mit der Low-Level Darstellung LLIR. Kenntnisse der High-Level Darstellung ICD-C IR sind zum weiteren Verständnis dieser Arbeit nicht notwendig. Daher wird an dieser Stelle auf eine Vorstellung verzichtet. Nachfolgend wird genauer auf den Aufbau der LLIR eingegangen.

3.1.1 Aufbau ICD-LLIR

Die vom Code Selector erzeugte ICD-LLIR[Inf10d] stellt eine Assembler-nahe Zwischendarstellung des zu übersetzenden Programms dar. Abbildung 3.2 zeigt, wie ein Programm innerhalb der Klassenstruktur der LLIR repräsentiert werden kann. Dabei besteht das Programm im Beispiel aus einer einzelnen Quelldatei, die zwei Funktionen enthält. Folgende Klassen sind dargestellt:

LLIR: Diese Klasse repräsentiert eine einzelne Quelldatei des zu übersetzenden Programms. Sie verwaltet Referenzen auf Objekte vom Typ LLIR_Function.

LLIR_Function: Diese Klasse repräsentiert eine Funktion innerhalb des zu übersetzenden Programms. Der Kontrollfluss innerhalb einer Funktion wird durch Referenzen auf Basisblöcke des Typs LLIR_BB dargestellt.

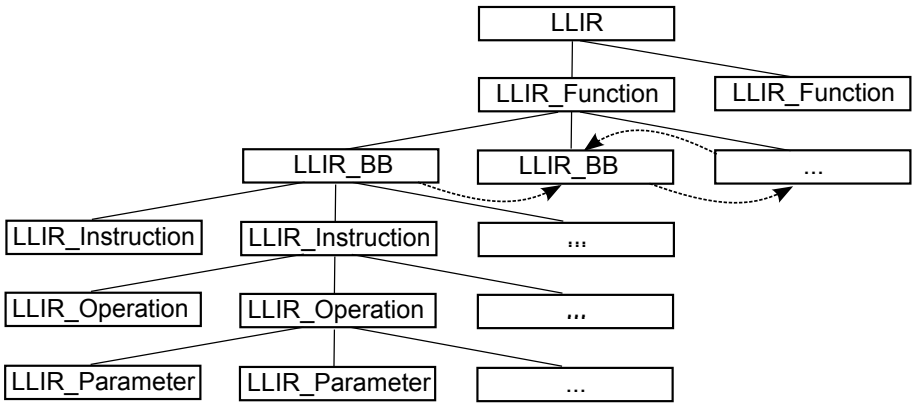


Abbildung 3.2: Darstellung eines Programms in ICD-LLIR

LLIR_BB: Objekte dieser Klasse repräsentieren einen Basisblock des Programms und verwalten eine geordnete Liste von hardwarenahen Anweisungen, die als Instanzen der Klasse LLIR_Instruction dargestellt werden. Der Kontrollfluss, der zwischen Basisblöcken möglich ist, ist in Abbildung 3.2 durch gestrichelte Pfeile dargestellt.

LLIR_Instruction: Objekten dieser Klasse sind ein oder mehrere Maschinenbefehle als Objekte der Klasse LLIR_Operation zugeordnet. Bei der LLIR für den TriCore enthält jedes Objekt dieser Klasse genau eine Referenz auf ein Objekt vom Typ LLIR_Operation. Wäre die Zielarchitektur ein VLIW-Prozessor, so wäre es möglich jedem Objekt dieser Klasse auch mehrere Referenzen auf unterschiedliche Objekte vom Typ LLIR_Operation zuzuordnen.

LLIR_Operation: Durch diese Klasse werden einzelne Maschinenbefehle der Zielarchitektur repräsentiert. Die benötigten Parameter eines Maschinenbefehls werden dabei durch Referenzen auf Objekte vom Typ LLIR_Parameter dargestellt.

LLIR_Parameter: Repräsentiert einen Parameter eines Maschinenbefehls. Es kann sich dabei um eine Integer-Konstante, ein Register, ein Label oder einen Operator handeln. Letztere dienen beispielsweise der Spezifikation von Adressierungsarten.

Erweiterbarkeit

Die auf der LLIR durchgeführten Optimierungen benötigen beziehungsweise erzeugen zahlreiche Informationen, die an geeigneter Stelle verwaltet werden müssen. Ein Beispiel für solche Informationen sind WCET-Werte, die von aiT erzeugt werden. Dabei kann es beispielsweise von Interesse sein, Daten für einen einzelnen Basisblock, für eine Funktion oder für eine komplette Kompilierungseinheit (LLIR) abzufragen. Um nicht für jedes neue Datum eine Änderung an den Datenstrukturen der LLIR vorzunehmen, wurde in der LLIR ein Konzept realisiert, das eine einfache Erweiterbarkeit ermöglicht. Wie in Abbildung 3.3 gezeigt, lassen sich beliebige Informationsquellen als Subklasse von LLIR_Objective darstellen und mittels eines Handlers an alle Objekte, die von der Superklasse LLIR_Tagged-Element erben, anhängen und abfragen.

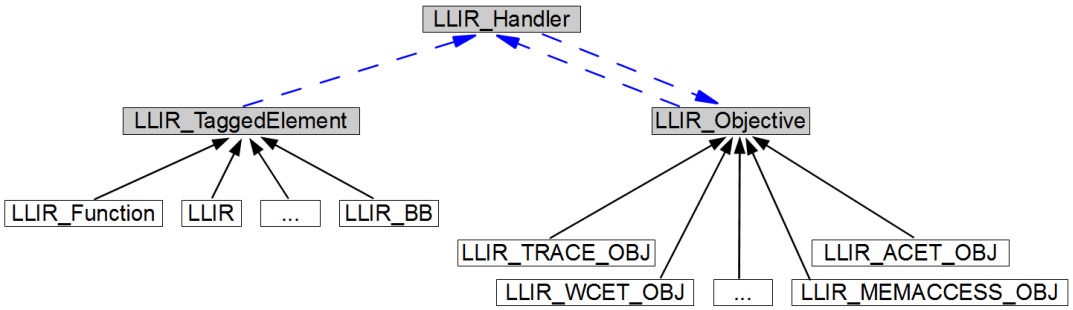


Abbildung 3.3: ICD-LLIR Handler, TaggedElements und Objectives

Im Folgenden sollen einige der vorhanden *Objectives*, die im Rahmen dieser Diplomarbeit eine Rolle spielen, kurz vorgestellt werden:

WCET: Objectives dieses Typs ermöglichen die Abfrage von WCET-Informationen, die durch aiT ermittelt wurden. Unter anderem kann auf die über alle Kontexte summierte Worst-Case Ausführungshäufigkeit ($WCEC_{sum}$) eines Objekts, die gesamte WCET über alle Kontexte ($WCET_{sum}$) oder die Anzahl an Worst-Case Instruction Cache-Misses eines Basisblocks zugegriffen werden. Weiterhin lässt sich ermitteln, ob der Pfad zwischen zwei Basisblöcken in dem analysierten Programm ausführbar (*feasible*) ist oder nicht.

ACET: ACET Objectives ermöglichen die Abfrage von durchschnittlichen Ausführungszeiten, die während eines Profilings durch die Simulationsplattform CoMET ermittelt wurden. Abfragbar sind unter anderem Informationen über die ermittelte durchschnittliche Ausführungszeit von Basisblöcken und Funktionen ($ACET_{sum}$) sowie Informationen über die durchschnittliche Ausführungshäufigkeit($ACEC_{sum}$).

MEMACCESS: Dieser Typ von Objectives speichert die von aiT per Abstrakter Interpretation ermittelten Intervalle von Speicherbereichen, auf die Lade- bzw. Speicherinstruktionen zugreifen. Dieses Objective wurde der LLIR im Rahmen dieser Diplomarbeit hinzugefügt (Kapitel 4.3), um bei der Aufstellung des Abhängigkeitsgraphen für das Instruction Scheduling unechte Abhängigkeiten zu erkennen. Ohne diese Informationen muss eine konservative Analyse zwischen allen Lade- und Speicherinstruktionen Datenabhängigkeiten erzeugen (Kapitel 4.2).

TRACE: Auch diese Art von Objectives wurde der LLIR innerhalb dieser Diplomarbeit hinzugefügt. Es dient der dezentralen Speicherung der durch die *Trace-Selektion* ermittelten Programmpfade (*Trace*), auf denen eine Optimierung durchgeführt werden soll. Mit Hilfe dieses Objectives kann für jeden Basisblock ermittelt werden, ob er Teil eines Trace ist. Mittels *getTraceSuccesor()* bzw. *getTracePredecessor()* kann auf Vorgänger- und Nachfolger-Basisblöcke eines Trace zugegriffen und so jederzeit der Trace rekonstruiert werden.

Unterschiede physikalische - virtuelle LLIR

Bei der Darstellung der LLIR kann zwischen der *virtuellen LLIR* und der *physikalischen LLIR* differenziert werden. Die virtuelle LLIR ist die Low-Level Darstellung des zu übersetzenden Programms vor der *Registerallokation*. In dieser Darstellung arbeiten alle Instruktionen mit *virtuellen Registern*. Diese stehen praktisch unbegrenzt zur Verfügung. In der Zielarchitektur ist die Anzahl *physikalischer Register* allerdings beschränkt. Bei der Transformation der virtuellen in die physikalische LLIR muss die Registerallokation daher eine Abbildung von virtuellen Registern auf physikalische Register durchführen. Ist die Anzahl gleichzeitig lebendiger virtueller Register dabei größer als die Anzahl physikalischer Register, müssen zusätzliche Lade- und Speicherinstruktionen eingefügt werden, um Registerinhalte in den Hauptspeicher ein- und auszulagern (*Spill Code*).

Im WCC können Low-Level Optimierungen für die virtuelle LLIR oder die physikalische LLIR implementiert werden. Die Unterschiede, die sich für Optimierungen ergeben, sollen anhand des lokalen und des Treegion-Scheduling dargestellt werden. Diese Optimierungen sind sowohl für die virtuelle, als auch für die physikalische LLIR implementiert.

In der virtuellen LLIR existieren durch die praktisch unbegrenzte Anzahl virtueller Register keine *unechten* Abhängigkeiten von Instruktionen. Solche Abhängigkeiten ergeben sich nur aufgrund der Wiederverwendung von Registern durch die Registerallokation. Die Möglichkeiten unabhängige Instruktionen zu finden sind daher für ein Instruction Scheduling vor der Registerallokation größer. Ein Instruction Scheduling auf der virtuellen LLIR kann also unter Umständen effektiver arbeiten als ein Instruction Scheduling auf der physikalischen LLIR.

Allerdings kann das Instruction Scheduling auf der virtuellen LLIR die Anzahl gleichzeitig lebendiger virtueller Register erhöhen. Damit erhöht sich auch der *Registerdruck*. Ein erhöhter Registerdruck kann dazu führen, dass die Registerallokation mehr Spill-Code erzeugen muss, um Registerinhalte ein- und auszulagern. Da die dazu benötigten Lade- und Speicherinstruktionen teuer sind, kann dies negative Auswirkungen auf die Laufzeit des zu übersetzenden Programms haben.

Das Instruction Scheduling nach der Registerallokation kann den durch die Registerallokation erzeugten Spill-Code berücksichtigen, und so negative Effekte der Lade- und Speicherinstruktionen eventuell abschwächen. Allerdings besteht bei diesem Instruction Scheduling die bereits erwähnte Einschränkung durch unechte Abhängigkeiten.

Die in dieser Diplomarbeit implementierten globalen Scheduling-Verfahren benötigen Informationen über die WCET (bzw. ACET). Da eine Ermittlung dieser Werte auf einer LLIR, die virtuelle Register benutzt, nicht möglich ist, können diese Verfahren nur als physikalische Optimierungen implementiert werden. Die Problematik, durch unechte Abhängigkeiten in der physikalischen LLIR nicht genügend unabhängige Instruktionen finden zu können, werden bei diesen Verfahren durch die globale Betrachtung des Programms entschärft.

3.1.2 WCET-Erweiterungen

Im Folgenden sollen die in Abbildung 3.1 gezeigten Erweiterungen *Flow Facts*, Schleifenanalyse sowie die Integration von aiT in den WCC kurz vorgestellt werden. Diese Erweiterungen sind WCET spezifisch und daher in einem herkömmlichen Compiler nicht vorhanden.

Flow Facts

Wie bereits in Kapitel 2.3 (Seite 14) angedeutet, benötigt die WCET-Analyse Informationen über die Ausführungshäufigkeit von Schleifen. Eine automatische Bestimmung der Ausführungshäufigkeit ist allerdings im Allgemeinen nicht möglich. Daher benötigt die WCET-Analyse an dieser Stelle weitere, durch den Benutzer bereitgestellte Informationen.

Im WCC können diese Informationen komfortabel als Annotation sogenannter Flow Facts direkt im C-Code des zu übersetzenden Programms angegeben werden. Dabei können für reguläre Schleifen, wie in Listing 3.1 gezeigt, *Loopbounds* annotiert werden, die die minimale und maximale Ausführungshäufigkeit von regulären Schleifen angeben.

```
Pragma( "loopbound min 10 max 10" )
  for ( Index = 1; Index <= 10; Index++ )
    Array[ Index ] = 1;
```

Listing 3.1: Loopbounds

Bei irregulären Schleifen lässt sich mittels Flussbeschränkungen (*Flowrestrictions*) das Verhältnis zwischen den Ausführungshäufigkeiten zweier Blöcke global beschränken.

Schleifenanalyse

Im WCC ist eine ausgefeilte Analyse integriert, die eine Vielzahl von Schleifen erkennen und mittels Abstrakter Interpretation, *Interprocedural Program Slicing* und *Polyhedral Loop Evaluation* [LCFM09] selbständig Flow Facts erzeugen kann, so dass dies nicht mehr vom Benutzer im Quellcode geschehen muss. Für Rekursionsschranken ist allerdings weiterhin eine manuelle Annotation notwendig.

aiT-Integration

Die Analyse der WCET mittels aiT ist ein integraler Bestandteil des WCC. Üblicherweise nimmt aiT ein ausführbares Binärprogramm zur Analyse entgegen und transformiert es in die interne Darstellung CRL2 (Kapitel 2.4). Die CRL2 ist, genau wie die LLIR, eine Low-Level Darstellung des Kontrollflusses des Programms, bestehend aus Funktionen und Basisblöcken, die mittels Kanten verbunden sind. Innerhalb des WCC ist es mittels eines

Konverters (*LLIR2CRL*) möglich, eine Transformation der LLIR in die CRL2 vorzunehmen und diese direkt an aiT zu übergeben.

Im Verlauf der Analyse wird der durch die CRL2 repräsentierte CFG mit WCET-Daten angereichert. Um diese Information im WCC nutzen zu können, erfolgt nach der WCET-Analyse eine Rückübertragung der Informationen in die LLIR mittels eines weiteren Konverters (*CRL2LLIR*). Innerhalb der LLIR werden die gewonnenen Daten durch die in Kapitel 3.1.1 (Seite 21) vorgestellten WCET Objectives repräsentiert und sind somit für folgende Optimierungen nutzbar.

3.2 TriCore Plattform

Die Kenntnis der unterstützten Zielhardware ist für die Low-Level Optimierung Instruction Scheduling unerlässlich. Daher erfolgt in diesem Kapitel eine Beschreibung der TriCore-Architektur.

Die Prozessoren Infineon TriCore TC1796 beziehungsweise TC1797 stellen die Zielplattform des WCC dar. Bei beiden handelt es sich um Prozessoren, die sowohl Eigenschaften eines DSPs, wie die Unterstützung von Multiply-Accumulate Instruktionen (*MAC*), Sättigungsarithmetik und Packed Operations, als auch Eigenschaften einer RISC CPU, wie mehrere unabhängige Ausführungspipelines, aufweisen.

Die meisten Instruktionen der TriCore-Architektur können in einem einzigen Zyklus abgearbeitet werden. Eine Ausnahme stellen Multiply-Accumulate Instruktionen dar, bei der die Execute-Phase der Pipeline aus zwei Schritten besteht, so dass diese Instruktionen eine Latenz von zwei Zyklen aufweisen.

Fließkommainstruktionen werden bei dem TC1797 auf einer dedizierten Fließkommaeinheit ausgeführt, die über einen Coprozessor-Port an die Pipelines angebunden ist. Die Latenzzeiten für diese Instruktionen variieren zwischen zwei und fünf Zyklen für einfache Instruktionen und 16 Zyklen für die Division. Während der Ausführung einer Fließkommainstruktion ist keine parallele Ausführung anderer Instruktionen möglich.

Weitere Merkmale der Architektur sind ein Scratchpad-Speicher, sowie die automatische Sicherung eines Teils der Register bei einem Funktionsaufruf. Bei den gesicherten Registern handelt es sich um die sogenannten *oberen Kontext-Register* A[10] bis A[15] (Adressregister), D[8] bis D[15] (Datenregister) sowie den Spezialregistern PSW und PCXI. Die Sicherung der Registerinhalte erfolgt in der *Context Save Area* (CSA). Die restlichen General Purpose Register stellen den *unteren Kontext* dar und haben nach der Rückkehr aus einer aufgerufenen Funktion keinen definierten Inhalt. Bei einem Rücksprung aus einer Funktion mittels der `ret` Instruktion wird der Inhalt der oberen Kontext-Register automatisch aus der CSA wiederhergestellt. Diese hardwareseitige Unterstützung erleichtert die Umsetzung von Funktionsaufrufen, da ein explizites Sichern von Registern mittels `mov` Instruktionen nicht notwendig ist.

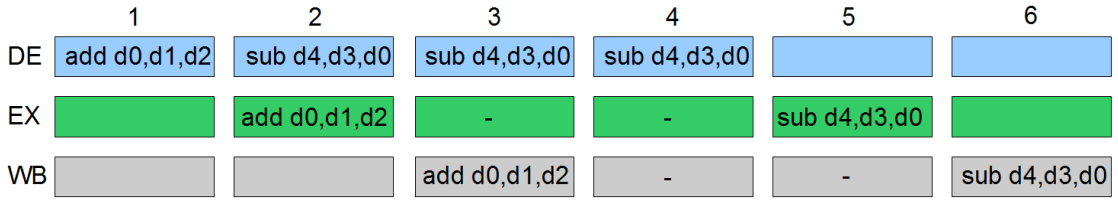
Von besonderem Interesse für die Implementierung eines Instruction Scheduling ist die

Funktionsweise der Pipeline-Architektur. Im Folgenden wird daher der Aufbau der Pipelines des TriCore sowie die Einschränkungen, die bei der Verarbeitung von Instruktionen beachtet werden müssen, erläutert.

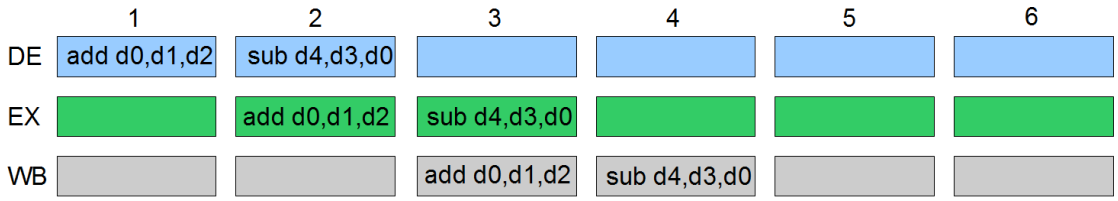
3.2.1 TriCore-Pipelines

Die TriCore-Architektur besitzt zwei reguläre Instruktions-Pipelines: die *Integer-Pipeline (IP-Pipeline)* und die *Load-Store-Pipeline (LS-Pipeline)*. Außerdem ist eine dritte Pipeline vorhanden, mit deren Hilfe sich *Zero-Overhead*-Schleifen realisieren lassen *(Loop-Pipeline)*.

Die Architektur weist einen *Pipeline Interlock* auf, der Konflikte (*Hazards*) bei der parallelen Ausführung von Instruktionen erkennen kann. Durch ein Verzögern der Ausführung am Beginn der Pipeline können solche Konflikte automatisch aufgelöst (*Stall*) werden. Daher ist es nicht notwendig *NOP*-Instruktionen durch den Compiler zu erzeugen.



(a) Ohne EX → EX Forwarding



(b) Mit EX → EX Forwarding

Abbildung 3.4: Vergleich Pipeline Forwarding

Die Pipelines des TriCore bearbeiten Instruktionen in vier Stufen: Instruction Fetch (FE), Decode (DE), Execute (EX) und Writeback (WB) und implementieren dabei ein ausgeklügeltes *Forwarding-System*. Dadurch lassen sich zahlreiche Konflikte aufgrund von Datenabhängigkeiten entschärfen. So ist es einer nachfolgenden Instruktion durch das EX → EX Forwarding möglich, direkt auf das Ergebnis einer unmittelbar vorher ausgeführten Instruktion zuzugreifen. Dies ist in Abbildung 3.4 für die folgende Sequenz zweier Instruktionen dargestellt:

```
sub d4, d3, d0;
add d0, d1, d2;
```

Die zweite Instruktion liest den von der vorhergehenden Instruktion `sub d4, d3, d0` berechneten Wert des Register `d0`.

Abbildung 3.4a stellt eine Pipeline dar, die kein Forwarding berechneter Werte unterstützt. Hier muss die Instruktion `sub d4, d3, d0` zwei Taktzyklen warten, da das Ergebnis der Instruktion `add d0, d1, d2` erst in der Writeback-Phase in das Register `d0` geschrieben wird. Die gesamte Ausführung benötigt daher sieben Zyklen.

In Abbildung 3.4b ist dieselbe Pipeline mit EX → EX Forwarding dargestellt. Hier kann das Ergebnis von `add d0, d1, d2` direkt an die Execute-Phase von `sub d4, d3, d0` weitergeleitet werden, und ein Umweg über das Register `d0` vermieden werden. Da so keine Stalls notwendig sind, verkürzt sich die Ausführungszeit auf vier Zyklen.

Auch für andere Pipeline-Stufen wie EX → DE ist in der TriCore-Pipeline ein Forwarding möglich. Eine einfache Sprungvorhersage (*Branch Prediction*) minimiert des Weiteren die Auswirkung von Kontrollflusskonflikten. Ohne diese Sprungvorhersage müssten bei der Ausführung bedingter Sprünge die Pipelines angehalten werden, bis feststeht, ob der Sprung genommen wird oder nicht.

Für das Instruction Scheduling lassen sich die Instruktionen der TriCore-Architektur, abhängig von der Pipeline auf denen sie ausgeführt werden, in vier Kategorien klassifizieren [Inf10a].

Integer-Pipeline (IP) Instruktionen: IP-Instruktionen führen logische und arithmetische Operationen auf Datenregistern aus.

Load-Store-Pipeline (LP) Instruktionen: LS-Instruktionen sind Lade- und Speicherinstruktionen, sowie arithmetische und logische Operationen auf Adressregistern. Auch unbedingte Sprünge und bedingte Sprünge, die Tests auf Adressregistern durchführen, werden auf der LS-Pipeline ausgeführt.

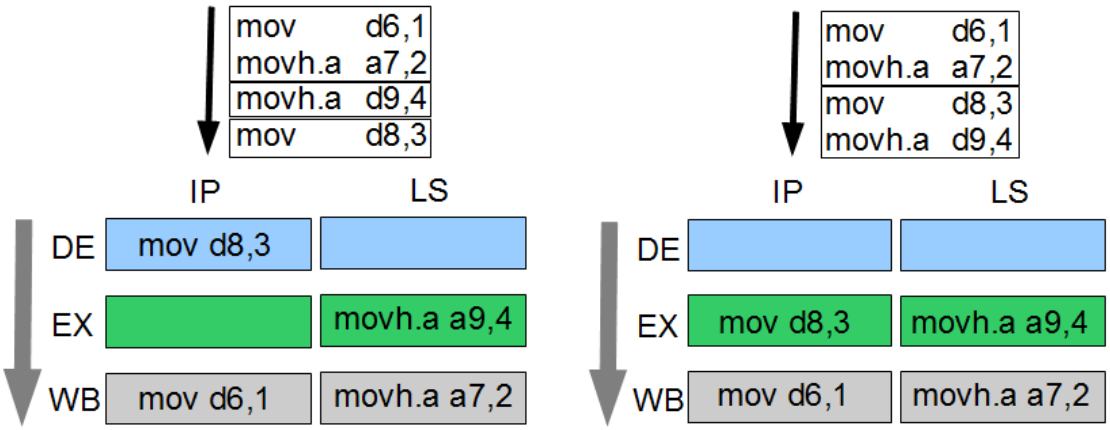
Dual-Pipeline (DP) Instruktionen: DP-Instruktionen sind Instruktionen, die gleichzeitig die IP- und LS-Pipeline belegen. Ein Beispiel dafür ist die `addsc.a` Instruktion, die Ressourcen beider Pipelines benötigt. Andere DP-Instruktionen benötigen zwar nicht tatsächlich Ressourcen beider Pipelines, um die Implementierung zu vereinfachen, werden sie beim TC1797 dennoch als DP-Instruktionen ausgeführt. Ein Beispiel dafür sind bedingte Sprünge, die von einem Datenregister abhängen.

Loop-Pipeline Instruktionen: Diese Kategorie besteht aus speziellen bedingten (`loop`) und unbedingten Schleifeninstruktionen (`loop.u`). Diese Instruktionen ermöglichen durch die Hardware-Unterstützung des TriCore eine effiziente Implementierung von Schleifenkonstrukten. Beide Instruktionen werden auf der Loop-Pipeline ausgeführt und benötigen dort nur beim Betreten und Verlassen der Schleife jeweils einen Zyklus. Die weiteren Iterationen der Schleife können ohne den Overhead zusätzlicher Sprunginstruktionen durchgeführt werden.

Da das in dieser Diplomarbeit eingesetzte Instruction Scheduling immer auf schleifenfreiem Code oder innerhalb einer umgebenden Schleife durchgeführt wird, und `loop` bzw. `loop.u` Instruktionen ausschließlich am Ende einer Schleife vorkommen, hat die Loop-Pipeline für das Scheduling keine besondere Bedeutung. Daher werden im Folgenden ausschließlich die IP- und LS-Pipeline betrachtet.

Dual Issue

Alle drei Pipelines der TriCore-Architektur teilen sich eine gemeinsame Fetch-Einheit, die in der Lage ist, innerhalb eines Zyklus sowohl die IP- als auch simultan die LS-Pipeline mit Instruktionen zu versorgen (*Dual Issue*). Im Idealfall – wenn eine ausgewogene Anzahl unabhängiger IP- und LS-Instruktionen zur Verfügung steht – wäre es daher möglich einen *IPC-Wert* (Instructions per Cycle) von zwei zu erreichen.



(a) 1 IP/LS-Bündel (b) 2 IP/LS-Bündel

Abbildung 3.5: Auswirkung der Instruktionsreihenfolge auf die Ausführungszeit

Eine Einschränkung der Instruction-Fetch Einheit ist allerdings, dass das gleichzeitige Laden von Instruktionen in die Pipelines (*issue*) nur möglich ist, wenn im Maschinencode eine LS-Instruktion auf eine IP-Instruktion folgt. Abbildung 3.5 verdeutlicht dies anhand eines einfachen Beispiels. Dargestellt sind jeweils die Decode-, Execute-, und Writeback-Stufen der LS- und IP-Pipeline. Da die Instruction Fetch-Stufe nur einmal vorhanden ist, ist eine Darstellung hier nicht notwendig.

In Abbildung 3.5a folgt auf eine `mov` Instruktion, die auf der IP-Pipeline ausgeführt wird, zunächst eine `movh.a` Instruktion, die auf der LS-Pipeline ausgeführt wird. Die Instruction Fetch Einheit kann daher in einem Zyklus ein *IP/LS-Bündel* (kurz Bündel) in die Pipeline laden. Die nächste Instruktion, die der Fetch-Stufe zur Verfügung steht, ist die LS-Instruktion `movh.a d9, 4`. Die nachfolgende IP-Instruktion `mov d8, 3` wäre prinzipiell gleichzeitig ausführbar. Da die beiden Instruktionen aber nicht in der geforderten Reihenfolge (IP-Instruktion vor LS-Instruktion) im Maschinencode vorkommen, muss die LS-Instruktion einzeln in die Pipeline geladen werden. Die IP-Instruktion kann dadurch erst im nächsten Zyklus geladen und einzeln ausgeführt werden.

In Abbildung 3.5b ist der Code semantisch äquivalent, aber in einer für die TriCore-Architektur günstigeren Reihenfolge im Maschinencode angeordnet: Hier folgt auf eine IP-Instruktion jeweils eine LS-Instruktion, so dass zwei Bündel gebildet werden können. Es muss also nicht, wie in Abbildung 3.5a gezeigt, eine LS-Instruktion und eine IP-Instruktion einzeln ausgeführt werden. Dadurch verkürzt sich in diesem Beispiel die Ausführungszeit um einen Zyklus.

Primäres Ziel eines Instruction Scheduling für die TriCore Plattform sollte daher die Bildung möglichst vieler Bündel sein, um die Hardware-Parallelität der Pipelines maximal auszunutzen.

4 Instruction Scheduling

Instruction Scheduling-Verfahren können nach lokalen und globalen Ansätzen differenziert werden. Während lokale Verfahren jeden Basisblock des Kontrollflussgraphen separat betrachten, arbeiten globale Verfahren gleichzeitig mit Instruktionen mehrerer Basisblöcke. Da globale Instruction Scheduler auf lokalen Verfahren basieren, gibt es zahlreiche Gemeinsamkeiten. Dieses Kapitel stellt einige wichtige Begriffe vor und liefert somit die Grundlagen für die weiteren Kapitel.

Zunächst wird in Kapitel 4.1 auf die unterschiedlichen Arten von Abhängigkeiten, die zwischen Instruktionen herrschen können, eingegangen. Anschließend wird in Kapitel 4.2 die Erzeugung eines *Abhängigkeitsgraphen* beschrieben. Ein konservativer Ansatz muss hierbei davon ausgehen, dass zwischen allen Lade- und Speicherinstruktionen eine Abhängigkeit besteht. In Kapitel 4.3 wird gezeigt, wie innerhalb des WCC ein Verfahren eingesetzt werden kann, das unnötige Abhängigkeiten verhindert.

Basierend auf dem Abhängigkeitsgraphen ist es möglich ein Instruction Scheduling mittels des bekannten *List Scheduling*-Algorithmus durchzuführen. Die Umsetzung eines List Schedulers im WCC ist abschließend in Kapitel 4.4 beschrieben.

4.1 Instruktionsabhängigkeiten

Ein Programm P lässt sich als Kontrollflussgraph CFG von Basisblöcken darstellen. Ein einzelner Basisblock bb_1 dieses CFG lässt sich wiederum als Sequenz der enthaltenen Instruktionen $bb_1 := (i_1, i_2, ..., i_e)$ darstellen. Wenn der Kontrollfluss während der Programmausführung zu Basisblock bb_1 wechselt, führt der Prozessor die in bb_1 enthaltenen Instruktionen sequentiell aus.

Eine exakte Abarbeitung in der gegebenen Instruktionsreihenfolge ist allerdings nur erforderlich, wenn zwischen den Instruktionen Abhängigkeiten bestehen. Unabhängige Instruktionen lassen sich ohne Auswirkung auf die Programmsemantik auch in einer anderen Reihenfolge ausführen. Diese Tatsache bildet die Grundlage des Instruction Scheduling:
Mittels eines Instruction Scheduling wird versucht, unter Beachtung aller Abhängigkeiten zwischen Instruktionen, eine möglichst gute Instruktionsreihenfolge (*Schedule*) für die Zielhardware zu ermitteln. Dabei sind unterschiedliche Kriterien zur Bewertung der Güte eines Schedules, wie Energieeffizienz oder durchschnittliche Ausführungszeit, möglich.

Im Folgenden werden die Abhängigkeiten, die zwischen zwei Instruktionen i1 und i2 bestehen können, definiert.

Datenabhängigkeit: Eine Instruktion i2 ist datenabhängig von einer im Kontrollfluss vorhergehenden Instruktion i1, falls i2 einen Operanden (Register- oder Speicheroperand) verwendet, der von i1 geschrieben wird.

```
mov d4, 10;
add d5, d4, 10
```

In diesem Beispiel ergibt sich eine Datenabhängigkeit (*data dependence*) der zweiten von der ersten Instruktion, da `add d5, d4, 10` das Register `d4` liest, das von `mov d4, 10` geschrieben wird. Ein Instruction Scheduling dieser Instruktionen muss zwingend in dieser Reihenfolge geschehen, da sich sonst die Semantik des Programms ändern könnte.

Durch die Verwendung indirekter Adressierung und durch *memory aliasing* wird die Erkennung von Datenabhängigkeiten, die sich aufgrund von Speicheroperanden ergeben, zur Kompilierungszeit aufwändig bis unmöglich. Memory Aliasing tritt dabei immer dann auf, wenn im Quellcode des zu übersetzenden Programms über mehrere *Pointer* auf die gleiche Speicheradresse zugegriffen wird. In einer konservativen Analyse muss daher davon ausgegangen werden, dass jede Ladeinstruktion von jeder vorhergehenden Speicherinstruktion datenabhängig ist. Das folgende Beispiel deutet die Schwierigkeiten bei der Bestimmung von Abhängigkeiten im Zusammenhang mit Speicheroperanden an:

```
movh.a a3, 512;
movh.a a4, 512;
st.w a3, d1;
ld.w d2, a4;
```

Die Speicherinstruktion beschreibt das Register `a3`, während die Ladeinstruktion das Register `a4` liest. Obwohl hier unterschiedliche Adressregister verwendet werden, besteht dennoch eine Datenabhängigkeit: Die Register `a3` und `a4` verweisen auf den gleichen Speicherbereich. In vielen Fällen sind die adressierten Speicherbereiche allerdings disjunkt, so dass tatsächlich keine Abhängigkeit existiert.

Wie in Kapitel 4.3 noch gezeigt, kann die Werte-Analyse von aiT im WCC genutzt werden um disjunkte Speicherzugriffe zu erkennen. Somit kann die Anzahl erkannter Abhängigkeiten reduziert werden.

Gegenabhängigkeit: Eine Instruktion i2 ist gegenabhängig von einer im Kontrollfluss vorhergehenden Instruktion i1, wenn i2 einen Operanden schreibt, der von i1 gelesen wird.

```
add d5, d4, 10
mov d4, 10;
```

Eine Ausführung der zweiten Instruktion kann nicht vor der Ersten erfolgen. Anders als bei der Datenabhängigkeit, bei der ein gültiges Schedule die Reihenfolge der Instruktionen zwingend einhalten muss, ist die Gegenabhängigkeit (*anti-dependence*) eine *falsche Abhängigkeit*: i1 und i2 erscheinen nur aufgrund der Wiederverwendung des gleichen Registers `d4` als abhängig.

Wie das folgenden Beispiel zeigt, sind auch bei der Bestimmung der Gegenabhängigkeit Speicheroperanden problematisch:

```
movh.a a3, 512;
movh.a a4, 1024;
ld.w d2, a4;
st.w a3, d1;
```

Eine konservative Analyse muss von einer Gegenabhängigkeit zwischen jeder Ladeinstruktion und jeder nachfolgenden Speicherinstruktion ausgehen. In diesem Beispiel arbeiten die Lade- und die Speicherinstruktion offensichtlich auf disjunkten Speicherbereichen. Ein Verschieben der Speicherinstruktion vor die Ladeinstruktion wäre also möglich. Das in Kapitel 4.3 vorgestellte Verfahren ist in der Lage solche Situationen zu erkennen.

Ausgabeabhängigkeit: Zwei Instruktionen i1 und i2 sind ausgabeabhängig, wenn sie das gleiche Register beziehungsweise die gleiche Speicheradresse beschreiben. In einer konservativen Analyse sind alle Speicherinstruktionen als ausgabeabhängig zu betrachten. Das folgende Beispiel zeigt eine Ausgabeabhängigkeit (*output dependence*) bezüglich Register `d3`.

```
mov d3, 10;
ld.w d3, a4;
```

Genau wie die Gegenabhängigkeit ist die Ausgabeabhängigkeit eine falsche Abhängigkeit, die nur aufgrund der Wiederverwendung von Registern entsteht.

Kontrollflussabhängigkeit: Die Kontrollflussabhängigkeit zweier Instruktionen spielt vor allem beim globalen Scheduling eine Rolle. Kontrollflussabhängigkeiten stellen sicher, dass der Kontrollfluss des Programms durch das Instruction Scheduling nicht verändert wird. So besteht beispielsweise zwischen zwei Sprunganweisungen eine Kontrollflussabhängigkeit.

4.2 Abhängigkeitsgraph

Die Abhängigkeiten die zwischen Instruktionen bestehen können, lassen sich in einem Abhängigkeitsgraphen darstellen.

Definition 4.1 (Abhängigkeitsgraph).
Für eine Instruktionsfolge $I := (ins_1, ..., ins_n)$ *lassen sich die Abhängigkeiten, die zwischen zwei Instruktionen* $ins_i, ins_j \in I$ *bestehen, in einem gerichteten, azyklischen* ***Abhängigkeitsgraphen*** *G=(V,E) (kurz: DAG für* directed acyclic graph*) darstellen. Dabei repräsentieren Knoten* $e \in E$ *Instruktionen aus* I*, während Abhängigkeiten durch gerichtete Kanten* $v \in V$ *dargestellt werden.*

Der Abhängigkeitsgraph lässt sich sowohl in einem *forward pass*, als auch in einem *backward pass* Verfahren aufbauen. Nachfolgend wird kurz ein *forward pass* Verfahren, das auch im WCC verwendet wird, beschrieben.

Im Scheduler des WCC iteriert das Verfahren zur Erzeugung des DAG über alle n Instruktionen der Instruktionsfolge $I := (ins_1, ..., ins_n)$ und erzeugt für die aktuell betrachtete Instruktion $ins_j \in I$ einen Knoten im DAG. Anschließend wird ins_j auf Abhängigkeiten zu allen in der Sequenz vorhergehenden Instruktionen $ins_i \in (ins_1, ..., ins_{j-1})$ untersucht. Falls eine Abhängigkeit zwischen ins_i und ins_j besteht, wird im DAG eine Kante zwischen den entsprechenden Knoten erzeugt. Bei einer Instruktionsfolge mit n Instruktion benötigt dieses Verfahren demnach $\mathcal{O}(n^2)$ Schritte.

Als zusätzliche Information wird an jeder Abhängigkeitskante annotiert, wie viele Zyklen mindestens vergehen müssen, bevor eine abhängige Instruktion ins_j in die Pipeline geladen werden kann, nachdem ins_i in die Pipeline geladen wurde (*Latenz*). In der TriCore-Architektur beträgt die Latenz für Instruktionen einer Instruktionsklasse, also von Instruktionen, die auf der gleichen Pipeline ausgeführt werden, für die meisten Instruktionen einen Zyklus. Eine abhängige Instruktion kann demnach im nächsten Zeitschritt in die Pipeline geladen werden. Eine Ausnahme davon stellen die wenigen *Multizyklen-Instruktionen*, wie die in Kapitel 3.2 (Seite 25) angesprochenen Multiply Accumulate Instruktionen, dar. Bei diesen Instruktionen müssen vor der Ausführung einer abhängigen Instruktion mehrere Zyklen vergehen.

Ein Problem des im WCC genutzen forward pass Verfahrens ist die hohe Anzahl redundanter Kanten, die im DAG erzeugt werden.

Redundante DAG-Kanten

Das folgende Beispiel deutet die Problematik redundanter Kanten innerhalb des Datenabhängigkeitsgraphen an.

```
i1: mov d8, 1;
i2: ... // weitere Instruktionen
i3: mov d8, 2;
i4: ... // weitere Instruktionen
i5: mov d8, 3;
```

Die Instruktionen i1 und i3 sind offensichtlich bezüglich des Registers `d8` ausgabeabhängig[1]. Auch i3 und i5 sind ausgabeabhängig. Da das verwendete Verfahren alle möglichen Abhängigkeiten erzeugt, wird auch eine Abhängigkeitskante zwischen i1 und i5 erzeugt. Da allerdings bereits die Abhängigkeiten zwischen i1 und i3 sowie zwischen i3 und i5 sicherstellen, dass in einem gültigen Schedule i5 niemals vor i1 ausgeführt werden kann, ist diese Kante redundant.

Während in diesem einfachen Beispiel redundante Kanten offensichtlich leicht erkennbar sind, ist dies bei realen Programmen deutlich aufwändiger. Nachdem der List Scheduler des WCC eine Instruktion selektiert hat, ist es notwendig zu bestimmen, ob dadurch weitere Instruktionen ausführbar geworden sind. Dazu müssen innerhalb des DAG für alle Instruktionen alle eingehenden Kanten auf unerfüllte Abhängigkeiten überprüft werden.

[1] In diesem Beispiel sind zwischen den `mov` Instruktionen weitere Instruktionen angedeutet, da die ersten beiden `mov` Instruktionen ansonsten tot wären.

Die Komplexität und damit die Laufzeit dieser Überprüfung steigt demnach unnötig mit der Anzahl erzeugter Kanten.

Aus diesem Grund wird im WCC nach der Erzeugung der Abhängigkeiten ein von Wilken et al. [WLH00] beschriebenes Verfahren durchgeführt, dass die Länge der kritischen Pfade zwischen zwei Instruktionen berücksichtigt, um redundante Kanten zu eliminieren. Auf diesen Algorithmus wird hier nicht weiter eingegangen.

Im nächsten Kapitel wird das Verfahren zur Reduktion falscher Abhängigkeiten zwischen Instruktionen, die auf den Speicher zugreifen, vorgestellt.

4.3 Vermeidung von Speicherabhängigkeiten

Wie bereits in Kapitel 4.2 angesprochen, muss eine konservative Analyse bei der Aufstellung des Abhängigkeitsgraphen Kanten zwischen allen Lade- und Speicherinstruktionen erzeugen. Die *Werte-Analyse* von aiT kann mittels Abstrakter Interpretation in vielen Fällen den Speicherbereich einer Lade- oder Speicherinstruktion genau bestimmen oder zumindest auf mögliche Bereiche eingrenzen. Wenn sicher ist, dass zwei Instruktionen immer auf disjunkte Speicherbereiche zugreifen, ist eine Erzeugung von Abhängigkeitskanten nicht notwendig. Dadurch werden die Möglichkeiten des Schedulers, unabhängige Instruktionen zu finden, vergrößert. Es erscheint daher sinnvoll, die von aiT ermittelten Informationen bei der Aufstellung des Abhängigkeitsgraphen zu nutzen.

Im Rahmen dieser Arbeit wurde der LLIR dafür das bereits in Kapitel 3.1.1 (Seite 21) kurz vorgestellte MEMACCESS Objective hinzugefügt. Dieses Objective speichert mögliche Speicherzugriffsbereiche einer Instruktion als Intervall [min, max]. Dabei geben min und max die möglichen Startadressen eines Speicherzugriffs an. Diese Werte werden von aiT während der Werte-Analyse ermittelt und vom CRL2LLIR Konverter in die Objectives übertragen. Ist ein Zugriff statisch exakt bestimmbar, sind min- und max-Werte identisch. Durch die Verwendung von Kontexten in aiT sind dabei für jede Instruktion mehrere Intervalle möglich, wie im folgenden Beispiel gezeigt[2].

```
int my_array[5] = { 1, 2, 3, 4, 5 };

int accessArray(int index){
  return my_array[index];
}

int main(void) {
  int i1 = my_array(1);
  int i2 = my_array(3);
  return i1+i2;
}
```

Der Zugriff auf das Array `my_array` erfolgt innerhalb der Funktion `accessArrray(...)`

[2]Für ein einfacheres Verständnis ist dieses Beispiel in C-Code angegeben. Die tatsächliche Analyse von aiT arbeitet, wie gezeigt, auf einer Low-Level Darstellung.

mittels einer `ld_w` Instruktion. In diesem einfachen Beispiel kann aiT die Speicherbereiche, die dabei gelesen werden, statisch bestimmen. Da `accessArray(...)` in zwei Kontexten aufgerufen wird (einmal für den Parameter 1 und einmal für Parameter 3), könnten dabei beispielsweise die nachfolgend angegebenen Speicherbereiche ermittelt worden sein.

```
Read Interval: [3221225472...3221225472] // my_array[1]
Read Interval: [3221225480...3221225480] // my_array[3]
```

Wäre die Anzahl betrachteter Kontexte zu klein (also in diesem Beispiel nur ein einzelner Kontext), so könnte nicht mehr zwischen den einzelnen Zugriffen differenziert werden. Das von aiT bestimmte Intervall sehe somit wie folgt aus: `[3221225472...3221225480]`. In diesem Fall ist der Zugriff demnach nicht mehr exakt bekannt, der mögliche Bereich konnte aber immerhin stark eingeschränkt werden.

Die Informationen der MEMACCESS Objectives werden während der Aufstellung des DAG genutzt, um nur dann Abhängigkeiten zwischen Instruktionen, die auf den Speicher zugreifen, zu erzeugen, wenn diese überlappende Speicherbereiche betreffen. Algorithmus 4.1 zeigt die Umsetzung im WCC als Pseudocode. Wenn eine Instruktion i1 und eine nachfolgende Instruktion i2 auf überlappende Speicherbereiche zugreifen, muss der Code zwischen vier möglichen Fällen differenzieren:

- Wenn i1 eine Speicherinstruktion und i2 eine Ladeinstruktion ist, besteht zwischen diesen Instruktionen eine Datenabhängigkeit.
- Wenn i1 eine Ladeinstruktion und i2 eine Speicherinstruktion ist, besteht zwischen diesen Instruktionen eine Gegenabhängigkeit.
- Wenn i1 und i2 Speicherinstruktionen sind, besteht zwischen diesen Instruktionen eine Ausgabeabhängigkeit,
- Aufeinanderfolgende Lesezugriffe auf den gleichen Speicherbereich führen nicht zu einer Abhängigkeit. Demnach besteht zwischen zwei Ladeinstruktionen i1 und i2 keine Abhängigkeit.

4.4 List Scheduling

Nachdem der Abhängigkeitsgraph aufgestellt ist, kann das bekannte List Scheduling Verfahren eingesetzt werden, um ein Schedule der betrachteten Instruktionsfolge zu erzeugen. Die Umsetzung dieses Verfahrens im WCC ist als Pseudocode in Algorithmus 4.2 dargestellt.

Als *Greedy*-Algorithmus selektiert dieses Verfahren unter allen Instruktionen, die im jeweils betrachten Zeitschritt ausführbar sind (Zeile 6), eine Instruktion höchster Priorität. Ausführbar (*ready*) sind dabei Instruktionen, deren Vorgänger alle bereits vom List Scheduler ausgewählt wurden, so dass deren Ergebnisse im aktuellen Zeitschritt zur Verfügung stehen. Informationen über ausführbare Instruktionen werden in der *Ready List* vorgehalten und ständig aktualisiert.

Algorithmus 4.1 Bestimmung der Lade-/Speicherabhängigkeiten

Eingabe: Lade-/Speicherinstruktionen *i1*, *i2* mit *i2* ist Nachfolger von *i1* im CFG
Ausgabe: Abhängigkeit (*i1*, *i2*) bezüglich Speicheroperanden

```
1: if ( MEMACCESS_OBJ( i1 )->overlaps( MEMACCESS_OBJ( i2 ) ) then
2:     // Beide Instruktionen adressieren überlappende Speicherbereiche
3:     if ( isSTInstruction( i1 ) && isLDInstruction( i2 ) ) then
4:         return DataDep( i1, i2 );
5:     else if ( isLDInstruction( i1 ) && isSTInstruction( i2 ) ) then
6:         return AntiDep( i1, i2 );
7:     else if ( isSTInstruction( i1 ) && isSTInstruction( i2 ) ) ) then
8:         return OutputDep( i1, i2 );
9:     end if
10:    // Keine Abhängigkeit bei aufeinanderfolgenden Ladeinstruktionen
11: end if
12: // Keine Abhängigkeit zwischen i1 und i2 aufgrund von Speicheroperanden
13: return NoDep( i1, i2 );
```

Zur Bestimmung der Priorität einer Instruktion können dabei diverse Heuristiken eingesetzt werden. Einige bekannte Heuristiken, wie die maximale Verzögerung einer Instruktion (*MaxDelay*) oder die Mobilität einer Instruktion (*Mobility*), sind generisch. Diese Heuristiken bestimmen die Priorität unabhängig von der betrachteten Zielhardware.

Innerhalb des WCC ist allerdings auch eine Heuristik implementiert, die speziell auf die Eigenschaften der TriCore-Architektur eingeht und Prioritäten abhängig von Instruktionstypen des TriCore vergibt. Diese Heuristik wird in Kapitel 4.4.1 kurz vorgestellt.

Die Bildung von IP/LS-Bündeln ist bei der TriCore-Architektur vorteilhaft. Vor der Bestimmung der Priorität einer Instruktion wird daher nach dem Typ der aktuell betrachteten Instruktion differenziert (Zeile 7). Nur LS-Instruktionen lassen sich mit einer IP-Instruktion bündeln. Für alle anderen Instruktionen wird daher die Priorität für den Fall einer getrennten Ausführung bestimmt (Zeile 8).

Handelt es sich bei der betrachteten Instruktion um eine LS-Instruktion, so wird die Priorität eines Bündels dieser LS-Instruktion mit allen laut Ready List ausführbaren Instruktionen bestimmt (Zeile 11 bis 14).

Dabei ist in Zeile 12 von Algorithmus 4.2 auch eine Besonderheit des List Schedulers im WCC zu sehen: Die Überprüfung, ob Abhängigkeiten durch ein gefundenes Bündel verletzt werden können, erscheint auf den ersten Blick redundant. Da die Instruktionen laut DAG ausführbar sind, ist intuitiv nicht klar, wieso ein Bündel Abhängigkeiten verletzen könnte. Aufgabe des DAG ist schließlich die Einhaltung von Abhängigkeiten sicherzustellen.

Begründet ist diese Überprüfung durch die Möglichkeiten des Dual-Issue innerhalb der TriCore-Pipeline. Eine IP-Instruktion und eine davon abhängige LS-Instruktion lassen sich in der Regel gleichzeitig ausführen und damit ein Bündel dieser Instruktionen bilden. Im Abhängigkeitsgraphen wird dies durch eine Kante zwischen beiden Instruktionen mit der

Algorithmus 4.2 List Scheduling WCC

Eingabe: Abhängigkeitsgraph DAG
Ausgabe: Instruction Schedule

```
1:  zyklus := 1;
2:  Bestimme Ready List aus DAG;
3:  schedule := {};
4:  while ( Ready List nicht leer ) do
5:      // Betrachte alle laut Ready List ausführbaren Instruktionen
6:      for ( ins ∈ Ready List ) do
7:          if ( Typ( ins ) != LS ) then
8:              Bestimme Prio( ins );
9:          else
10:             // Mögliche Bündel mit IP-Instruktionen bestimmen
11:             for ( ip_ins ∈ Ready List ) do
12:                 if ( Typ( ip_ins ) == IP ) &&
                        bündel( ip_ins, ins ) verletzt keine Abhängigkeiten ) then
13:                     Bestimme Prio( bündel( ip_ins, ins ) );
14:                 end if
15:             end for
16:         end if
17:     end for
18:     selIns := Instruktion oder IP/LS-Bündel mit größter Priorität;
19:     Füge selIns zu schedule hinzu;
20:     Entferne selIns aus Ready List;
21:     if ( selIns ist Bündel ) then
22:         // Maximale Latenz der Bündel-Instruktionen bestimmen
23:         latenz := max( Latenz( IP_INS ), Latenz( LS_INS ) );
24:     else
25:         latenz := Latenz( selIns );
26:     end if
27:     zyklus += latenz;
28:     Aktualisiere Ready List;
29: end while
30: return schedule;
```

Latenz 0 dargestellt. Problematisch wird diese Darstellung, wenn Abhängigkeiten zwischen mehreren IP-Instruktionen und einer LS-Instruktion bestehen, wie im folgenden Beispiel dargestellt.

```
i1: and d12, d8, d9;
i2: nor d13, d8, 0;
i3: ld_w d8, a13, 8;
```

Zwischen Instruktion i1 und i3 besteht eine Gegenabhängigkeit, da i3 das Register `d8` beschreibt, das von i1 gelesen wird. Ein Scheduling von i3 vor i1 ist daher nicht möglich.

Da i1 eine IP-Instruktion ist und i3 eine LS-Instruktion, beträgt die Latenz zwischen beiden Befehlen allerdings 0. Eine gleichzeitige Ausführung als IP/LS-Bündel wäre also prinzipiell laut Datenabhängigkeitsgraph möglich.

Allerdings besteht auch eine Gegenabhängigkeit zwischen i2 und i3. Auch zwischen diesen Instruktionen beträgt die Latenz 0. Zwischen den Instruktionen i1 und i2 besteht keine Abhängigkeit. Beide Instruktionen lesen das gleiche Register `d8`, arbeiten aber ansonsten auf unterschiedlichen Registern. In der Ready List erscheinen daher alle drei Instruktionen als im gleichen Zyklus ausführbar. Tatsächlich ist dies allerdings nicht der Fall, da ein Bündel von i1 und i3 die Abhängigkeit zu i2 verletzen würde. Die Instruktion i2 könnte anschließend nicht mehr den ursprünglichen Wert von Register `d8` lesen, sondern den von i3 geschriebenen. Das gleiche gilt analog für ein Bündel aus i2 und i3.

Aus diesem Grund lassen sich keine Bündel mit LS-Instruktion bilden, die Abhängigkeiten zu mehr als einer IP-Instruktion aufweisen. Dies wird in Zeile 12 sichergestellt.

Sobald alle Instruktionen, die ohne Ressourcenkonflikte ausgeführt werden können, ausgewählt sind, wird der Zeitschritt erhöht. Anschließend wird die Ready List aktualisiert. Das Verfahren wird solange fortgeführt, bis die Ready List leer ist.

4.4.1 Instruction-Priority Heuristik

Um eine hohe Auslastung der IP-Pipeline und LS-Pipeline zu erreichen, werden beim List Scheduling für den TriCore IP/LS-Bündel bevorzugt selektiert. Erreicht wird dies, indem einem IP/LS-Bündel die summierten Prioritäten der enthaltenen IP- und LS-Instruktion zugewiesen werden. Die einzelnen Prioritäten können dabei beispielsweise über eine der bekannten, generischen Heuristiken wie MaxDelay oder Mobility bestimmt werden.

Im WCC ist neben den generischen eine weitere TriCore-spezifische Prioritäts-Heuristik implementiert. Diese *Instruction-Priority* Heuristik soll dafür sorgen, dass Instruktionen so gewählt werden, dass im weiteren Verlauf des List Scheduling möglichst viele Bündel gebildet werden können. Die Heuristik weist einer Instruktion, in Abhängigkeit von der zuletzt selektierten Instruktion, einen Wert zwischen 1 und 3 zu. Tabelle 4.1 zeigt die dabei möglichen Werte.

Vorgänger Instruktion	Instruktionstyp aktuelle Instruktion		
	IP-Instruktion	LS-Instruktion	DP-Instruktion
Anfang eines Basisblocks	2	1	3
IP-Instruktion	1	3	2
LS-Instruktion	2	1	3
DP-Instruktion	2	1	3

Tabelle 4.1: Faktoren der Instruction-Priority Heuristik

Wie die erste Zeile der Tabelle zeigt, werden DP-Instruktionen am Anfang eines Basisblocks bevorzugt, indem die maximale Priorität von 3 zugewiesen wird. Dies hat den Hin-

tergrund, dass sich keine Bündel bilden lassen, die DP-Instruktionen enthalten. Es könnte aber IP- sowie LS-Instruktionen geben, die von diesen DP-Instruktionen abhängen. Somit sollten DP-Instruktionen so früh wie möglich selektiert werden.

Die Selektion einer LS-Instruktion am Anfang eines Basisblocks ist hingegen nicht sinnvoll, da diese Instruktion so nicht mehr für ein später eventuell mögliches Bündel mit einer IP-Instruktion zur Verfügung steht. Der Faktor beträgt in diesem Fall nur 1, so dass LS-Instruktion am Anfang eines Basisblocks im Vergleich zu anderen Instruktionen benachteiligt werden.

Wie in der zweiten Zeile der Tabelle gezeigt, ist die Selektion einer IP-Instruktion direkt nach einer vorhergehenden IP-Instruktion nicht sinnvoll. Es sollte stattdessen, wenn möglich, eine LS-Instruktion selektiert werden, da sich so ein IP/LS-Bündel ergibt. Daher erhalten IP-Instruktionen nur einen Wert von 1, während LS-Instruktion mit einem Wert von 3 bevorzugt werden.

Im List Scheduler des WCC wird die Instruction-Priority Heuristik in Kombination mit einer generischen Heuristik genutzt: Nach Bestimmung der generischen Priorität werden bestimmte Instruktionen mittels der Instruction-Priority Heuristik bevorzugt, indem der ermittelte Wert mit dem Faktor zwischen 1 und 3 multipliziert wird.

5 WCET-fähiges Trace Scheduling

Das bisher vorgestellte Instruction Scheduling-Verfahren ist ein lokales Scheduling, das jeweils nur einzelne Basisblöcke betrachtet. Wie bereits angedeutet, ist dieses Verfahren nur eingeschränkt in der Lage die WCET zu minimieren: Einerseits stellt die geringe Anzahl unabhängiger Instruktionen innerhalb eines Basisblocks eine Einschränkung des lokalen Instruction Scheduling dar. Andererseits hat das Verfahren keine Kenntnis des WCEP und optimiert dadurch auch Basisblöcke, die nicht zur WCET beitragen können.

Das von Fisher [Fis81] vorgestellte Trace Scheduling ist ein globales Instruction Scheduling. Es umgeht die Einschränkungen der lokalen Verfahren, indem Programmpfade (Traces) und nicht isolierte Basisblöcke betrachtet werden. Die Idee des Trace Scheduling ist es, auf diesen Pfaden ein List Scheduling so durchzuführen, als wären sie ein einzelner Basisblock. Durch die simultane Betrachtung mehrerer Basisblöcke steht diesem Verfahren somit eine größere Anzahl unabhängiger Instruktionen zur Verfügung.

Das ursprüngliche Ziel Fishers war es, durch das Trace Scheduling eine Reduktion der durchschnittlichen Programmlaufzeit (ACET) zu erreichen. Daher sollte eine Optimierung der am häufigsten ausgeführten Programmpfade durchgeführt werden. Im Gegensatz dazu ist das Ziel dieser Diplomarbeit eine Reduktion der WCET. Dies wird durch die Selektion von Programmpfaden erreicht, die zur WCET beitragen.

In Kapitel 5.1 wird zunächst der Begriff Trace definiert und ein kurzer, allgemeiner Überblick über das Trace Scheduling-Verfahren gegeben. In den weiteren Kapiteln wird anschließend detaillierter auf einzelne Phasen des Trace Scheduling eingegangen.

Kapitel 5.2 liefert eine allgemeine Vorstellung der Trace-Selektion. Diese dient der Wahl eines Programmpfades innerhalb des Kontrollflussgraphen, der für das angestrebte Optimierungsziel möglichst geeignet erscheint. Anschließend wird beschrieben, wie ein Trace selektiert werden kann, um damit eine Reduktion der WCET zu erreichen.

Das Verschieben von Instruktionen innerhalb eines Traces durch den List Scheduler kann dazu führen, dass die Semantik von Pfaden, die nicht zum Trace gehören, verändert wird. Kapitel 5.3 beschreibt die verschiedenen Fälle, die dabei auftrete können und zeigt, wie im Trace Scheduling durch die Einfügung von Kompensationscode darauf reagiert wird.

Trace Scheduling baut auf einem lokalen Instruction Scheduling auf. Daher soll zur Bestimmung eines Schedules ein List Scheduler ohne weitere Modifikation genutzt werden. Allerdings sind nicht alle Basisblock-übergreifenden Verschiebungen von Instruktionen beim Trace Scheduling erlaubt. Einschränkungen an mögliche Verschiebungen müssen daher innerhalb des Abhängigkeitsgraphen festgehalten werden. Die Änderungen, die im Vergleich

zum Abhängigkeitsgraphen des lokalen Instruction Scheduling notwendig sind, werden in Kapitel 5.4 erläutert.

Ein Großteil der WCET eines Programms ergibt sich häufig aufgrund der Ausführung von Schleifen. Traces sind allerdings schleifenfrei und können damit keine Schleifenrückkanten enthalten. In 5.5 wird beschrieben, wie innere Schleifen als sogenannte *Loop Representatives* dargestellt werden können und so Teil eines Trace werden können.

5.1 Trace Scheduling

Dieses Kapitel soll zunächst einen groben Überblick über das generelle Vorgehen beim Trace Scheduling geben. Anschließend werden die einzelnen Phasen detaillierter vorgestellt. Zunächst folgen benötigte Definitionen.

Definition 5.1 (Followers).
Mit ***Followers(****b_i****)****, definiert auf dem Kontrollflussgraphen $G = (V, E)$, wird eine Funktion bezeichnet, die einem Basisblock $b_i \in V$ die Menge aller $b_j \in V$ zuordnet für die gilt: $(b_i, b_j) \in E$.*
Stellt ein Basisblock b_i eine Senke des Kontrollflussgraphen dar, so ist $Followers(b_i) = \{\}$.

Mit $Followers(b_i)$ können demnach alle Basisblöcke ermittelt werden, zu denen der Kontrollfluss nach Ausführung von b_i unmittelbar wechseln kann. Wenn es mehr als einen Basisblock in $Followers(b_i)$ gibt, dann ist an b_i eine Verzweigung des Kontrollflusses gegeben. In Abbildung 5.1 ist beispielsweise an Basisblock A mit $Followers(A) = \{B, G\}$ eine mögliche Verzweigung des Kontrollflusses dargestellt.

Mit Hilfe der Funktion $Followers$ kann der Begriff Trace definiert werden.

Definition 5.2 (Trace).
Ein ***Trace T*** *auf einem Kontrollflussgraphen $G = (V, E)$ ist eine Folge $T_i = (bb_1, ..., bb_t)$ aus V, so dass für alle $j, 1 \leq j \leq t-1$ gilt:*
$bb_{j+1} \in Followers(bb_j)$ *und*
$\nexists bb_k \in T_i : bb_j = bb_k$ *und*
$bb_j \in T_i \Rightarrow bb_j \notin T_l, i \neq l$.

Ein Trace ist demnach eine azyklische Folge unterschiedlicher Basisblöcke durch den Kontrollflussgraphen. Dabei kann jeder Basisblock maximal Teile eines Trace sein, so dass ein Trace niemals einen anderen Trace kreuzt.

In Abbildung 5.1 ist beispielhaft ein Trace als Folge der grau hinterlegten Basisblöcke (A, B, C, D, E, F) dargestellt. An Basisblöcken, an denen eine Verzweigung des Kontrollflusses möglich ist (*Split*), kann nur einer der Nachfolger Teil des Trace sein. Diesen Basisblock bezeichnet man, wie in Abbildung 5.1 dargestellt, als *On-Trace Nachfolger*. Alle anderen Nachfolger werden als *Off-Trace Nachfolger* bezeichnet. Im Beispiel ist Basisblock B der On-Trace Nachfolger des Splits A, während G einen Off-Trace Nachfolger darstellt.

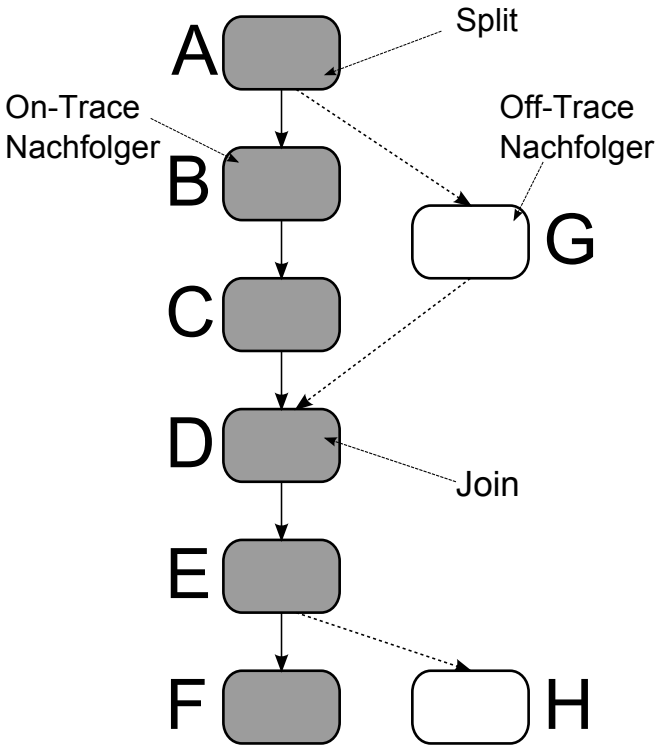


Abbildung 5.1: Beispiel für einen Trace

Für On-Trace Vorgänger und Off-Trace Vorgänger ist eine analoge Definition an Programmpunkten, an denen der Kontrollfluss zusammengeführt wird (*Join*), möglich. Somit stellt der Basisblock C für den Join-Basisblock D den On-Trace Vorgänger dar, während Basisblock G ein Off-Trace Vorgänger ist.

Algorithmus 5.1 stellt eine grobe Übersicht über den allgemeinen Ablauf des Trace Scheduling dar. In den folgenden Kapiteln wird auf die einzelnen Schritte detaillierter eingegangen.

Als erster Schritt des Trace Scheduling wird mittels der Trace-Selektion (Zeile 3) ein zu optimierender Trace gewählt. Auf diesem Trace wird ein für das Trace Scheduling angepasstes Verfahren zur Bestimmung des Abhängigkeitsgraphen durchgeführt (Zeile 5). Dieses Verfahren stellt sicher, dass zusätzlich zu lokalen Abhängigkeiten auch Abhängigkeiten innerhalb des Traces durch das anschließend durchgeführte List Scheduling eingehalten werden. Der in Zeile 6 benutze List Scheduler entspricht dabei dem in Kapitel 4.4 (Seite 36) vorgestellten lokalen List Scheduler.

Nach der Bestimmung eines Schedules und dem Verschieben der Instruktionen wird in der Bookkeeping-Phase (Zeile 8) Kompensationscode erzeugt. Dieser Code dient dazu die ursprüngliche Semantik des Programms wieder herzustellen, falls diese durch das Verschieben von Instruktionen über Basisblock-Grenzen verändert wurde. Das gesamte Verfahren wird so lange wiederholt, bis kein weiterer Trace mehr gefunden wird.

Im nächsten Kapitel wird die Trace-Selektion, die eine Auswahl der zu optimierenden Programmpfade durchführt, ausführlicher beschrieben.

Algorithmus 5.1 Trace Scheduling

Eingabe: Programm P.
Ausgabe: Optimiertes Programm P.

1: **repeat**
2: // Trace-Selektion
3: *Trace*:= Schleifenfreie Folge von Basisblöcken aus P, die keine anderen Traces kreuzt;
4: // Trace Scheduling
5: Bestimme DAG_{Trace} für Trace;
6: Führe List Scheduling auf DAG_{Trace} aus;
7: //Bookkeeping
8: Füge Kompensationscode in P ein, um Semantik auf Off-Trace Pfaden von $Trace$ zu korrigieren;
9: **until** alle $Traces$ aus P betrachtet

5.2 Trace-Selektion

Bevor das Trace Scheduling durchgeführt werden kann, muss ein Trace selektiert werden. In Fishers Arbeit [Fis81] bestand das Ziel des Trace Scheduling in einer Reduktion der durchschnittlichen Ausführungszeit des Programm. Daher sollte ein Trace so gewählt werden, dass er dem am häufigsten ausgeführten Programmpfad entspricht. Dazu wurden erwartete Ausführungshäufigkeiten der Basisblöcke benutzt. Bei einem Basisblock, der mit einem bedingten Sprung endet, sollte derjenige Basisblock als Nachfolger für den Trace selektiert werden, zu dem die Ausführung wahrscheinlich am häufigsten wechselt.

In späteren Arbeiten wurden die erwarteten Ausführungshäufigkeiten durch reale, gemessene Ausführungshäufigkeiten ersetzt. Mittels Versuchsläufen mit Beispieldaten können diese Werte ermittelt werden (Profiling). Aufbauend auf den Ausführungshäufigkeiten wurden von Fisher und nachfolgend von Chang Heuristiken zur Selektion von Traces vorgestellt. Beide Verfahren beginnen mit der Wahl eines *schwersten* Startknoten b_{seed} auf dem Kontrollflussgraphen $G = (V, E)$, der noch nicht Teil eines anderen Trace ist. Dabei dienen die ermittelten Ausführungshäufigkeiten der Basisblöcke als Gewichtsfunktion $w(b)$ für alle $b \in V$. Die Ausführungshäufigkeiten von Kontrollflusskanten werden als Gewichtsfunktion $w(e)$ für alle $e \in E$ benutzt.

Ausgehend von diesem Knoten wird der Trace $T_i = (b_a, ..., b_{seed}, ...b_e)$ abwechselnd am Anfang und am Ende verlängert, solange dabei keine Schleifenrückkanten benutzt werden müssen, und der betrachtete Block nicht bereits Teil eines anderen Trace T_j ist.

Nachfolgend werden zwei der möglichen Heuristiken zur Trace-Selektion für eine Vorwärts-Verlängerung am Ende des Trace beispielhaft dargestellt. Für die Rückwärts-Richtung kann eine Verlängerung analog erfolgen.

Selektion durch Knotengewichte

Der aktuell letzte Basisblock des Trace wird als *pred* bezeichnet. Ein Trace-Nachfolger *succ* wird bei der Selektion durch Knotengewichte, wie in Algorithmus 5.2 dargestellt, ermittelt. Dabei bezeichnet *TRACES* die Menge aller bisher auf dem Kontrollflussgraph erzeugten Traces.

Algorithmus 5.2 Knotengewicht

Eingabe: Knoten *pred*, *CFG V=(V,E)*.
Ausgabe: Bester Nachfolgeknoten *succ*.

```
1: Für alle x aus Followers(pred) : Wähle succ als x mit maximalen Knotengewicht;
2: if succ ∈ TRACES then
3:     return 0;
4: else
5:     return succ;
6: end if
```

Unter allen möglichen Nachfolgern wird derjenige Basisblock als Trace-Nachfolger gewählt, der das höchste Knotengewicht $w(b)$ aufweist. Da das Knotengewicht der ermittelten/erwarteten Ausführungshäufigkeit entspricht, wird damit unter allen möglichen Nachfolgern von *pred* derjenige Basisblock als Nachfolger gewählt, der am häufigsten ausgeführt wird. Ist der gewählte Basisblock bereits Teil eines anderen Trace, so kann der Trace nicht weiter nach vorne verlängert werden.

Die Selektion durch Knotengewichte ist zwar sehr leicht zu implementieren, hat aber Mängel. Wie bereits angedeutet, ist es für das Scheduling vorteilhaft, wenn der Trace so selektiert wird, dass bei einem bedingten Sprung der Basisblock als Nachfolger gewählt wird, zu dem der Kontrollfluss häufig wechselt. Da die Selektion durch Knotengewichte ausschließlich die Ausführungshäufigkeit von Basisblöcken und nicht von Kanten berücksichtigt, ist dies bei diesem Verfahren nicht sichergestellt: Ein Basisblock kann zwar häufig ausgeführt werden, es ist aber nicht sicher, ob der Basisblock nicht in den meisten Fällen über einen anderen Kontrollfluss erreicht wird.

Selektion durch Kantengewichte

Eine Alternative zur Selektion durch Knotengewichte stellt die Selektion nach Kantengewichten dar. Die Kanten $e = (b_i, b_j) \in E$ sind mit der Ausführungshäufigkeit dieser Kante annotiert und geben so an, wie häufig ein Übergang von b_i nach b_j stattfindet. Wenn b_i nicht mit einem bedingten Sprung endet gilt demnach $w(e) = w(b_i) = w(b_j)$.

Die Auswahl eines Nachfolgeknoten *succ* bei gegebenem Knoten *pred* erfolgt ähnlich wie in Algorithmus 5.2 gezeigt. Zur Selektion wird allerdings nicht das Knotengewicht $w(b)$ sondern das Kantengewicht $w(e)$ genutzt. Auch bei diesem Verfahren stoppt die Verlängerung, sobald ein Basisblock selektiert wurde, der bereits Teil eines anderen Trace ist.

Eine Erweiterung stellt das in [LFK+92] vorgestellte *mutually most likely* Kriterium dar:

Der Trace wird nur um den Basisblock *succ* verlängert, wenn e unter allen eingehenden Kanten von *succ*, die Kante mit dem höchsten Kantengewicht ist (*"if we are at succ, we are most likely to come from pred"*). Wenn dieses Kriterium nicht erfüllt ist, wird *succ* meistens durch einen anderen Kontrollpfad erreicht. Es erscheint daher sinnvoll, dass *succ* Teil eines anderen Trace wird.

5.2.1 WCET-fähige Trace-Selektion

Eine WCET-fähige Trace-Selektion muss in der Lage sein, einen Trace so zu selektieren, dass ein anschließendes Trace Scheduling zu einer Reduktion der WCET führen kann. Auf den ersten Blick erscheint es, als wäre die Trace-Selektion für dieses Ziel trivial: Als Trace kann einfach der WCEP (Worst-Case Execution Path) gewählt werden.

Aufgrund der bereits angesprochenen Unterstützung von Kontexten (Kapitel 2.4, Seite 14) ist der von aiT ermittelte WCEP allerdings meist kein *einfacher* Pfad durch den Kontrollflussgraphen. Es kann vorkommen, dass in unterschiedlichen Schleifendurchläufen oder Funktionsaufrufen unterschiedliche Ziele einer Verzweigung auf dem WCEP liegen. In solch einem Fall ist also eine Trace-Selektion notwendig, die entscheidet, welcher Zweig für den Trace gewählt wird.

Problematisch an der Trace-Selektion ist, dass nicht garantiert werden kann, dass ein gewählter Pfad wirklich in dieser Form ausführbar ist. Durch die mögliche Aggregation der Ausführungsfrequenzen für Kanten über mehrere Kontexte, kann es zu einer Selektion von Zweigen kommen, die sich innerhalb eines einzelnen Kontext ausschließen. Da dies allerdings ein Problem aller Selektionsverfahren ist, wird darauf nicht weiter eingegangen.

In dieser Arbeit liegt der Schwerpunkt auf der Reduktion der WCET. Daher wird eine Trace-Selektion zur Reduktion der WCET benötigt. Andererseits soll alternativ auch eine Trace-Selektion zur Reduktion der ACET ermöglicht werden, um so eine Vergleichsmöglichkeit zu schaffen. Wie gezeigt benutzen die Trace-Selektions-Verfahren nach Kantengewicht und nach Knotengewicht als Gewichtsfunktionen per Profiling ermittelte Ausführungshäufigkeiten. Zur Reduktion der WCET sind diese Gewichtsfunktionen allerdings nicht geeignet. Es wird daher eine Alternative benötigt.

Kostenmetriken

Bei einer Trace-Selektion zur Reduktion der WCET können die Kosten von Basisblöcken und Ausführungskanten mit Hilfe von aiT ermittelt werden. Als Gewichtsfunktion $w(b)$ für Basisblöcke b wird bei der Trace-Selektion zur WCET-Reduktion die in den WCET Objectives gespeicherte $WCET_{sum}$ gewählt. Diese repräsentiert die über alle Kontexte summierte WCET eines Basisblocks. Die Gewichtsfunktion $w(e)$ für Kontrollflusskanten e entspricht der Worst-Case Ausführungshäufigkeit der entsprechenden Kante. Diese lässt sich als $WCEC_{sum}$ der Kante bestimmen. Auch dieser Wert ist über alle Kontexte summiert.

Wird alternativ, statt einer Reduktion der WCET, eine Reduktion der ACET angestrebt,

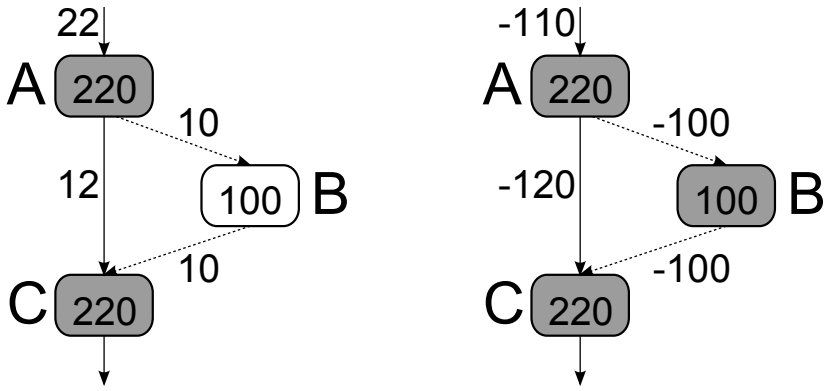
so können im WCC die Ausführungshäufigkeiten von Basisblöcken und von Kontrollflusskanten über ein Profiling mittels CoMET ermittelt werden. Die Trace-Selektion zur Reduktion der ACET müsste in diesem Fall $ACET_{sum}$ und $ACEC_{sum}$ Werte der ACET Objectives nutzen, um Gewichtsfunktionen zu bestimmen.

Um in der Implementierung der Trace-Selektion zahlreiche Unterscheidung zwischen WCET Objectives und ACET Objectives und den entsprechenden Zugriffsmethoden zu vermeiden, wurde in dieser Arbeit eine andere Lösung entwickelt. Kostenmetriken für die Trace-Selektion werden mittels eines *Adapter*-Designs abstrahiert, so dass eine einfache Integration beliebiger Kostenmetriken möglich ist.

Eine Kostenmetrik zur Bestimmung von Traces kann als Subklasse von `LLIR_TraceMetric` implementiert werden. Dabei müssen folgende Methoden zur Verfügung gestellt werden.

- `getCost()` - Gibt die Kosten für die Ausführung eines Basisblocks an.
- `getExecutionCountSum()` - Gibt die Ausführungshäufigkeit eines Basisblocks an.
- `getExecutionCount(LLIR_BB &successor)` - Gibt an, wie häufig die Ausführung vom aktuell betrachteten Basisblock zum Basisblock `successor` wechselt.

Aktuell ist im WCC mittels der Klassen `LLIR_WCETTraceMetric` und `LLIR_ACTTraceMetric` eine Trace-Selektion nach ACET und WCET verfügbar. Diese Adapterklassen implementieren die beschriebenen Methoden, indem sie auf die $WCET_{sum}$ und $WCEC_{sum}$ bzw. $ACET_{sum}$ und $ACEC_{sum}$ Werte zugreifen. Das Hinzufügen weiterer Metriken, wie beispielsweise der Energiebedarf für die Ausführung eines Basisblocks, wäre auf diese Weise ohne Änderungen der Trace-Selektion möglich.



(a) Selektion nach Kanten-/Knotengewicht

(b) Selektion nach Longest Path

Abbildung 5.2: Vergleich unterschiedlicher Verfahren zur Trace-Selektion

Eine Selektion nach Kantengewichten soll anhand eines Beispiels demonstriert werden. Abbildung 5.2 deutet den Kontrollflussgraph einer *if-then*-Verzweigung an. Dabei entspricht der Basisblock B dem *then*-Teil. Innerhalb der Basisblöcke ist als Gewichtsfunktion die gesamte WCET, die durch die Ausführung des Basisblocks entsteht, annotiert. Die Kanten zwischen den Basisblöcken sind mit der WCEC annotiert. Die Selektion nach Kantenge-

wichten (und auch die Selektion nach Knotengewichten) würde in diesem Beispiel den Pfad bestehend aus Basisblock A und C mit einer WCET von $220 + 220 = 440$ Zyklen wählen (Abbildung 5.2a).

Allerdings würde ein Trace, der zusätzlich den *then*-Teil der Verzweigung (Basisblock B) enthält, stärker zur WCET beitragen ($220 + 100 + 220 = 540$ Zyklen). Für eine Reduktion der WCET wäre es daher wünschenswert diesen Pfad selektieren zu können.

In [Kel09] wurde eine Trace-Selektion für High-Level Traces entwickelt, die explizit das Ziel hat, längste Pfade bezüglich der WCET zu selektieren. Bei diesen High-Level Traces handelt es sich, im Gegensatz zu den in dieser Arbeite benutzten Basisblock-Traces, um Traces auf der High-Level Darstellung ICD-C IR. Die *Longest Path*-Selektion soll in dieser Arbeit zur Selektion von Low-Level Traces adaptiert werden und so als Alternative zur Selektion nach Kantengewichten dienen. Das Verfahren wird im Folgenden kurz erläutert.

Selektion des längsten Pfades

Das Longest Path-Verfahren selektiert, wie auch die Selektion nach Kantengewichten, zunächst einen Startknoten b_{seed}. Anschließend wird versucht, einen längsten (schwersten) Pfad zwischen einem Startknoten b_{source} und b_{seed} sowie zwischen b_{seed} und einem als *Supersenke* bezeichneten Knoten zu ermitteln. Der Knoten b_{source} entspricht dabei entweder einem Schleifenkopf, falls b_{seed} Teil einer inneren Schleife ist, oder dem ersten Basisblock des Kontrollflussgraphen. Die Supersenke wurde eingeführt, um einen gemeinsamen Punkt zu haben, an dem alle Kontrollflüsse zusammengeführt werden. Die Konkatenation beider Teilpfade ergibt den selektierten Trace.[1]

Das Kantengewicht $w_l(e)$ mit $e = (b_i, b_j) \in E$ wird bei diesem Verfahren so gewählt, dass dadurch eine Abschätzung der WCET möglich ist, die sich aufgrund der Ausführung dieser Kante ergibt. Konkret erfolgt die Berechnung $w_l(e)$ auf folgende Weise:

$$w_l(e) = -1 * (WCET_{sum}(b_j) * \frac{WCEC_{sum}(e)}{WCEC_{sum}(b_j)}) \text{ für } b_i \neq b_j$$

Jede Kante enthält dadurch den Anteil der WCET, der durch Kontrollflüsse über diese Kante am Zielknoten anfällt. Da es eventuell keine Kante gibt, die b_{source} als Ziel hat (falls b_{source} die Quelle des CFG ist), wird das Kantengewicht für alle $(b_{source}, b_j) \in E$ getrennt berechnet.

$$w_l(b_{source}, b_j) = -1 * (WCET_{sum}(b_j) * \frac{WCEC_{sum}(b_{source}, b_j)}{WCEC_{sum}(b_j)} + WCET_{sum}(b_{source}))$$

Dabei werden in diesem Verfahren negative Kantengewichte $w_l(e)$ ermittelt, da so effizient ein kürzester Pfad gesucht werden kann. Dieser kürzeste Pfad mit negativen Gewichten entspricht dem gewünschten längsten Pfad. Abbildung 5.2b zeigt die von diesem Verfahren berechneten Kantengewichte für die *if-then*-Verzweigung. Wie gewünscht, ist dieses Verfahren in der Lage, den Trace zu selektieren, der auch den *then*-Teil enthält.

[1] Es müssen in diesem Verfahren zwei Teilpfade konkateniert werden, da ein Pfad, der ausschließlich zwischen b_{source} und der Supersenke ermittelt wird, nicht zwangsläufig b_{seed} enthält.

Nachdem ein Trace mittels einer der beschriebenen Trace-Selektions Verfahren ausgewählt wurde, wird auf diesem das List Scheduling so durchgeführt, als wäre es ein einzelner Basisblock. Anschließend werden Instruktionen, abhängig von der Position im Schedule, innerhalb der LLIR in neue Basisblöcke verschoben. Dabei sind unterschiedliche Arten von Verschiebungen möglich, deren Auswirkung auf die Programmsemantik im Folgenden detaillierter betrachtet werden.

5.3 Kompensation

Beim Trace Scheduling sind einerseits, genau wie beim lokalen Instruction Scheduling, Verschiebungen von Instruktionen innerhalb eines Basisblocks möglich. Diese Verschiebungen können offensichtlich keinerlei negativen Auswirkungen auf die Semantik anderer Ausführungspfade haben. Andererseits sind auch Verschiebungen möglich, die Basisblock-Grenzen überschreiten. Da Trace Scheduling Off-Trace Pfade während des List Scheduling nicht besonders berücksichtigt, kann es durch solche Verschiebungen zu einer geänderten Programmsemantik kommen. Daher kann es nach dem List Scheduling erforderlich sein Kompensationscode in Off-Trace Pfaden zu erzeugen, um die ursprüngliche Semantik wiederherzustellen.

Die unterschiedlichen Möglichkeiten von Verschiebungen werden im Folgenden genauer betrachtet. Zunächst werden einige benötigte Begriffe definiert.

Definition 5.3 (ContainingBB).
*Als **ContainingBB** wird eine Funktion bezeichnet, die einer Instruktion ins_i den Basisblock bb_j aus einem Trace $T = (bb_1, ..., bb_t)$ zuordnet, in dem sie enthalten ist.*

Definition 5.4 (Split).
*Eine Split-Instruktion oder kurz **Split**, ist eine Instruktion ins_{split} aus einem Basisblock bb_j auf einem Trace $T = (bb_1, ..., bb_t)$, für die gilt:*
$|Followers(\ ContainingBB(\ ins_{split}\ )\ )| \geq 2$.
Der Basisblock bb_j wird im Folgenden als Split-Basisblock bezeichnet.

Beispielsweise stellt ein bedingter Sprung einen Split dar. Da Verzweigungen innerhalb des Kontrollflusses nur am Ende eines Basisblocks möglich sind, existiert in jedem Split-Basisblock genau eine Split-Instruktion. Aus der Sicht des Kontrollflusses stellt ein Split einen *Seitenausgang* aus dem Trace dar.

Definition 5.5 (Join).
*Eine Join-Instruktion oder kurz **Join**, ist eine Instruktion ins_{join} aus einem Basisblock bb_j auf einem Trace $T = (bb_1, ..., bb_t)$, für die gilt:*
Am Beginn von bb_j werden alternative Kontrollflüsse zusammengeführt.
ins_{join} ist dabei die erste Instruktion die in bb_j ausgeführt wird. Der Basisblock bb_j wird im Folgenden als Join-Basisblock bezeichnet.

Ein Join-Basisblock muss also mindestens zwei eingehende Kontrollflusskanten $e \in V$ aufweisen. Aus der Sicht des Kontrollflusses stellt ein Join einen *Seiteneingang* in den Trace dar.

Wird eine Instruktion ins_x aus einem Basisblock bb_j in einen Basisblock bb_i verschoben, mit bb_i, bb_j aus $T = (bb_1, ..., bb_t)$ und $bb_i \neq bb_j$, hängt die Erzeugung von Kompensationscode davon ab, ob dabei Splits oder Joins auf dem Trace passiert wurden. Die unterschiedlichen Fälle werden im Folgenden betrachtet.

5.3.1 Join-Kompensation

Eine *Join-Kompensation* wird notwendig, sobald eine verschobene Instruktion ins_x, die Grenzen eines Join-Basisblocks passiert. Für einen Trace $T = (bb_1, ..., bb_{join}, ..., bb_t)$ mit Join-Basisblock bb_{join} sind dabei zwei Fälle zu unterscheiden:

- Eine Instruktion ins_x aus einem bb_j, mit $j < join$ wird in einen Basisblock bb_i mit $i \geq join$ verschoben. Bei dieser Verschiebung *von oben* würde ins_x ohne Kompensation auch beim Durchlaufen der Off-Trace Pfade, die zu dem Join-Basisblock bb_{join} führen, ausgeführt werden. Dieser Fall ist in Abbildung 5.3a und Abbildung 5.3b für ein Verschieben von Instruktion ins_x in den Join-Basisblock B dargestellt.

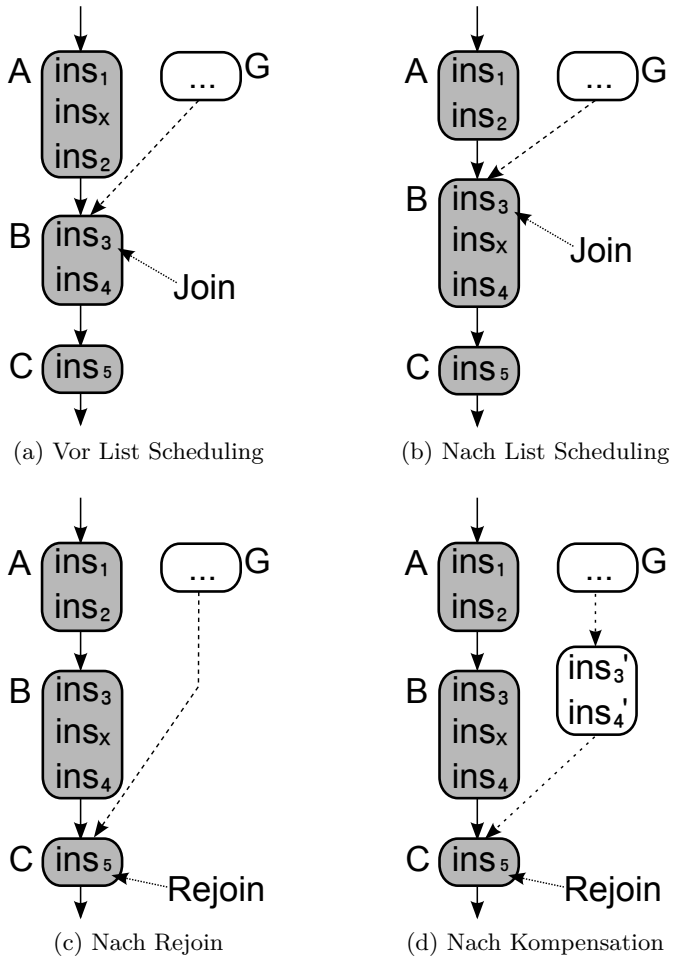
- Eine Instruktion ins_x aus einem bb_j, mit $j \geq join$ wird in einen Basisblock bb_i mit $i < join$ verschoben wird. Ohne Kompensation würde ins_x nach dieser Verschiebung *von unten* beim Durchlaufen der Off-Trace Pfade nicht ausgeführt werden, obwohl dies im ursprünglichen Kontrollfluss der Fall war.

Für beide Fälle kann die Kompensation in gleicher Weise erfolgen. In Abbildung 5.3 werden die einzelnen Schritte einer Join-Kompensation für ein Verschieben von oben gezeigt. In Abbildung 5.3a ist ein Ausschnitt aus einem Kontrollflussgraphen vor dem List Scheduling dargestellt. In diesem Beispiel besteht der Trace aus den Basisblöcken A, B und C. Instruktion ins_3 aus Basisblock B stellt eine Join-Instruktion dar. Demnach ist nach Definition Basisblock B ein Join-Basisblock beziehungsweise Seiteneingang. An diesem Seiteneingang stellt Basisblock G einen Off-Trace Vorgänger dar.

List Scheduling auf dem Trace (A, B, C) führt zu einer Verschiebung von Instruktion ins_x aus Basisblock A unter die Join-Instruktion ins_3 (Abbildung 5.3b). In diesem Fall wurde durch das Verschieben von Instruktion ins_x die Grenzen eines Join-Basisblocks überschritten. Dies führt ohne Kompensation zu einer eventuell geänderten Semantik des Programms: Wird in einem Lauf des Programms der Off-Trace Pfad über Basisblock G nach Basisblock B benutzt, so wird Instruktion ins_x ausgeführt. Im ursprünglichen Programm war dies vor dem List Scheduling nicht der Fall.

Zur Kompensation werden zunächst zwei Funktionen benötigt.

Definition 5.6 (TracePosition).
TracePosition(*ins_i****)*** *ist eine Funktion, die die eindeutige Position einer Instruktion ins_i innerhalb der sequentiellen Folge von Instruktionen eines Traces liefert.*



(a) Vor List Scheduling (b) Nach List Scheduling

(c) Nach Rejoin (d) Nach Kompensation

Abbildung 5.3: Kompensation einer Verschiebung unter eine Join-Instruktion

Definition 5.7 (SchedulePosition).
SchedulePosition*(*ins_i*)* *ist eine Funktion, die die eindeutige Position einer Instruktion* ins_i *innerhalb des sequentiellen Schedules, das durch das List Scheduling bestimmt wurde, liefert.*

In dem in Abbildung 5.3 gezeigten Beispiel ist Kompensation notwendig, da für Instruktion ins_x und die Join-Instruktion ins_3 gilt:
$TracePosition(ins_x) < TracePosition(ins_3)$, aber
$SchedulePosition(ins_x) > SchedulePosition(ins_3)$.

Um die ursprüngliche Semantik wieder herzustellen, ist zunächst die Bestimmung einer *Rejoin-Instruktion* notwendig.

Definition 5.8 (Rejoin-Instruktion).
Als ***Rejoin-Instruktion*** ins_{ri} *einer Join-Instruktion* ins_{join} *aus einer Folge* $I_T = (ins_1, ..., ins_t)$ *von Instruktionen* ins_n*, die auf dem Trace T liegen, wird eine Instruktion bezeichnet, die nachfolgende Bedingung erfüllt:*

Für alle ins_i mit TracePosition(ins_i) < TracePosition(ins_{join}) gilt:
SchedulePosition(ins_i) < SchedulePosition(ins_{ri})

Eine Rejoin-Instruktion ins_{ri} für eine Instruktion ins_{join} wird demnach so gewählt, dass alle Instruktionen, die bei einer Ausführung des Traces vor der Join-Instruktion ins_{join} ausgeführt worden wären, nach dem List Scheduling vor der Rejoin-Instruktion ins_{ri} ausgeführt werden. Ist eine Rejoin-Instruktion bestimmt, werden alle Off-Trace Pfade, die ins_{join} als Ziel haben, so modifiziert, dass sie ins_{ri} als Ziel haben.

In Abbildung 5.3b erfüllt Instruktion ins_5 die Bedingungen an eine Rejoin-Instruktion. Der Sprung des Off-Trace Vorgängers G wird daher, wie in Abbildung 5.3c gezeigt, auf den Basisblock C umgelenkt.

Durch diese Modifikation wird Instruktion ins_x beim Durchlaufen des Off-Trace Pfades nicht mehr ausgeführt. Allerdings werden dadurch auch die Instruktionen ins_3 und ins_4 nicht mehr ausgeführt. Dies wird beim Trace Scheduling durch Einfügung von *Join-Kompensationskopien* korrigiert.

Definition 5.9 (Join-Kompensationskopien).
*Als **Join-Kompensationskopien** einer Join-Instruktion ins_{join} mit Rejoin-Instruktion ins_{ri} wird ein Tupel $JK = (ins_1, ..., ins_m)$ von Instruktionen bezeichnet, falls für alle $ins_l \in JK$ gilt:*
TracePosition(ins_l) ≥ TracePosition(ins_{join}) ∧
SchedulePosition(ins_l) < SchedulePosition(ins_{ri})

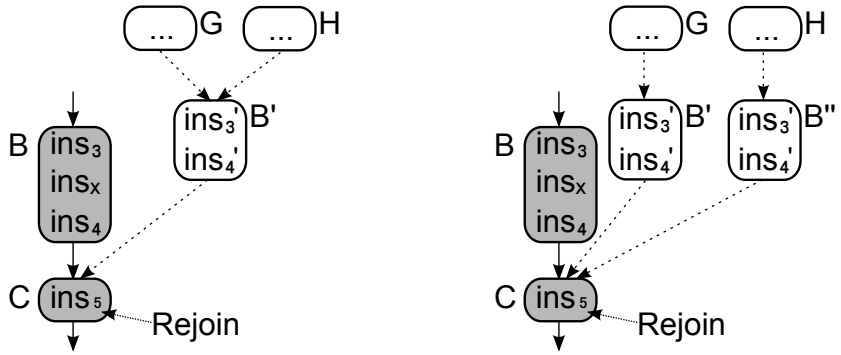
Das Tupel der Join-Kompensationskopien besteht aus allen Instruktionen, die im Trace nach dem Join ausgeführt worden wären, durch die beschriebene Korrektur des Kontrollflusses, beim Durchlaufen von Off-Trace Pfaden allerdings nicht mehr ausgeführt werden. Im Beispiel in Abbildung 5.3d besteht das Tupel der Join-Kompensationskopien aus den Instruktionen ins_3 und ins_4, die als Kopien ins_3´ und ins_4´ in den Kontrollflussgraphen eingefügt werden.

Einfügung von Join-Kompensationskopien

Die ermittelten Join-Kompensationskopien müssen zwischen allen Off-Trace Vorgängern und dem ermittelten Rejoin-Basisblock eingefügt werden. Abbildung 5.3d zeigt das Ergebnis dieser Einfügung im Beispiel.

Falls ein Rejoin mehrerer Off-Trace Vorgänger notwendig ist, sind dabei zwei Strategien denkbar, wie die Instruktionen in den Kontrollflussgraphen und in den Maschinencode eingefügt werden können. Auf diese Alternativen soll im Folgenden kurz eingegangen werden. Dabei wird davon ausgegangen, dass alle Join-Kompensationskopien innerhalb eines einzigen Basisblocks bb_j platziert werden können[2].

[2]Ist auch eine Join-Kopie einer Sprung-Instruktion notwendig, so müssen eventuell mehrere Basisblöcke neu erzeugt werden. Prinzipiell wird die Kompensation aber in der selben Weise durchgeführt.



(a) Eine gemeinsame Join Kopie (b) Mehrere Join Kopien

Abbildung 5.4: Alternativen zur Einfügung von Join-Kompensationkopien

- Der neue Basisblock bb_j, der die Join-Kopien enthält, wird wie in Abbildung 5.4a dargestellt, ein einziges mal im Maschinencode eingefügt. Die Sprünge der Off-Trace Vorgänger werden so korrigiert, dass diese bb_j als Ziel haben. Am Ende von bb_j ist dann in vielen Fällen ein weiterer Sprung notwendig, der den Kontrollfluss zu bb_{ri} führt.

- Für jeden Off-Trace Vorgänger wird, wie in Abbildung 5.4b dargestellt, eine eigene Kopie von bb_j angelegt und in den Maschinencode eingefügt.

Der Vorteil der ersten Lösung besteht darin, dass dadurch weniger Codewachstum auftritt: Unabhängig von der Anzahl der Off-Trace Vorgänger muss jede Join-Kopie nur ein einziges Mal erzeugt werden. Der Vorteil der zweiten Lösung besteht darin, dass die kopierten Basisblöcke bb_j in vielen Fällen mit dem Off-Trace Vorgängern verschmolzen werden können, so dass weniger Sprünge notwendig sind.

Ein mögliches Verschmelzen von Basisblöcken erscheint aus Gründen der *Code-Lokalität* vorteilhaft. Des Weiteren haben zusätzliche Sprünge, die in der ersten Alternative häufiger notwendig sind, oft eine negative Auswirkung auf die WCET. Da es außerdem nur selten Fälle gibt, in denen tatsächlich zahlreiche Kopien für Off-Trace Vorgänger notwendig sind, kann das stärkere Codewachstum dieser Lösung vernachlässigt werden. In dieser Arbeit wurde daher die zweite Alternative implementiert.

Wahl einer Rejoin-Instruktion

In Abbildung 5.3 wurde Instruktion ins_5 als Rejoin-Instruktion ins_{ri} gewählt. Innerhalb eines Trace existieren allerdings meist mehrere Instruktionen, die die Bedingungen an eine Rejoin-Instruktion erfüllen. Beispielsweise würde in Abbildung 5.3c auch Instruktion ins_4 die geforderte Bedingung erfüllen. Stehen mehrere Instruktionen als Rejoin-Instruktion zur Verfügung, erscheint es sinnvoll, eine Instruktion zu wählen, die im Trace möglichst weit vorne vorkommt. Dadurch kann die Anzahl benötigter Join-Kompensationskopien oft reduziert werden. Im Beispiel wäre bei Wahl von Instruktion ins_4 als Rejoin-Instruktion nur eine Join-Kompensationskopie von Instruktion ins_3 notwendig gewesen.

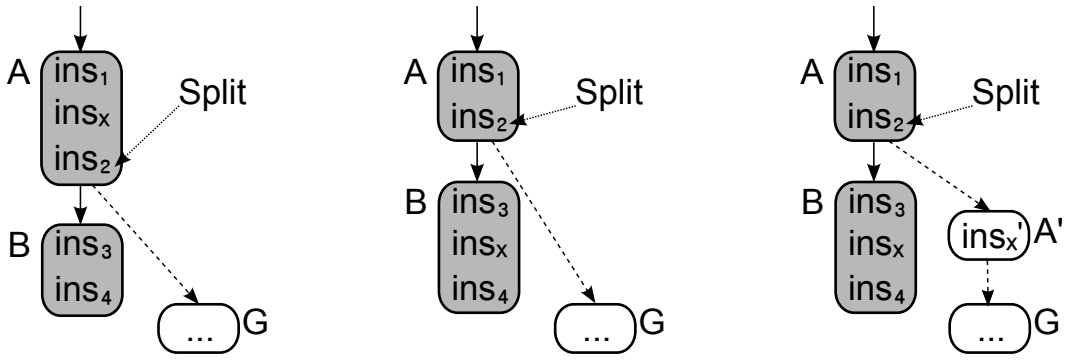
Offensichtlich ist ein Rejoin an dieser Instruktion allerdings nicht direkt möglich, da Kontrollflüsse eines Programms immer nur am Beginn eines Basisblocks zusammengeführt werden können. Bevor der Kontrollfluss der Off-Trace Vorgänger angepasst wird, ist es daher in solch einem Fall notwendig, den Basisblock bb_{ri} aus dem ins_{ri} stammt, so zu teilen, dass ins_{ri} den Beginn des Basisblocks darstellt. Alle Instruktionen, die im Basisblock vor ins_{ri} lokalisiert sind, werden in einen neuen Basisblock verschoben, der im Kontrollflussgraphen und im Maschinencode direkt vor bb_{ri} eingefügt wird. Anschließend kann der Kontrollfluss aller Off-Trace Vorgänger so modifiziert werden, dass bb_{ri} das neue Ziel darstellt.

5.3.2 Split-Kompensation

Bei der Kompensation von Joins kann das Passieren einer Join-Grenze von unten nach oben (aus Sicht des Kontrollflussgraphen), sowie das Passieren von oben nach unten in exakt gleicher Weise behandelt werden. Bei dem Verschieben von Instruktionen über eine Split-Instruktion muss dagegen zwischen beiden Richtungen unterschieden werden.

Zunächst wird der Fall eines Passierens von oben nach unten behandelt. Dabei wird davon ausgegangen, dass beim Verschieben eine einzelne Split-Instruktion passiert wurde. Falls mehrere Split-Instruktionen passiert wurden, kann die Kompensation analog erfolgen.

Abbildung 5.5 deutet diesen Fall für einen Trace, bestehend aus den Basisblöcken A und B, an. Dabei stellt Basisblock A einen Split dar. Die letzte Instruktion in A (ins_2) ist demnach eine Split-Instruktion. Abbildung 5.5b stellt beispielhaft das Ergebnis eines Verschiebens



(a) Vor List Scheduling (b) Nach List Scheduling (c) Nach Kompensation

Abbildung 5.5: Kompensation einer Verschiebung unter eine Split-Instruktion

der Instruktion ins_x unter den Split dar. Beim Passieren des Off-Trace Pfades von A nach G wird diese Instruktion dadurch nicht mehr ausgeführt.

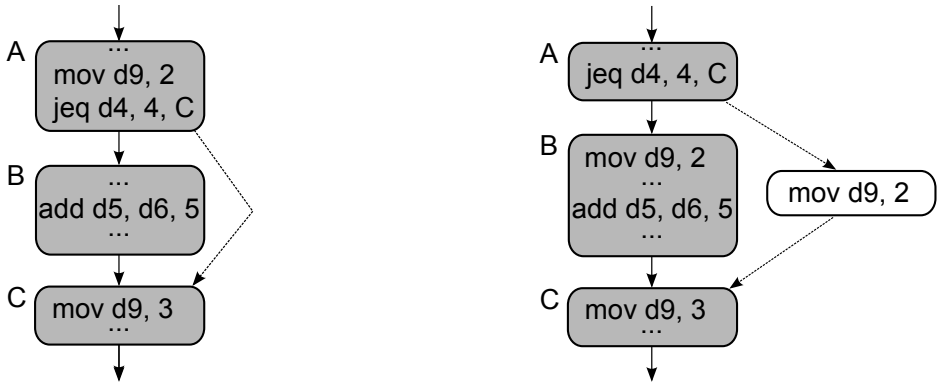
Eine Korrektur der Programmsemantik findet statt, indem Split-Kompensationskopien ermittelt werden. Diese werden anschließend zwischen dem Split und allen Off-Trace Nachfolgern eingefügt.

Definition 5.10 (Split-Kompensationskopien).
*Als **Split-Kompensationskopien** einer Split-Instruktion ins_{split} wird ein Tupel $SK = (ins_1, ..., ins_m)$ von Instruktionen bezeichnet, falls für alle $ins_l \in SK$ gilt:*
$TracePosition(ins_l) < TracePosition(ins_{split}) \wedge$
$SchedulePosition(ins_l) > SchedulePosition(ins_{split})$

Split-Kompensationskopien bestehen demnach aus allen Instruktionen, die im Trace vor dem Split ausgeführt worden wären, nach dem List Scheduling allerdings nicht mehr ausgeführt werden, falls der Kontrollfluss über den Off-Trace Nachfolger läuft. Wie in Abbildung 5.5c gezeigt, werden diese Instruktion als Kopien zwischen dem Split und dem Off-Trace Nachfolger eingefügt.

Wie im Folgenden gezeigt, ist es in einigen Fällen allerdings möglich, auf eine Kopie zu verzichten, und so die Menge an Kompensationscode zu verringern.

Vermeidung von Split-Kompensation



(a) Trace vor List Scheduling

(b) Trace nach List Scheduling und Kompensation

Abbildung 5.6: Trace mit Potential zur Vermeidung einer Split-Kopie

Die Erzeugung einer Split-Kompensationskopie kann vermieden werden, wenn die zu kopierende Instruktion im Off-Trace Pfad nicht *lebendig* ist. Um dies zu veranschaulichen wird im folgenden Beispiel, anders als bisher, mit konkreten Instruktionen in einer Pseudo-Assemblernotation gearbeitet.

Abbildung 5.6 stellt das Ergebnis eines Verschiebens von Instruktion `mov d9, 2` aus Basisblock A in Basisblock B dar. Dabei wurde die Split-Instruktion `jeq d4, 4, C` passiert. Dadurch ist nach der bisherigen Beschreibung von Split-Kompensation eine Kopie von `mov d9, 2` zwischen Basisblock A und dem Off-Trace Nachfolger C notwendig.

Das von `mov d9, 2` beschriebene Register `d9` ist allerdings am Anfang von Basisblock C nicht lebendig: Bevor das Register `d9` in Basisblock C (oder einem Nachfolger) zum ersten Mal verwendet wird, wird `d9` durch die Instruktion `mov d9, 3` in C ein neuer Wert zugewiesen. Eine Split-Kopie von `mov d9, 2` ist in diesem Fall redundant und nicht notwendig.

Um entscheiden zu können, ob eine Instruktion Register beschreibt, die auf dem jeweiligen Off-Trace Pfad lebendig sind, muss zunächst bekannt sein, wo der Kontrollfluss wieder zusammengeführt wird. Wie gezeigt, kann die Kompensation von Joins den Kontrollfluss von Off-Trace Pfaden ändern. Daher erfolgt die Kompensation von Splits in dieser Diplomarbeit erst, nachdem die Kompensation aller Joins abgeschlossen ist. Dadurch sind für alle Splits die korrekten Off-Trace Nachfolger bekannt.

Mittels einer im WCC verfügbaren *Lebendigkeitsanalyse* kann an einem Off-Trace Nachfolger bb_{offt} die Menge aller Register $REG_{live-in}$ bestimmt werden, die zu Beginn von bb_{offt} lebendig sind. Mit dieser Information kann bestimmt werden, welche der Split-Kompensationskopien wirklich notwendig sind und welche redundant sind.

Im Folgenden wird davon ausgegangen, dass ein Tupel von Split-Kompensationskopien $JK = (ins_1, ins_n)$ zwischen bb_{split} und bb_{offt} kopiert werden soll. Um das Tupel der notwendigen Kopien zu bestimmen, muss rückwärts über alle $ins_j \in ins_n ... ins_1$ iteriert werden. Eine Kopie ins_j ist notwendig, falls

- ins_j eine Speicherinstruktion ist.
- ins_j Register beschreibt, die an bb_{offt} lebendig sind.

Sobald eine Kopie als notwendig erkannt wurde, müssen auch alle von dieser Instruktion gelesenen Register als lebendig betrachtet werden, und der Menge $REG_{live-in}$ hinzugefügt werden.

Eine zu kopierende Speicherinstruktion wird dabei immer als zwingend notwendig betrachtet. Nur die Instruktionen, die als notwendig erkannt wurden, werden zwischen Split und Off-Trace Nachfolgern erzeugt. Auf diese Weise kann die Anzahl benötigter Kopien häufig reduziert werden.

5.3.3 Keine Kompensation

Falls beim Verschieben einer Instruktion ins_k aus einem bb_j nach bb_i weder Split- noch Join-Instruktionen passiert wurden, kann diese Verschiebung offensichtlich keine Off-Trace Pfade negativ beeinflussen. Die Instruktion ins_k wurde in diesem Fall innerhalb einer rein sequentiellen Folge von Basisblöcken ohne Seiteneingang und -ausgang verschoben. Es ist keine Kompensation notwendig. Dies ist beispielsweise bei einer Verschiebung von Instruktionen aus Basisblock B nach Basisblock C (und umgekehrt) in Abbildung 5.1 (Seite 43) der Fall. Ein anderer Fall von Basisblock-übergreifenden Verschiebungen, die beim Trace Scheduling keine Kompensation erfordern, sind Verschiebungen von Instruktionen von unten über die Grenzen eines Splits. Diese spekulativen Verschiebungen werden im Folgenden beschrieben.

Spekulative Verschiebungen

Es wird im Folgenden davon ausgegangen, dass eine einzelne Instruktion, die im Trace unterhalb einer Split-Instruktion lokalisiert war, an eine Position oberhalb des Splits verschoben wurde. Das Gesagte gilt analog für die Verschiebung mehrerer Instruktionen.

Bei dieser Art von Verschiebung werden Instruktionen *spekulativ* im Kontrollfluss nach oben verschoben. Eine Instruktion wird dadurch früher ausgeführt, als es eigentlich innerhalb des Kontrollflusses vorgesehen war. Abhängig vom gewählten Kontrollfluss wäre die verschobene Instruktion während einer normalen Programmausführung eventuell auch überhaupt nicht ausgeführt worden.

Für spekulative Verschiebungen ist beim Trace Scheduling keine Kompensation notwendig. Allerdings sind sie nur möglich, wenn sie *sicher* sind. Dazu müssen drei Bedingungen erfüllt sein.

1. Eine spekulativ verschobene Instruktion beschreibt keine Register, die in Off-Trace Pfaden lebendig sind. Da beim spekulativen Verschieben keine Kompensation durchgeführt wird, würde sonst die Semantik geändert werden.

2. Eine spekulativ verschobene Instruktion ändert den Speicherinhalt des Systems nicht.

3. Eine spekulativ verschobene Instruktion kann keine Ausnahmen auslösen. Würde solch eine Instruktion dennoch verschoben, so könnte es nur aufgrund der spekulativen Ausführung innerhalb des Programms zu Ausnahmen kommen, die sonst nicht aufgetreten wären.

Die Einhaltung dieser Bedingungen wird während der Aufstellung des Abhängigkeitsgraphen für das Trace Scheduling sichergestellt. Die Änderungen, die im Vergleich zur Bestimmung des Abhängigkeitsgraphen beim lokalen Instruction Scheduling notwendig sind, werden im Folgenden beschrieben.

5.4 Abhängigkeitsgraph für Traces

Das List Scheduling, das auf einem Trace durchgeführt wird, unterscheidet sich prinzipiell nicht von einem List Scheduling auf einem einzelnen Basisblock: Mit Hilfe des Abhängigkeitsgraphen können ausführbare Instruktionen selektiert und verschoben werden.

Allerdings sind dabei, wie bereits angesprochen, nicht alle denkbaren Verschiebungen möglich. Die zusätzlichen Einschränkungen werden, wie im Folgenden beschrieben, innerhalb des Abhängigkeitsgraphen durch Kanten repräsentiert. Diese Vorgehensweise ermöglicht den Einsatz des lokalen List Schedulers ohne weitere Änderungen.

Grundsätzlich werden die Abhängigkeitskanten zwischen einer Instruktion wie in Kapi-

tel 4.2 beschrieben gebildet. Alle Instruktionen ins_i und ins_j aus der Folge von Instruktionen $I_T = (ins_1, ..., ins_n)$ auf dem Trace T, mit $i < j$, werden auf mögliche Abhängigkeiten untersucht. Falls eine Abhängigkeit besteht, wird eine Kante im DAG erzeugt. Die Unterschiede, die beim Trace Scheduling beachtet werden müssen, werden im Folgenden erläutert.

Abhängigkeiten zwischen Sprunginstruktionen

Mit Sprüngen sind im Folgenden alle Instruktionen gemeint, die eine Änderungen des Kontrollflusses bewirken. Dazu gehören also auch die `call` und `ret`-Instruktionen, die als unbedingte Sprünge betrachtet werden können. In der Arbeit von Fisher[Fis81] sind ursprünglich zwischen zwei Sprunginstruktionen keine Abhängigkeiten vorgesehen. Es wäre damit prinzipiell möglich die Reihenfolge zweier Sprünge zu tauschen. Dies erschwert nicht nur die Kompensation deutlich, sondern erfordert in den meisten Fällen auch eine große Menge an Kompensationscode[LFK+92].

In dieser Arbeit wird daher auf solche Verschiebungen verzichtet. Alle Sprünge bleiben in der Reihenfolge, in der sie auch im Trace vorkommen. Sichergestellt wird dies durch zusätzliche Kontrollflusskanten, die zwischen allen Sprunginstruktionen erzeugt werden.

Abhängigkeiten durch bedingte Sprünge

Um sicherzustellen, dass spekulatives Verschieben nicht zu einer Änderung der Programmsemantik führt, ist zwischen einer bedingten Sprunginstruktion ins_i und einer im Trace folgenden Instruktion ins_j eventuell die Erzeugung einer zusätzlichen Abhängigkeitskante notwendig. Zur Bestimmung dieser Abhängigkeiten wird die Funktion Conreadregs benötigt.

Definition 5.11 (Condreadregs).
Auf der Folge von Instruktionen $I_T = (ins_1, ..., ins_n)$ eines Traces T ist eine Funktion $Condreadregs$ definiert, die bedingten Sprüngen in T eine Menge von Registern zuordnet. Sei ins_i ein bedingter Sprung in I_T. Ein Register r ist in $Condreadregs(ins_i)$ enthalten, wenn es in $Followers(ContainingBB(ins_i)) \setminus \{ContainingBB(ins_{i+1})\}$ mindestens einen Basisblock bb_j gibt, an dessen Beginn r lebendig ist.

Informell ist ein Register r in $Condreadregs(ins_i)$ enthalten, wenn es, abgesehen vom direkt im Trace folgenden Basisblock, mindestens einen weiteren Nachfolger gibt, an dessen Beginn r lebendig ist. Die Informationen von $Condreadregs$ werden während der Aufstellung des DAG genutzt, um ein Verschieben einer Instruktion über einen bedingten Sprung zu verhindern, wenn dadurch Werte überschrieben würden, die in Off-Trace Pfaden benötigt werden (Punkt 1 der Bedingungen für spekulatives Verschieben).

Dazu wird bei der Bestimmung von Abhängigkeiten zwischen einem bedingten Sprung ins_i und einer nachfolgenden Instruktion ins_j eine zusätzliche Kante erzeugt, wenn ins_j

Register aus Condreadregs(ins_i) beschreibt. Diese Abhängigkeit kann also als eine Art der Gegenabhängigkeit betrachtet werden.

Abhängigkeiten durch Instruktionen die Ausnahmen auslösen können

Punkt 3 der Bedingungen für spekulatives Verschieben (Seite 57) fordert, dass eine Instruktion nicht spekulativ über einen Split verschoben werden darf, wenn sie eine Ausnahme auslösen könnte. In der TriCore-Architektur können Lade- und Speicherinstruktion Ausnahmen auslösen. Diese Instruktionen sind daher beim Trace Scheduling von einer spekulativen Verschiebung ausgeschlossen. Sichergestellt wird dies durch eine zusätzliche Kontrollflusskante innerhalb des DAG zwischen allen bedingten Sprüngen und Lade- und Speicherinstruktionen.

Zusammen mit den erzeugten Kontrollflusskanten zwischen allen Sprunginstruktionen ist damit auch die Forderung sichergestellt, dass keine Instruktionen spekulativ verschoben werden, die den Speicherinhalt ändern könnten (Punkt 2 der Bedingungen für spekulatives Verschieben).

5.5 Innere Schleifen

Das bisher in dieser Arbeit beschriebene Trace Scheduling-Verfahren konnte nur auf schleifenfreiem Code durchgeführt werden. Programme enthalten aber meist zahlreiche Schleifen und ein Großteil der WCET kommt oft durch die Ausführung von Schleifen zustande. Fisher beschreibt in seiner Arbeit [Fis81] daher zwei Möglichkeiten, wie Trace Scheduling mit Schleifen umgehen kann. Dabei geht er davon aus, dass der betrachtete Kontrollflussgraph *reduzierbar* ist.

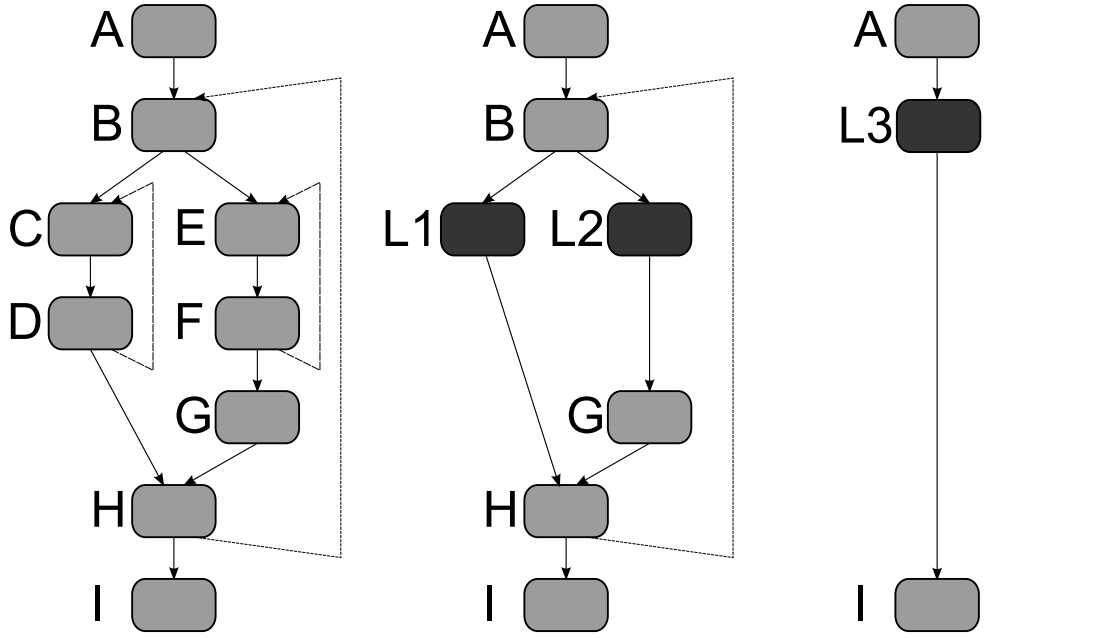
Dadurch lassen sich Schleifen, die im Programm P vorkommen, als Sequenz von Schleifen darstellen: Alle Schleifen L_I in P bilden eine Sequenz $L_1, L_2, ...L_P$, so dass gilt: Wenn L_i und L_j Basisblöcke gemeinsam haben, und $i < j$, dann ist L_i eine innere Schleife von L_j.

Zwei Schleifen innerhalb von P sind also entweder disjunkt oder ineinander geschachtelt und bezüglich des *Enthaltenseins* topologisch sortiert. Als äußerste Schleife L_P wird dabei das gesamte Programm betrachtet.

In Abbildung 5.7a ist ein Ausschnitt aus einem reduzierbaren Kontrollflussgraphen dargestellt, der drei innere Schleifen enthält. Dabei stellen gestrichelte Linien Schleifenrückkanten dar. Die beiden innersten Schleifen $L1 = (C, D)$ und $L2 = (E, F)$ sind disjunkt. Die äußerste Schleife $L3$, die zwischen Basisblock B und H iteriert, enthält die inneren Schleifen komplett. Eine mögliche topologische Sortierung für das Trace Scheduling wäre demnach $L1, L2, L3$.

Dies Forderung nach reduzierbaren Kontrollflussgraphen stellt keine wesentliche Einschrän-

kung für das Trace Scheduling dar, da ein Kontrollflussgraph, der aus strukturiertem Code ohne `gotos` erzeugt wird, automatisch reduzierbar ist.



(a) Reduzierbarer Kontrollflussgraph (b) Innere Schleifen ersetzt (c) Äußerste Schleife durch Loop Representatives ersetzt

Abbildung 5.7: Trace Scheduling auf Kontrollflussgraphen mit inneren Schleifen

Nachfolgend werden die beiden Ansätze von Fisher zum Trace Scheduling von Programmen, die Schleifen enthalten, vorgestellt.

- Im ersten, einfachen Ansatz von Fisher beginnt die Trace-Selektion mit der innersten Schleife L_i der topologischen Sortierung. Basisblöcke für den Trace werden innerhalb der umgebenden Schleife L_i selektiert. Wird eine Schleifenrückkante entdeckt, wird diese wie ein Sprung aus dem Trace behandelt. Der dadurch selektierte Trace ist demnach schleifenfrei und das Trace Scheduling kann darauf ohne weitere Änderungen durchgeführt werden.

  Sobald das Trace Scheduling für L_i abgeschlossen ist, betrachtet das Verfahren anschließend alle weiteren Schleifen L_j der topologischen Sortierung. Wird innerhalb einer Schleife eine Kontrollflusskante entdeckt, die zu einem Schleifenkopf bb_{lh} einer inneren Schleife L_i, mit $i < j$ führt, so muss die Schleife L_i bereits vom Trace Scheduling behandelt worden sein. Demnach ist der Basisblock bb_{lh} bereits Teil eines anderen Traces. Da ein Basisblock immer nur Teil eines einzelnen Traces sein kann, ist während der Trace-Selektion für L_j keine Selektion von Basisblock bb_{lh} möglich.

  Traces sind durch dieses Verfahren also immer schleifenfrei. Der Nachteil ist allerdings, dass mögliches Optimierungspotential ungenutzt bleibt: In typischen Programmen kommen oft zahlreiche relativ kleine Schleifen vor. Die Instruktionen dieser Schleifen nutzen oft nur wenige Ressourcen und Register. Es wäre wünschenswert,

wenn solche Schleifen für das Trace Scheduling keine Grenzen darstellen würden. Dadurch wäre es möglich Instruktionen vor oder hinter eine Schleife zu verschieben.

- Im zweiten von Fisher beschriebenen Ansatz ist es durch den Einsatz von Loop Representatives möglich, Instruktionen vor oder hinter innere Schleifen zu verschieben. Auch bei diesem Ansatz beginnt das Trace Scheduling mit der ersten Schleife der topologischen Sortierung. Nachdem das Trace Scheduling darauf abgeschlossen ist, wird die Schleife im Kontrollflussgraphen durch einen Stellvertreter, der als Loop Representative bezeichnet wird, ersetzt.

  Bei der Betrachtung äußerer Schleifen können Loop Representatives wie elementare Instruktionen behandelt werden. Abbildung 5.7b zeigt den Kontrollflussgraphen aus Abbildung 5.7a, nachdem das Trace Scheduling die beiden innersten Schleifen durch Loop Representatives L1 und L2 ersetzt hat.

  Wird in einem der nächsten Schritte die Trace-Selektion auf einer äußeren Schleife durchgeführt (im Beispiel die Schleife zwischen B und H), so kann einer der beiden Loop Representatives Teil des Trace werden. Die Trace-Selektion sieht an dieser Stelle keine Schleifenrückkante. Um beim nachfolgend durchgeführten List Scheduling keine Abhängigkeiten zu verletzten, muss die Erzeugung des Abhängigkeitsgraphen modifiziert werden: Zwischen einer ins_j, die im Trace vor einem Loop Representative lr_k kommt, werden Abhängigkeitskanten erzeugt, falls eine Abhängigkeit zu einer der Instruktionen besteht, die in lr_k enthalten sind.

  Das List Scheduling kann anschließend ohne Modifikationen durchgeführt werden, da alle Besonderheiten die sich durch Loop Representatives ergeben, im DAG abgebildet werden. Da Loop Representatives für das List Scheduling wie elementare Instruktionen wirken, ist es möglich Instruktionen vor oder hinter innere Schleifen zu verschieben.

Das zweite Verfahren biete durch Loop Representatives offensichtlich größeres Potential für Optimierungen. Daher wurde in dieser Arbeit bei der Implementierung des Trace Scheduling eine Variante dieses Verfahrens genutzt.

5.6 Implementierungs-Details

In diesem Kapitel wird kurz auf einige weitere Aspekte der Implementierung des WCET-fähigen Trace Scheduling innerhalb des WCC eingegangen, die bisher nicht behandelt wurden.

Das Trace Scheduling wird innerhalb des WCC als eine der zahlreichen physikalischen LLIR Optimierungen aufgerufen. Einer Instanz der Klasse `LLIR_TCInstScheduling` wird dazu eine Liste von `LLIR` Objekten übergeben, auf der das Instruction Scheduling durchgeführt werden soll. Die `LLIR` Objekte stellen dabei die Quelldateien des zu übersetzenden Programms nach der Registerallokation dar. Der erste Schritt des Trace Scheduling, die

Trace-Selektion, benötigt für die Auswahl eines Trace Kosteninformationen der Basisblöcke und Kontrollflusskanten.

Kosten Neubestimmung

Zunächst ist daher die Bestimmung dieser Kosten-Informationen notwendig. Dazu wird auf der übergebenen Liste von `LLIR` Objekten die WCET Analyse mittels aiT durchgeführt. Ist alternativ eine Optimierung basierend auf ACET-Werten angestrebt, wird stattdessen ein ACET-Profiling mittels CoMET durchgeführt. Anschließend sind Kosteninformationen in Form von WCET Objectives (bzw. ACET Objectives) verfügbar, die von der Trace-Selektion genutzt werden können.

Nachdem die Kosten bestimmt sind, wird über alle Funktionen aller `LLIR`-Instanzen iteriert und mittels der Trace-Selektion jeweils ein Trace bestimmt. Auf dem ermittelten Traces wird das Trace Scheduling durchgeführt. Wie in Kapitel 2.5.1 beschrieben, kann es durch das Trace Scheduling zu Pfadwechseln kommen. Daher ist eine erneute Bestimmung der Kosteninformationen notwendig, nachdem alle Funktionen einmal betrachtet wurden. Allerdings kann die Neuberechnung der WCET für einige Programme sehr zeitintensiv sein.

Daher ist über einen Konfigurationsparameter bestimmbar, nach wie vielen Iterationen eine neue Analyse durchgeführt werden soll. An dieser Stelle ist es demnach möglich auf Präzision bei der Trace-Selektion zu verzichten, um so eine bessere Laufzeit des Trace Scheduling zu ermöglichen. Wie die Auswertung in Kapitel 7 zeigt, bringt eine Neuberechnung der WCET nach jeder Iteration meist wenig Vorteile. Daher kann darauf in vielen Fällen verzichtet werden.

Das beschriebene Trace Scheduling-Verfahren wird solange wiederholt bis in keiner Funktion des zu optimierende Programms weitere Traces ermitteln werden können. Bei komplexen Programmen kann dies sehr zeitintesiv sein. Gleichzeitig sind mit fortschreitender Dauer des Trace Scheduling meist kaum noch Verbesserungen zu erwarten, da alle kritischen Pfade schon zu Beginn betrachtet worden sind. Um die Laufzeit zu beschränken ist alternativ ein Abbruch des Trace Scheduling nach einem konfigurierbaren *Timeout*-Wert möglich.

Rollback Mechanismus

Wie bereits angedeutet, kann nicht verhindert werden, dass es durch das Trace Scheduling zu einer Verschlechterung der WCET/ACET kommt. Aus diesem Grund wurde in dieser Diplomarbeit ein *Rollback-Mechanismus* für das Trace Scheduling implementiert. Dieser Mechanismus stellt sicher, dass auf das beste Ergebnis, das während des Trace Scheduling erzielt wurde, zurückgegriffen werden kann. Konnte durch das Trace Scheduling keine Verbesserung erzielt werden, so wird auf das unoptimierte Programm zurückgefallen.

Nach jeder Analysephase mittels aiT beziehungsweise CoMET werden dazu die aktuellen

Kosten (WCET/ACET) des gesamten Programms bestimmt. Ist der neue Werte besser als der aktuelle Bestwert, so wird der neue Werte als Bestwert übernommen. Durch die Darstellung des zu übersetzenden Programms in Form einer Liste von `LLIR` Objekten lässt sich leicht eine Kopie des aktuellen Zustands des Programms anfertigen. Diese Kopie wird als lokales Optimum gespeichert.

Nach Abschluss der Trace Scheduling wird auf die `LLIR` zurückgegriffen, die dem ermittelten Bestwert entspricht. Somit kann es einerseits durch das Trace Scheduling nie zu einer Verschlechterung der WCET/ACET gegenüber dem unoptimierten Programm kommen. Andererseits ist es dadurch auch möglich zeitweise Verschlechterungen zu akzeptieren, um so lokale Minima zu überwinden.

6 WCET-fähiges Superblock Scheduling

Wie Kapitel 5.3 gezeigt hat, ist die Kompensation von Instruktionen, die über Splits verschoben wurden, relativ simpel und führt meist zu wenig Kompensationscode. Handelt es sich bei den Verschiebungen um spekulativen Verschiebungen ist keine Kompensation notwendig. Die Möglichkeit redundante Splitkopien zu erkennen und zu verhindern, schränkt die benötigte Menge an Kompensationscode weiter ein.

Die Kompensation von Verschiebungen die Seiteneingänge eines Trace betreffen, ist hingegen aufwändig und führt oft zu starken Änderungen des Kontrollflussgraphen und damit zu einer großen Menge an Kompensationscode. Um die Schwierigkeiten bei der Kompensation an Seiteneingängen zu umgehen, führten Mahlke et al. [HMC+93] im IMPACT-Projekt das Konzept des Superblocks ein. Ein Superblock ist ein eingeschränkter Trace, auf dem keine Seiteneingänge vorkommen dürfen. Dadurch ist bei einem Superblock Scheduling offensichtlich keine Kompensation an Seiteneingängen notwendig.

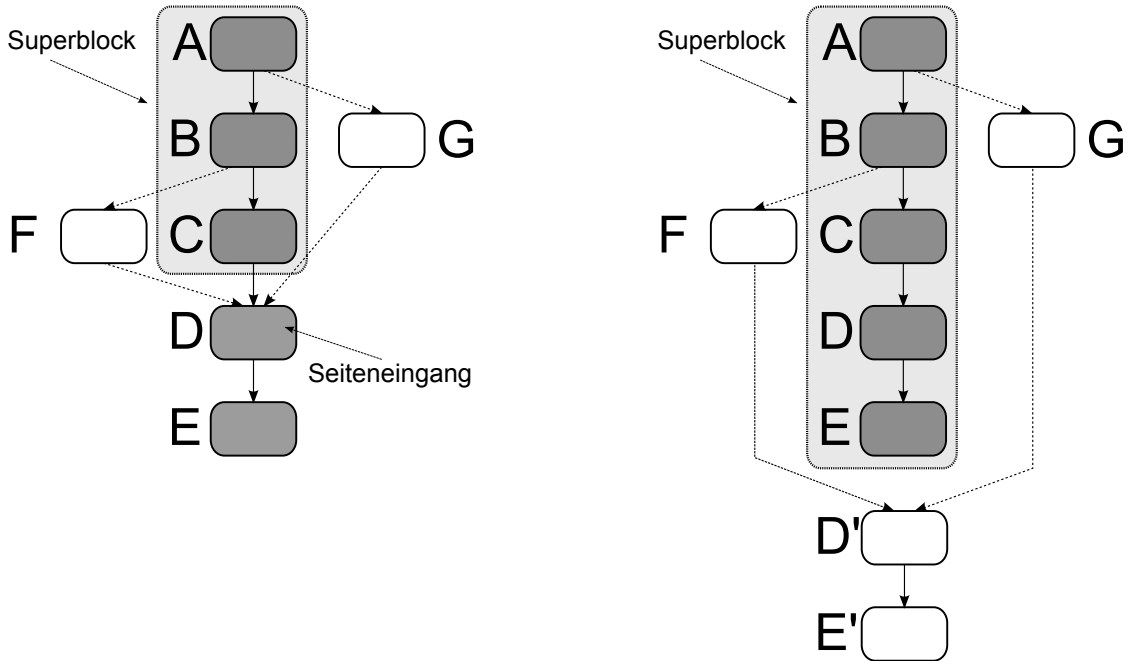
Das in dieser Diplomarbeit entwickelte Trace Scheduling führt ein Instruction Scheduling auf nicht eingeschränkten Traces durch. Demnach ist es auch in der Lage ein Instruction Scheduling auf Superblöcken durchzuführen. Die Unterschiede, die sich dadurch bezüglich einer WCET ergeben soll in diesem Kapitel erläutert werden.

Zunächst wird dazu das Konzept des Superblocks ausführlicher vorgestellt. Anschließend wird gezeigt, wie im Kontrollflussgraphen vorkommende natürliche Superblöcke durch den Prozess der Tail Duplication vergrößert werden können um so mehr Potential für das Instruction Scheduling zu schaffen. Abschließend wird eine in dieser Arbeit entwickelte Erweiterung vorgestellt, die Pfadwechsel, die durch die Tail Duplication entstehen, verringern soll.

6.1 Superblock

Definition 6.1 (Superblock).
*Ein **Superblock** S ist ein Trace $T = (b_{start}, ..., b_{end})$, der nur am Startknoten b_{start} mehr als eine eingehende Kontrollflusskante $e \in E$ aufweisen darf.*

In einem Superblock können demnach keine Seiteneingänge beziehungsweise Joins vorkommen. Die Schwierigkeiten im Zusammenhang mit der Kompensation von Seiteneingängen, die beim Trace Scheduling auftreten, sind bei einem Superblock dadurch nicht möglich. Ein Superblock kann allerdings weiterhin Seitenausgänge beziehungsweise Splits enthalten.



(a) Superblock vor Tail Duplication

(b) Superblock nach Tail Duplication

Abbildung 6.1: Superblock vor und nach Tail Duplication

In Abbildung 6.1a ist ein Kontrollflussgraph mit einem Superblock bestehend aus den Basisblöcken A, B und C dargestellt. Da die in einem Kontrollflussgraph vorkommenden natürlichen Superblöcke im Vergleich zu uneingeschränkten Traces meist sehr klein sind, werden Superblöcke durch eine Tail Duplication vergrößert. Die Bildung eines Superblock S läuft demnach in zwei Phasen ab:

1. Bestimmung eines Trace T. Die Selektion eines Trace für einen Superblock kann dabei prinzipiell genau wie in Kapitel 5.2 gezeigt erfolgen.

2. Elimination eventuell vorhandener Seiteneingänge des Trace T durch die Tail Duplication. Anschließend stellt T einen Superblock S dar.

Ist ein Superblock gebildet, kann darauf ein Trace Scheduling genau wie auf allgemeinen Traces durchgeführt werden. Es ist also möglich den Aufbau des Abhängigkeitsgraphen ohne Änderungen vom Trace Scheduling zu übernehmen. Alle zusätzlichen Kontrollflusskanten, die dabei bei Traces mit Seiteneingängen eventuell erzeugt werden, können bei einem Superblock nicht auftreten.

Auch an der Implementierung des List Schedulers und bei der anschließenden Kompensation sind keine Änderungen notwendig. Zusätzliche Arbeit zur Entwicklung des WCET-fähigen Superblock Scheduling ist, aufbauend auf dem Trace Scheduling, also grundsätzlich nur bei der Implementierung der Tail Duplication notwendig. Im Folgenden wird auf dieses Verfahren eingegangen.

6.2 Tail Duplication

Abbildung 6.1a zeigt einen Ausschnitt aus einem Kontrollflussgraphen, in dem ein Trace bestehend aus den Basisblöcken A, B, C, D und E selektiert wurde. Da ein Superblock keine Seiteneingänge aufweisen darf und Basisblock D einen Seiteneingang darstellt, besteht ein natürlicher Superblock in diesem Beispiel nur aus den markierten Basisblöcken A, B und C.[1]

Durch eine Tail Duplication ist es möglich, den Superblock zu vergrößern und damit mehr Möglichkeiten für weitere Optimierungen auf diesem Superblock zu schaffen. Dazu wird beginnend mit dem Basisblock, der den ersten Seiteneingang des Trace darstellt, eine Kopie aller Basisblöcke bis zum Ende des Trace (*Tail*) erzeugt. Anschließend wird der Kontrollfluss aller Off-Trace Vorgänger so korrigiert, dass diese die kopierten Blöcke als Ziel haben. Am Trace sind danach keine Seiteneingänge mehr vorhanden und er stellt somit einen Superblock dar.

Das Ergebnis einer Tail Duplication ist in Abbildung 6.1b dargestellt. In diesem Beispiel wurde die Tail Duplication beginnend mit dem Seiteneingang an Basisblock D durchgeführt. Dazu wurden die Basisblöcke D und E kopiert, und der Kontrollfluss der Off-Trace Vorgänger am Seiteneingang D zu der Kopie D´ korrigiert. Dadurch stellt Basisblock D keinen Seiteneingang in den Trace mehr dar, so dass der Superblock auch die Basisblöcke D und E umfassen kann.

Anschließend kann das Superblock Scheduling auf dem Superblock A, B, C, D, E durchgeführt werden. Hierbei wird die in Kapitel 5 beschriebene Implementierung des Trace Scheduling wiederverwendet. Nachdem das Instruction Scheduling auf dem Superblock abgeschlossen ist und Kompensationscode erzeugt wurde, kann wie beim Trace Scheduling ein weiterer Trace selektiert werden. Dabei könnte ein neuer Trace auch die kopierten Tail-Basisblöcke D´ und E´ enthalten.

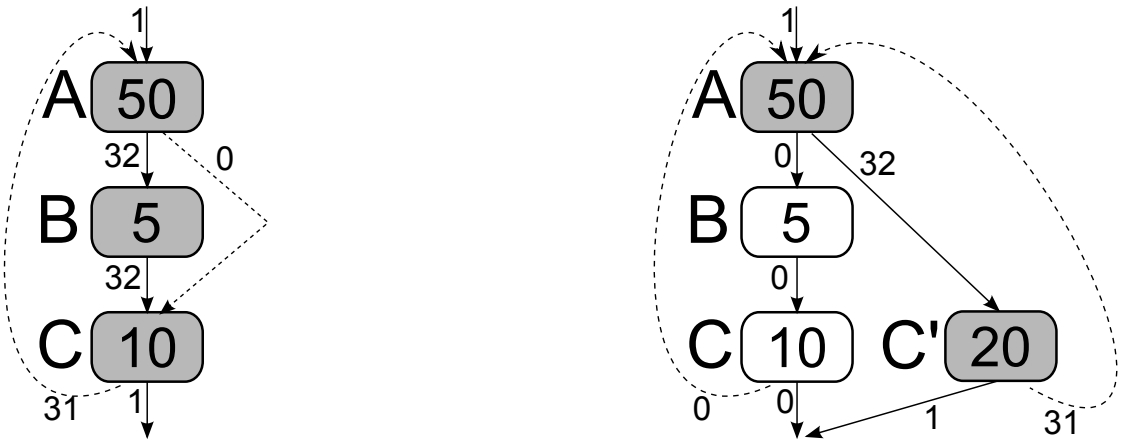
Beispielsweise wäre es in Abbildung 6.1b möglich, einen Trace bestehend aus den Basisblöcken F, D´, E´ zu selektieren. Für diesen Trace stellt D´ wiederum einen Seiteneingang dar, wodurch eine erneute Tail Duplication notwendig wird. Es wäre daher auch möglich, die Tail Duplication direkt so durchzuführen, dass für jeden Off-Trace Vorgänger an einem Seiteneingang separate Kopien der Tail-Basisblöcke erzeugt werden. Da während der Tail Duplication allerdings noch nicht feststeht, ob die kopierten Tail-Basisblöcke tatsächlich Teil eines weiteren Traces werden, wäre solch ein Vorgehen allerdings nicht sinnvoll. Es würde dadurch nur ein unnötiges Codewachstum verursacht werden.

Das mit der Tail Duplication verbundene Codewachstum wird im Zusammenhang mit dem Superblock Scheduling meist als größter Nachteil gegenüber dem Trace Scheduling gesehen. Im Folgenden soll betrachtet werden, welche Auswirkungen die Tail Duplication auf die WCET haben kann.

[1] Die Basisblöcke D und E würden zusammen einen weiteren Superblock darstellen.

6.2.1 Pfadwechsel durch Tail Duplication

Im Gegensatz zum Trace Scheduling, bei dem es nur zu Codewachstum kommt, falls dies zur Kompensation von Verschiebungen notwendig ist, führt eine Tail Duplication grundsätzlich zu Codewachstum. Aufgrund des relativ großen Speichers des TriCore stellt ein Codewachstum prinzipiell kein ernsthaftes Problem dar. Die Duplikation von Basisblöcken kann allerdings zu Pfadwechseln führen und damit eine negative Auswirkung auf die WCET haben. Darauf soll im Folgenden eingegangen werden.



(a) WCEP in *Do-While*-Schleife vor Tail Duplication

(b) WCEP in *Do-While*-Schleife nach Tail Duplication

Abbildung 6.2: Beispiel eines durch Tail Duplication verursachten Pfadwechsels

Abbildung 6.2a stellt eine Ausschnitt aus einem Kontrollflussgraphen dar, der für den Benchmark startup_fixed vom WCC erzeugt wird. Innerhalb des Kontrollflussgraphen stellen die Basisblöcke A, B und C eine Schleife dar. Diese Schleife ist aus einer *Do-While*-Schleife im Programmcode erzeugt worden. Dabei wird Basisblock B abhängig von einer Bedingung, die innerhalb des Basisblock A geprüft wird, ausgeführt. Die Basisblöcke A, B und C bilden demnach eine *if-then*-Verzweigung.

Für diese Verzweigung kann während der WCET-Analyse durch aiT die Sprungbedingung nicht statisch bestimmt werden. Es ist damit nicht klar, in welchem Kontext der *then*-Teil, und damit Basisblock B, ausgeführt wird und in welchem Kontext die Ausführung von Basisblock B übersprungen wird. Da die Ausführung von Instruktionen in Basisblock V offensichtlich teurer als ein Sprung von Basisblock A nach C ist, muss aiT annehmen, dass Basisblock B auf dem WCEP liegt. Der grau markierte WCEP beinhaltet demnach die Basisblöcke A, B und C. Die von aiT ermittelten WCET-Werte sind in den Basisblöcken annotiert. An den Kanten sind die ermittelten WCEC-Werte dargestellt.

Soll auf der dargestellten Schleife ein Superblock Scheduling durchgeführt werden, so ist zunächst eine Tail Duplication notwendig, da Basisblock C einen Seiteneingang darstellt. Die Tail Duplication erzeugt eine Kopie C´ von Basisblock C und lenkt den Kontrollfluss zwischen Basisblock A und C zu Basisblock C´ um. Das Ergebnis ist in Abbildung 6.2b dargestellt.

Auf dem so gebildeten Superblock A, B, C könnte ein Superblock Scheduling durchgeführt werden. Dies würde allerdings nicht zur erhofften WCET-Reduktion führen. Wie in Abbildung 6.2b dargestellt, hat der kopierte Basisblock C´ eine deutlich höhere WCET als Basisblock C. Daher kommt es an dieser Stelle zu einem Pfadwechsel.

Dieser Pfadwechsel lässt sich durch die Positionierung von Basisblock C´ innerhalb des Maschinencodes erklären. Dabei spielen zwei Punkte eine Rolle.

***Instruction-Cache Misses*:** Bei der Bestimmung der WCET berücksichtigt aiT auch den Zustand des Instruction-Caches. Durch das Einfügen von C´ in den Maschinencode verschlechtert sich die *räumliche Lokalität* des Codes. Dadurch kann es im gezeigten Beispiel einerseits bei einem Sprung von Basisblock A nach Basisblock C´ und anschließend beim Rücksprung zu Basisblock A zu teuren Instruction-Cache Misses kommen. Vor der Tail Duplication kam es bei einem Sprung zu Basisblock C noch nicht zu diesen Instruction-Cache Misses. Dies führt zu der dargestellten erhöhten WCET von Basisblock C´.

Zusätzliche Sprung-Instruktionen: Durch die Position des kopierten Basisblocks C´ im Maschinencode ist beim Verlassen der Schleife ein zusätzlicher Sprung notwendig, der den Kontrollfluss zum Schleifennachfolger (nicht dargestellt) führt. Vom nicht kopierten Basisblock C aus kann der Schleifennachfolger dagegen direkt erreicht werden, da dieser auch physikalisch im Maschinencode folgt.

Da ähnliche Konstellationen wie in Abbildung 6.2 häufig in Programmen vorkommen, wurde im Rahmen dieser Diplomarbeit eine Heuristik entwickelt, die eine Tail Duplication und damit einen Pfadwechsel in solchen Situationen verhindern soll.

6.2.2 Vermeidung von Tail Duplication bei kritischen *if-then*-Verzweigungen

Das Problem des Pfadwechsels in Abbildung 6.2 tritt nur auf, da die WCET des *then*-Teils (Basisblock B) relativ gering ist. Wäre die WCET groß, so würde eine Tail Duplication von Basisblock C wahrscheinlich nicht zu einem Pfadwechsel führen, da die zusätzlichen Kosten durch Instruction-Cache Misses weniger stark wiegen, als die Kosten der Ausführung von Basisblock B.

Eine andere Situation, in der es nicht zu einem Pfadwechsel kommen kann, ist gegeben, wenn aiT erkennt, dass die alternative Kante (in diesem Beispiel von A nach C) *infeasible* ist. Ein Kontrollfluss über diese Kante ist demnach unmöglich.

Die in Algorithmus 6.1 gezeigte Heuristik, um diese Situationen zu erkennen, wurde in die Trace-Selektion des WCC integriert. Sie sorgt dafür, dass ein Trace vor einem kritischen Basisblock, wie Basisblock C, stoppt, so dass keine Tail Duplication notwendig wird. Die Heuristik wird für eine Vorwärts-Verlängerung des Trace beschrieben. Die Rückwärts-Richtung läuft analog ab.

In Algorithmus 6.1 ist dargestellt, wie entschieden wird, ob ein Trace, der aktuell mit Basisblock b endet, um einen Basisblock *succ* verlängert werden soll. Falls *succ* kein Join

Algorithmus 6.1 Vermeidung von *if-then*-Tail Duplication

Eingabe: Trace Basisblock *b*, Verlängerungskandidat *succ*.
Ausgabe: true, wenn Trace-Verlängerung gestoppt werden soll.

```
1: if ( ! istJoin(succ) ) then
2:     // Kein Seiteneingang -> keine Tail Duplication
3:     return false;
4: end if

5: // Minimale Kosten des then-Teils
6: MinKostenThen:=50;
7: for ( Alle CFG-Vorgänger pred von succ ohne b ) do
8:     if ( infeasible( pred, succ ) ) then
9:         //  Tail Duplication bei infeasible Pfad nicht kritisch
10:        continue;
11:    end if

12:    thenKosten:=summiere WCET-Werte auf Pfad zwischen pred -> succ;
13:    if ( thenKosten < MinKostenThen ) then
14:        // Stoppe Trace-Verlängerung, da Pfadwechsel möglich
15:        return true;
16:    end if
17: end for

18: return false;
```

ist, kann die Verlängerung ohne weitere Überprüfung stattfinden, da eine Tail Duplication nur notwendig ist, falls Joins auf dem Trace vorkommen (Zeile 2). Ist *succ* ein Join, so werden alle Off-Trace Vorgänger (Zeile 7) von *succ* betrachtet. Ist die Kontrollflusskante zwischen Off-Trace Vorgänger *pred* und *succ* von aiT als infeasible erkannt (Zeile 8), kann es an dieser Stelle nicht zu einem Pfadwechsel kommen. Solche Kanten sind demnach nicht kritisch und werden nicht weiter betrachtet.

Für alle anderen Vorgänger werden die WCET-Werte der Basisblöcke, die zwischen *pred* und *succ* liegen, summiert (Zeile 12). Im Beispiel aus startup_fixed (Abbildung 6.2) würde an dieser Stelle für *pred* = Basisblock A und *succ* = Basisblock C die WCET von Basisblock B mit 5 ermittelt. Ist die ermittelte Summe auf dem Pfad für einen der betrachteten Off-Vorgänger niedriger, als ein festgelegter Schwellenwert (Zeile 13), so stoppt die Trace-Verlängerung vor dem Join-Basisblock *succ*. In dieser Diplomarbeit wurde der Schwellenwert *MinKostenThen* nach Versuchen auf 50 festgelegt. Im Beispiel wäre Basisblock C demnach nicht in den Trace mit aufgenommen worden und es wäre keine Tail Duplication notwendig.

Die Heuristik erwies sich auch beim Trace Scheduling in einigen Benchmarks als nützlich. Beim Trace Scheduling verhindert sie, dass es durch das Verschieben von Instruktionen in den *then*-Teil zu Kompensation kommt. Diese Kompensation hat den gleichen Effekt wie die beschriebene Tail Duplication, indem sie aus einer *if-then*-Verzweigung quasi eine *if-then-else*-Verzweigung macht. Auch dabei besteht die Gefahr des Pfadwechsels. Dies kommt beim Trace Scheduling allerdings seltener vor, da dies dort nur passiert, wenn es auch einen Grund für eine Verschiebung von Instruktionen in den *then*-Teil gibt.

7 Ergebnisse

Um die erstellten Optimierungen zu evaluieren, wurden umfangreiche Tests durchgeführt, deren Ergebnisse in diesem Kapitel zusammengefasst sind. Einerseits dienten diese Tests dazu, die Korrektheit der implementierten Optimierungen sicherzustellen (Kapitel 7.1) und andererseits dazu, die Optimierungen durch Messung der erzielten WCET- sowie ACET-Werte zu bewerten (Kapitel 7.2).

7.1 Korrektheitstests

Um sicherzustellen, dass die implementierten Optimierungen korrekt arbeiten, wurde das Trace Scheduling und das Superblock Scheduling jeweils anhand der *Testbench* des WCC überprüft. Diese Testbench enthält zahlreiche Benchmarks, die mit speziellen Aufrufen an *check-Funktionen* durchsetzt sind. Dadurch lassen sich Werte von internen Variablen des Benchmarks und Ergebnisse, die der Benchmark nach einer Übersetzung ohne Optimierungen berechnet, mit den Werten nach den Optimierungen vergleichen.

Falls es durch eine Optimierung zu einer Codetransformation kommt, die die Semantik ändert, besteht eine sehr große Wahrscheinlichkeit, dass dadurch auch in mindestens einem der Benchmarks die Ausgaben verfälscht werden. Falls es zu einer Abweichung kommt, gilt dieser Benchmark in der Testbench als nicht bestanden. Das Trace Scheduling und das Superblock Scheduling haben alle Benchmarks bestanden.

7.2 Quantitative Evaluation

Die Leistungsfähigkeit des WCET-fähigen Trace Scheduling und des WCET-fähigen Superblock Scheduling wurden anhand von 51 ausgewählten Benchmarks der bestehenden *WCET-Testbench* des WCC analysiert. Der Code wurde für die Zielhardware Infineon TC1797 kompiliert und optimiert. Für die Analysen und Simulationen wurde der Programmcode im gecachten Flash-Speicher des TriCore abgelegt. Die ACET-Werte wurden mit dem Simulator CoMET ermittelt, wobei die in der Testbench des WCC für die Benchmarks verfügbaren Eingabedaten benutzt wurden.

Alle Tests wurden mit der höchsten Optimierungsstufe (O3) durchgeführt, da ein bereits stark optimierter Code weiter verbessert werden soll, statt direkt auf dem Eingabecode zu arbeiten. Standardmäßig wird bei dieser Optimierungsstufe im WCC das lokale Instruction Scheduling nach der Registerallokation durchgeführt. Dieses Instruction Scheduling-

Verfahren wurde zur Ermittlung der WCET- und ACET-Werte der beiden neu implementierten Instruction Scheduling-Verfahren deaktiviert.

7.2.1 Auswirkung von Trace Scheduling auf die WCET

Abbildung 7.1 zeigt die durch das WCET-fähige Trace Scheduling erreichte Reduktion der von aiT ermittelten WCET-Werte. Dabei sind einmal die relativen Werte im Vergleich zur WCET von Code, der bei der Optimierungsstufe O3 ohne lokales Instruction Scheduling generiert wird, dargestellt. Standardmäßig wird im WCC bei O3 ein lokales Instruction Scheduling durchgeführt. Der Vergleich mit der WCET dieses Codes ist ebenfalls dargestellt. Das Trace Scheduling wurde für diese Versuchsreihe mit folgenden Parametern durchgeführt:

- Kostenmetrik zur Trace-Selektion: WCET.
- Neuberechnung der WCET durch aiT nachdem jede Funktion einmal betrachtet wurde.
- Timeout für das Scheduling: 3600 Sekunden.

Zunächst sollen die dargestellten Ergebnisse kurz erläutert werden. Das Ziel des Trace Scheduling sollte eine Reduktion der WCET im Vergleich zu O3 ohne lokales Instruction Scheduling sein. Daher sollte die dargestellte relative WCET kleiner als 100% sein. Für die meisten betrachteten Benchmarks ist dies der Fall. Wie bereits angesprochen existieren allerdings auch Benchmarks, in denen durch das Trace Scheduling keine Reduktion der WCET möglich ist oder es sogar zu einer Verschlechterung der WCET kommen kann.

Durch den implementierten Rollback-Mechanismus ist es möglich, solche Situationen zu erkennen und die Optimierungen rückgängig zu machen. Es sollten demnach keine Werte größer als 100% durch das Trace Scheduling erreicht werden. Benchmarks in denen ein Rollback notwendig ist, sind beispielsweise codecs_codrle1 und codecs_codhuff. Die WCET-Werte sind dadurch im Vergleich zu O3 ohne lokales Instruction Scheduling unverändert.

Im Vergleich zur WCET, die bei O3 mit lokalem Instruction Scheduling ermittelt wird, tritt bei diesen Benchmarks allerdings eine relative Verschlechterung der WCET auf. Dies kommt zustande, da das lokale Instruction Scheduling zu einer Reduktion der WCET führt, während beim Trace Scheduling auf das unoptimierte Programm zurückgefallen wird.

Interessant ist der Effekt, der beispielsweise bei fir2dim_fixed und matrix1_fixed auftritt. Trace Scheduling führt hier im Vergleich zu O3 ohne lokales Instruction Scheduling nicht zu einer Reduktion der WCET. Im Vergleich zu O3 mit lokalen Scheduling tritt aber eine recht deutliche Reduktion der WCET auf. Der Grund dafür liegt in der nicht vorhandenen Berücksichtigung von WCET-Daten beim lokalen Instruction Scheduling. Die Optimierungen, die dadurch vorgenommen werden, erscheinen zur Reduktion der ACET prinzipiell sinnvoll, führen aber zu einer Erhöhung der WCET. Im relativen Vergleich

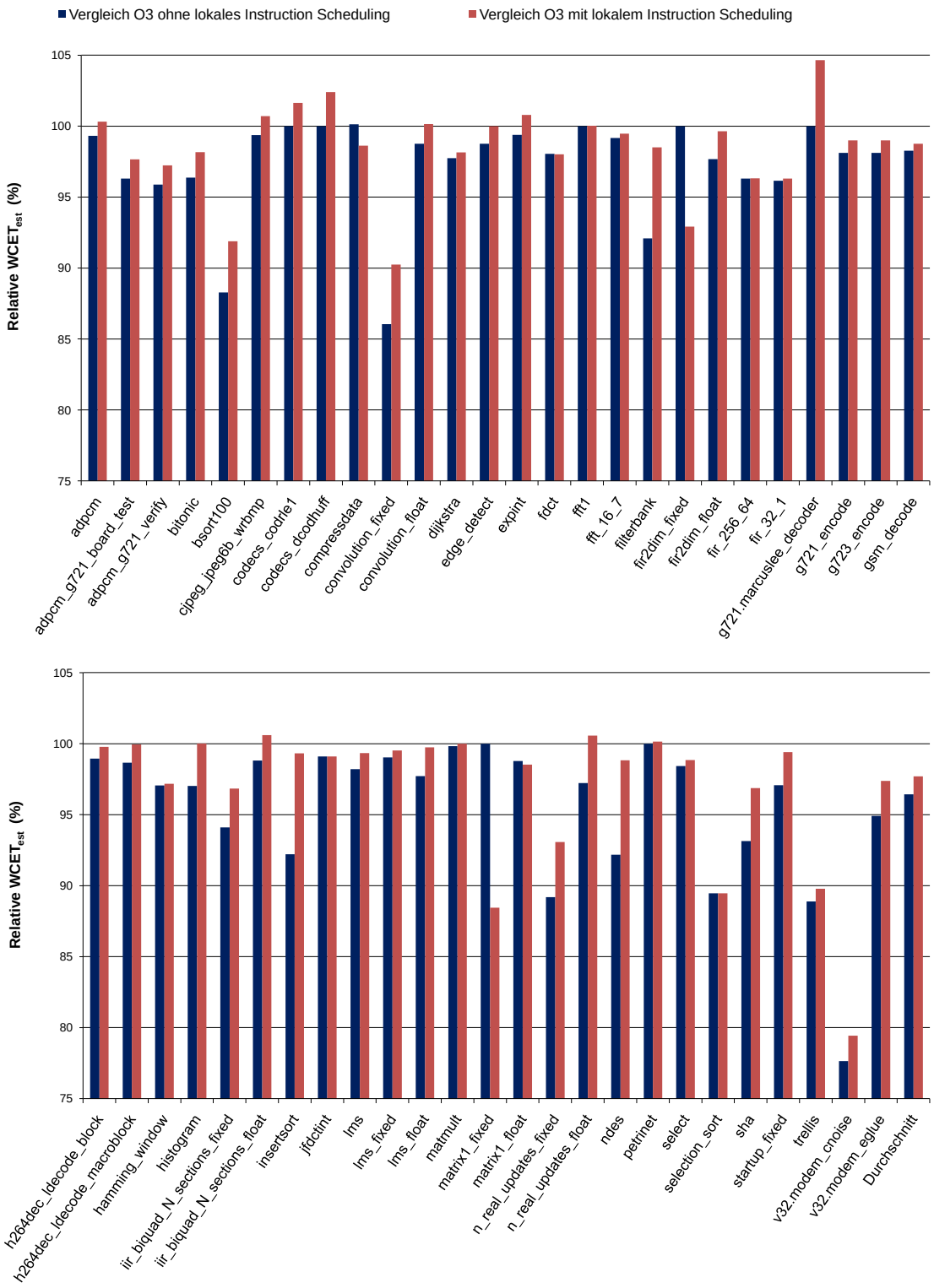


Abbildung 7.1: Relative WCET-Werte nach Trace Scheduling
(100% ≡ WCET O3, mit/ohne lokales Instruction Scheduling).

erzielt das Trace Scheduling durch den Rollback auf den unoptimierten Stand daher in diesen Benchmarks eine relative Verbesserung der WCET.

Insgesamt ergibt sich durch das Trace Scheduling eine relative Reduktion der WCET der betrachten Benchmarks auf 96,4% (ohne lokales Instruction Scheduling) beziehungsweise 97,7% (mit lokalem Instruction Scheduling).

Für die weiteren Auswertungen dienen nun nur noch die Werte, die durch das lokale Instruction Scheduling bei O3 erzielt werden, als Vergleichsbasis.

7.2.2 Auswirkung unterschiedlicher Kostenmetriken

Als nächstes wurde die Auswirkung unterschiedlicher Kostenmetriken der Trace-Selektion auf die WCET sowie die ACET untersucht. Dabei sollte das Trace Scheduling, das mit WCET-Daten arbeitet, eine größere Reduktion der WCET erreichen, als das Profiling-basierte Trace Scheduling, das ACET-Daten von CoMET nutzt. Umgekehrt sollte das Profiling-basierte Trace Scheduling eher dazu geeignet sein, die ACET zu reduzieren.

Abbildung 7.2 stellt die ermittelten WCET-Werte nach dem Trace Scheduling im Vergleich zu den Werten bei der Optimierungsstufe O3 mit lokalem Instruction Scheduling dar. Dabei wurde das Trace Scheduling einmal, wie in Abbildung 7.1, basierend auf WCET-Daten von aiT durchgeführt. Zu Vergleichszwecken werden die WCET-Werte dargestellt, die sich ergeben, wenn das Trace Scheduling basierend auf Profiling-Daten von CoMET durchgeführt wird.

Dabei erreicht das WCET-basierte Trace Scheduling in allen Benchmarks Ergebnisse, die mindestens so gut sind, wie die Ergebnisse des Profiling-basierten Trace Scheduling. Im Schnitt beträgt die relative WCET dabei, wie bereits gezeigt, 97,7%. Das Profiling-basierte Trace Scheduling kommt auf durchschnittliche WCET-Werte von 99,9%.

In Abbildung 7.3 wurden die gleichen Optimierungen durchgeführt, dabei aber die ACET-Werte nach dem Trace Scheduling mit den Werten bei O3 mit lokalem Instruction Scheduling verglichen. Wie erhofft, führt hier das Profiling-basierte Trace Scheduling im Durchschnitt zu einer größeren Reduktion der ACET und kommt auf einen relativen Wert von 97,8%. Das WCET-basierte Trace Scheduling erreicht eine Reduktion der ACET auf durchschnittlich 98,5%.

Das bessere Abschneiden des WCET-basierten Trace Scheduling zur Reduktion der WCET sowie des Profiling-basierten Trace Scheduling zur Reduktion der ACET liegt einerseits an der unterschiedlichen Wahl von Traces zur Optimierung. Wie erwartet entsprechen die am häufigsten ausgeführten Programmpfade nicht zwangsläufig dem WCEP.

Ein weiterer wichtiger Grund ist andererseits der Rollback-Mechanismus (Kapitel 5.6). Während eines WCET-basierten Trace Scheduling werden WCET-Daten als Grundlage für den Rollback benutzt, während des Profiling-basierten Trace Scheduling dementsprechend ACET-Daten. Gut sichtbar sind die Unterschiede, die sich dadurch ergeben, bei dem Benchmark petrinet. Bei diesem Benchmark ist keine Reduktion der WCET durch

das Trace Scheduling möglich. Das WCET-basierte Trace Scheduling erreicht während der Optimierung allerdings eine leichte Reduktion der ACET. Da das Ziel allerdings eine Reduktion der WCET ist, fällt der Rollback-Mechanismus auf das unoptimierte Programm zurück, so dass die relativen ACET- und WCET-Werte für petrinet 100% betragen.

Das Profiling-basierte Trace Scheduling führt ein Rollback basierend auf ACET-Werten durch. Da es während des Trace Scheduling zu einer Reduktion der ACET-Werte kommt, ist kein Rollback notwendig. Die gefundene Lösung ist wie in Abbildung 7.3 gezeigt deutlich besser als die Lösung des lokalen Instruction Scheduling. Allerdings wurde wie in Abbildung 7.2 gezeigt dadurch eine Verschlechterung der WCET erzielt.

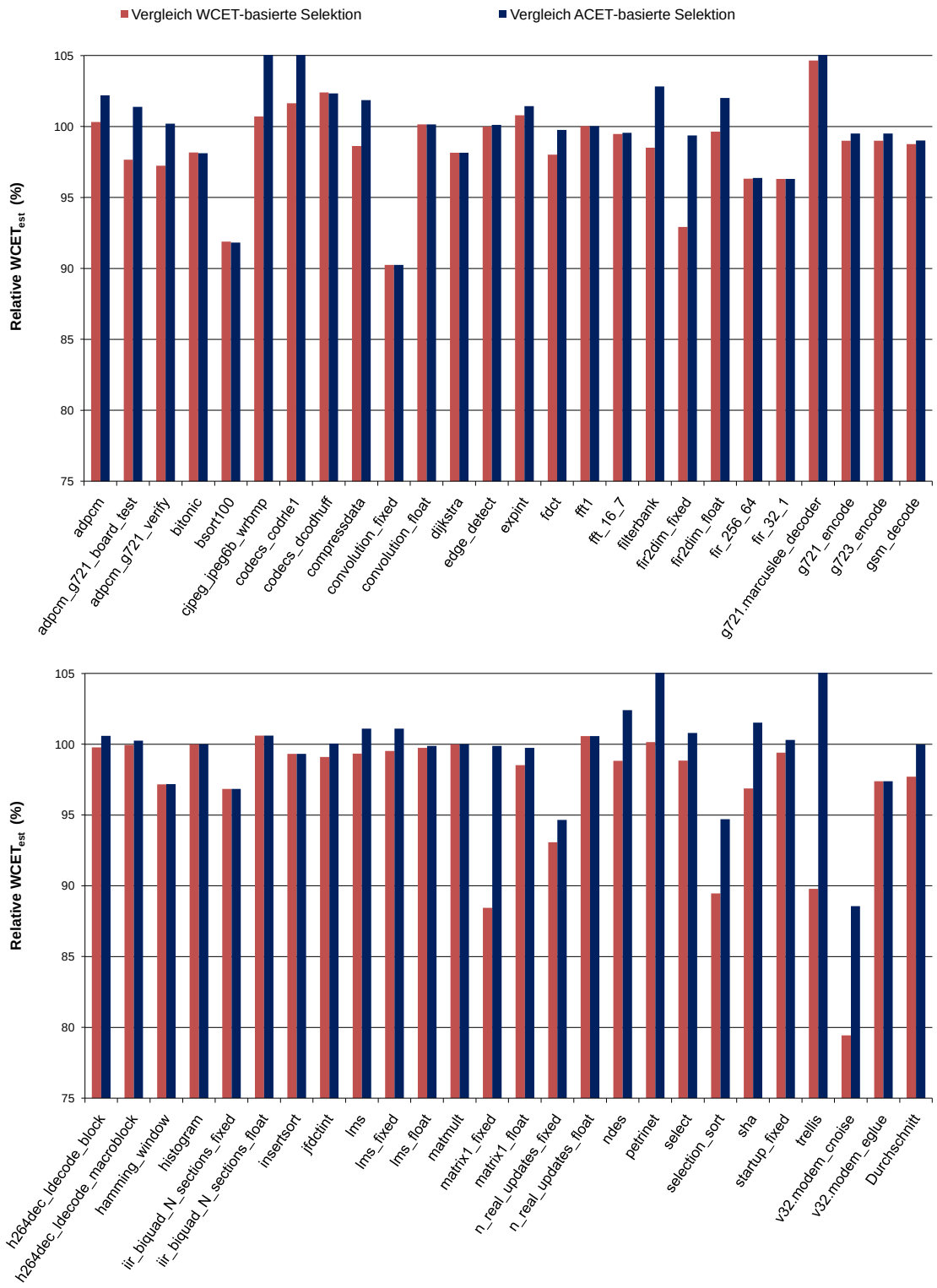


Abbildung 7.2: Einfluss unterschiedlicher Kostenmetriken auf die WCET (100% ≡ WCET O3, lokales Instruction Scheduling).

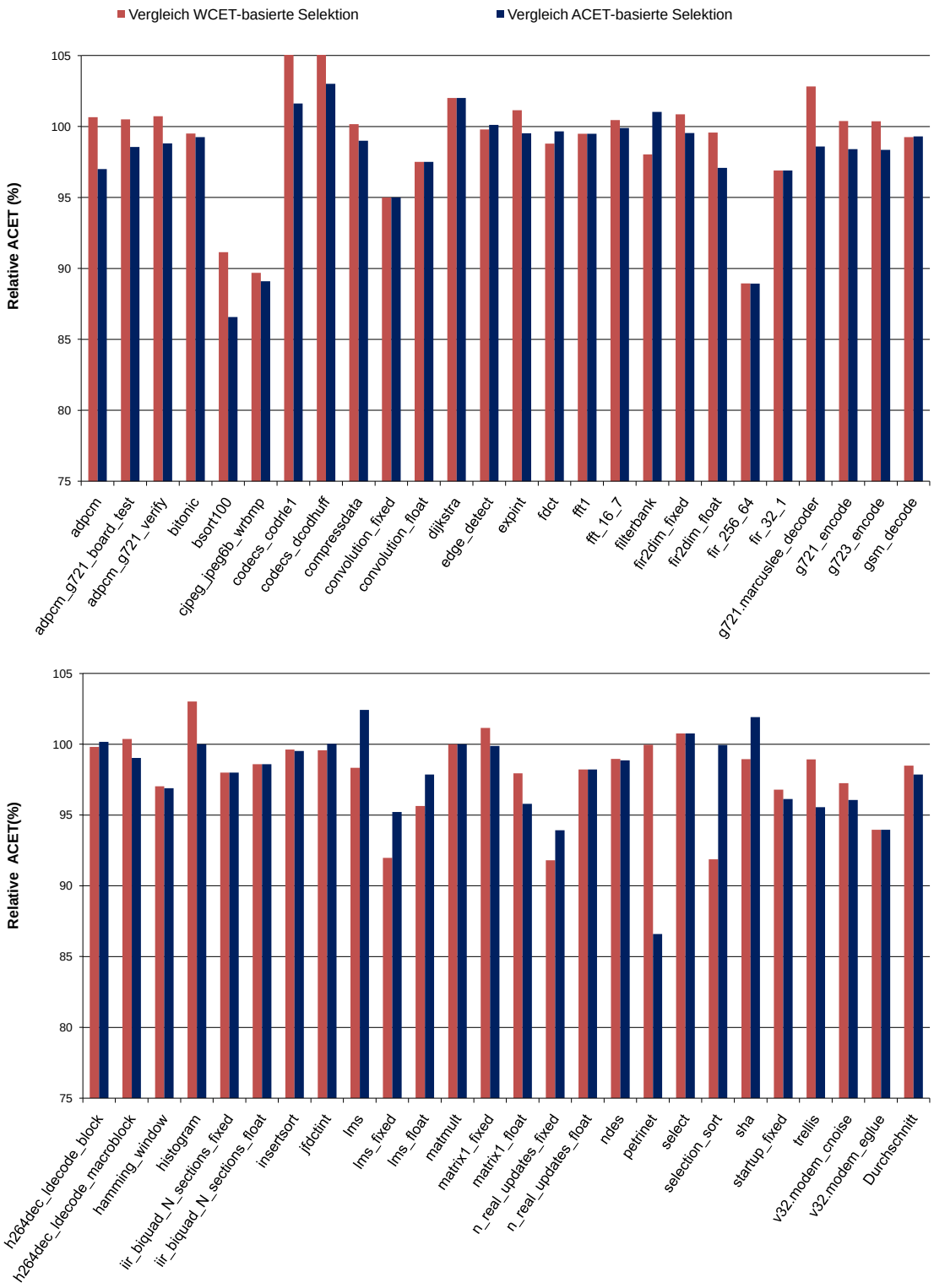


Abbildung 7.3: Einfluss unterschiedlicher Kostenmetriken auf die ACET (100% ≡ ACET O3, lokales Instruction Scheduling).

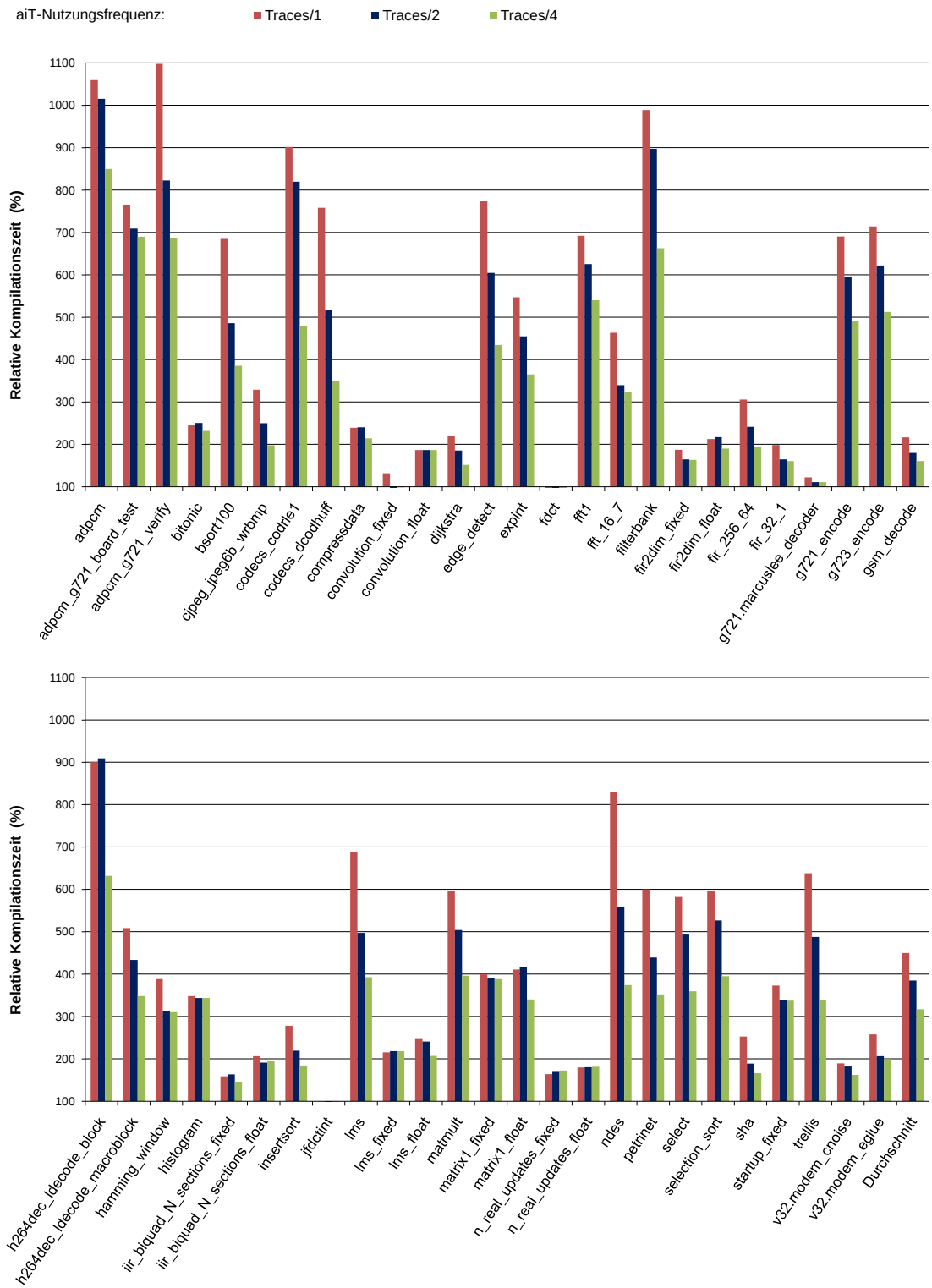


Abbildung 7.4: Auswirkungen reduzierter aiT-Neuberechnungen auf die Kompilationszeit (100% ≡ Kompilationszeit O3, lokales Instruction Scheduling).

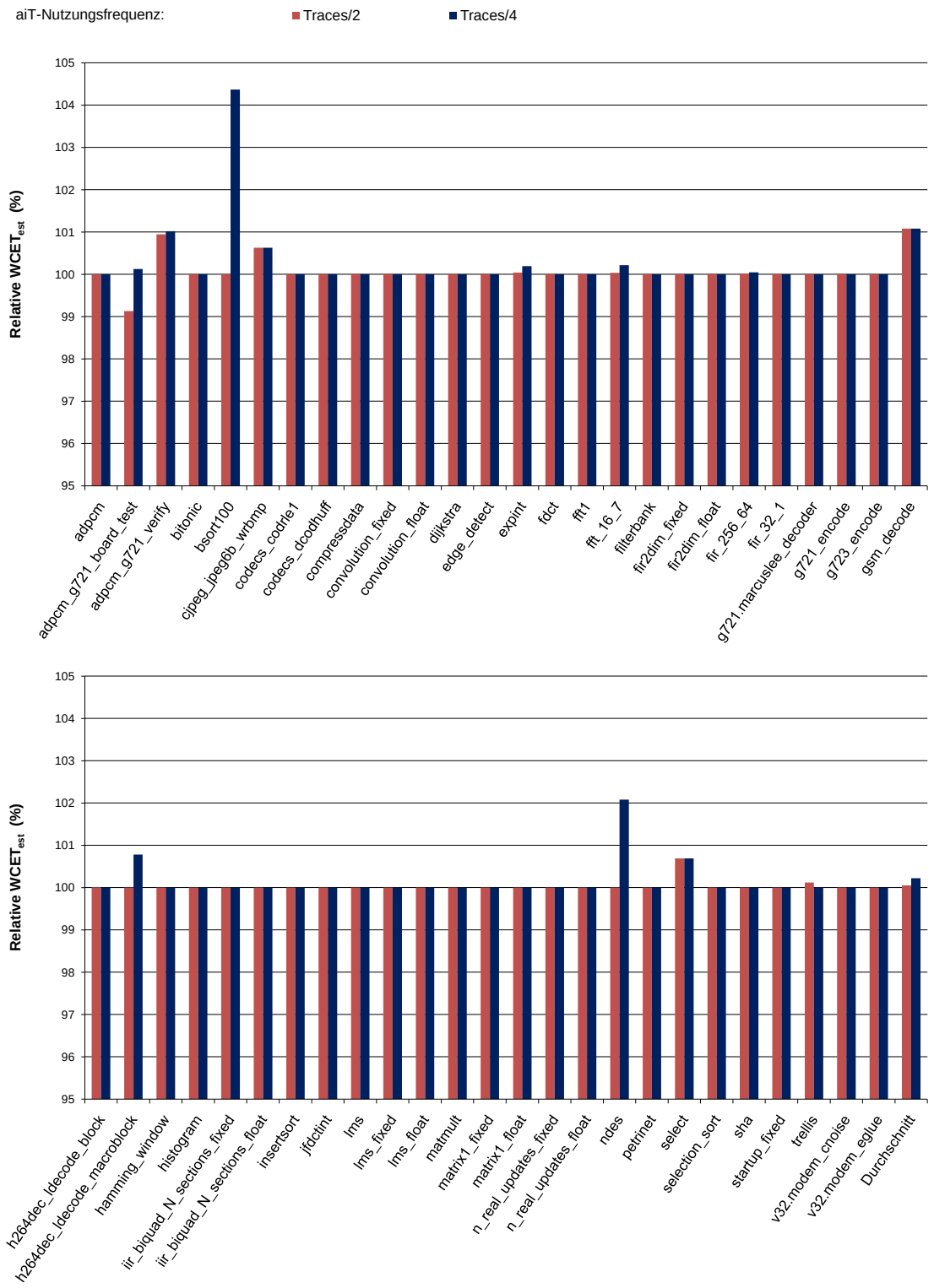


Abbildung 7.5: Auswirkungen reduzierter aiT-Neuberechnungen auf die WCET (100% ≡ Neuberechnung nach jeder Iteration).

7.2.3 Auswirkung reduzierter aiT-Neuberechnung auf die Kompilationszeit

Die Neuberechnung von WCET-Daten nach jedem Trace Scheduling-Durchlauf führt zu einer hohen Präzision der nachfolgenden Trace-Selektion. Dadurch kann ausgeschlossen werden, dass Optimierungen auf Pfaden durchgeführt werden, die nicht mehr auf dem WCEP liegen. Allerdings führt die häufige Neuberechnung durch aiT bei einigen komplexen Benchmarks zu einer deutlich gesteigerten Kompilationszeit im Vergleich zum lokalen Instruction Scheduling bei O3. Im Folgenden soll untersucht werden, welche Auswirkungen sich durch eine Reduktion der Anzahl von aiT-Läufen einerseits auf die Kompilationszeit und andererseits auf die WCET ergeben.

Abbildung 7.4 stellt die relative Kompilationszeit im Vergleich zu O3 mit lokalem Instruction Scheduling dar. Dabei wurde das Trace Scheduling einmal so konfiguriert, dass eine aiT-Neuberechnung durchgeführt wird, nachdem in jeder Funktion des zu übersetzenden Programms ein Trace betrachtet wurde. Die anderen Werte stellen die Ergebnisse bei einer Neuberechnung nach zwei beziehungsweise vier Traces dar.

Wie erwartet führt die notwendige Berechnung von WCET-Daten zu einer teilweise deutlich gesteigerten Kompilationszeit im Vergleich zum lokalen Instruction Scheduling, das nicht auf diese Daten zurückgreift. Durch eine Reduktion der Neuberechnungen kann die benötigte Kompilationszeit in vielen Fällen allerdings deutlich reduziert werden.

Interessant ist das Ergebnis bei den Benchmarks fdct und jfdctint. Bei einer Neuberechnung nach jedem Trace (Traces/1) ist die Kompilationszeit minimal geringer, als bei dem lokalen Instruction Scheduling (96% bzw. 99%). Bei einer Neuberechnung nach jedem vierten Trace (Traces/4) erfolgt bei jfdctint sogar eine Reduktion auf 90%. Eine Erklärung dafür wird nachfolgend geliefert.

Verringerte Kompilationszeit bei fdct und jfdctint

Der Maschinencode für fdct und jfdctint besteht hauptsächlich aus Schleifen, die aus einzelnen, großen Basisblöcken bestehen. So besteht in jfdctint beispielsweise ein Basisblock aus 478 Instruktionen bei insgesamt 821 Instruktionen. Da Trace Scheduling und Superblock Scheduling nicht die Grenzen dieser Schleifen überschreiten können, wird das Instruction Scheduling an diesen Stellen prinzipiell genau wie beim lokalen Instruction Scheduling durchgeführt.

Innerhalb der Basisblöcke kommen zahlreiche Lade- und Speicherinstruktionen vor. Das lokale Instruction Scheduling erzeugt, wie beschrieben, zwischen all diesen Instruktionen Abhängigkeitskanten. Beim Trace Scheduling und Superblock Scheduling kann durch das in Kapitel 4.3 (Seite 35) beschriebene Verfahren in vielen Fällen auf Kanten verzichtet werden. Dadurch ist die anschließend durchgeführte Elimination redundanter Kanten weniger aufwändig und damit schneller durchführbar. Da außerdem die Ermittlung der WCET für diese Benchmarks sehr schnell möglich ist und nur wenige Läufe von aiT notwendig sind, kommt es zu der beobachteten Reduktion der Kompilationszeit.

Deaktiviert man die Verhinderung von nicht benötigten Abhängigkeitskanten zwischen Lade- und Speicherinstruktionen, ergibt sich eine Kompilationszeit, die größer ist als die des lokalen Instruction Scheduling. Außerdem ist in diesem Fall keine Reduktion der WCET mehr möglich, da durch die Einschränkungen der unechten Abhängigkeiten praktisch keine Verschiebungen von Instruktionen möglich sind.

7.2.4 Auswirkung reduzierter aiT-Neuberechnung auf die WCET

Abbildung 7.5 zeigt die Auswirkungen, die sich durch eine reduzierte Anzahl von aiT-Neuberechnungen auf die WCET ergeben. Als Vergleichswert wird die WCET benutzt, die sich ergibt, wenn aiT aufgerufen wird, nachdem in jeder Funktion des zu übersetzenden Programms ein Trace behandelt wurde. Interessanterweise ergibt sich dabei, abgesehen von wenigen Ausnahmen, trotz der verringerten Präzision praktisch keine Verschlechterung der WCET. Im Fall von adpcm_g721_board_test führen seltenere Neuberechnungen von aiT, und die damit verbundene verringerte Präzision sogar zu einer leicht verbesserten WCET.

Insgesamt ergibt sich so eine Erhöhung der WCET im Durchschnitt auf 100,05% beziehungsweise 100,24%. Zusammen mit den deutlich besseren Kompilationszeiten erscheint es daher nicht notwendig und sinnvoll, aiT nach dem Scheduling jedes Traces aufzurufen.

7.2.5 Auswirkungen von Superblock Scheduling auf die WCET/ACET

Als nächstes werden die Ergebnisse des WCET-fähigen Superblock Scheduling untersucht. Abbildung 7.6 stellt die durch das Superblock Scheduling erzielte relative Reduktion der WCET und ACET im Vergleich zum Trace Scheduling dar. Dabei wurde zur Ermittlung der WCET Daten wie bisher die Trace-Selektion eingesetzt, die auf den von aiT ermittelten Daten arbeitet. Zur Ermittlung der ACET-Werte wurde wieder die Profiling-basierte Trace-Selektion genutzt. Abgesehen von der Tail Duplication, die vor dem Superblock Scheduling durchgeführt wird, wurden beide Verfahren mit exakt gleichen Parametern aufgerufen.

Offensichtlich gibt es bei den meisten Benchmarks keine großen Unterschiede zwischen dem Trace Scheduling und dem Superblock Scheduling. Einerseits liegt dies daran, dass in einigen Benchmarks wie schon bei jfdctint gezeigt, die WCET fast ausschließlich innerhalb von verzweigungsfreiem Code zustande kommt. Innerhalb solchen Codes ohne Seiteneingänge verhält sich das Superblock Scheduling exakt wie das Trace Scheduling.

Bei anderen Benchmarks führt das Trace Scheduling Verschiebungen in solch einer Weise durch, dass der benötigte Kompensationscode an Seiteneingängen praktisch dem Code entspricht, der auch von einer Tail Duplication erzeugt wird. Es kommt dadurch zu einer ähnlichen Programmstruktur und zu ähnlichen WCET- sowie ACET-Werten.

Auffällig ist das Ergebnis, das durch das Superblock Scheduling für den Benchmark select erzielt wird. Im Folgenden wird dies genauer betrachtet.

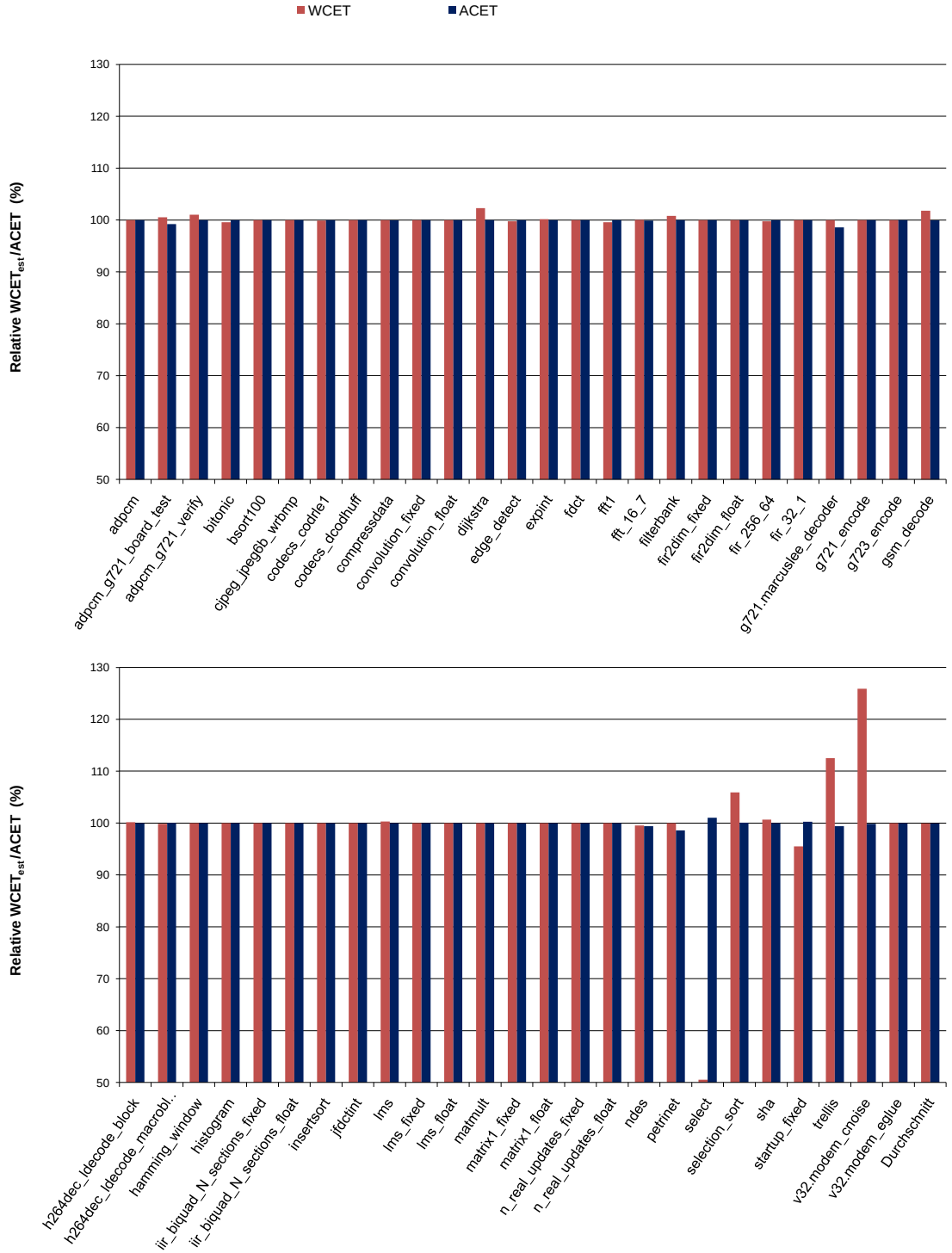


Abbildung 7.6: Relative WCET- und ACET-Werte nach Superblock Scheduling (100% ≡ WCET und ACET nach Trace Scheduling).

WCET-Verbesserung bei Benchmark select durch Superblock Scheduling

Während die WCET-Werte für die meisten Benchmarks bei Trace Scheduling und Superblock Scheduling ähnlich sind, kann im Benchmark select durch das Superblock Scheduling eine Reduktion der WCET auf ungefähr 50% der Werte des Trace Scheduling erzielt werden. Interessant ist dies vor allem, wenn man die Ergebnisse der Analyse von aiT im Verlauf des Superblock Scheduling, wie in Tabelle 7.1 dargestellt, betrachtet. Dabei ist die WCET, die Größe des Programmcodes sowie die Anzahl der von aiT ermittelten Instruction-Cache Misses dargestellt.

Insgesamt wurde aiT acht mal während des Superblock Scheduling von select aufgerufen. Die erste Zeile der Tabelle entspricht den Werten, die vor dem Superblock Scheduling ermittelt wurden.

aiT-Analyse	WCET	Codegröße	Instruction-Cache Misses
1	6924	912	45
2	6874	912	45
3	7125	1002	75
4	8313	1124	127
5	8139	1156	127
6	3445	1202	3221201056
7	3633	1286	3221201059
8	3630	1320	3221201060

Tabelle 7.1: WCET Entwicklung bei Benchmark select während des Superblock Scheduling

Anhand dieser Tabelle lassen sich mehrere Effekte des Superblock Scheduling (und teilweise auch des Trace Scheduling) erkennen.

- Wie bereits angesprochen, kann es während der Optimierung zeitweise zu Verschlechterungen kommen, die teilweise allerdings akzeptiert werden müssen, um lokale Minima zu überwinden. Dies ist beispielsweise zwischen der zweiten und der dritten Zeile der Tabelle zu sehen. Die Werte in der zweiten Zeile entsprechen den Ergebnissen nachdem das Superblock Scheduling einmal durchgeführt wurde. Durch das nächste Superblock Scheduling kommt es zu einer Erhöhung der WCET. Es wäre allerdings ein Fehler gewesen, das Superblock Scheduling direkt nach der festgestellten Verschlechterung abzubrechen, da die Lösung in der zweiten Zeile mit einer WCET von 6874 nur ein lokales Minimum darstellt. Die beste Lösung wird während der sechsten Analyse mit einer WCET von 3445 ermittelt.

- Der in dieser Arbeit implementierte Rollback-Mechanismus ist nicht nur sinnvoll, um auf das unoptimierte Programm zurückfallen zu können, falls Superblock Scheduling und Trace Scheduling zu Verschlechterungen der WCET führen. Es kann immer auf die beste gefundene Lösung zurückgegriffen werden.

- Superblock Scheduling kann unter Umständen zu deutlichem Codewachstum führen. Während des Superblock Scheduling wächst der Programmcode in diesem Bench-

mark auf 144% der ursprünglichen Größe an. Bei der besten ermittelten Lösung liegt das Codewachstum bei 133% im Vergleich zur ursprünglichen Größe.

- Die Tail Duplication und das damit verbundene Codewachstum kann zu einer Steigerung der von aiT ermittelten Instruction-Cache Misses führen. Dies sieht man zwischen Lauf eins und fünf. Zusammen mit der Codegröße steigt auch die Anzahl der Instruction-Cache Misses von 45 auf 127. Die dramatische Steigerung der Instruction-Cache Misses, die ab dem sechsten Lauf ermittelt wird, erscheint dabei unrealistisch.

  Es ist davon auszugehen, dass bei einer solch massiven Anzahl von Instruction-Cache Misses auch die WCET höher sein müsste. In diesem Beispiel sinkt die WCET aber sogar deutlich.

Die Gründe für die deutliche Senkung der WCET sind im Detail schwer nachvollziehbar, da das Superblock Scheduling zu massiven Änderungen des Kontrollflussgraphen des Programms führt. Ein Grund liegt aber offensichtlich darin, dass in der durch die Tail Duplication neu formierten Programmstruktur mehr Programmpfade von aiT als infeasible ausgeschlossen werden können. Dies lässt sich auch am Ergebnis der Werte-Analyse von aiT im fünften Lauf (WCET: 8139) ablesen, das im Folgenden ausschnittsweise dargestellt ist.

```
generating value analysis output
1114 total reads : 184 exact (16.5%)
 448 total writes: 207 exact (46.2%)
```

Es konnte nur von 16,5% der 1114 möglichen lesenden Speicherzugriffe das exakte Ziel bestimmt werden. Von 488 Schreibzugriffen konnten immerhin 46,2% exakt bestimmt werden. Nach der Tail Duplication und dem Superblock Scheduling zeigt sich im sechsten Lauf, wie nachfolgend dargestellt, ein wesentlich besseres Ergebnis.

```
generating value analysis output
444 total reads : 441 exact (99.3%)
335 total writes: 333 exact (99.4%)
```

Offensichtlich konnten einige Programmpfade, die durch die Tail Duplication infeasible wurden, von der weiteren Analyse ausgeschlossen werden, so dass nur noch 444 lesende und 335 schreibende Speicherzugriffe betrachtet werden müssen. Diese Zugriffe konnten von der Werte-Analyse von aiT sehr exakt bestimmt werden (99,3% bzw. 99.4%). Insgesamt ergibt sich dadurch, dass eine innere Schleife, die vorher stark zur WCET beigetragen hat, nach der Tail Duplication und dem Superblock Scheduling nicht mehr auf dem WCEP liegt.

7.2.6 Auswirkungen von Trace Scheduling und Superblock Scheduling auf die Codegröße

Die Verfahren Trace Scheduling und Superblock Scheduling für die TriCore-Architektur zeigen ähnliche Ergebnisse bei der Reduktion der WCET und ACET. Da die Implemen-

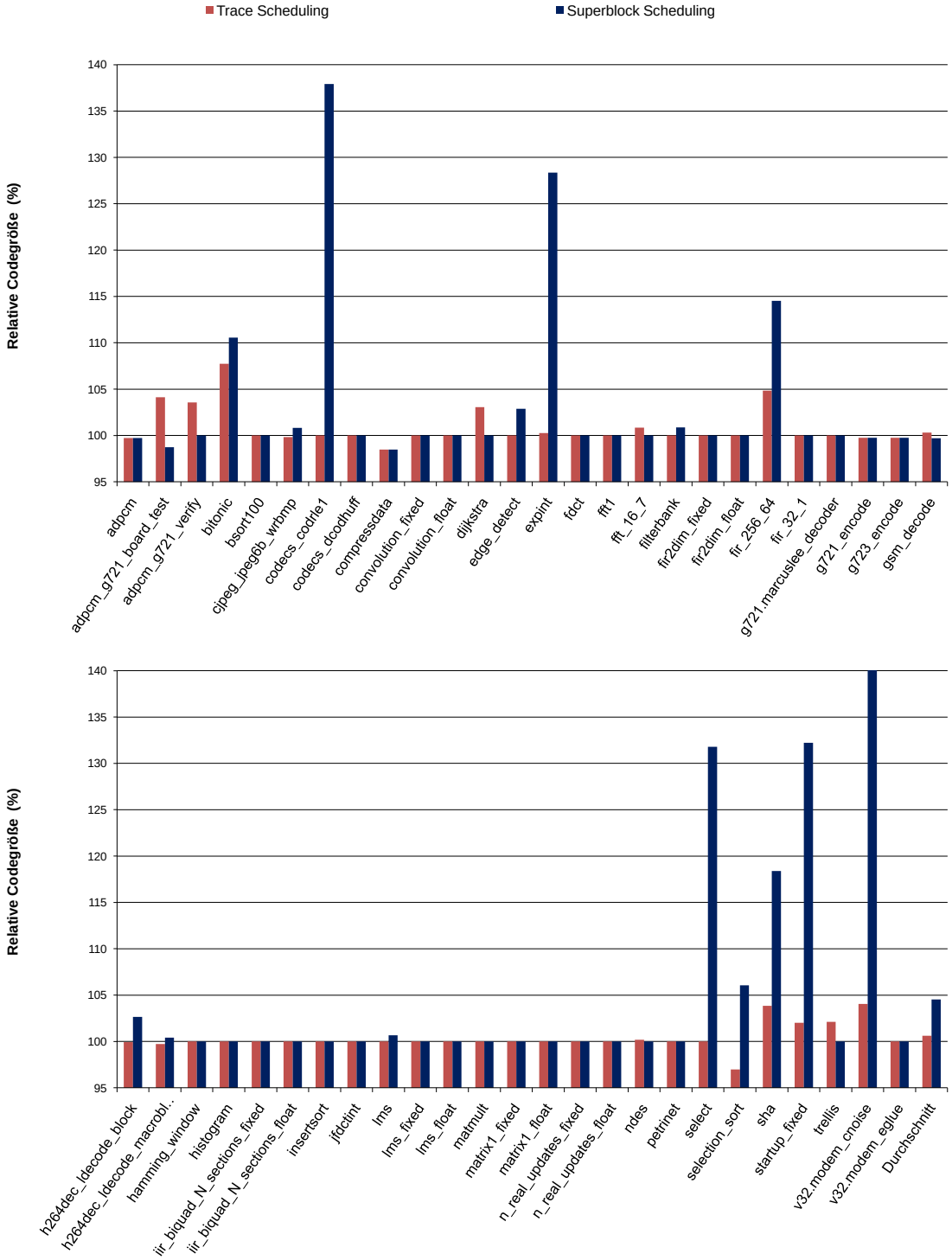


Abbildung 7.7: Relative Codegröße Trace Scheduling und Superblock Scheduling (100% ≡ Codegröße O3, lokales Instruction Scheduling).

tierung des Superblock Scheduling weniger komplex ist, scheint dieses Verfahren Vorteile gegenüber dem Trace Scheduling zu bieten. Als Nachteil des Superblock Scheduling wird allerdings meist ein vergrößertes Codewachstum genannt.

Die Auswirkungen beider Instruction Scheduling-Verfahren auf die Codegröße im Vergleich zum Code, der bei der Optimierungsstufe O3 mit lokalem Instruction Scheduling erzeugt wird, ist in Abbildung 7.7 dargestellt. Insgesamt zeigt sich dabei das erwartete Ergebnis, dass Trace Scheduling im Durchschnitt zu geringerem Codewachstum führt, als Trace Scheduling. Allerdings ist auch das Codewachstum beim Superblock Scheduling abgesehen von einigen deutlichen Ausreißern zu vernachlässigen.

Interessanterweise führen beide Instruction Scheduling-Verfahren bei einigen Benchmarks wie compressdata oder adpcm wider Erwarten sogar zu einer Reduktion der Codegröße. Erklären lässt sich dies durch spekulative Verschiebungen, die anschließend zu einer Vereinfachung des Kontrollflussgraphen führen können. Dies macht es nachfolgend möglich, den Maschinencode kompakter darzustellen. Bei dem Benchmark selection_sort sind solche Vereinfachungen nur beim Trace Scheduling möglich, so dass es hier zu einer Verkleinerung des Programmcodes auf 97% kommt, während es beim Superblock Scheduling zu einer Vergrößerung auf 106% kommt.

7.3 Zusammenfassung

Insgesamt zeigt sich, dass sowohl das WCET-fähige Trace Scheduling als auch das WCET-fähige Superblock Scheduling teilweise zu deutlichen Änderungen der Programmstruktur führen. Durch die Einfügung von Kompensationscode kann die Semantik der optimierten Programm trotzdem erhalten bleiben. Durch den Zugriff auf Profiling-Daten ist des Weiteren neben der Reduktion der WCET auch eine gezielte Reduktion der ACET möglich.

Im Vergleich zum lokalen Instruction Scheduling sind die erzielten Gewinne allerdings bei beiden Verfahren und bei beiden Optimierungszielen, ausgenommen von einigen Ausnahmen, meist gering. Mögliche Gründe dafür sollen hier zusammengefasst werden.

- Trace Scheduling und darauf aufbauend Superblock Scheduling sind ursprünglich für VLIW Architekturen beschrieben worden, die zahlreiche *Issue Slots* aufweisen, und somit einen hohen Instruction Level Parallelism ermöglichen (ILP). Innerhalb eines einzelnen Basisblocks existieren allerdings meist nur sehr wenige unabhängige Instruktionen, so dass viele Slots ungenutzt bleiben. Globale Instruction Scheduling-Verfahren bieten durch die simultane Betrachtung mehrerer Basisblöcke bei diesen Archtitekturen Vorteile.

  Bei der TriCore-Architektur ist allerdings maximal ein gleichzeitige Laden von zwei Instruktionen in die Pipeline möglich. Dadurch ließe sich also selbst im Idealfall maximal ein Instructions per Cycle-Wert(IPC) von zwei erreichen. Das mögliche Optimierungspotential ist für Trace Scheduling und Superblock Scheduling demnach weniger hoch, als bei VLIW Architekturen.

- Um einen IPC-Wert von zwei zu erreichen, müsste der Fetch-Stufe des TriCore in jedem Zyklus eine IP-Instruktion und eine gleichzeitig ausführbare LS-Instruktion zur Verfügung stehen. Für die betrachten Benchmarks ist das Verhältnis von IP- zu LS-Instruktionen allerdings oft sehr unausgeglichen.

  Bei dem Benchmark jfdctint kommen beispielsweise nur 158 IP-Instruktionen bei insgesamt 821 Instruktionen vor. 645 Instruktionen sind bei dem vom WCC für die Optimierungsstufe O3 generierten Code LS-Instruktionen. Selbst wenn man davon ausgehen würde, dass vor dem Instruction Scheduling kein einziges Bündel aus IP- und LS-Instruktionen existiert hat, könnte ein Instruction Scheduling im Idealfall nur 158 Bündel bilden. 487 LS-Instruktionen müssten weiterhin einzeln ausgeführt werden. Das unausgeglichene Verhältnis von Instruktionen stellt hier also allgemein eine Einschränkung für Instruction Scheduling-Verfahren dar.

- Eine weiterer Punkt, an dem Instruction Scheduling-Verfahren generell eine große Möglichkeit zur Reduktion der Laufzeit aufweisen, ist eine Verhinderung von Stall-Zyklen. Allerdings sind auch hierfür bei der TriCore-Architektur die Möglichkeiten eingeschränkt. Beispielsweise kann bei vielen Prozessorarchitekturen ein Instruction Scheduling zu positiven Effekten führen, indem zwischen einer Ladeinstruktion und der ersten Verwendung des beschriebenen Registers, eine unabhängige Instruktion platziert wird. Mögliche Pipeline-Stalls, die bei Cache-Misses auftreten, können so verhindert werden.

  Bei der TriCore-Architektur bringt solch ein Vorgehen keine Vorteile: Ist das zu ladende Datum im Cache vorhanden, so kann es direkt im nächsten Zyklus benutzt werden. Kommt es dagegen zu einem Cache-Miss, führt dies zwangsläufig zu einem Pipeline-Stall. Es ist in diesem Fall in der TriCore-Architektur keine Ausführung einer unabhängigen Instruktion möglich.

- Dabei kann ein Cache-Miss vor allem aufgrund der verwendeten Flash-Architektur unter Umständen sehr teuer sein. Dies ist von zahlreichen Faktoren abhängig, auf die ein Instruction Scheduling keinen Einfluss hat.

- Weiterhin können bereits kleinste Änderungen des Codelayouts zu einem für die TriCore-Architektur ungünstigerem Alignment führen. Dadurch können teure *Line-Crossing*-Effekte auftreten, die kaum vorhersehbar sind, aber vor allem eine große Auswirkung auf die WCET haben.

Ein weiteres Problem hängt nicht direkt mit der TriCore-Architektur zusammen, sondern ist ein generelles Problem beider betrachteten Instruction Scheduling-Verfahren:
Trace Scheduling und Superblock Scheduling setzen das Vorhandensein eines eindeutig präferierten Programmpfades voraus. Falls solch ein Pfad existiert, spielen Verschlechterungen durch Kompensationscode auf selten ausgeführten Pfaden keine entscheidende Rolle.

In kontrollflussintensiven Programmen ist solch ein Pfad allerdings oft nicht vorhanden, so dass eine Verbesserung auf einem Pfad durch Kompensationscode auf einem anderen Pfad ausgeglichen wird. Durch die angesprochenen Eigenschaften der Hardware-Architektur

können die negativen Effekte derart ausgeprägt sein, dass insgesamt keinerlei Vorteile durch das Instruction Scheduling erzielt werden. Bei der Optimierung der WCET kommt dabei das bereits häufiger angesprochene Problem des Pfadwechsels hinzu.

Ein Instruction Scheduling-Verfahren wie Hyperblock-Scheduling [MLC+92] würde solch eine Benachteiligung alternativer Kontrollflusspfade verhindern. Allerdings ist eine Implementierung für die TriCore-Architektur nicht möglich, da die dazu benötigten, bedingten Instruktionen nicht unterstützt werden.

8 Zusammenfassung und Ausblick

Ziel dieser Diplomarbeit war es, durch einen Einsatz globaler Instruction Scheduling-Verfahren eine gezielte Reduktion der WCET zu ermöglichen. Um dieses Ziel zu erreichen war zunächst die Implementierung eines Trace-Selektions-Verfahren notwendig, das von dem Analysewerkzeug aiT ermittelte WCET-Daten nutzen kann, um so einen für die Optimierung möglichst gut geeigneten Programmpfad zu bestimmen (Kapitel 5.2). Um zu zeigen, dass zur Reduktion der WCET andere Pfade, als zu einer Reduktion der durchschnittlichen Programmlaufzeit (ACET) gewählt werden müssen, wurde eine Möglichkeit geschaffen, während der Trace-Selektion unterschiedliche Kostenmetriken zu nutzen (Kapitel 5.2.1).

Aufbauend auf den ermittelten Traces wurde eine Möglichkeit geschaffen, den Abhängigkeitsgraph der in den Traces enthaltenen Instruktionen so zu erzeugen (Kapitel 5.4), dass der im WCC vorhandene lokale List Scheduler (Kapitel 4.4) genutzt werden kann, um ein Instruction Scheduling auf Traces durchzuführen. Da es dadurch zu Änderungen der Semantik auf Off-Trace Pfaden kommen kann, wurde das von Fisher [Fis81] beschriebene Bookkeeping implementiert, um Kompensationscode zu erzeugen (Kapitel 5.3). Um dabei unnötige Kopien von Instruktionen, wenn möglich, zu verhindern, wurde die im WCC vorhandene Lebendigkeitsanalyse für Register genutzt, um tote Instruktionen nicht zu kopieren Kapitel 5.3.2.

Darauf aufbauend wurde ein weiteres WCET-fähiges Instruction Scheduling-Verfahren implementiert, dass den von Mahlke et al. [HMC+93] beschriebenen Superblock als Basis nutzt (Kapitel 6). Da dazu die implementierte Trace-Selektion und das Bookkeeping des Trace Scheduling wiederverwendet werden konnte, war zusätzlicher Implementierungsaufwand hauptsächlich für die Tail Duplication notwendig (Kapitel 6.2). Die Tail Duplication erzeugt aus allgemeinen Traces Superblöcke, auf denen anschließend das Instruction Scheduling durchgeführt wird (Kapitel 6.2).

Bei der Evaluation der beiden neu implementierten Verfahren zeigte sich, dass trotz der teilweise stattfinden massiven Änderungen der Programmstruktur die Semantik der Programme durch Kompensationscode in korrekter Weise korrigiert wird (Kapitel 7.1).

Bei der Evaluation der Leistungsfähigkeit zeigte sich allerdings, dass die Reduktion der WCET bei beiden Verfahren über einen großen Querschnitt an Benchmarks, im Vergleich zum lokalen Instruction Scheduling, relativ gering ist (Kapitel 7.2). Einerseits liegt dies an der bekannten Problematik der Pfadwechsel, die während des globalen Instruction Scheduling auftreten konnen (Kapitel 2.5.1).

Andererseits führte auch das Profiling-basierte Trace Scheduling und Superblock Scheduling zu relativ geringen Verbesserungen der ACET im Vergleich mit dem lokalen Instruc-

tion Scheduling. Offensichtlich spielen daher auch andere Gründe eine Rolle für die begrenzte Leistungsfähigkeit beider Verfahren bei einem Einsatz für die TriCore-Architektur. Mögliche Gründe wurden in Kapitel 7.3 zusammengefasst.

8.1 Ausblick

Es wurde in dieser Arbeit gezeigt, dass die Menge an benötigtem Kompensationscode beim Trace Scheduling meist relativ gering ist. Falls allerdings Kompensation an Seiteneingängen notwendig wird, führt dies oft zu deutlichen Änderungen der Programmstruktur und in der TriCore-Architektur zu negativen Effekten, die die positiven Effekte oft überwiegen.

In [FGL94] beschreiben Freudenberger et al. mehrere Möglichkeiten um die Menge an benötigtem Kompensationscode beim Trace Scheduling zu verringern. Eine Vermeidung von redundanten Split-Instruktionen wurde in dieser Diplomarbeit bereits implementiert. Freudenberg et al. ermittelten relativ geringe Verbesserungen der ACET durch eine mögliche Vermeidung von Rejoin-Kompensation. Begründet wurde dies dadurch, dass selten ausgeführte Pfade wenig zur ACET beitragen, und daher zusätzlichen Join-Kopien nicht kritisch sind.

Bei einer Optimierung der WCET könnte die Vermeidung von Rejoin-Kompensation eventuell Pfadwechsel verhindern. Daher wäre eine Überprüfung interessant, wie groß die Möglichkeiten in der TriCore-Architektur sind um Rejoin-Kompensation zu verhindern und welche Auswirkungen dies auf die WCET hat.

Für ein Superblock Scheduling empfehlen Mahlke et al. [HMC+93] als ersten Schritt nach Bestimmung eines Superblocks eine Restrukturierung des Low-Level Code des Superblocks. Dabei werden Basisblöcke, die im Superblock benachbart sind, so verschoben, dass sie auch im Maschinencode benachbart sind. Dadurch soll sich ein verbessertes Cacheverhalten bei der Ausführung des Superblocks ergeben. Allerdings kann, durch das dadurch verursachte schlechtere Cacheverhalten anderer Basisblöcke, ein Pfadwechsel auftreten. Es wäre zu untersuchen, welche Effekte diese Superblock-basierte Code Restrukturierung auf die WCET hat.

Als weitere Optimierungen vor einem Superblock Scheduling werden in [HMC+93] die Superblock-basierten Optimierungen *Branch Target Expansion*, *Loop Peeling* und *Loop Unrolling* empfohlen. Ziel dieser Optimierungen ist eine Vergößerung der ermittelten Superblöcke. Dies soll die Möglichkeiten, unabhängige Instruktionen für das Instruction Scheduling zu finden, erhöhen.

Auch durch ein *Register Renaming* [Kuc78] lässt sich eine erhöhte Anzahl unabhängiger Instruktionen und damit mehr Potential für ein Instruction Scheduling erreichen. Register Renaming kann im Allgemeinen zahlreiche unechte Ausgabe- und Gegenabhängigkeiten, die durch die Wiederverwendung von Registern entstehen, auflösen. Durch die gesteigerten Möglichkeiten Instruktionen zu verschieben, könnnte dabei sowohl das Trace Scheduling, als auch das Superblock Scheduling durch ein Register Renaming profitieren.

Literaturverzeichnis

[Abs10] AbsInt Angewandte Informatik GmbH: *Worst-Case Execution Time Analyzer aiT for TriCore*. `http://www.absint.com/ait`, March 2010

[BBI10] BITKOM Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.: *Eingebettete Systeme – Ein strategisches Wachstumsfeld für Deutschland*. `http://www.bitkom.org/files/documents/EingebetteteSysteme_web.pdf`, March 2010

[EJ09] Ebert, Christof ; Jones, Capers: Embedded Software: Facts, Figures, and Future. In: *Computer* 42 (2009), April

[FGL94] Freudenberger, Stefan M. ; Gross, Thomas R. ; Lowney, P. G.: Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler. In: *ACM Transactions on Programming Languages and systems* 16 (1994)

[Fis81] Fisher, Joseph A.: Trace Scheduling: A Technique for Global Microcode Compaction. In: *IEEE Transactions on Computers* 30 (1981), Nr. 7

[FK09] Falk, Heiko ; Kleinsorge, Jan C.: Optimal static WCET-aware scratchpad allocation of program code. In: *Proceedings of the 46th Annual Design Automation Conference (DAC)*. New York, NY, USA, 2009

[FS06] Falk, Heiko ; Schwarzer, Martin: Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In: *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Seoul/Korea, October 2006

[Gre01] Gregg, David: Comparing Tail Duplication with Compensation Code in Single Path Global Instruction. In: *Proceedings of the 9th International Conference on Compiler Construction (CC)*, 2001

[HBC98] Havanki, W. ; Banerjia, S. ; Conte, T.: Treegion Scheduling for Wide Issue Processors. In: *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA, 1998

[HD86] Hsu, P. Y T. ; Davidson, E. S.: Highly concurrent scalar processing. In: *Proceedings of the 13th annual international symposium on Computer architecture (ISCA)*. Los Alamitos, CA, USA, 1986

[HMC+93] Hwu, Wen-Mei W. ; Mahlke, Scott A. ; Chen, William Y. ; Chang, Po-

hua P. ; WARTER, Nancy J. ; BRINGMANN, Roger A. ; OUELLETTE, Roland G. ; HANK, Richard E. ; KIYOHARA, Tokuzo ; HAAB, Grant E. ; HOLM, John G. ; LAVERY, Daniel M.: The Superblock: An effective technique for VLIW and superscalar compilation. In: *THE JOURNAL OF SUPERCOMPUTING* 7 (1993)

[HMNS01] HANSMANN, Uwe ; MERK, Lothar ; NICKLOUS, Martin ; STOBER, Thomas: *Pervasive Computing Handbook*. Springer-Verlag, Heidelberg, 2001

[Inf10a] INFINEON: *TriCore® Compiler Writer's Guide*. `http://www.infineon.com/dgdl/inf0010_v1_4Dec2003_1.pdf?folderId=db3a304412b407950112b40f8a931422&fileId=db3a304412b407950112b40f8aad1423`, March 2010

[Inf10b] INFORMATIK 12, TU D.: *The WCET-aware C Compiler WCC*. `http://ls12-www.cs.tu-dortmund.de/research/activities/wcc/infrastructure/index.html`, March 2010

[Inf10c] INFORMATIK CENTRUM DORTMUND: *ICD-C Compiler framework*. `http://www.icd.de/es/icd-c`, March 2010

[Inf10d] INFORMATIK CENTRUM DORTMUND: *ICD Low Level Intermediate Representation Backend Infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund, 2010

[Kel09] KELTER, Timon: *Superblock-basierte High-Level WCET-Optimierungen*, TU Dortmund, Diplomarbeit, 2009

[Kuc78] KUCK, David L.: *Structure of Computers and Computations*. New York, NY, USA : John Wiley & Sons, Inc., 1978

[LCFM09] LOKUCIEJEWSKI, Paul ; CORDES, Daniel ; FALK, Heiko ; MARWEDEL, Peter: A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models. In: *International Symposium on Code Generation and Optimization (CGO)*. Seattle / USA, March 2009

[LFK+92] LOWNEY, P. G. ; FREUDENBERGER, Stefan M. ; KARZES, Thomas J. ; LICHTENSTEIN, W. D. ; NIX, Robert P. ; O'DONNELL, John S. ; RUTTENBERG, John C.: *The Multiflow Trace Scheduling Compiler*. 1992

[LFM08] LOKUCIEJEWSKI, Paul ; FALK, Heiko ; MARWEDEL, Peter: WCET-driven Cache-based Procedure Positioning Optimizations. In: *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*. Washington, DC, USA, 2008

[LFMT08a] LOKUCIEJEWSKI, Paul ; FALK, Heiko ; MARWEDEL, Peter ; THEILING, Henrik: WCET-Driven, Code-Size Critical Procedure Cloning. In: *The 11th Interna-*

tional Workshop on Software & Compilers for Embedded Systems (SCOPES). Munich / Germany, March 2008

[LFMT08b] LOKUCIEJEWSKI, Paul ; FALK, Heiko ; MARWEDEL, Peter ; THEILING, Henrik: WCET-driven, code-size critical procedure cloning. In: *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*. New York, NY, USA, 2008

[LKM10] LOKUCIEJEWSKI, Paul ; KELTER, Timon ; MARWEDEL, Peter: Superblock-Based Source Code Optimizations for WCET Reduction. In: *Proceedings of the 7th IEEE International Conferences on Embedded Software and Systems (ICESS)*. Bradford, UK, June 2010

[Mar06] MARWEDEL, Peter: *Embedded System Design*. 2nd edition. Springer Verlag, 2006

[MLC+92] MAHLKE, Scott A. ; LIN, David C. ; CHEN, William Y. ; HANK, Richard E. ; BRINGMANN, Roger A.: Effective Compiler Support for Predicated Execution Using the Hyperblock. In: *In Proceedings of the 25th International Symposium on Microarchitecture*, 1992

[PLM09] PLAZAR, Sascha ; LOKUCIEJEWSKI, Paul ; MARWEDEL, Peter: WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In: *Proceedings of The 9th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2009

[Puc09] PUCYK, Thomas: *Lokale und globale Instruction Scheduling-Verfahren für den TriCore Prozessor*, TU Dortmund, Diplomarbeit, 2009

[Sch08] SCHMOLL, Florian: *ILP-basierte Registerallokation unter Ausnutzung von WCET-Daten*, TU Dortmund, Diplomarbeit, 2008

[SS08] SRIKANT, Y.N. ; SHANKAR, Priti: *The Compiler Design Handbook - Optimizations and Machine Code Generation*. CRC Press, 2008

[Tur09] TURLEY, Jim: Embedded Systems - Connecting Everyone to Everything. In: *Technology Radar Feature Paper* (2009)

[WLH00] WILKEN, Kent ; LIU, Jack ; HEFFERNAN, Mark: Optimal instruction scheduling using integer programming. In: *SIGPLAN Not.* 35 (2000), May

[ZKW+06] ZHAO, Wankang ; KREAHLING, William ; WHALLEY, David ; HEALY, Christopher ; MUELLER, Frank: Improving WCET by applying worst-case path optimizations. In: *Real-Time Syst.* 34 (2006), October

Abbildungsverzeichnis